Canevas de domaines pour l'intégration de données

Mourad Alia

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque

|_|_|_|_|_|_|_|_|_|_|

T H E S E

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : « Informatique : Systèmes et Logiciels »

préparée au laboratoire MAPS/AMS de France Télécom R&D et au laboratoire Logiciels, Systèmes Réseaux (LSR)

dans le cadre de **l'Ecole Doctorale** ***"Mathématique, Sciences et Technologies de l'Information, Informatique"***

présentée et soutenue publiquement

par

Mourad Alia

Le 30 juin 2005

Titre :

Canevas de domaines pour l'intégration de données

Directeur de thèse :

Christine Collet

JURY

M. Jacques Mossière	Président
M. Bruno Defude	Rapporteur
M. Philippe Merle	Rapporteur
Mme. Christine Collet	Directeur de thèse
M. Alexandre Lefebvre	Co-Directeur de thèse
M. Jean-Marc Geib	Examinateur
M. Pascal Déchamboux	Examinateur

Je dédie ce travail à :

la mémoire de ma mère et de ma soeur Taous,

mon père,

Remerciements

Cette thèse a été effectuée au laboratoire de France Télécom Recherche et Développement MAPS/AMS en coopération avec le laboratoire Logiciel Systèmes et Reseaux (LSR) de l'Institut d'Informatique et de Mathématiques Appliquées de Grenoble (IMAG). La diversité et la richesse des travaux menés au sein de ces deux laboratoires m'ont véritablement permis de progresser et d'accéder au monde de la recherche.

Je tiens à remercier M. Jacques Mossière, professeur à l'Institut National Polytechnique de Grenoble, pour avoir accepté de présider le Jury de cette thèse. Mes remerciements vont également aux rapporteurs, M. Bruno Defude, professeur à l'Institut National des Télécommunications (INT) d'Evry et M. Philippe Merle, Chargé de recherche à l'INRIA et au Laboratoire de l'Informatique Fondamentale de Lille (LIFL), pour avoir pris le temps de lire et de commenter ce manuscrit. Je remercie également M. Jean-Marc Geib, Directeur du Laboratoire d'Informatique Fondamentale de Lille (LIFL) et M. Pascal Déchamboux, responsable de recherche et de développement au sein du groupe France Télécom.

Je tiens à remercier M. Alexandre Lefebvre, responsable d'unité R&D au sein de France Télécom, avec qui j'ai fait mon cheminement. Les moments de travail passés ensemble m'ont permis d'avancer, et la libérté qu'il m'a laissée m'a aidé à me forger. Je lui suis aussi reconnaissant pour le temps qu'il a passé à relire et corriger mes rapports et nos articles.

Je n'oublierai pas Mme Christine Collet, Professeur à l'Institut National Polytechnique de Grenoble, pour les conseils et l'encadrement qu'elle m'a prodigués durant cette thèse. Sa confiance et ses encouragements ont été décisifs pour l'aboutissement de ce travail.

Je remercie Mme Kathleen Milsted, Responsable chef du laboratoire Architecture et Ingénierie des plateformes Middlewares et Services Intégrées (MAPS/AMS) de France Télécom pour son accueil au sein du laboratoire.

Je tiens absolument à remercier M. Frank Eliassen, Professeur de l'université d'Oslo et responsable de recherche au sein de l'équipe Systèmes Répartis et Réseaux du laboratoire Simula, pour son accueil chaleureux dans son pays, la Norvège, ainsi que dans son équipe de recherche.

Toute mon amitié à Mickaël et Abdeslam pour leur soutien pendant les dernières minutes de rédaction, Romain pour les agréables moments qu'on a passés au bureau, Aimé, avec qui les discussions sur le caravaning, les roses et RT Linux ont créé d'agréables pauses café, Marc, Thierry, Braham, Brahim, Mustapha, Younes, Naga, Levent, Tahar, Jean-Charles, Jacques, Jean-Philippe, François, Anne marie, Ayman, Hacen, Djamel, Tanguy, Trinh, Genarro, Fabrice, Christophe (et beaucoup d'autres) pour tous les instants partagés.

Je remercie chaleureusement ma grande famille qui m'a toujours soutenu.

Je remercie Nathalie pour sa tendresse et sa grande qualité humaine et aussi pour le temps considérable qu'elle a passé à la relecture de ce manuscrit.

Résumé

Canevas de domaines pour l'intégration de données :

Beaucoup de travaux de recherche autour des systèmes d'intégration de données se sont concentrés sur les problèmes liés aux schémas, aux langages et au traitement de requêtes. Dans cette thèse, nous nous intéressons à la problématique de construction de tels systèmes. Nous appliquons les concepts architecturaux des systèmes répartis, notamment de canevas et de composant logiciel, pour proposer un intergiciel d'intégration de données offrant quatre niveaux d´adaptabilité.

Au niveau architectural, nous définissons le canevas de domaine de données qui est le composant central de l'intégration. Un système d'intégration de données est défini par la composition de domaines de données. Au niveau modèle de données, nous adoptons un modèle hybride doté d'un typage flexible, associé au langage d'expression des vues de domaines, qui permet de prendre en compte l'hétérogénéité structurelle des sources (ou domaines) de données à intégrer. Au niveau langage de requêtes, nous proposons un canevas d'expression qui permet de décrire les requêtes à la fois pour l'intégration et pour l'interrogation de domaines, indépendamment d'une syntaxe particulière. Au niveau optimisation, nous proposons un canevas d'optimisation de requêtes adaptable, dans le sens où il permet de construire et de supporter diverses stratégies de recherche.

Le canevas d'expression et le canevas d'optimisation de requête ont été implantés dans le cadre du consortium ObjectWeb. Ces implantations ont été utilisées dans la mise en uvre de deux standards de persistance d'objets Java, à savoir JDO (Java Data Object) et EJB-CMP (Container Managed Persistance). Dans le contexte de recherche de composants pour des besoins d'administration des systèmes à large échelle, nous avons proposé une utilisation de notre intergiciel pour proposer un service de requête qui permette de rechercher des composants dans un gisement et dans des systèmes en cours d'exécution, par introspection associative.

Mots clés : *Intégration de données, Architecture de systèmes, Requêtes, Modèles de données, Canevas logiciel, Composants logiciels, Architectures adaptables.*

Abstract

Data domains framework for data Integration :

The focus of many research works on data integration has been on problems inherent to semantic and schema integration, query languages and query processing. In this thesis, one we focus on the problematic of the construction of such systems. We apply architectural concepts of distributed systems, namely those of framework and software components, in order to propose a data integration middleware offering four levels of adaptability.

At the architectural level, we define a framework of data domains, the main software component unit for data integration. A data integration system is defined by the composition of data domains. At the data model level, we adopt a hybrid data model with a flexible type system that takes into account the structural heterogeneity of the data models of data sources. At the query language level, we propose a query expression framework that allows the description of queries for both views definition and data domain interrogation, independently of a particular syntax. At the optimization level, we propose an adaptable query optimization framework, in sense that it allows the construction and support of diverse search strategies.

The query expression and optimization frameworks have been implemented as part of ObjectWeb consortium projects. These implementations have been used for developpingment two java object persistence standards, namely JDO (Java Data Object) and EJB-CMP (Container Managed Persistence). In the context of component retrieval for large scale administration needs, we propose to use our middleware for a query service system making it possible to retrieve components from both repository of component templates and running systems through associative introspection.

Key words : *Data integration, system architecture, query languages and processing, software framework, software components, adaptable architectures.*

Table des matières

Table des figures

Chapitre 1

Introduction

Tout art et toute recherche, de même que toute action et toute délibération réfléchie, tendent, semble–t–il, vers quelque bien, toutefois il parait bien qu'il y a une différence entre les fins.

Aristote " L'Ethique à Nicomaque "

1.1 Contexte et Motivations

Les avancées considérables en réseaux et en infrastructures de communication ont conduit à la prolifération et à l'expansion des systèmes d'information à large échelle. Dans ces systèmes cohabitent des données et des services qui manipulent ces données. Ces données sont souvent gérées par des sources de données hétérogènes et réparties, et les systèmes d'intégration de données ont pour objectif de répondre au besoin d'offrir des interfaces d'accès uniformes à ces données. Cependant, la prise en compte de ce besoin conduit à deux phénomènes : la complexité des systèmes et l'émergence de standards.

– *Complexité des systèmes d'intégration :*

La construction des systèmes d'intégration n'est pas une tâche facile et les vingt dernières années de recherche en la matière témoignent de cette complexité. En effet, les sources stockent souvent des données de différents domaines ayant donc des sémantiques variables. Ces données sont également représentées dans différents formats. Elles peuvent être structurées, comme c'est le cas des bases de données classiques (relationnel et objet), semi-structurées, comme c'est le cas des données Web, ou encore non structurées tels que des fichiers plats. Dans la majorité des cas, les sources de données exhibent des fonctionnalités et des capacités différentes, notamment en termes de langages de requêtes et de capacité d'évaluation de ces requêtes. Par exemple, certaines sources de données peuvent évaluer n'importe quel opérateur de requêtes et d'autres, notamment les sources Web, ne peuvent exécuter que certains opérateurs, voire aucun. Par ailleurs, certaines sources imposent des restrictions pour l'accès aux données. Le défi dans l'intégration de sources de données est de faire cohabiter ces sources, de plus en plus nombreuses, souvent réparties, qui existent indépendamment les unes des autres, dans un seul système uniforme, appelé système

d'intégration, sans contraindre le comportement ni l'autonomie de chacune d'elles.

Dans les premières solutions proposées [47], les systèmes d'intégration se présentent sous forme de "boîtes noires", dont les sources de données sont indissociables du système lui-même. Ces systèmes ne sont pas extensibles, dans le sens où il n'est pas possible, par exemple, d'ajouter une nouvelle source avec un nouveau modèle de données ou de nouvelles fonctionnalités [25]. Ceci a conduit à la proposition de l'architecture de médiation [185, 186]. Cette architecture décompose les systèmes d'intégration en deux composants fonctionnels : les wrappers et les médiateurs. Les wrappers sont des composants dédiés aux sources qui permettent d'abstraire et de cacher l'hétérogénéité des sources aux médiateurs. Quant aux médiateurs, ils permettent de concilier les données provenant des différents wrappers à travers un langage de requête et un modèle de données communs. L'avantage de cette architecture est qu'elle sépare systématiquement les problèmes liés à l'hétérogénéité et l'accès aux sources, à travers les wrappers, des problèmes de l'intégration qui sont pris en charge par les médiateurs. Les médiateurs sont donc des composants génériques indépendants des formats de données des sources, des langages de requêtes des sources, etc.

Cependant, même avec cette décomposition, la construction de tels systèmes de médiation[1] reste difficile et le développeur est toujours contraint de se concentrer sur plusieurs problèmes à la fois notamment, les problèmes liés aux schémas, au traitement de requêtes, ou aux langages de requêtes. Les recherches dans le domaine de l'intégration de données, ces dernières années, ont permis de développer plusieurs systèmes d'intégration tels que [169, 71, 41, 19, 168, 54, 57, 81, 97, 108, 91], dont l'architecture obéit au principe de l'architecture de médiation. Cependant, en partie à cause de la complexité de la problématique de l'intégration, ces travaux n'ont pas apporté de solution générique au problème de l'intégration ; l'objectif de chacun a été de mettre en avant une approche, un algorithme ou un modèle pour la résolution d'un problème particulier de l'intégration et souvent dans un domaine d'application spécifique.

– *Emergence des standards :*

Pendant de longues années, les bases de données relationnelles avec l'adoption du langage de requêtes SQL se sont imposées sur le marché des bases de données mais aussi dans quelques systèmes d'intégration. Cependant, avec l'expansion des systèmes d'information, le besoin de l'intégration est plus fort que jamais et de nombreuses technologies ont alors émergé sous différentes formes, notamment J2EE [161], JDO [162]. Ces technologies permettent de faire persister les objets des applicatifs du langage de programmation Java dans une ou plusieurs bases de données, et en particulier les bases de données relationnelles. Chacune de ces technologies spécifie des modèles et des langages pour l'accès et l'exploitation des données. Elles constituent, dans un sens, des systèmes d'intégration de données avec un schéma intégré objet Java [153]. Par exemple, J2EE (EJB-CMP) offre un langage de requêtes et un modèle objet particulier pour accéder aux objets Java EJB QL. JDO offre une autre variété de langage de requêtes objet pour accéder aux sources de données JDO QL. On accède donc aux mêmes données à travers plusieurs langages de requêtes. Devant la prolifération de ces standards, il est important de disposer d'une solution générique d'un système d'intégration indépendamment de ces standards, afin de faciliter leurs implantations.

La complexité des systèmes d'intégration de données, la problématique de l'intégration et la

[1] Dans ce qui suit le terme "système d'intégration" désigne aussi un "système de médiation".

prolifération des standards nous conduisent inévitablement à reconsidérer le développement des systèmes d'intégration et à fournir une approche générale d'intégration de données qui permette de couvrir les problèmes de l'intégration tout en étant adaptable aux techniques, aux modèles et aux langages existants.

1.2 Objectifs et démarche

L'objectif général de cette thèse est d'offrir un intergiciel d'intégration de données adaptable.

1.2.1 Vers un intergiciel d'intégration de données adaptable

Le terme "*adaptable*" est employé dans le sens où notre travail doit prendre en compte les différents problèmes de l'intégration de données, que nous appelons les *aspects* d'intégration, et chacun de ces aspects doit pouvoir être résolu de plusieurs manières. Dans la littérature de l'architecture logicielle, la notion d'adaptabilité est aussi désignée par d'autres termes tels que *extensibilité, réutilisabilité, composabilité* ou *modularité.* Pour plus de détails, [105] est un bon état de l'art qui couvre la notion d'adaptabilité des systèmes en général.

Le terme "*intergiciel*"[2] désigne une couche logicielle intermédiaire entre deux autres couches logicielles. Dans un système informatique réparti, un intergiciel est défini comme une couche logicielle qui relie le système d'exploitation et les applications sur chaque site d'un système, tel le bus CORBA [167] (Common Object Request Broker) ou Java RMI [164]. Dans la littérature des bases de données, le terme "*intergiciel de base de données*" ("database middleware") est employé pour désigner un système de médiation de sources de données [71, 132, 143]. Ce terme a été repris par la suite dans la classification des systèmes de bases de données à composants proposée dans [50]. En effet, un système de médiation est interposé entre les sources de données et les applications ; il est donc considéré comme un intergiciel. L'une des caractéristiques d'un intergiciel est son adaptabilité car il doit prendre en compte les fonctionnalités des couches inférieures (les sources) et des couches supérieures (les applications). Plus précisément, les aspects considérés par notre intergiciel sont les suivants :

- *Modèle de données* : Un modèle de données commun est le premier pré-requis dans un système d'intégration. Il permet de représenter d'une manière homogène les données des sources et de les échanger avec les composants du système d'intégration (les médiateurs et les wrappers). Les systèmes d'intégration existants sont généralement dotés d'un modèle de données relationnel, objet ou semi-structuré.
  Pour être adaptable à ces différents modèles, le modèle de données de notre intergiciel doit pouvoir prendre en compte aussi bien les données structurées que les données semi-structurées, voire même non structurées. Il doit aussi permettre de représenter les données intégrées ou composées, i.e, les données résultats de la fusion des données des différents sources, comme l'union ou la jointure entre deux objets. Ceci revient à dire que le modèle de données doit pouvoir raisonner sur la structure des données pour produire d'autres structures. Ceci est possible si le modèle de données dispose d'un système de types flexible ayant la capacité d'inférence de type.

- *Langages de requêtes* : La finalité du processus d'intégration est de donner une vue homogène d'un ensemble hétérogène à travers un langage de requête commun, qui est le

[2] Le terme Intergiciel est la traduction québécoise du terme anglais *middleware*.

deuxième pré-requis dans un système d'intégration. Les recherches autour des langages de requêtes dans les systèmes d'intégration ont suivi l'évolution des modèles de données et tous les langages de requêtes classiques ont été adoptés, notamment SQL, OQL et XQuery [176]. Avec la prolifération des nouvelles technologies, d'autres langages de requêtes sont apparus, comme EJB QL et JDO QL.
Contrairement aux systèmes d'intégration classiques, l'objectif de notre intergiciel n'est pas d'offrir un langage de requête unique pour l'interrogation des données, mais de donner la possibilité d'être interrogé par différents langages de requêtes. L'intergiciel doit donc être indépendant d'un langage de requêtes particulier et offrir un formalisme d'expression de requêtes indépendant d'une syntaxe. L'avantage de cette approche est de permettre l'extension des capacités d'expression de requêtes. Ceci peut être utile pour répondre aux besoins d'une application ou d'un utilisateur qui sont d'ajouter un nouvel opérateur de calcul, un nouvel agrégat, etc.

- *Optimisation des requêtes* : Probablement, l'optimisation des requêtes est l'un des problèmes les plus difficiles à résoudre dans les systèmes d'intégration. L'objectif de l'optimisation est de trouver une meilleure décomposition de la requête globale sur les différentes sources concernées. En plus des caractéristiques de l'environnement et de la répartition des systèmes d'intégration, les stratégies de recherche doivent considérer plusieurs paramètres : les capacités d'évaluation des sources, les restrictions d'accès à ces sources et d'éventuelles méta-données sur les statistiques et les formules de coûts exportées par les sources. Plusieurs travaux ont proposé des techniques pour représenter ces capacités et ces restrictions, des approches pour modéliser les coûts, même en l'absence de méta-données sur les modèles de coût des sources [11, 196, 195, 73, 149, 169]. Les stratégies de recherche utilisées varient d'un système à un autre et peuvent être des stratégies de type bottom-up [73] ou de type top-down [169], avec différents modèles de coûts. La performance de ces stratégies dépendent de l'environnement (réparti ou non) et de la disponibilité des méta-données.
  Notre intergiciel doit permettre de prendre en compte ces différents paramètres et de supporter différentes stratégies de recherche. Ceci va permettre d'adapter l'utilisation de l'optimiseur selon les besoins des applications.

- *Évolution des systèmes* : Les systèmes d'intégration de données permettent d'extraire et d'organiser les données des sources dans un ou plusieurs composants interconnectés. Cette organisation dépend du domaine d'application et des objectifs du système d'intégration de données. En effet, les composants médiateurs de l'architecture de médiation peuvent former une topologie hiérarchique ou encore une topologie pair à pair, donnant ainsi plus de liberté à l'évolution de la topologie du système. Cependant, le nombre de composants devient important et les systèmes nécessitent une administration et un contrôle.
  L'intergiciel doit permettre de considérer les différentes topologies et offrir des fonctions qui permettent de superviser et de faire évoluer la topologie des systèmes d'intégration.

1.2.2 Approche architecturale

Pour prendre en compte les différents aspects de l'adaptabilité dans notre intergiciel, nous avons suivi une approche architecturale basée sur le paradigme de la séparation des préoccupations [80, 138] (*separation of concerns*). Dans le domaine de l'architecture logicielle, la séparation

des préoccupations consiste à décomposer un système en un ensemble de composants logiciels[3] interdépendants, où chacun incarne un phénomène du système. Ce paradigme permet de gérer la complexité des systèmes pour leur conception et leur implémentation notamment les systèmes de bases de données. La figure 1.1 illustre un système de base de données composé de quatre composants : gestionnaire de caches, gestionnaire de transaction, gestionnaire d'accès et gestionnaire de requêtes.

Plus précisément, nous adoptons une approche *canevas (framework)* logiciels. Dans le petit Robert, un canevas est "*une grosse toile claire et à jour qui sert de fond aux ouvrages de tapisserie à l'aiguille...donnée première d'un ouvrage...Improviser, travailler sur un canevas...Ce n'est encore qu'un canevas*". Dans un contexte architectural, un canevas logiciel est "*a reusable conception of all or parts of a system that is represented by a set of abstract classes and the way their instances interact*" [90]. L'un des principes de base d'un canevas logiciel est qu'il sépare la spécification de l'implantation, i.e, les interfaces des classes implantant ces interfaces. Cette séparation se fait suivant le principe de l'*ouverture/fermeture*. L'ouverture/fermeture se pratique en faisant reposer le code "fixe" sur une abstraction du code amené à évoluer. En d'autres termes, il s'agit de séparer le commun du variable, en faisant reposer le commun sur une définition stable du variable en utilisant des classes abstraites ou encore des patrons de conception. Remarquons que c'est le principe suivi également par l'architecture de médiation en séparant les wrappers des médiateurs. Un canevas ne doit donc pas être dépendant d'une implantation particulière mais en contrepartie, il doit être le résultat d'une large expertise du domaine concerné afin de prendre en compte tous les aspects du système.

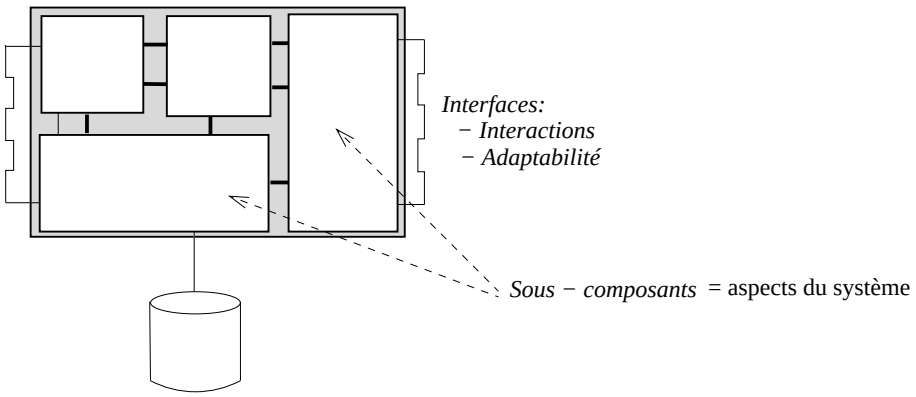


FIG. 1.1 – Un système de bases de données à composants

Un canevas logiciel peut aussi être vu comme un ensemble de composants. Dans ce manuscrit, nous réservons le terme "*composant logiciel*" pour dénoter un composant en cours d'exécution (run-time), conformement à la définition suivante : "*A software component is a unit of composition with contextually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition*" [165].

[3] Dans le domaine des bases de données, [187] a proposé une étude exploratoire et conceptuelle de l'utilisation du paradigme de composant logiciel pour la construction des systèmes de bases de données. Les composants sont appelés des mégamodules où chacun capture les fonctionnalités et le comportement des services autonomes de grandes organisations telles que les banques, les systèmes de réservation et les systèmes de transport décrits par leurs ontologies. Ces mégamodules sont reliés d'une manière asnychrone par des langage de composition de service tel que CLAM [152, 22](Composition Langage for Autonomous Megamodules).

Notre intergiciel se présente donc sous forme d'un ensemble de canevas, où chaque canevas incarne un aspect de l'adaptabilité de l'intergiciel que nous avons cité dans la section précédente.

1.3 Contribution

Nous proposons une approche d'intégration qui permet de construire des systèmes d'intégration par *la composition récursive de composants appelés domaines de données* [15, 14]. Sur la figure 1.2, cette composition est représentée par la couche intergicielle interposée entre les sources de données et les applications d'intégration. Une application d'intégration, dotée d'un langage de requêtes spécifique, utilise les domaines de données pour l'intégration des données. Une telle application peut être aussi bien un gestionnaire de persistance d'objets EJB-CMP [161], avec son langage de requête EJBQL [161], ou une implantation de JDO[162] (Java Data Object) avec son langage de requête JDOQL [162], qu'une application d'intégration manipulant des objets respectant les recommandations de l'ODMG [56] (Object Data Management Group) accessibles par le langage de requêtes OQL [56] ou bien manipulant des données semi-structurées XML avec le langage de requête XQuery [176].

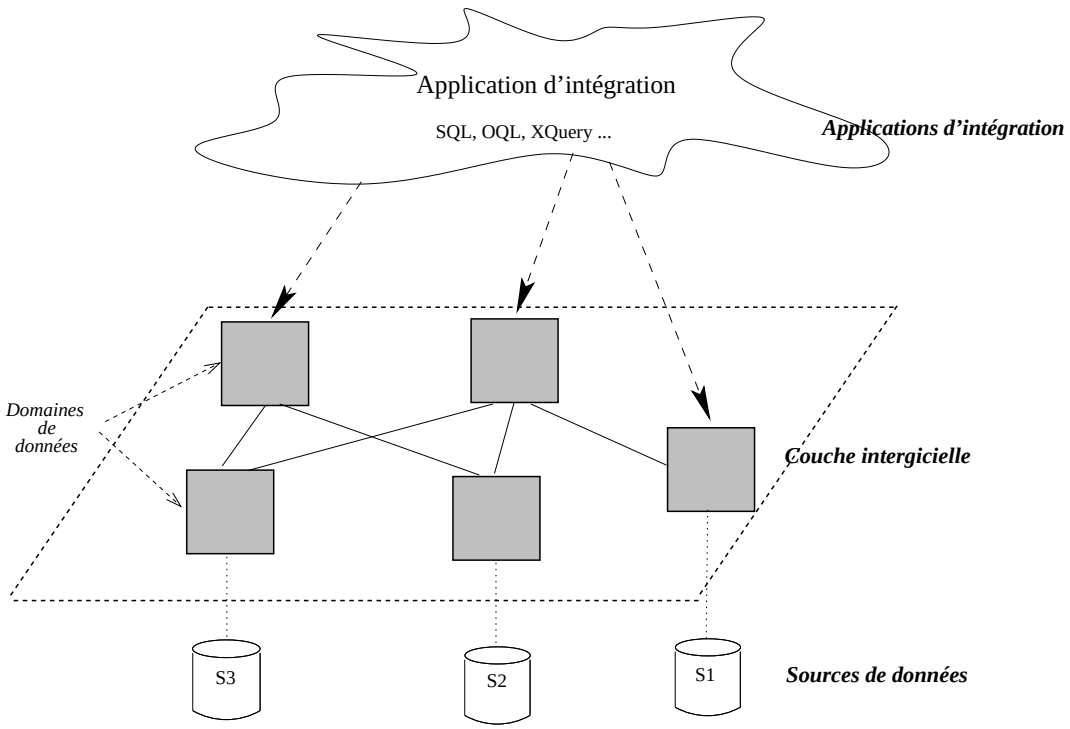


FIG. 1.2 – Approche d'intégration par la composition

Notre intergiciel offre donc un ensemble de canevas et un modèle de données qui permettent de spécifier, de construire et de composer les domaines de données en considérant les différents aspects de l'adaptabilité. Notre contribution est résumée à travers les points suivants :

– **Un canevas de domaines de données** :

L'un des principes de l'architecture de médiation est d'organiser et de partager les données intégrées sous forme de composants. Dans la construction d'une telle architecture, l'un des problèmes que l'on doit résoudre concerne la composition de ces composants. Nous proposons le composant *domaine de données* qui est à la base de la construction des systèmes d'intégration par les vues. Nous définissons un système d'intégration de données par une composition de domaines de données avec différentes topologies.
Un domaine de données peut aussi être vu comme une abstraction architecturale et une spécification minimale d'un médiateur. La spécification de ce composant nécessite de définir les sous-composants qui le constituent ainsi que les interfaces qu'il exhibe. Cette spécification a été guidée par la prise en compte les différentes fonctionnalités que doit offrir un domaine de données de manière à prendre en compte les différents aspects d'adaptabilité, notamment en terme d'optimisation. Les interfaces spécifiées permettent d'effectuer plusieurs types de tâches. Elles sont utilisées par les applications d'intégration pour interroger et faire évoluer le contenu de ces domaines, et dans la composition pour permettre l'interaction des domaines de données.

– **Un modèle de données flexible** *:*

Le modèle de données sur lequel reposent les domaines de données est un modèle hybride. Il peut être instanciable aussi bien vers un modèle structuré que semi-structuré. Il est doté d'un système de types flexible permettant de construire et d'inférer des structures de données typées à partir d'autres structures. Nous proposons aussi un mécanisme qui permet de gérer les identificateurs et les références entre les objets.

– **Un canevas d'expression de requêtes** *:*

Contrairement aux systèmes d'intégration de données classiques, les domaines de données n'imposent pas de langages de requêtes. Les requêtes sont décrites dans un formalisme canonique, qui permet de décrire les requêtes indépendamment d'une syntaxe. Ce formalisme permet à la fois l'intégration de données, en construisant de nouveaux types de données, et l'interrogation du domaine de données. Pour interroger les domaines de données, les requêtes des applications d'intégration sont projetées dans ce formalisme selon les méta-données de projection.
L'approche canevas permet d'offrir un formalisme adaptable aux différents langages de requêtes. Ainsi, il ne s'agit pas de citer tous les opérateurs possibles mais d'identifier et de spécifier l'interface des types d'opérateurs que l'on peut avoir dans une expression de requêtes. Une requête est alors décrite par une composition d'opérateurs.

– **Un canevas d'optimisation de requêtes :**

Il s'agit de prendre en compte l'aspect d'optimisation de requêtes dans un système d'intégration de données à base de domaines. Nous prenons en compte alors les paramètres et les contraintes relatives à l'optimisation des requêtes dans les systèmes d'intégration de données, en proposant un canevas d'optimisation de requêtes. L'approche canevas permet à la fois de décrire les mécanismes utilisés dans l'optimisation, telles que les règles, et de spécifier un ensemble de composants responsables de cette optimisation. Ces composants nous permettent de construire et de supporter différentes stratégies de recherches dans une composition de domaines.

– **Implantation et validation :**

Les canevas d'expression et d'optimisation de requêtes ont été implantés en Java. Dans une première validation, ces deux canevas ont permis d'implanter les deux langages de requêtes EJB QL et JDO QL dans le cadre de plusieurs projets d'ObjectWeb liés à la persistance d'objets Java. Ceci a permis de valider principalement l'adaptabilité du canevas d'expression de requêtes aux langages de requêtes de type objet [13].
Afin de montrer la portée et la faisabilité de notre approche, nous avons également utilisé notre intergiciel dans un autre domaine qui est la recherche et l'administration des composants logiciels. Nous avons proposé une architecture à base de domaines de données, pour un système d'introspection des architectures réflexives à composants [16].

1.4 Structure du mémoire

Ce manuscrit est composé de neuf chapitres, dont ce premier chapitre d'introduction. Les chapitres 2, 7, 8 et 9 représentent respectivement l'état de l'art, l'implantation et la validation, la conclusion et enfin les annexes. Notre contribution est décrite dans les chapitres 3, 4, 5 et 6. Ces quatre chapitres sont organisés selon le plan thématique qui décrit les différents aspects d'adaptabilité de notre intergiciel d'intégration, schématisé par la figure 1.3 : le canevas de domaine de données, le modèle de données, le canevas d'expression de requêtes et le canevas d'optimisation de requêtes.

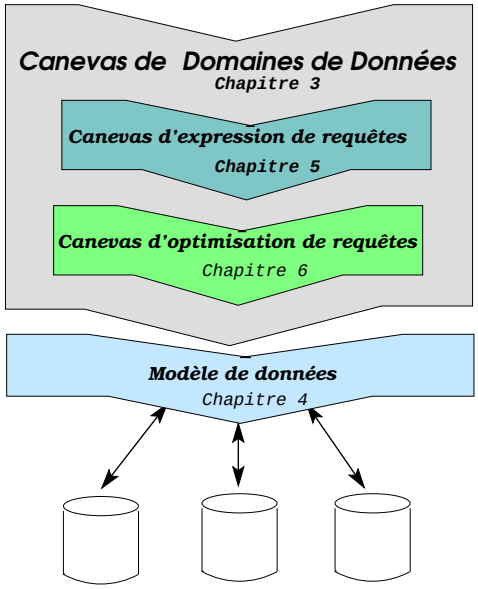


FIG. 1.3 – Aspects d'adaptabilité d'un intergiciel d'intégration.

Le chapitre 2 est un état de l'art de la problématique de l'intégration de données et des systèmes d'intégration de données. Il examine, d'un coté, les modèles et les techniques inhérents à la problématique d'intégration de données, et principalement l'intégration des schémas et le traitement de requêtes, et d'un autre coté, les architectures des systèmes d'intégration de données. On montre alors que la notion de vue couplée avec une architecture de médiation

permet d'avoir des architectures flexibles. Cependant, même si la majorité des architectures des systèmes d'intégration de données suivent cette architecture, les recherches dans le domaine de l'intégration de données sont disparates et chaque système d'intégration aborde uniquement un aspect particulier. Ce chapitre permet donc de conclure que la conception d'une architecture adaptable d'un intergiciel d'intégration est nécessaire.

Le chapitre 3 présente une nouvelle approche de construction de systèmes d'intégration par la composition de composants appelés domaines de données. Un domaine de données est une abstraction logicielle qui permet de réifier les sources de données et d'offrir des méthodes d'importation et d'exportation des données et des fonctions de traitement des requêtes. Ce composant est indépendant d'un langage de requêtes et de techniques d'optimisation particulières. On montre que cette abstraction permet de construire diverses topologies de systèmes d'intégration, allant des médiateurs classiques aux simples services d'intégration comme dans l'architecture I3 [17].

Le chapitre 4 présente le modèle de données sous-jacent aux domaines de données. C'est un modèle typé hybride : semi-structuré, objet et relationnel. Dans un sens, ceci montre le rapprochement entre les modèles de données semi-structurées et les modèles de données structurées, et le fait qu'ils peuvent être ramenés à une représentation intergicielle commune. Ce modèle peut être instancié vers le modèle relationnel, le modèle objet ou vers un modèle semi-structuré. Le typage flexible de ce modèle de données permet d'inférer les types de données intégrées à partir de leurs définitions.

Le chapitre 5 se concentre sur la description des requêtes. Contrairement aux approches proposées jusqu'à présent, et pour permettre l'adaptabilité aux différents standards de langages de requêtes, les requêtes sont exprimées indépendamment d'une structure particulière en adoptant une approche canevas. Ce canevas permet de considérer d'une manière uniforme les opérateurs utilisés pour l'intégration de données et les opérateurs qui permettent d'interroger les domaines de données. On montre alors que la séparation entre les différentes interfaces d'opérateurs du canevas permet d'avoir un canevas adaptable où de nouveaux opérateurs peuvent être facilement ajoutés.

Le chapitre 6 s'attache à l'aspect d'optimisation de requêtes dans une architecture d'intégration de données à base de domaines. L'objectif du canevas d'optimisation de requêtes proposé est de pouvoir construire des optimiseurs avec de multiples stratégies de recherche, en prenant en compte les problèmes liés à l'optimisation, notamment les capacités des sources. Comme dans les travaux évoquant la construction des optimiseurs extensibles pour les SGBD : Starbust [72, 141], Exodus [68] et son successeur Volcano [69], Genesis [20], EROC [113], TIGUKAT [134], OPT++ [93], et autres [103, 59], nous nous basons aussi sur les règles et nous spécifions un ensemble de composants qui permettent d'atteindre cet objectif. La personnalisation d'une stratégie de recherche revient à définir le type et l'ordre des interactions effectuées entre ces composants.

Le chapitre 7 présente une implantation des différents canevas de l'intergiciel et deux validations de ce travail. L'implantation concerne le canevas d'expression de requêtes et le canevas d'optimisation de requêtes. Ces deux canevas sont utilisés dans le projet de persistance d'objets Java, dans le cadre du consortium d'intergiciels libres Objectweb. Ils ont permis l'implantation des aspects d'expression et de traitement de requêtes relatifs aux standards EJB et JDO. La deuxième validation de notre travail consiste à utiliser l'intergiciel dans le contexte de la recherche des composants logiciels dans les architectures à composants. Un système à base de domaines de données pour la recherche des composants est alors proposé.

Le chapitre 8 dresse un bilan du travail effectué et montre la portée de cet intergiciel par rapport aux systèmes existants. Les perspectives pour les évolutions de notre travail et son utilisation sont discutées dans la dernière partie de ce chapitre.

Le chapitre 9 est une annexe qui regroupe quelques illustrations auxquelles le lecteur peut se référer, notamment une présentation des modèles de données et de leurs systèmes de types.

Chapitre 2

Systèmes d'intégration de données : modèles et architectures

L'important est de faire en sorte que tout soit aussi simple que possible mais pas plus simple.

Albert Einstein

2.1 Introduction

Un système d'intégration de données (SID) est une couche intergicielle entre une collection de sources de données et des utilisateurs et/ou des applications ; il permet un accès associatif et uniforme aux données des sources, comme illustré par la figure 2.1. L'application se "concentre" donc sur les données à récupérer au lieu de se "soucier" de la manière de récupérer ces données. La construction des SID est une tâche difficile et a été le centre d'intérêt des recherches en bases de données depuis une trentaine d'années. Les principales dimensions que doivent prendre en compte de tels systèmes sont [133, 74] :

- *Hétérogénéité* : L'hétérogénéité concerne aussi bien les sources de données que les caractéristiques de leurs plates-formes d'exécution, tels que les systèmes d'exploitation et les protocoles de communication. L'hétérogénéité des sources concerne les points suivants :
  - L'hétérogénéité des schémas, des modèles de données et de la sémantique associée. Cette hétérogénéité se traduit par des conflits d'intégration.
  - L'hétérogénéité des fonctionnalités offertes par chaque source en termes de capacité d'évaluation et de restriction d'accès. Certaines sources sont capables d'exécuter tout type d'opérateurs, comme ceux des bases de données, alors que d'autres sources ne peuvent effectuer qu'un simple parcours séquentiel sur leur contenu. Aussi, certaines sources ne permettent qu'un accès restreint aux données selon certains patterns. Par

exemple, on ne peut accéder aux informations d'une source de données Web "bibliothèque" qu'à travers le titre d'un livre ou le nom d'un auteur.

- *Répartition* : Les sources de données à intégrer peuvent être largement réparties. L'architecture des systèmes d'intégration de données doit prendre en compte cette répartition, notamment dans le traitement de requêtes, et garantir un certain niveau de transparence.

- *Autonomie* : Les sources de données et toutes leurs fonctionnalités existent avant le système d'intégration lui-même. L'intégration de données est effectuée en gardant et en respectant l'autonomie des sources dans le sens où celles-ci s'exécutent et évoluent indépendamment du système.

Ces trois dimensions sont renforcées dans le contexte des systèmes d'information actuels. En effet, ces systèmes sont généralement composés de services et de données largement répartis qui cohabitent souvent d'une manière autonome, notamment à travers le Web. L'hétérogénéité des données se manifeste davantage à travers la nature des sources de données, qui peuvent aller d'une base de données classique à un capteur. De plus, ces systèmes sont de plus en plus dynamiques dans le sens où de nouvelles sources peuvent être ajoutées ou supprimées à tout moment.

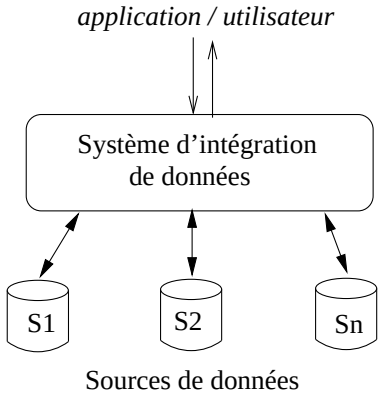


FIG. 2.1 – Système d'intégration de données

Ce chapitre présente la problématique d'intégration de données de différents points de vues, en distinguant principalement les modèles et mécanismes utilisés pour l'intégration et les architectures des systèmes d'intégration. La section 2.2 présente une taxonomie des systèmes d'intégration. Puis, la section 2.3 évoque la problématique de l'intégration des schémas et des mécanismes permettant de les résoudre. La section 2.4 montre les différents aspects et les principales étapes de traitement de requêtes dans les systèmes d'intégration. Dans la section 2.5, nous présentons quelques systèmes d'intégration existants, en mettant l'accent sur leurs architectures et leurs principaux apports. Enfin, une analyse de ces systèmes est présentée dans la dernière section.

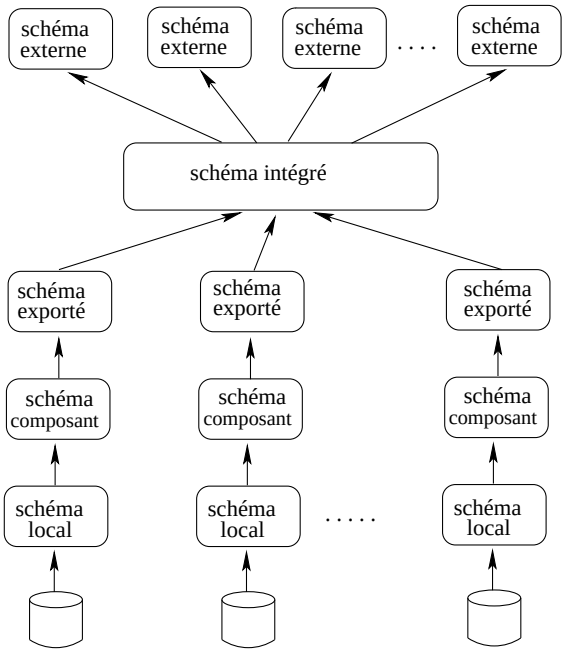


FIG. 2.2 – Les niveaux de schémas dans les systèmes d'intégration

2.2 Taxonomie des systèmes d'intégration

La figure 2.2 illustre les cinq niveaux de schémas de référence que l'on peut avoir dans un système d'intégration, tels qu'identifiés dans [156]. Le principe de l'intégration consiste alors à réaliser des "mappings" (ou des correspondances) entre les différents schémas. Les schémas appelés *schémas composants*[1] décrivent les *schémas locaux* des différentes sources dans un *modèle de données commun* du système qui peut être différent des modèles des données des sources. Une source de données peut exporter la totalité ou une partie de ces données à travers un *schéma exporté*. Les différents schémas exportés de différentes sources sont conciliés et représentés dans un schéma cohérent appelé *schéma intégré*. Les parties des données des schémas intégrés qui sont utilisées par les programmes et/ou les utilisateurs sont représentées par des *schémas externes*.

Généralement, selon l'approche d'intégration, le terme schéma intégré désigne le *schéma global* dans le cas où l'on a un seul schéma intégré, et le *schéma fédéré* dans le cas où le système est constitué de plusieurs schémas intégrés. En effet, dans la majorité des taxonomies proposées [25], on distingue trois approches d'intégration :

- *Approche à schéma global* : Dans cette première génération de systèmes (ANSI/SPARC) [47], l'intégration est forte car les schémas exportés des sources sont intégrés en un seul schéma intégré. Par rapport à la figure 2.2, les schémas exportés sont identiques aux schémas composants. Même si cette architecture offre une transparence totale face à l'hétérogénéité et à la répartition des sources, elle a été identifiée comme étant rigide, sa mise en oeuvre est difficile et elle ne prend en compte ni l'autonomie des sources ni la dynamicité des architectures.

[1] Dans [156] le nom *schéma composant* est appelé *component schema*.

- *Approche à requêtes multibases* : Dans cette approche d'intégration, on ne dispose pas de schéma intégré. Ces systèmes n'offrent pas de mécanisme de coopération et de partage d'informations. L'accès aux données par les applications et/ou les utilisateurs se fait directement sur les sources à travers un langage de requêtes appelé langage multibases. Par rapport à la figure 2.2, cela revient à dire que les sources de données (les schémas composants), ne sont pas transparents aux applications et/ou aux utilisateurs. Notons que dans cette approche, les sources de données sont généralement des bases de données classiques.

- *Approche à schémas fédérés* : Dans ces systèmes, les sources de données exportent seulement une partie de leurs schémas, i.e que les schémas exportés représentent seulement une partie des schémas composants. Les schémas intégrés ne fédèrent pas forcément tous les schémas exportés. Cette approche est donc plus flexible et plus adéquate pour les systèmes actuels, et les schémas intégrés peuvent être définis selon les besoins des applications et/ou des utilisateurs. Un exemple type d'une architecture se basant sur cette approche est l'architecture de médiation [186, 185].

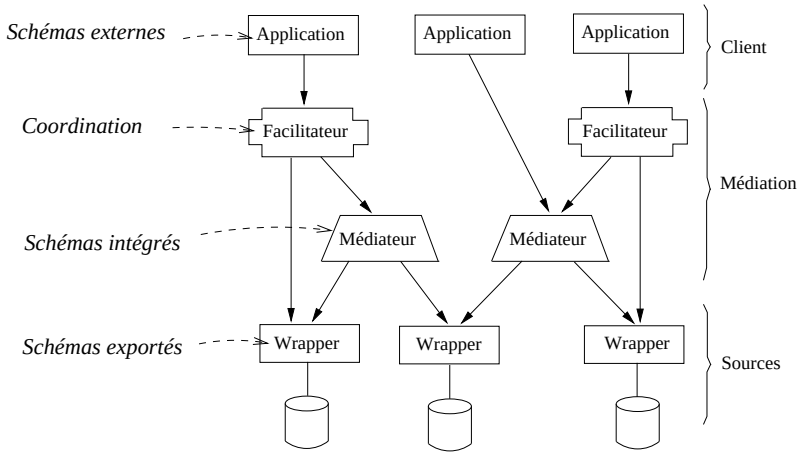


FIG. 2.3 – Architecture de médiation

L'architecture de médiation (voir figure 2.3), initialement définie dans le cadre du projet DARPA [186, 185], est une architecture de fédération de données à trois couches.

- *Couche source* : Cette couche regroupe l'ensemble des sources concernées par l'intégration. Chaque source est représentée (ou réifiée) par un composant appelé wrapper. Par rapport à la figure 2.2, un *wrapper* représente le schéma exporté d'une source. Ce composant offre un premier niveau d'intégration en permettant de réécrire les requêtes depuis le langage commun du système vers le langage natif des sources, et inversement, de transformer le résultat des requêtes évaluées par les sources dans le modèle de données commun du système.

- *Couche médiation* : Cette couche, que l'on appelle aussi couche intergicielle, est constituée d'une hiérarchie de composants, appelés médiateurs, et de facilitateurs. Les *médiateurs* sont les composants clés de l'architecture, ils représentent le schéma intégré qui concilie l'hétérogénéité des schémas exportés des wrappers. Un médiateur est généralement doté d'un modèle de données et d'un langage de requêtes communs. Les *facilitateurs* sont des composants utilisés par les applications pour coordonner les interactions relatives aux re-

quêtes avec les médiateurs ou les wrappers.

- *Couche client* : Cette couche représente les acteurs qui utilisent (ou qui consomment) les données fournies par le système, notamment : des utilisateurs, des applications, des interfaces graphiques, etc.

Cette architecture est une généralisation des architectures de systèmes de fédération de données. Elle ne résoud pas explicitement les problèmes d'intégration de données, mais offre les composants de base qui permettent de construire des systèmes d'intégration de données en prenant en compte la répartition, l'autonomie et l'hétérogénéité des sources. En effet, les différents composants de l'architecture, notamment les wrappers et les médiateurs, peuvent être largement répartis. Aussi, l'architecture ne fait aucune supposition sur les propriétés des sources qui peuvent être de n'importe quel type et qui peuvent avoir des capacités d'évaluation limitées et/ou des contraintes d'accès. L'ajout d'une nouvelle source de données se traduit par l'ajout d'un wrapper à l'architecture existante. Même si le principe de médiation de données est communément admis, le partage des tâches entre les médiateurs et les wrappers varie d'un système à un autre, comme nous allons le voir dans la section 2.5.

2.3 Intégration des schémas

"L'intégration de données est le processus de standardisation des définitions de données et des structures de données par l'utilisation d'un schéma conceptuel commun sur une collection de sources de données" [75].

Le processus d'intégration de données passe systématiquement par la résolution d'un certain nombre de conflits appelés conflits d'intégration. L'hétérogénéité se traduit par des conflits d'intégration entre les différentes sources. Dans la majorité des taxonomies de description de ces conflits, on distingue les conflits schématiques (ou syntaxique) et les conflits sémantiques [88, 173].

- *Conflits schématiques :* On distingue deux cas de conflits liés aux schémas :

  1. *Conflits de noms* : Les schémas des sources peuvent utiliser les mêmes noms pour désigner des concepts différents, on parle alors *d'homonymie*. Elles peuvent aussi utiliser deux noms différents pour désigner le même concept, on parle alors de *synonymie*.

  2. *Conflits structurels* : Lorsqu'une même information est représentée différemment dans deux sources de données, on parle de conflits structurels. Par exemple, une source utilise une seule relation "*livre*" qui contient à la fois des informations sur les livres et les auteurs alors qu'une autre source utilise deux relations "*livre*" et "*auteur*" pour représenter les mêmes informations. Le même problème peut se poser lorsqu'on a deux sources ayant deux modèles de données différents, par exemple, la première représente les livres sous forme d'objets et la deuxième sous forme de tuples. Ce type de conflit se décline souvent en conflit de typage, où la même information a des types différents selon les sources.

- *Conflits sémantiques* : Les conflits sémantiques sont liés au fait que les données des différentes sources peuvent être sujettes à des interprétations diverses. On distingue :

1. *Conflits de nommage* : Les conflits de nommage sont dûs au fait que les sources utilisent des conventions de nommage différentes. Les données sont alors décrites dans des formats différents. Des exemples classiques de ce type de conflits sont le format des dates (e.g : "01/04/2005, "01-04-2005") et les sigles (e.g : "LNCS", "Lecture Notes in Computer Science").

2. *Conflits de mesure* : Les sources de données peuvent utiliser des unités de mesures différentes pour quantifier la même information. Par exemple, une source utilise le *dollar* pour estimer les prix des livres, alors que l'autre source utilise *l'euro*. On peut citer d'autres exemples : centigrade, Celsius et Farhrenheit ou encore Mètres, Centimètres et Inches.

2.3.1 Intégration par les vues

Deux approches classiques permettent de décrire les règles mappings entre les différents schémas d'un système, et particulièrement entre les schémas exportés et les schémas intégrés : l'approche GAV (Global As View) [62] et l'approche LAV (Local As View) [107].

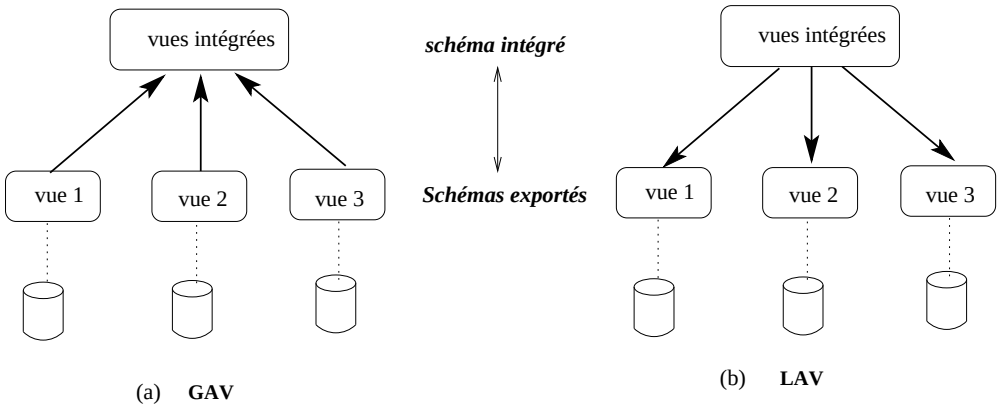


FIG. 2.4 – Approche GAV vs LAV

Ces deux approches se basent sur le mécanisme de vues (voir figure 2.4). L'utilisation des vues dans l'intégration apporte beaucoup de flexibilité [2, 6]. Elles permettent d'avoir plusieurs points de vues sur les sources de données, de transformer les données, et sont aussi utilisées pour préserver la confidentialité des données entre les utilisateurs. Les vues sont alors utilisées pour représenter les différents schémas, notamment les schémas exportés et les schémas intégrés. Les vues représentant les schémas intégrés sont aussi connues sous le nom de vues intégrées.

– *Approche Global-as-View :* Dans cette approche, chaque vue d'un schéma intégré est définie en fonction des vues exportées [62], i.e des schémas exportés. Ces définitions montrent comment récupérer les données de ces vues intégrées à partir des sources (voir partie (a) de la figure 2.4).

Exemple :

Soient trois vues, `vue1, vue2 et vue3`, représentant les schémas exportés des sources et deux vues intégrées `Bib` et `Review`. Les relations (ou les requêtes) qui définissent `Bib` et `Review` à partir de vues `vue1, vue2 et vue3` sont les suivantes :

```
Bib (title, author) :- vue1 (title, author, year, price)
Bib (title, author) :- vue2 (title, editor, author, price)
Review (title, review) :- vue1 (title, author, year, price) ∧ vue3 (title, review)
Review (title, review) :- vue2 (title, editor, author, price) ∧ vue3 (title, review)
```

Cette approche est la plus communément utilisée dans les systèmes d'intégration.

– *Approche Local-As-View :* A l'inverse de l'approche GAV, dans l'approche LAV (voir partie (b) de la figure 2.4), ce sont les vues exportées qui sont définies en fonction des vues intégrées [107]. Ces définitions montrent, pour chacune des sources, quelles sont les données du schéma intégré que l'on retrouve dans cette source. Les schémas intégrés représentent des vues universelles qui sont définies indépendamment des sources.

Exemple :

Les relations ci-dessous décrivent les schémas exportés (vue1, vue2 et vue3) en fonction des schémas intégrés `Bib`, `Review` et Style.

```
vue1 (title, author, year, price) :- Bib (title, author) ∧
                                     Style (title, style, year) ∧ year > 1973

vue2 (title, editor, author, price) :- Bib (title, author) ∧
                                  Style (title, style, year) ∧ Style = "roman"

vue3 (title, review) ⇒ Review (title, review)
```

Cette approche est utilisée dans très peu de systèmes, notamment dans Infomaster [54], Information Manifold [97] et Xperanto [154].

Les règles de mapping entre les schémas sont décrites par des opérateurs algébriques ou des requêtes. Ceci permet de prendre en compte certains conflits d'intégration, notamment les conflits de noms et les conflits de mesures, en utilisant des opérateurs de renommage et de conversion d'unités de mesures dans les requêtes de mapping. Ces opérateurs permettent donc de changer les noms des aspects, de convertir des unités de mesures, etc. Les exemples ci-dessus illustrent de simples vues qui sont définies par des requêtes conjonctives. La définition de ces opérateurs dépend du modèle de données sous-jacent (voir section suivante).

Ces deux approches ont toutes deux des points forts et des points faibles. Le principal avantage de l'approche GAV est la simplicité de la réécriture des requêtes (voir section 2.4.1). Son inconvénient est qu'elle n'est pas performante lorsqu'il s'agit de construire un système d'intégration où les schémas des sources évoluent constamment. En effet, l'ajout d'une source de données demeure une tâche difficile et demande de considérer les relations de cette source avec les sources existantes, et de modifier les règles de mapping. A l'opposé de l'approche GAV, l'approche LAV est plus appropriée dans les systèmes d'intégration de données dont les schémas sont sujets à

des évolutions. L'ajout d'une source se fait indépendamment des autres sources de données. Cependant, la réécriture de requêtes nécessite des algorithmes plus compliqués, comme nous le verrons dans la section 2.4.1. Notons qu'il existe aussi des approches hybrides, appelées GLAV, qui mélangent les deux approches LAV et GAV et qui permettent de passer d'une approche à une autre [36]. Ces aspects ne sont pas développés dans ce manuscrit.

Selon qu'on matérialise ou non les données des vues, on distingue les systèmes d'intégration qui se basent sur les vues virtuelles, et les systèmes d'intégration qui matérialisent les vues. Le deuxième type de systèmes regroupe les systèmes de fouilles de données (Data Warehouse). Un Data Warehouse est un repository qui stocke les données des vues intégrées. Ces systèmes nécessitent plus d'opérations, notamment la maintenance et le rafraîchissement des données des vues matérialisées.

2.3.2 Modèles de données

Chaque système d'intégration fédéré utilise un modèle de données commun qui permet de représenter les données des différentes sources et l'intéropérabilité entre les différents composants du système. Le modèle de données utilisé doit être assez expressif pour couvrir les modèles de données des sources de données sous-jacentes. La conversion des données des sources dans un modèle commun permet de prendre en compte les conflits structurels entre les sources. Plusieurs standards de modèles de données existants ont été utilisés comme modèles communs d'intégration, notamment des modèles relationnels, des modèles objets et des modèles semi-structurés :

- *Modèle relationnel* : Les systèmes MAIRMED [28, 157] LeSelect [81] et XPERANTO [38] utilisent le modèle relationnel pour l'intégration de données. Les vues relationnelles sont définies en utilisant le langage SQL.

- *Modèle Objet* : Le modèle objet a été adopté dans plusieurs systèmes d'intégration, notamment dans PEGASUS [95, 12], DIOM [108], DISCO [169], IRO-DB [57] et AMOS II [145]. Les vues dans le modèle objet sont construites à partir d'opérateurs spéciaux (union, sélection, intersection,etc) qui permettent la création de nouveaux types d'objets (appelés aussi classes virtuelles) à partir d'autres types [99, 101]. Le type du résultat est inféré à partir de la définition de la vue. Notons que ces opérateurs prennent en compte particulièrement la gestion des identificateurs des nouveaux types créés.

- *Modèle semi-structuré* : Dans la littérature, les données semi-structurées sont représentées sous forme de graphes ou d'arbres. Le modèle de données OEM [137] figure parmi les premiers modèles de données semi-structurées utilisés pour l'intégration de données, initialement dans le projet TSIMMIS [62]. Ce modèle de données représente les données d'une manière simple sous forme d'objets complexes avec prise en compte d'identificateurs et références entre objets (voir section annexes 9.1.3.1). Par la suite, d'autres systèmes tels que YAT [42] et STRUDEL [58] se sont basés sur ce modèle pour représenter leurs données. D'autres systèmes tel que Xyleme [8] et MIX [24] adoptent le modèle de données XML (DTD et XML Schema). La définition des vues sur les données semi-structurées a fait l'objet de plusieurs travaux de recherches [44, 174, 2]. Généralement, comme dans le modèle objet, les vues sont construites par des requêtes et le type des vues résultat est inféré à partir d'autre types [119, 42, 24]. Par exemple dans [24], où l'on utilise les DTDs pour décrire la structure des données, les DTDs relatives aux vues intégrées sont inférées.

Ces modèles différent par leurs systèmes de types. L'annexe 9.1 détaille ces modèles de données.

2.4 Traitement de requêtes

"Un système d'intégration de données permet d'interroger d'une manière uniforme et plus ou moins transparente plusieurs sources de données à travers un langage de requêtes" [190].

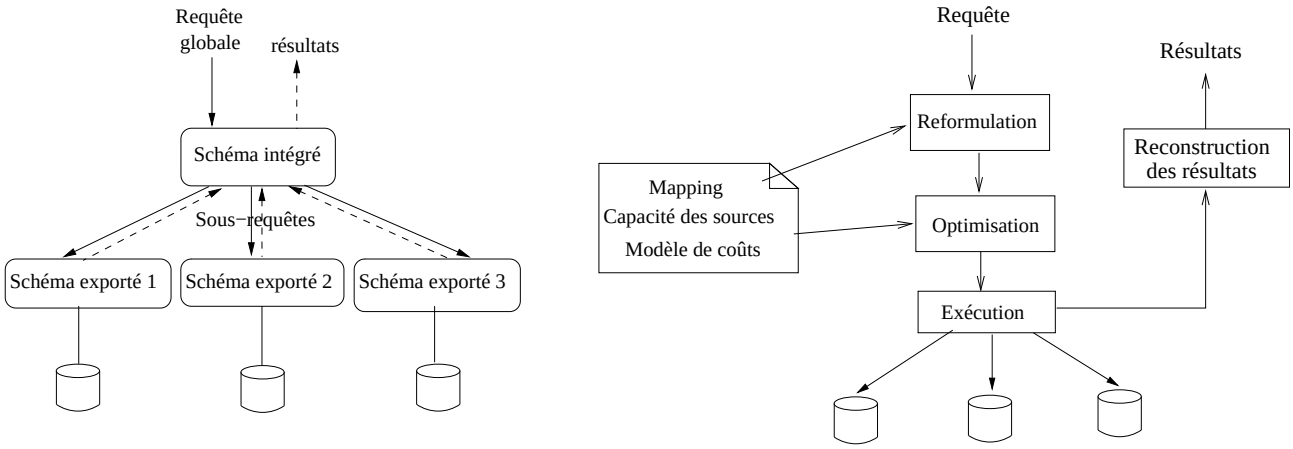


FIG. 2.5 – Traitement de requêtes dans les systèmes d'intégration

L'un des objectifs majeurs d'un système d'intégration est l'accès associatif transparent aux données intégrées représentées par le schéma intégré. Cet accès se fait communément à travers un langage de requêtes. La figure 2.5 représente les différentes phases de traitement d'une requête globale définie sur le schéma intégré. La suite de cette section décrit ces phases.

2.4.1 Reformulation

En se basant sur les méta-données de mapping entre le schéma intégré et les schémas exportés (ou encore entre les vues intégrées et les vues exportées), la phase de reformulation a pour objectif de produire des requêtes qui portent sur les schémas exportés des sources (i.e chaque feuille de l'arbre de requête porte sur un wrapper conformément à l'architecture de médiation). L'algorithme de décomposition ou de reformulation dépend de l'approche d'intégration entre les schémas exportés et intégrés adoptée par le système.

- *Cas de l'approche GAV* : Dans cette approche, l'algorithme de reformulation est très simple. Il consiste à remplacer les vues intégrées de la requête globale par la définition ou les requêtes de ces vues. On obtient alors un arbre de requête où les feuilles représentent les vues exportées des sources.

- *Cas de l'approche LAV* : La reformulation des requêtes dans ce type de mapping est connue sous le nom de *réécriture des requêtes en utilisant les vues*[2] ou encore "*contenance*

[2] Answering queries using views

des requêtes"[3]. Pour savoir si une requête est incluse dans une autre, il faut pouvoir reformuler la deuxième requête en fonction de la première. D'une manière générale, l'exercice consiste à réécrire une requête Q sur un schéma intégré en fonction d'un ensemble de vues V qui portent sur le même schéma intégré. Si plusieurs réécritures sont possibles, le travail des algorithmes de reformulation consiste à rechercher la meilleure reformulation de la requête appelée *Reformulation à contenance maximale*. Ceci revient à dire que si *Q1* est la meilleure reformulation de Q, alors il n'existe pas d'autre reformulation *Q2* telle que : $Q1 \subseteq Q2 \subseteq Q$. Plusieurs algorithmes ont été proposés : l'algorithme *bucket* [107], puis l'algorithme dans le cadre du système d'intégration Information Manifold [97], puis l'algorithme *inverse rule* [107] qui a été proposé pour le système d'intégration Infomaster [54] et enfin l'algorithme *MiniCon* [142] qui englobe les deux autres algorithmes et qui supporte le passage à l'echelle des vues. La complexité de ces algorithmes a été identifiée comme étant de l'ordre de NP-Complete. Aussi, il faut noter que ces algorithmes opèrent généralement sur de simples requêtes conjonctives. Pour en savoir plus, nous invitons le lecteur à se référer à [107].

Notons que la résolution des requêtes en utilisant les vues est aussi utilisée dans d'autres contextes : l'optimisation des requêtes, les data Warehouse et les caches sémantiques [107]. Par exemple, dans les architectures client/serveur, elle est utilisée pour résoudre la requête cliente en utilisant les caches des données locales (matérialisées), qui peuvent être le résultat d'autres requêtes.

2.4.2 Optimisation

Cette phase consiste à rechercher un meilleur partitionnement de l'arbre de requête entre les sources concernées. Comme dans les SGBD classiques, il s'agit de rechercher le meilleur plan[4] dans l'espace de recherche de la requête. Les stratégies de recherches utilisées doivent prendre en compte les coûts et les statistiques sur les données des sources, l'hétérogénéité des capacités d'évaluation des sources et les contraintes d'accès aux sources :

- *Modèles de coûts :* L'optimisation à base de coût dans les systèmes d'intégration centralisés est difficile. Ceci est dû à la difficulté d'obtenir un modèle de coût générique, qui est à son tour dûe à l'absence des statistiques et à la difficulté d'avoir des formules de coût sur les données des sources. Plusieurs techniques d'optimisation ont alors été proposées. Dans [11], la technique d'optimisation à base de coût utilisée consiste à conserver dans le cache les statistiques sur les sources à chaque accès ou requête. L'estimation des coûts des plans d'exécution est basée sur l'historique de ces statistiques. Dans [195, 52], une technique appelée optimisation par calibration a été proposée. Elle consiste à poser des requêtes types de jointure aux sources afin d'estimer et de déduire les différents paramètres du modèle de coût. Cette technique a été affinée dans [196] afin de mieux choisir et échantillonner les requêtes posées.

- *Capacités d'évaluation et contraintes d'accès aux sources :* Comme déjà évoqué dans l'introduction du chapitre, certaines sources présentent des capacités d'évaluation limitées et/ou des restrictions d'accès. L'optimiseur doit donc coopérer avec les sources en prenant en compte toutes ces informations exportées par les sources pour générer des plans va-

[3] Query containement

[4] Le moins coûteux à l'accès, le moins coûteux en consommation de ressources ou le moins coûteux financièrement.

lides. Selon la manière dont ces informations sont exportées par les sources, on distingue globalement deux approches.

1. La première approche consiste à utiliser un formalisme (ou une sorte de langage) pour décrire ces informations, comme dans les systèmes TSIMMIS, DISCO et YAT. Ces informations souvent contenues dans de simples fichiers par les wrappers sont communiquées par les médiateurs pour la recherche du plan.

2. Dans la deuxième approche, adoptée par le système GARLIC, ces informations sont exportées à travers des méthodes d'optimisation par les wrappers. Les médiateurs ignorent donc le format de description des capacités et des contraintes des sources. Ils interagissent directement avec les wrappers par appels de méthodes lors de la recherche du plan.

Dans GARLIC, chaque Wrapper exporte son propre modèle de coût [73, 149] sous forme de méthode de calcul. L'estimation des coûts des sous-requêtes est donc directement effectuée par les wrappers correspondants en appellant les méthodes de calcul. La stratégie de recherche proposée est une variété de stratégie de recherche de type *bottom-up*.
Le modèle de coût proposé dans DISCO est un modèle ouvert qui peut utiliser aussi bien les techniques d'estimation par calibration que par historique [120, 121]. Les wrappers exportent les méta-données relatives aux coûts et aux statistiques sous forme de méta-données. La stratégie de recherche utilisée dans ce système est une variété de stratégie de recherche de type *top-down* [169].
Dans les deux approches, le principe est que lors de l'exploration de l'espace de recherche, il est demandé à chaque fois à la source si elle peut ou non exécuter une partie de l'arbre de requête. Celle-ci retourne un arbre annoté où l'on distingue la partie prise en compte de celle non prise en compte. La stratégie de recherche évalue les coûts et continue son exploration. Les stratégies utilisées pour savoir quelle partie de l'arbre une source peut prendre en compte sont des stratégies de type *bottom-up*. Par exemple, les wrappers dans GARLIC utilisent STARs (Strategy Alternatives Rules) [104] pour décrire les capacités des sources et énumérer les différentes alternatives possibles. Notons que, devant des capacités limitées, l'espace de recherche des requêtes diminue.

2.4.3 Exécution et construction des résultats

L'exécution des requêtes se fait en deux étapes : (i) Tout d'abord, les sous-requêtes des différentes sources sont réécrites dans le langage cible des sources. Ce travail est généralement effectué par les wrappers. (ii) Ont lieu ensuite l'exécution et la coordination effectives des différents évaluateurs des sous-requêtes. Les parties non prises en compte par les sources sont exécutées au niveau des médiateurs.

Dans certains systèmes, le traitement de requêtes se termine par une phase de construction de résultat. Ceci consiste à transformer les collections de données résultats de l'évaluation de la requête dans le modèle de données de l'application d'intégration. Cette étape peut être représentée par un ou plusieurs opérateurs algébriques de construction et de formattage de données au-dessus de l'arbre global de la requête. Par exemple, si nous considérons une architecture d'intégration pour projeter et stocker les données XML dans des sources de données relationnelles, la restructuration consistera à créer les différents noeuds du document XML à partir des

collections de tuples [154]. Notons aussi que, d'une manière générale, le traitement des requêtes XML se termine toujours par une phase de formatage qui consiste à créer les différentes balises du résultat de la requête [42, 122, 188, 189]. Cette étape peut être réalisée en implantant des opérateurs de restructuration appropriés. Ces opérateurs doivent prendre en compte la gestion des identificateurs et des références entre les entités suivant les règles de mapping.

2.5 Architecture des systèmes existants

Il existe de nombreux systèmes d'intégration de données. Dans ce qui suit, nous présentons une sélection de systèmes d'intégration. Pour chaque système, nous présentons ses objectifs ou ses apports majeurs ainsi que ses architectures.

2.5.1 TSIMMIS et LORE

TSMMIS (The Stanford-IBM Manager of Multiple Information Sources)[62] est un projet collaboratif entre l'université de Stanford et IBM (Almaden Research Center). Ce système fait partie des pionniers dans la médiation des données structurées et semi-structurées. Les principaux apports de TSIMMIS sont :

- L'utilisation du modèle de données semi-structurées OEM (Object Exchange Model) [137] présentée dans la section 9.1.3.1, élaborée dans le cadre de ce projet. Ce modèle a par la suite été adopté dans d'autres systèmes et est considéré comme un modèle de référence des données semi-structurées.

- La génération des médiateurs en utilisant le langage MSL (Mediator Specification Langage). Ce langage se base sur les règles pour décrire les vues exportées des sources au niveau des wrappers et les vues intégrées au niveau des médiateurs.

Tel que le montre la figure 2.6, l'architecture de TSIMMIS est composée des modules suivants.

- *Les Classificateurs/Extracteurs* : Ces composants permettent d'identifier des données provenant des sources non structurées (fichiers textes, pages Web HTML) selon des patterns, et les exportent dans le système via les wrappers.

- *Les wrappers* : Le rôle des wrappers est double ; ils traduisent des requêtes écrites dans le langage OEM-QML vers le langage des sources et convertit le résultat de la requête des sources dans le modèle de données OEM.

- *Les médiateurs* : Les médiateurs représentent des vues intégrées. Ces vues sont le résultat de la combinaison des vues provenant des sources ou d'autres médiateurs, ce qui permet d'obtenir une hiérarchie de médiateurs. Ces vues sont décrites dans le langage MSL.

- *Les Générateurs* : Les wrappers et les médiateurs sont générés automatiquement à partir des définitions respectivement grâce aux deux composants Générateur de Médiateur et Générateur de Wrapper. Ces définitions sont exprimées dans les deux langages orientés règles MSL.

La suite de TSIMMIS a vu le jour sous le nom de LORE (Lighweight Object REpository) [9]. Si l'objectif de TSIMMIS a été la prise en compte des données structurées et semi-structurées à

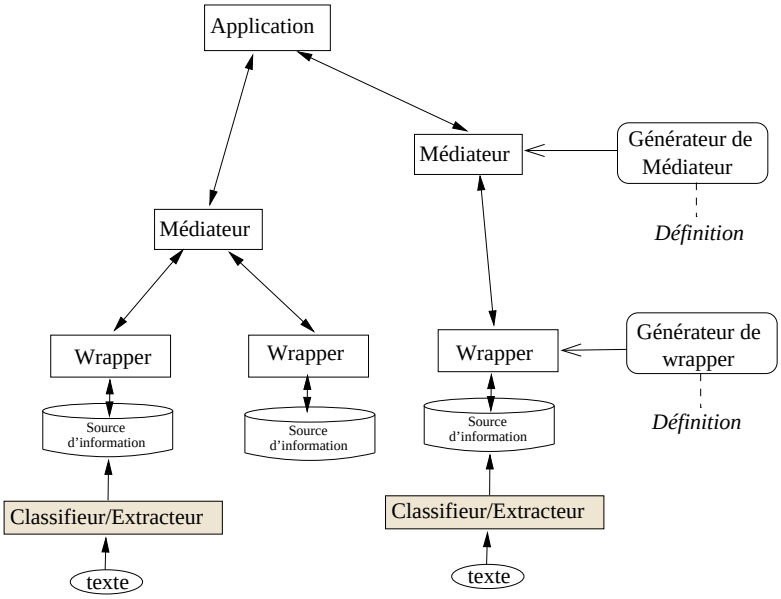


FIG. 2.6 – Architecture de TSMMIS

travers le modèle OEM et le développement des outils permettant d'automatiser la génération des médiateurs et des wrappers, l'objectif de LORE est de construire un repository de stockage et de mapping des objets OEM. Un langage de requêtes qui se base sur les mécanismes des langages de requêtes semi-structurées, notamment l'utilisation des expressions régulières, a aussi été spécifié. Ce langage, appelé LOREL, est une extension du langage objet OQL.

2.5.2 GARLIC

Le système GARLIC est développé dans le laboratoire d'IBM (Almaden Research Center) [71]. Ce système suit l'architecture de médiation et s'est spécialement focalisé sur l'intégration des données multimédia (images, cartes géographiques, textes, etc) ; il supporte les sources matrimoniales. Le modèle de données adopté est le modèle orienté-objet de l'ODMG et le langage de requête est le langage GQL (GARLIC Query Language), une extension de SQL.

L'apport de GARLIC est double : tout d'abord au niveau optimisation de requêtes et au niveau architectural. Une stratégie de recherche de type bottom-up (programmation dynamique) a été proposée. Pour une requête donnée, l'optimiseur du médiateur coopère avec les wrappers concernés pour la recherche du plan en se basant sur les coûts et en prenant en compte les capacités des sources.

L'architecture de GARLIC (voir figure 2.7) suit l'architecture de médiation où l'on distingue les composants suivants :

- Un composant représentant des méta-données qui permettent de décrire le schéma global et les règles de mapping.

- Un composant qui fait office de médiateur dont la principale tâche est le traitement des requêtes en coopération avec les wrappers.

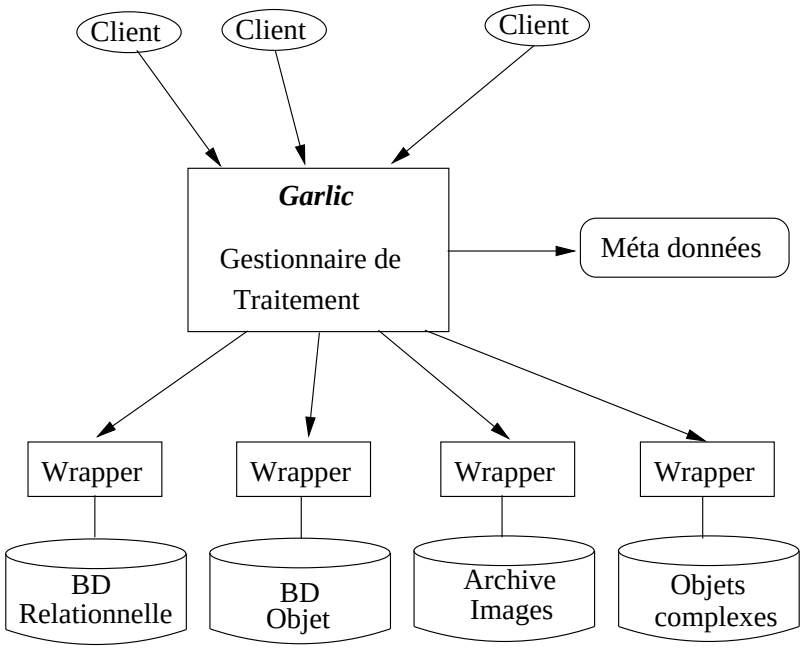


FIG. 2.7 – Architecture de GARLIC

- Les wrappers dont le rôle est plus riche et plus important que les wrappers dans TSIMMIS. En effet, chaque wrapper offre les services suivants [148, 50] : (i) Il permet la création des OIDs des objets qui représentent les données de la source en se basant sur GDL (GARLIC Description Language). Ce langage décrit quelles sont les clés utilisées pour la création des identificateurs. (ii) Il offre des méthodes d'accès aux sources ainsi que d'autres méthodes qui permettent aux applications ou à l'évaluateur de requêtes d'interagir avec les sources. (iii) Il offre les méthodes responsables de l'optimisation, notamment les méthodes `plan_access` et `plan-join`, et la méthode `translate` qui permet de créer un itérateur sur le résultat d'évaluation d'une requête.

L'apport de GARLIC a été important en ce qui concerne l'architecture des wrappers. En effet, à l'égard de plusieurs autres systèmes, le wrapper GARLIC effectue plus de tâches et coopère dans l'optimisation des requêtes avec le médiateur. Toutes les fonctionnalités des wrappers sont exportées via des méthodes. Ceci donne une flexibilité quant à l'évolution des wrappers, notamment en termes de capacités des sources, car on ne fait aucune supposition, ni sur le modèle de données des sources, ni sur ses capacités de traitement de requêtes.

2.5.3 DISCO

DISCO (Distributed Information Search Component) a été développé dans les laboratoires de l'INRIA [169]. C'est un système de médiation doté d'un modèle de données orienté-objet d'ODMG et du langage de requête OQL. Les principaux apports et les caractéristiques de DISCO sont les suivantes.

- L'intégration de données au niveau du médiateur est réalisée par des vues intégrées objets. Ces vues sont spécifiées selon l'approche GAV en utilisant le langage OQL et ne prennent pas en compte tous les conflits d'intégration.

- Plusieurs aspects de traitement de requêtes ont été abordés. Les plans logiques des requêtes OQL sont décrits dans un formalisme désigné par UAM (Unified Abstract Machine), qui se base sur les opérateurs algébriques relationnels. L'optimisation se fait en coopération avec les sources selon une stratégie de recherche de type top-down, qui se base sur les capacités d'évaluation des sources et sur des formules de coûts et des statistiques. Les capacités des sources sont exportées vers le médiateur via les wrappers dans un formalisme de description. Ce formalisme permet d'exprimer de manière simple si la source peut effectuer ou non une opération donnée. Comme déjà évoqué dans la section 6.2.2, les coûts et les statistiques sur les données des sources sont aussi exportés par les wrappers en se basant sur un modèle de coûts, proposé dans [120, 121], pour les données objets. Pour l'exécution des requêtes sur plusieurs sources, DISCO a proposé des techniques et des algorithmes qui permettent de prendre en compte l'indisponibilité des sources au moment de l'évaluation en récupérant les résultats intermédiaires [23].

2.5.4 YAT

YAT est un système de médiation développé par l'équipe Verso de l'INRIA [42]. Il vise spécialement à définir des modèles et des langages pour l'intégration des données du Web. Il se base sur un modèle de données sous forme de graphe, variété du modèle OEM pour la représentation des données [41]. Deux principaux aspects caractérisent ce système.

- Un langage de requêtes à base de règles appelé YATL [43]. Ce langage offre surtout des possibilités de restructuration, de conversion des données et de fonctions skolems pour la création des identificateurs. Une algèbre pour les données semi-structurées a aussi été proposée dans le cadre de ce langage. Cette algèbre est composée des opérateurs classiques algébriques sur les objets en plus de nouveaux opérateurs *bind* et *tree*. L'opérateur *bind* permet de lier des variables de la requête sur les noeuds du graphe de données sous forme d'une table (appelé *tab*), où chaque attribut représente une variable. Inversement, l'opérateur *tree* reconstruit le graphe d'objet à partir de la représentation tabulaire. L'opérateur *bind* représente la feuille de l'arbre de requêtes et l'opérateur *tree* est le dernier opérateur appliqué, i.e il est la racine de la requête.

- Un système de types qui permet un typage statique des données semi-structurées. Ce système considère globalement les types primitifs, des collections typées dénotées par le symbole "*" qui signifie zéro ou plusieurs occurrences, une alternative entre deux types dénotée par "|", les identificateurs et les références. Se basant sur ces types, le vérificateur de types de YAT permet de vérifier les concordances des types dans un programme (ou dans une requête) et aussi d'inférer le type des résultats des requêtes. Notons que beaucoup de travaux autour du typage statique des données semi-structurées XML [114, 35, 78] se basent les mêmes principes que ce système de types (voir section annexe 9.1.3.3).

Dans l'architecture de YAT, le médiateur est représenté par le composant *Runtime environement,* comme illustré par la figure 2.8. Ce composant utilise les wrappers pour importer les données, puis les exporter via d'autres wrappers et est composé de :

- *YAT patterns/YATL rules management module* : Ce composant permet de gérer l'ensemble des règles et des patterns du système.

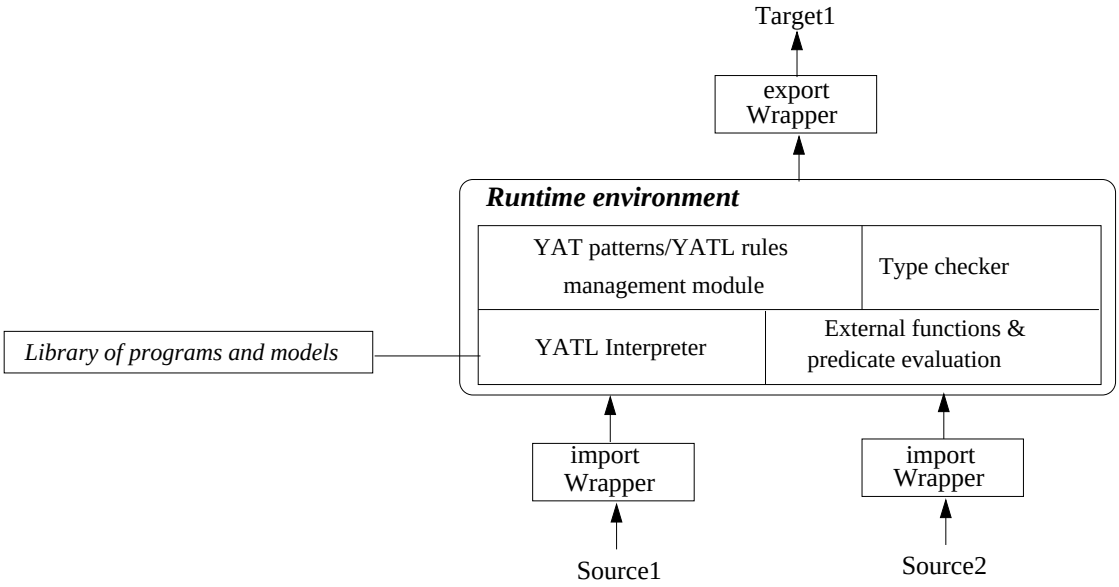


FIG. 2.8 – Architectures de YAT

- *External functions & predicate evaluation* : Ce composant offre des fonctionnalités d'évaluation, des fonctions et des prédicats des programmes.

- *Type checker* : Ce composant permet la vérification et l'inférence de types.

- *YATL interpreter* : Un interpréteur de règles est l'équivalent d'un évaluateur de requêtes. Il interagit respectivement avec les deux composants *Type checker* et *External functions & predicate evaluation* pour la vérification et l'inférence de type et pour l'exécution des opérateurs et des fonctions de calcul.

2.5.5 DIOM

DIOM (Distributed Interoperable Model) a été élaboré à l'université d'Alberta [108]. Il a pour objectif de fournir une plate-forme qui permet un accès uniforme à des sources de données réparties, hétérogènes et autonomes à large échelle.

Le modèle de données utilisé est basé sur le modèle de données ODMG-93, et le langage de requêtes utilisé, IQL (Interface Query Language), est une extension du langage SQL. L'intégration de données dans DIOM se fait selon l'approche GAV. Les vues intégrées sont construites en utilisant des *méta-opérateurs.* Ces méta-opérateurs, appliqués aux interfaces décrivant les schémas des sources ou les schémas intégrés, i.e, d'autres vues intégrées, permettent de créer de nouvelles interfaces appelées des interfaces composées. Ils correspondent aux opérateurs d'intégration sur les vues orientées-objets (voir section 2.3.2), notamment : *l'opérateur d'agrégation*, *l'opérateur de généralisation, l'opérateur de spécialisation* et *l'opérateur import/hide.*

DIOM suit l'architecture de médiation. Les médiateurs de DIOM sont instanciés à partir d'un méta-médiateur et d'une description du schéma intégré. L'architecture du méta-médiateur, *Diorama,* représentée par la figure 2.9, est constituée de deux couches : la couche médiateur et la couche wrapper. La couche médiateur est composée de :

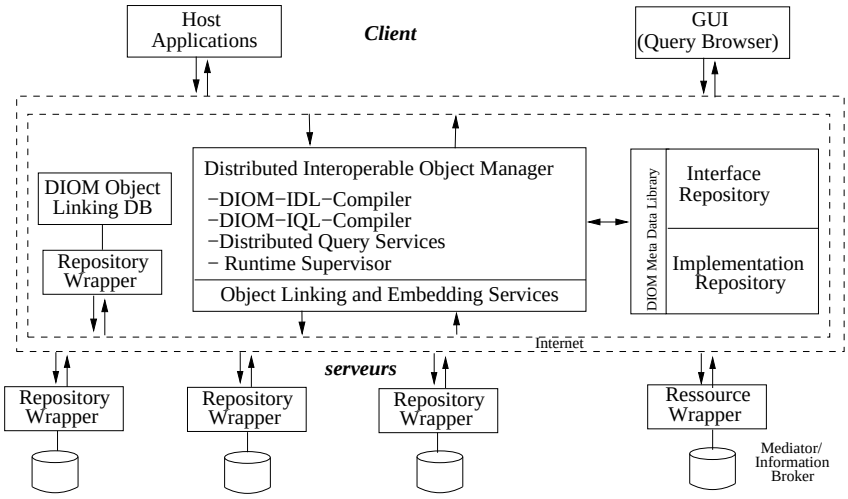


FIG. 2.9 – Architecture de Diorama

- *Interface manager* : ce composant fournit une interface graphique et une API qui expose les fonctionnalités du médiateur aux utilisateurs.

- *Distributed query mediation services* : ce composant offre des fonctionnalités liées au traitement de requêtes dans les systèmes d'intégration, notamment la décomposition de la requête, la génération du plan d'évaluation et la reconstruction des résultats. La décomposition des requêtes est effectuée sur les vues intégrées selon les méta-opérateurs utilisés. Les méta-opérateurs d'agrégation et de généralisation sont respectivement réecrites en jointures et en unions. Les sélections et les projections sont poussées vers les sources et les jointures effectuées dans le même site sont groupées. Pour l'optimisation de requêtes, un arbre de jointures est généré selon des heuristiques puis la répartition de l'arbre sur les sites se fait selon une stratégie de recherche à base de coût type bottom-up [144].

- *Runtime supervisor* : ce composant est responsable de la délégation de l'exécution des sous-requêtes aux wrappers.

- *Information source catalog manager* : ce composant est responsable de la gestion des méta-données sur les sources ainsi que du repository des interfaces. Il interagit avec la couche wrapper via le composant *Resource wrapper* pour récupérer les méta-données des sources sous-jacentes.

La couche wrapper est constituée des deux composants suivants :

- *Query wrapper service manager* : ce composant reçoit les requêtes du *Runtime supervisor*, transforme cette requête dans le langage de la source en s'appuyant sur les méta-données des sources contenues dans le *Resource wrapper*, exécute la requête et retourne les résultats.
- *Implementation repository manager* : ce composant est responsable de la maintenance des données entre les données des sources et leurs correspondants dans DIOM.

L'architecture de DIOM est modulaire et entièrement orientée-objet dans le sens où le modèle de

données commun utilisé est un modèle objet et que les wrappers et les médiateurs sont spécifiés par des interfaces. L'un des aspects pointé par DIOM est la *composabilité* des médiateurs grâce aux méta-opérateurs d'intégration qui permettent de créer des interfaces composées. Une architecture DIOM permet donc la construction incrémentale et récursive d'un système d'intégration de données. L'aspect non considéré par DIOM est la prise en compte des capacités des sources lors de l'optimisation de requêtes.

2.5.6 AMOS II

AMOS II est un système de médiation développé à l'université de Upsala en Suède [145]. AMOS II est basé sur un modèle de données qui suit le paradigme objet et qui est une extension du modèle fonctionnel Daplex [91]. L'intégration se fait selon l'approche GAV par la création de nouveaux types d'objets, en utilisant les deux types d'objets : *Derived Type* (DT) et *Integration Union Types* (IUT). Les instances DT sont dérivées à partir d'instances de leurs super-types, selon des prédicats spécifiés dans la définition des DT. Inversement, les IUT sont définies comme des super-types d'autres types, permettant ainsi de modéliser l'union des entités à intégrer. On peut résumer les principaux apports de ce système en deux points.

- *La répartition et le traitement des requêtes* : Les médiateurs d'AMOS II sont considérés comme des serveurs largement répartis. Les liaisons entre les médiateurs sont gérées à l'aide d'un serveur de noms. Pour gérer la répartition, deux types d'objets sont distingués : les objets locaux appelés *Stored Type* et les objets répartis appelés *Proxy Type*. Les *Proxy Type* permettent de créer des références et de maintenir des liaisons entre les objets répartis entre plusieurs médiateurs répartis. L'optimisation de requêtes se fait par la répartition de la requête selon un modèle de coût et en appliquant quelques heuristiques (*tree distribution*) [92]. Le plan généré est représenté par un graphe où les noeuds représentent les sous-arbres assignés aux médiateurs et les arcs désignent l'ordre du flot de données (data shipping) entre les médiateurs.

- *La composition des médiateurs* : AMOS II met en avant la composition des médiateurs par la création des types ou des vues intégrées (IUT et DT) entre les médiateurs. Ces types représentent en quelque sorte les dépendances entre les différents médiateurs. Cette composition a été généralisée dans [94] pour des topologies de médiateurs pair à pair.

Sur l'architecture d'AMOS II, représentée par la figure 2.10, on distingue les trois niveaux d'une architecture de médiation, à savoir le niveau source, le niveau client et le niveau médiateur. Le médiateur est constitué de :

- *AMOS II Kernel* : Ce composant représente le noyau du médiateur. Il offre les fonctionnalités de base d'un SGBD. Ceci permet au médiateur de supporter plusieurs interfaces d'accès et plusieurs types de wrappers.

- *Interfaces* : Le médiateur peut être accessible par plusieurs interfaces ; ODBC/JDBC ou inter-AMOS permet à un autre médiateur d'interagir avec celui-ci.

- *Wrappers* : AMOS II supporte plusieurs types de wrappers (XML, ODBC), ou encore des wrappers permettant d'accéder à d'autres médiateurs. Les rôles de ces wrappers sont les suivants.

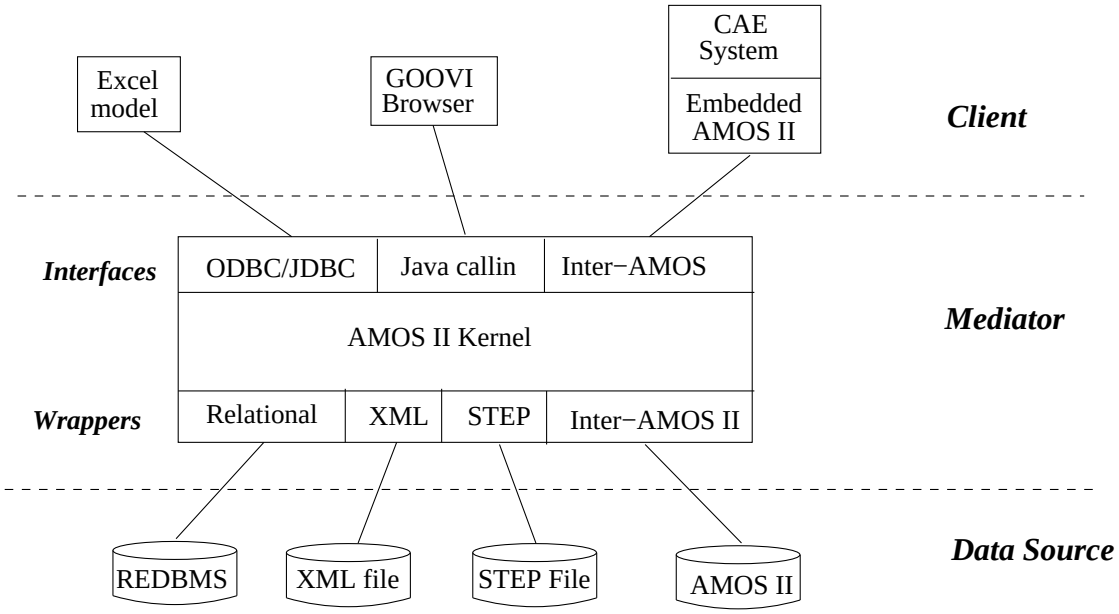


FIG. 2.10 – Architecture AMOS II

- *Importation des schémas* : Les informations implicites ou explicites sur les sources sont converties vers un ensemble de types AMOS II.

- *Translation de requêtes* : La représentation interne d'une requête est réécrite dans le langage de requêtes de la source.

- *Génération du code des OIDs* : Lorsque il est question d'assigner des OIDs aux données des sources, le wrapper utilise des méta-données incluses dans la requête pour la génération du code relatif aux OIDs.

Sur le plan architectural, l'idée d'avoir un noyau d'un médiateur est intéressante, car elle permet de dissocier les liens de composition et les langages d'interaction avec le médiateur. Cependant, dans la description de AMOS II, la véritable architecture de ce noyau n'a pas été présentée et l'on ignore si celle-ci pourrait supporter plusieurs interfaces d'accès.

2.5.7 COIL

COIL (Component Object Interconnection Language) est un langage de haut niveau de définition de médiateurs à travers un ensemble de composants (*COIL components*) proposé par l'université du Colorado [130, 112, 168]. COIL offre une bibliothèque de composants où chacun représente un aspect de l'intégration et qui permettent de définir l'architecture interne d'un médiateur par composition.

Comme le montre la figure 2.11, COIL suit l'architecture de médiation. Comme dans [168], cette architecture est définie par trois "*sections*".

- *Export* : Cette section spécifie l'interface qu'offre l'objet médiateur aux clients. Cette section est déclarative.

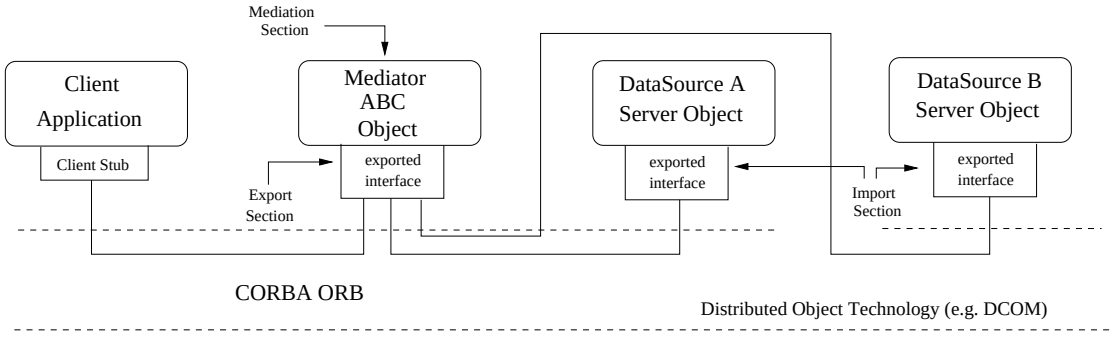


FIG. 2.11 – Architecture de COIL.

- *Import* : Cette section spécifie les interfaces des sources de données qui sont utilisées par les objets médiateurs. Cette section est également déclarative.

- *Mediation* : Cette section représente la couche médiation de l'architecture de médiation. Contrairement aux deux autres, elle est impérative ; le programmeur construit les fonctionnalités de cette section en utilisant les composants COIL. Des composants COIL sont combinés au run-time, formant un programme COIL, pour permettre l'interopérabilité selon une sémantique donnée, entre les sources et les clients.

Les composants COIL sont :

- *Dock* : ce composant est toujours placé au plus bas et au plus haut dans la chaîne de composition. Il interagit directement avec l'interface *import* pour récupérer les données selon une structure donnée depuis les sources, et interagit avec l'interface *export* du médiateur pour fournir les données. Plusieurs *docks* peuvent interagir avec une même interface.

- *Route* : Le flot de données entre les *docks* et les composants *operations* est représenté par le composant *Route*. Entre ces deux composants, d'autres composants : *filters*, *transformations* et *operations* peuvent être placés.

- *Mapping* : Ce composant permet d'effectuer la transformation structurelle de données lorsque les données sont envoyées par deux *docks*, représentant différents types de données, vers un autre *dock*. Ce composant est l'équivalent de la phase de transformation de données dans le modèle de données commun dans un système d'intégration.

- *Filters* : Ce composant représente un prédicat de filtrage de données sur un type d'item. Notons que ce type de composants peut modifier le nombre d'items.

- *Transformation* : Comme le composant *filter*, ce composant permet d'effectuer des transformations structurelles sur les items mais sans aucun filtrage.

- *Operations* : Ce composant représente les opérateurs algébriques tels que l'intersection, l'union, etc. Il prend en entrée un ou plusieurs flots de données (des composants *route*) et retourne le résultat vers un ou plusieurs *docks*.

- *Comparator* : Ce composant représente des opérateurs de comparaison entre différents types de données.

- *Object Binding* : Ce type de composant offre des méthodes uniformes permettant de spécifier d'une manière uniforme les liaisons entre les sources de données et les médiateurs dans un environnement tel que CORBA [167] ou DCOM [116].

L'architecture de COIL vise à décomposer le médiateur en un ensemble de composants qui constituent les services offerts par celui-ci. Des composants peuvent être ajoutés ou supprimés au run-time. Ces composants sont de type flot de données et les aspects d'optimisation de données n'a pas été pris en compte.

2.5.8 CoDIMS

CoDIMS (Configurable Data Integration Middleware System) est un environnement orienté-objet basé pour la construction de systèmes d'intégration de données Configurable, développé à l'université de Rio au Brésil [19, 18]. L'approche de CoDIMS est d'offrir un certain nombre de services (DIMS : Data Integration Middleware Services) permettant d'effectuer les tâches d'intégration de données que sont : la gestion des méta-données, le traitement de requêtes, la gestion des transactions, le contrôle de concurrence, la gestion de règles et la communication.

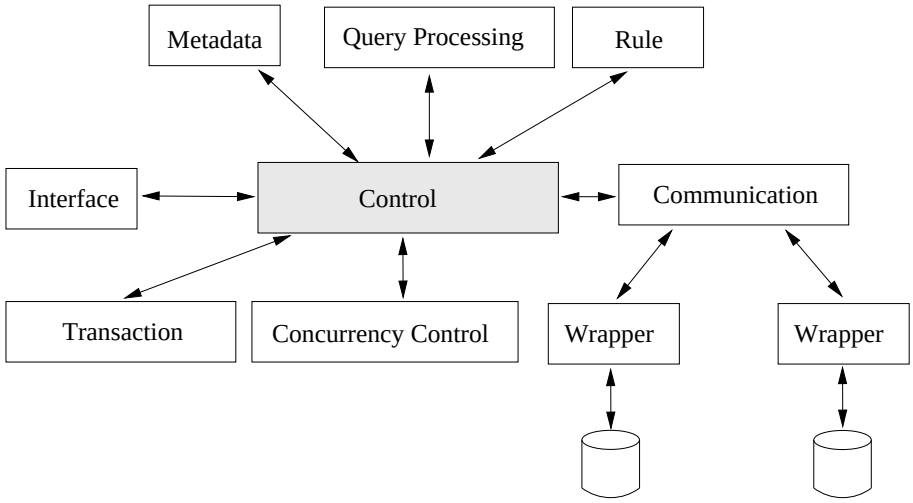


FIG. 2.12 – Architecture de CoDIMS

Sur la figure 2.12 qui représente l'architecture de CoDIMS, chacune de ces tâche est représentée par un composant. Ces composants sont :

- *Metadata Manager* : ce composant est responsable de la gestion des méta-données relatives aux sources et au schéma global. Il offre une interface qui permet d'accéder au schéma global, selon un modèle de données commun, et d'accéder aux schémas exportés des sources selon le modèle de données des sources. Conformément à [19], cette interface est spécifiée par deux méthodes `define-metadata()` pour l'ajout de méta-données et `get-object-MD()` pour récupérer les méta-données contenues par ce composant.

- *Query Processing* : ce composant encapsule toutes les phases du cycle de vie d'une requête depuis l'optimisation jusqu'à l'exécution de la requête globale. L'interface de composants est constituée des méthodes : `parser` pour le parsing de la requête, `re-writer` pour réécrire un graphe de requêtes et la méthode `optimizer` pour l'optimisation et l'évaluation de la requête. Ce composant interagit avec le composant *Metadata Manager* pour effectuer l'analyse syntaxique et l'optimisation de la requête globale. Une fois la requête globale décomposée, les sous-requêtes sont déléguées aux sources en utilisant le composant *Communication.* Enfin, ces sous-requêtes sont exécutées et le résultat est intégré dans le résultat global avant de retourner les résultats à l'utilisateur.

- *Communication* : ce composant est responsable de la gestion des communications physiques entre l'intergiciel et les sources de données. Comme dans les architectures de médiation, les wrappers sont responsables de la translation des requêtes dans le langage des sources.

- *Transaction Manager* : les propriétés transactionnelles ACID (Atomicité, Consistance, Isolation et Durabilité) sont garanties par ce composant lors de l'accès aux sources.

- *Concurrency control* : fournit les mécanismes d'implantation pour l'isolation des transactions concurrentes.

- *Rule Manager* : Ce composant permet d'ajouter des règles actives au système d'intégration.

- *Control* : toutes les propriétés de configuration de CoDIMS sont fournies par ce composant. Son rôle est d'orchestrer les différentes tâches, demandées par un client, en appelant les composants correspondants. Deux types de configuration sont distinguées : la configuration physique et la configuration logique. La configuration physique consiste à choisir les composants d'une application donnée. Par exemple, pour une application dont le besoin se limite à l'interrogation des données, les deux composants *Transaction Manager* et *Concurrency Manager* peuvent être écartés de l'architecture. La configuration logique consiste à spécifier l'ordre dans lequel les tâches sont effectuées, tant au niveau du composant *Control* qu'au niveau des autres composants. Cette spécification se base sur les règles précédentes, comme dans les systèmes workflow. Dans [19], seulement un exemple de configuration relatif au traitement de requêtes est cité. Cette configuration consiste à spécifier dans le composant *Query Processing* la séquence de méthode de traitement d'une requête, i.e, le cycle de vie d'une requête : parser, réécrire, optimiser puis exécuter la requête.

CoDIMS offre une architecture originale qui permet de couvrir tous les aspects d'intégration, y comprise la gestion de la cohérence grâce à une vision workflow du run-time du système. Cependant, cette architecture demeure centralisée et plutôt orientée SGBD. En effet, par exemple, les wrappers ne font partie de l'architecture elle-même et ne sont donc même pas spécifiés. L'intégration de schémas n'est pas mise en avant et l'on ne voit pas comment elle est effectuée. Le composant de traitement de requêtes reste très général. Il est donc impossible de spécialiser de nouvelles stratégies de recherche afin de changer (ou de configurer) le comportement de l'optimiseur.

2.5.9 Intégration de données par AXML

Cette section regroupe des architectures de systèmes d'intégration qu'on appelle architectures orientées-services. Un tel système d'intégration de données est construit par un ensemble de services d'intégration tels que les services de wrapping, les services de transformation, les services d'évaluation de requêtes, etc. Le médiateur est alors construit en appelant les différents services d'intégration. Cette architecture a été initialement proposée dans l'architecture de référence I3 ARPA [17]. Ces services sont répartis et publiés dans des environnements ou des plate-formes tels que CORBA sous forme d'objets, ou dans le Web, sous forme de services Web (Web Services). On peut résumer la construction de ce type d'architecture en trois phases.

- *Publication des services* : La description des différents services d'intégration est publiée, soit en WSDL [177] et RDF [178] dans le cas des Web services, soit l'équivalent de l'IDL CORBA.

- *Localisation des services d'intégration* : Pour localiser les services d'intégration, les services de courtage de la plateforme sous-jacente peuvent être utilisés. Pour CORBA, c'est l'équivalent des pages jaunes (trader corba), et pour le Web, c'est l'équivalent d'UDDI [170].

- *Invocation des services d'intégration* : Après la localisation, l'invocation et l'interopérabilité des services se font à travers IIOP sur le bus de courtage ORB dans les plates-formes CORBA, et à travers SOAP [179] dans le cas du Web.

Un exemple type de systèmes d'intégration utilisant ce type d'architecture est AXML (Active XML) [4, 5]. Le grand avantage de ce type d'architecture est l'interopérabilité à large échelle. En effet, ce type d'architecture évolue librement en utilisant les ressources et les protocoles des systèmes sous-jacents.

2.5.10 Analyse

Cette section offre une analyse des systèmes d'intégration de données présentés dans les sections précédentes. Remarquons d'abord que même si tous ces systèmes suivent l'architecture de médiation, les objectifs et donc les architectures de ces systèmes sont différents. Selon les objectifs, on peut distinguer les systèmes qui se sont concentrés sur le traitement des requêtes et ceux qui se sont concentrés sur l'architecture des systèmes d'intégration.

1. *Traitement des requêtes* : Le traitement de requêtes a été l'un des objectifs des deux systèmes DISCO et GARLIC. Les deux systèmes permettent de prendre en compte les méta-informations sur les sources, notamment les capacités et les restrictions d'accès aux sources. Cependant, ces deux systèmes abordent la problématique d'optimisation différemment du point de vue représentation des méta-informations sur les sources et des stratégies de recherche ainsi que des modèles de coûts utilisés.

   Dans DISCO, les méta-informations sur les sources sont représentées dans un langage de description de capacités. Chaque source exporte ses méta-informations à travers un wrapper associé. Le wrapper transmet ensuite ces informations au médiateur dès qu'il est relié à celui-ci. Ces méta-informations représentent les capacités des sources et les formules de coût et statistiques sur les données des sources. L'optimiseur d'une requête globale au niveau du médiateur se base sur ces informations pour construire un modèle de coût et

rechercher des plans en prenant en compte les capacités des sources selon une stratégie top-down. L'utilisation d'un langage de description des capacités des sources a été adoptée dans une majorité d'autres systèmes tels que : TSIMMIS, YAT, Information Manifold.

Dans GARLIC, il n'y a pas de langage de description de méta-données des sources. Les wrappers exportent toutes les fonctionnalités des sources à travers des méthodes telles que `accessPlan` et `joinAccess` pour l'optimisation. La méthode `accessPlan` permet de retourner les plans d'accès associés aux opérations de sélection et de projection. Quant à la méthode `joinAccess`, elle retourne les plans physiques associés aux opérations de jointure. Le médiateur coopère directement avec les wrappers à travers ces méthodes pour la recherche d'un plan d'évaluation selon une stratégie de recherche de type bottom-up.

Ces approches présentent toutes les deux des avantages et des inconvénients. L'avantage de l'approche DISCO est que l'optimisation est effectuée localement au niveau du médiateur, ce qui peut réduire le temps d'optimisation dans le cas où les wrappers et les médiateurs sont répartis. L'inconvénient de cette approche est que les médiateurs et les wrappers sont contraints de partager le même formalisme de description et qu'il est donc difficile de prendre en compte de nouveaux cas de description de capacités. Dans GARLIC, on n'impose pas de formalisme de description des capacités des sources. Les médiateurs interagissent directement avec les wrappers pour optimiser, calculer les coûts et exécuter les requêtes. Les médiateurs n'ont pas connaissance de la représentation des capacités des sources et toutes les méta-informations sont cachées (abstraction) par les méthodes. L'avantage de cette approche est qu'elle couvre tous les cas de capacités des sources et que ces sources peuvent évoluer librement. En contrepartie, le coût de l'optimisation peut être important dans le cas où le wrapper et le médiateur sont répartis, car les interactions se traduisent par des appels de méthodes à distance. Remarquons que la différence entre ces approches se traduit par des architectures de wrappers différentes. Dans l'approche GARLIC, les wrappers effectuent plus de tâches que dans les approches à la DISCO.

2. *Architecture* : Du point de vue architecture, nous distinguons les systèmes qui se sont intéressés à l'interopérabilité et à la composition des médiateurs, et les systèmes qui se sont intéressés à la construction des médiateurs.

   (a) *Composition des médiateurs* : La composition des médiateurs est montrée dans TSIMMIS, DIOM et AMOS II. Dans ces trois systèmes, cette composition est mise en oeuvre par le mécanisme de vues intégrées entre les médiateurs. Dans TSIMMIS, les médiateurs sont organisés d'une manière hiérarchique, où les vues sont décrites dans un langage simple à base de règles. Cette organisation est restée au niveau conceptuel et n'a pas été mise en oeuvre.

   En mettant en avant la composabilité des médiateurs, DIOM et AMOS II ont proposé des architectures évolutives où de nouveaux médiateurs peuvent être ajoutés par la création de nouvelles vues objets. Ces vues sont construites à partir d'objets provenant d'autres médiateurs. Les médiateurs peuvent alors former des topologies hiérarchiques ou pair à pair. Notons que l'utilisation de AXML pour l'intégration de données peut aussi conduire à considérer la composition de médiateurs, comme dans AMOS II et DIOM. Les vues intégrées sont alors construites en utilisant des requêtes XQuery imbriquées dans des services Webs.

La composition des médiateurs permet de construire des systèmes d'intégration interopérables où de nouveaux médiateurs peuvent être ajoutés à l'architecture existante. Cependant, les systèmes ayant abordé cette problématique ont proposé des algorithmes d'optimisation figés (DIOM et AMOSII, ou encore des algorithmes ne prenant en compte les capacités d'évaluation des sources comme c'est le cas de DIOM.

(b) *Construction des médiateurs* : La construction flexible d'un médiateur est l'objectif des deux systèmes COIL et CoDIM. Le principe suivi dans COIL et dans CoDIM est qu'un médiateur est défini par un ensemble de tâches ou de composants.

COIL a suivi le principe de la programmation par mégamodule [187] et a proposé un ensemble de composants permettant de construire des médiateurs. Ces composants peuvent être ajoutés ou supprimés dynamiquement. CoDMS a aussi défini des composants pour la construction de médiateurs configurables. Ces composants sont composés (liés ou tissés) par un composant noyau appelé *Control,* en utilisant des règles de précédence de workflow.

Le fait de considérer un médiateur comme un ensemble de composants où chacun effectue une tâche particulière permet d'avoir une architecture configurable comme dans CoDIM et COIL. Cependant, ces systèmes (CoDIM et COIL) n'ont pas pris en compte dans leurs architectures l'aspect d'optimisation des requêtes.

2.6 Conclusion

L'intégration de données a été le centre de recherches actives depuis plusieurs années. Les architectures des systèmes d'intégration ont convergé vers l'architecture de médiation, une architecture à composants ayant des propriétés architecturales intéressantes (hétérogénéité, autonomie, répartition). Plusieurs systèmes ont alors été élaborés. Comme montré dans la section précédente, ces systèmes n'apportent pas de solutions génériques à la problématique d'intégration de données, mais chacun se focalise sur un aspect de l'intégration : de nouveaux langages de requêtes et de modèles de données, des algorithmes de traitement de requêtes, de nouvelles architectures, etc. Il serait donc intéressant d'avoir un intergiciel d'intégration adaptable qui permette de supporter les différentes techniques des différents aspects de l'intégration[5]. Nous identifions les caractéristiques de cet intergiciel dans les points suivants.

- *Architecture* : Dans la construction et les architectures des systèmes d'intégration, l'utilisation des vues reste un mécanisme flexible pour la description des correspondances entre les schémas. Dans tous les modèles de données (plus particulièrement les modèles objets et semi-structurés), les vues intégrées sont construites en utilisant des opérateurs algébriques et leurs types peuvent être inférés directement à partir de leurs définitions. Les vues permettent d'organiser les données à travers les médiateurs par la composition des médiateurs comme dans DIOM et AMOS II. Aussi, l'architecture des deux systèmes CoDIM et COIL montre qu'il est aussi intéressant de décomposer un médiateur en sous-composants pour plus de flexibilité. On peut dire que ces deux architectures manifestent le besoin de *compo-*

[5]Ceci rejoint ce qui a été mentionné dans [147] : *"Meanwhile, despite the many insights generated by years of data integration research, the research results are fragmented - we have many individual techniques but few results about how to combine them. Commercial data integration tools also seem to deal with a narrow range of issues"*

sition des médiateurs ou des services de médiation. L'intergiciel d'intégration de données doit proposer un mécanisme de composition qui permette à la fois la composition de médiateurs et les sous-composants de médiateurs.

- *Modèles de données et langages* : Dans tous les systèmes d'intégration, l'accès associatif s'effectue à travers un langage de requêtes selon un modèle de données. Ces langages apportent une transparence à l'accès aux sources, qui est justement l'objectif visé par ces systèmes. En contrepartie, ces systèmes se voient comme des boîtes noires et il n'est pas possible, par exemple, d'étendre le langage par de nouveaux opérateurs (par exemple des agrégats) pour l'utilisateur, d'ajouter de nouveaux opérateurs d'intégration (par exemple des opérateurs de conversion d'unités de mesures) ou d'utiliser un autre langage de requêtes. Pour permettre cette extensibilité, l'expression de requêtes dans l'intergiciel d'intégration doit donc être indépendante d'un langage de requêtes avec un modèle de données qui permet de capturer les modèles de données standards, i.e, les modèles objets, semi-structurés et le modèle relationnel. L'indépendance vis à vis des langages de requêtes se manifeste aussi par le besoin de prendre en compte des technologies de persistance d'objets notamment J2EE (EJB-CMP) [161] et JDO [162] pour l'implantation de leurs langages de requêtes respectifs EJB QL et JDO QL (Les principes de JDO et EJB-CMP sont présentés dans le chapitre 7).

- *Optimisation de requêtes* : L'optimisation des requêtes dans les systèmes d'intégration doivent prendre en compte l'hétérogénéité des capacités de calcul, des restrictions d'accès et des contraintes d'accès et des modèles de coûts. Plusieurs techniques et stratégies de recherche tenant compte de cette hétérogénéité ont été proposées. L'optimiseur de l'intergiciel d'intégration de données doit donc être capable de supporter plusieurs stratégies de recherche, indépendamment d'un modèle de coût.

Cette thèse contribue à la spécification et à l'implantation d'éléments d'un tel intergiciel. Dans le chapitre suivant, nous présentons l'architecture générale de cet intergiciel.

Chapitre 3

Canevas de domaines de données

C'est en rentrant dans l'objet qu'on rentre dans sa propre peau ...
Respecter l'objet!
Tout doit etre contruit ... composé de parties qui forment un tout. Un arbre comme un corps humain, un corps humain comme une cathédrale.
J'ajoute, je retranche, je déplace ...

Matisse & *Picasso*

3.1 Introduction

Dans le chapitre précédent, nous avons montré le besoin de composition dans les systèmes d'intégration. Nous définissons l'architecture d'un système d'intégration de données comme une *composition récursive de composants appelés domaines de données*. Comme illustré par la figure 3.1, les données exportées depuis les sources sont organisées sous forme de domaines de données. Ces domaines forment une couche intergicielle qui est interposée entre les sources de données et les applications d'intégration.

Dans ce chapitre, à travers la section 3.2, nous commençons par définir la notion de domaines de données, qui est au coeur de notre approche. Nous décrivons alors le mécanisme de composition de domaines et montrons comment il permet l'intégration de données d'une manière flexible. Dans les sections 3.3 et 3.4, nous proposons une architecture de référence d'un domaine qui est décrite à travers les sous-composants le constituant et ses principales interfaces. En conclusion, nous discutons de cette architecture et de sa portée par rapport à l'architecture de médiation.

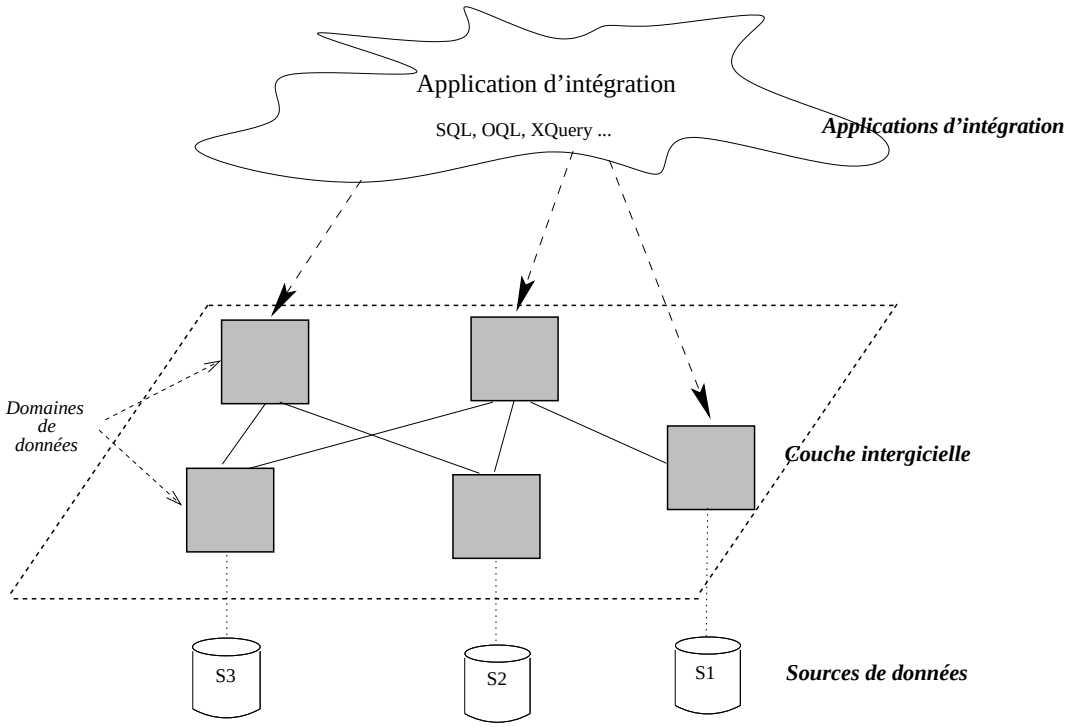


FIG. 3.1 – Approche d'intégration par la composition

3.2 Domaine de données

3.2.1 Définition

Un *domaine de données* est un *composant logiciel* défini par un contenu, le traitement et le contrôle sur ce contenu.

- *Contenu d'un domaine* : Le contenu d'un domaine de données est défini par un ensemble de vues exportées. *Les vues exportées* peuvent être *primitives* dans le cas où elles représentent les sources de données ou des vues intégrées construites à partir des vues importées d'autres domaines de données (voir section 3.2.2). *Une vue importée* dans un domaine est considérée comme une vue exportée dans son domaine d'origine. On peut dire qu'un domaine de données prend en entrée un ensemble de vues importées et retourne en sortie un ensemble de vues exportées. La figure 3.2 illustre un domaine de données qui représente des livres. Chacune des trois vues exportées de ce domaine représente un style de livre, à savoir : Histoire, Informatique et Roman.

- *Le traitement* : Un domaine de données permet le traitement de requêtes qui portent sur son contenu. Pour cela, il exporte des fonctions et des méta-données qui permettent l'évaluation et l'optimisation des requêtes.

- *Le contrôle* : Un domaine de données offre des fonctions de contrôle sur les vues importées et les vues exportées gérées par le domaine. Ces fonctions sont :

– *L'évolution* : Le contenu d'un domaine de données peut évoluer. Des vues peuvent être ajoutées, supprimées ou mises à jour dynamiquement.

– *L'introspection* : L'introspection d'un domaine consiste à découvrir l'ensemble des vues du domaine, à savoir les vues importées et les vues exportées ainsi que leurs structures (ou schémas). Cette fonction est nécessaire car le contenu d'un domaine change et évolue dynamiquement.

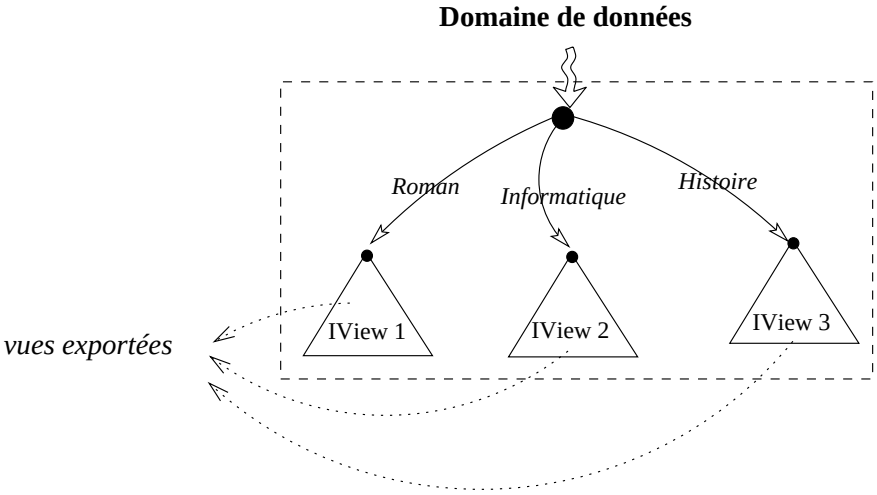


FIG. 3.2 – Contenu d'un domaine de données

Chaque vue manipulée dans un domaine est définie par un nom, une requête et un type :

– *Le nom* permet de désigner une vue dans un domaine de données d'une manière non ambigüe. Il représente aussi une information sémantique sur la nature des données représentées par la vue. Ce nom pourra être utilisé dans une expression de requête ou lors de l'introspection du contenu d'un domaine de données.

– *La requête* d'une vue est une expression d'arbre d'opérateurs. Une telle expression est décrite indépendamment d'un langage de requêtes, grâce au canevas d'expression de requêtes qui fait l'objet du chapitre 5. Dans le cas des vues exportées non primitives, les requêtes sont décrites à partir des requêtes des vues importées d'autres domaines de données. La figure 3.3 illustre la requête d'une vue exportée qui est construite à partir de trois vues importées. L'arbre de requête d'une vue exportée peut donc porter sur plusieurs domaines de données. L'approche d'intégration est de type GAV (Global-as-View), dans la mesure où les vues exportées sont des requêtes sur les vues importées.

– *Le type* correspond à la structure de données associée à la vue. Il représente le type du résultat de l'évaluation de la requête associée à la vue, qui est défini à partir de l'arbre de requêtes correspondant à la vue. Ce type fait partie du modèle de données sous-jacent aux domaines de données.

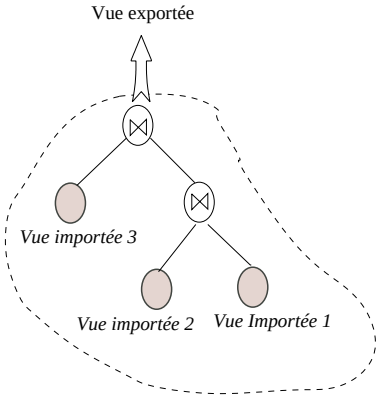


FIG. 3.3 – Vues importées vs vues exportées

Vues vs Domaine :

A l'égard des domaines de données que nous considérons comme des unités architecturales d'intégration, les vues sont considérées comme des unités conceptuelles d'intégration. Elles expriment une sorte de contrat entre les domaines pour le partage d'espaces de données. En effet, les dépendances entre les domaines sont exprimées par les vues exportées/importées entre les domaines. Les vues exportées dans un domaine réalisent alors l'intégration conciliant les vues importées d'autres domaines. Cette conciliation est exprimée par les opérateurs qui forment la requête des vues exportées (voir figure 3.3). Ces opérateurs prennent donc en compte les problèmes liés à l'intégration, notamment les conflits de nommage, de typage, comme nous le verrons dans le chapitre 5.

Domaine = concept architectural :

Il faut souligner que la notion de "*domaine*" existe aussi bien dans la terminologie des bases de données que dans les concepts architecturaux des systèmes répartis.

- Dans les bases de données relationnelles, un domaine représente l'ensemble des valeurs que peut prendre un attribut. Un domaine est aussi utilisé pour désigner un domaine d'application : finances, biologie, etc, notamment dans les systèmes d'information ou les Data Warehouses, où les données d'un même domaine partagent une même sémantique.

- Dans la terminologie des systèmes répartis, un domaine désigne un concept architectural. Dans la spécification RM-ODP [84, 85], un domaine est défini par des objets contrôleurs et des objets contenus du domaine. Chaque objet contenu est relié par un type de relation avec un objet contrôleur. Cette relation permet de distinguer plusieurs types de domaines, comme par exemple : les *domaines de nommage* que l'on retrouve dans les annuaires de type DNS ou LDAP, qui regroupent un ensemble d'entités nommées suivant une politique de désignation, ou encore les *domaines de sécurité* qui regroupent un ensemble de processus partageant une même politique de sécurité. Pour plus de détails, nous vous invitons à regarder [100], où l'abstraction de *"domaine"* est généralisée et considérée comme un modèle de programmation pour la construction des systèmes dans un contexte de Global

Computing, notamment dans les environnements mobiles.

Dans notre contexte, un *domaine de données* incarne à la fois la signification d'un domaine dans les bases de données, dans le sens où il peut représenter des données propres à un domaine d'application, et la signification d'un domaine dans les architectures des systèmes répartis, dans le sens où il contient des objets que sont les vues intégrées et offre du contrôle.

3.2.2 Domaines de données primitifs vs domaines de données non primitifs

Selon qu'un domaine est directement rattaché aux sources de données ou non, on distingue deux types de domaines de données : les domaines de données primitifs et les domaines de données non primitifs.

Un domaine de données primitif est un domaine qui encapsule les données exportées d'une source de données, comme le montre la figure 3.4. Les vues exportées d'un domaine de données primitif sont appelées *les vues primitives*. Selon le modèle de données de la source sous-jacente, on distingue des domaines de données primitifs relationnels, des domaines de données objets, des domaines de données semi-structurées, etc. Un domaine de données primitif réalise donc le premier niveau d'intégration en convertissant les données des sources dans le modèle de données des domaines. Un domaine primitif peut avoir des capacités de traitement limitées. Ces capacités sont décrites par l'ensemble des opérateurs que peut exécuter la source sous-jacente et sont exportées par le domaine en vue d'un traitement efficace des requêtes.

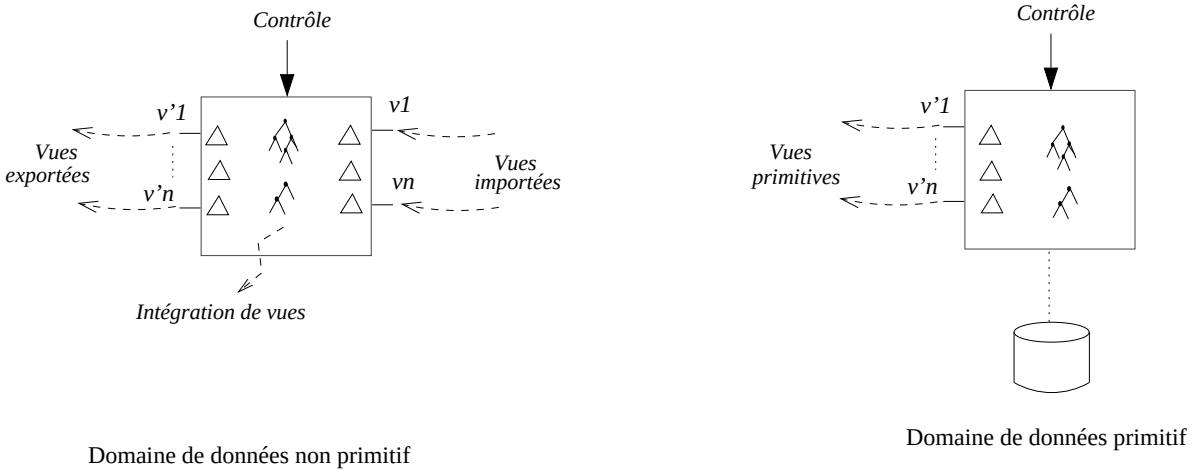


FIG. 3.4 – Domaines de données primitif et non primitif

Un domaine de données non primitif est un domaine qui contient des vues virtuelles importées d'autres domaines de données, comme le montre la figure 3.4. Il réalise un autre niveau d'intégration en réconciliant les vues importées d'autres domaines, qui proviennent potentiellement de sources différentes, dans des vues exportées intégrées. Ce domaine prend en compte les problèmes liés aux conflits de nommage, structurels et sémantiques lors de la transformation des vues importées en vues exportées, comme déjà évoqué dans la section précédente.

3.2.3 Intégration de données par la composition de domaines : *composer pour intégrer !*

3.2.3.1 Composition de domaines

La composition de domaines de données est le mécanisme d'importation et d'exportation du contenu et du traitement entre les domaines de données. L'ensemble des domaines de données qui importent et qui exportent les données forme un système d'intégration, comme l'illustre la figure 3.5. Le choix des domaines dans un système d'intégration dépend des besoins et de la sémantique du système d'intégration. Plus précisément, la composition permet le partage des ressources entre les domaines de données ; il se décline sur deux niveaux :

- Le partage des données à travers des vues importées/exportées. Ces vues permettent de spécifier les dépendances entre les différents domaines de données d'un système d'intégration. Elles peuvent former des topologies hiérarchiques, comme dans les systèmes de médiation classiques, ou encore de type graphe comme dans les topologies pair à pair (peer to peer).

- La répartition du traitement entre les domaines de données. L'évaluation et l'optimisation d'une requête sur un domaine sont effectuées en coopération avec les autres domaines concernés. Nous verrons dans le chapitre 6 que l'optimisation des requêtes dans une composition de domaines implique aussi la composition des stratégies de recherche des domaines non primitifs avec les domaines primitifs, qui ont des capacités d'évaluation limitées.

Notons que la notion de composition existe aussi dans les modèles de programmation à composants. Elle est considérée comme la principale opération de déploiement. Elle consiste aussi à assembler les différents composants qui constituent l'architecture du système. Contrairement aux domaines de données, les dépendances entre ces composants sont généralement liées au traitement au sens "service". Elles sont exprimées par les deux types d'interfaces : *les interfaces requises* et *les interfaces offertes,* qui expriment le fait qu'un composant requiert des services pour en offrir d'autres. Ainsi, si l'on considère la composition de deux composants comme un *opérateur* entre deux composants [158], alors les *opérateurs* de composition dans notre cas sont plus précisément les opérateurs algébriques des requêtes des vues exportées sur les vues importées.

De plus, du fait que la composition de domaines permette l'intégration de données, l'architecture résultat de la composition est très flexible. Les fonctions de contrôle offertes par chacun des domaines permettent aux applications la gestion et l'administration du système d'intégration comme illustré par la figure 3.5.

En effet, une architecture à base de domaines de données est dynamique. Les dépendances représentent des contrats initiaux entre les différents composants car l'architecture peut évoluer. De nouveaux domaines de données peuvent être déployés dans l'architecture existante. Des vues importées ou exportées peuvent aussi être ajoutées, supprimées ou mises à jour dans un domaine de données. Ceci est possible par le mécanisme de contrôle qu'offre l'interface d'évolution d'un domaine de données.

Les domaines de données d'un système d'intégration de données peuvent aussi être observés grâce aux interfaces d'introspection qu'offre chaque domaine de données. Ceci permet de visualiser les vues exportées par les domaines de données ainsi que les vues importées d'autres domaines pour contrôler les dépendances entre les différents domaines de données dynamiquement.

Remarquons que, selon les règles de projection des données des applications sur les vues exportées des domaines, le traitement d'une requête d'une application peut concerner plusieurs domaines. Pour éviter que cette application interagisse directement avec tous ces domaines lors du traitement, il suffit d'utiliser un domaine de données non primitif qui va représenter localement - au niveau de l'application d'intégration - toutes les vues exportées de tous les domaines utilisées par l'application. Ce domaine est l'équivalent d'un "facilitateur" dans l'architetcure de médiation (voir section 2.2).

3.2.3.2 Exemple

Dans cet exemple, nous faisons abstraction de la couche application d'intégration. Soient trois sources de données S1, S2 et S3 telles que :

- S1 représente une base de données relationnelle d'une bibliothèque d'une université avec des informations sur les livres et leurs auteurs.

- S2 et S3 représentent des sources de données semi-structurées proposées par deux fournisseurs de vente de livres sur le Web.

On veut intégrer ces trois sources de données pour que l'université puisse acheter d'autres exemplaires de ces livres en choisissant le fournisseur le moins cher. Bien entendu, plusieurs solutions sont possibles. Dans la solution donnée par la figure 3.5, à chaque source de données est associé un domaine de données primitif : D1, D2 et D3. Les composants D4 et D5 permettent de croiser les livres de la bibliothèque avec les livres des deux fournisseurs. Les vues exportées par ces domaines représentent les livres vendus par les fournisseurs de la bibliothèque. Enfin, le domaine de données D6 permet de croiser les deux vues exportées par les deux domaines D4 et D5 pour répondre à la question posée. Une solution plus simple est d'avoir quatre domaines de données : les trois domaines de données primitifs et un domaine de données non primitif qui exporte une "grosse" requête correspondant à la question. L'avantage d'utiliser la configuration de la figure 3.5 est de pouvoir réutiliser les deux domaines D4 et D5 pour poser d'autres requêtes sur leurs contenus sans passer par D6, notamment dans le cas où plusieurs applications d'intégration interagissent avec les domaines de données.

Cet exemple n'est qu'une illustration de l'utilisation des domaines de données pour l'intégration de données. L'utilisation des domaines peut aussi être plus simple. En effet, un domaine peut contenir une seule vue construite à partir d'une autre vue importée où le domaine joue de simples rôles, comme par exemple filtrer les données ou renommer les éléments de la vue importée. Dans ce cas, les domaines de données peuvent être utilisés pour construire des services d'intégration comme dans les architectures orientées-services (voir section 2.5.9) ou pour construire des composants COIL [168] (voir section 2.5.7). La figure 3.6 montre une composition de trois domaines de données sous forme de simples opérateurs en pipeline. Le domaine de données D0 permet d'accéder aux données et les deux autres domaines effectuent de simples opérations de filtrage et de renommage.

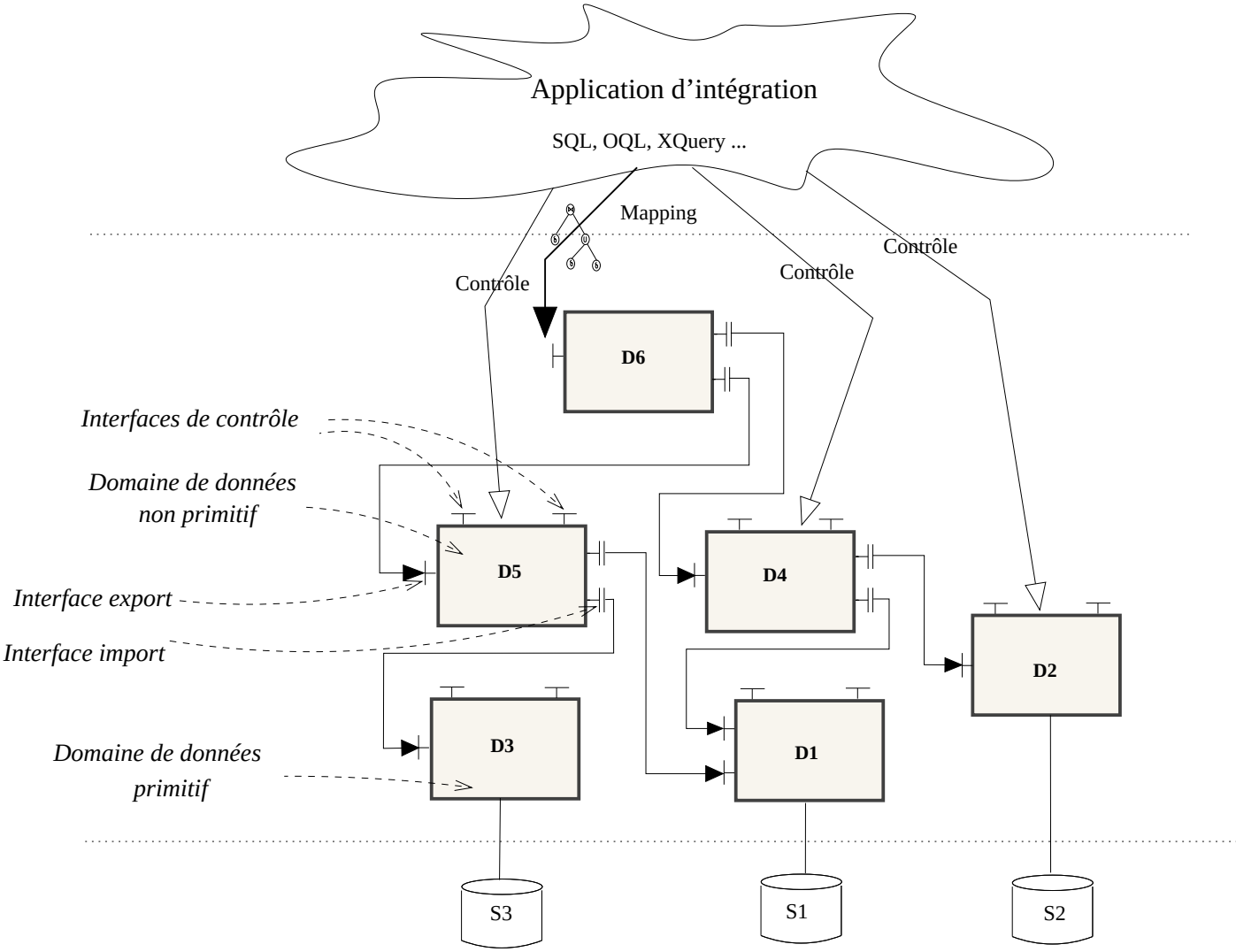


FIG. 3.5 – Composition des domaines pour l'intégration de données

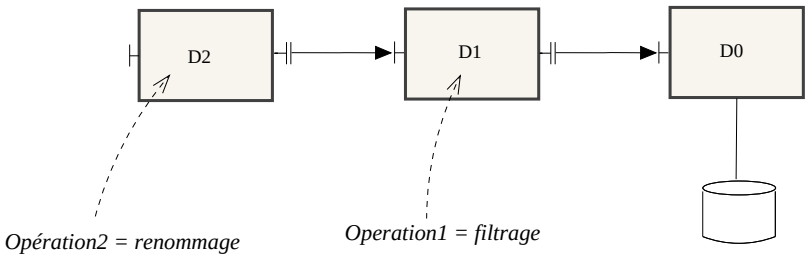


FIG. 3.6 – Des domaines de données comme de simples opérateurs

Dans les sections suivantes, nous décrivons l'architecture d'un domaine de données. Cette architecture est définie par ses sous-composants ainsi que les spécifications des interfaces qu'offre chaque domaine.

3.3 Composants d'un domaine de données

Un domaine de données offre des fonctions qui permettent le traitement des requêtes sur son contenu. Ces fonctions sont offertes par des sous-composants, comme le montre la figure 3.7 qui représente l'architecture d'un domaine. Parmi ces composants, on distingue :

- Le gestionnaire de vues : `View Manager`.
- Les composants responsables du traitement des requêtes : `ComOptimizer`, `LocOptimizer`, `Rule Manager`, `Compiler` et `Evaluation Manager`.
- Le gestionnaire du domaine : `Domain Manager`.

Dans cette architecture, les composants interagissent à travers des interfaces. Les flèches représentent les liaisons (bindings) entre les interfaces des différents composants. Le sens de dépendance entre les composants (appelant - appelé ou client - serveur) suit l'orientation de ces flèches.

3.3.1 Gestionnaire de vues

Le gestionnaire de vues contient toutes les informations sur les définitions des vues exportées. Il maintient aussi des méta-données sur les vues importées, notamment les interfaces qui permettent d'interagir avec leurs domaines d'origine. Le gestionnaire de vues offre aussi des fonctions qui permettent d'introspecter et de mettre à jour ces informations. On peut accéder aux informations sur une vue à partir de son nom ou directement à partir de son identificateur. On peut aussi accéder à toutes les vues gérées par ce gestionnaire. Ces fonctions sont utiles pour les fonctions d'introspection et d'évolution du domaine. Le gestionnaire de vues offre aussi une autre fonction qui consiste à étendre les vues exportées du domaine. Cette phase de réécriture des vues est développée dans la section 6.2.

3.3.2 Composants de traitement de requêtes

Les composants de traitement sont responsables des phases d'optimisation, de compilation et d'évaluation des requêtes.

- L'optimisation est effectuée par les trois composants : `ComOptimizer`, `LocOptimizer`, `Rule Manager`. Le composant `ComOptimizer` interagit avec les deux composants `LocOptimizer` et `Rule Manager` pour contrôler le processus de recherche d'un plan d'évaluation. En présence d'une composition de plusieurs domaines, le processus de recherche nécessite aussi la coopération avec les domaines de données primitifs ayant des capacités limitées. Dans le chapitre 6, nous présentons le canevas d'optimisation de requêtes et montrons comment construire des stratégies de recherche simples et des stratégies de recherche composites à l'aide de ce canevas.

- La compilation des plans d'évaluation, en vue d'optimiser son code, est effectuée par le composant `Compiler`. Comme dans [109], ceci permet d'éviter les vérifications dynamiques

des types lors de l'exécution des requêtes. Les principes de fonctionnement de ce composant sont présentés dans la section 6.2.3.

- Le composant `Evaluation Manager` est responsable de l'exécution des requêtes. Dans le cas d'un domaine de données non primitif, l'exécution d'une requête est répartie entre plusieurs domaines. Nous en parlons davantage dans la section 6.2.4.

Les interfaces qu'offrent tous ces composants sont présentées à travers le chapitre 6.

3.3.2.1 Gestionnaire du domaine

Un domaine de données peut être vu comme un composant qui permet d'effectuer plusieurs types de tâches à travers ses interfaces d'interactions que nous présentons dans la section suivante, à savoir : l'optimisation, l'évaluation et la consultation. Le composant `Domain Manager` permet d'intercepter les demandes d'exécution de tâches pour les diriger vers les composants appropriés du domaine. Ce composant permet aussi de coordonner l'évaluation de plusieurs requêtes en parallèle dans le cas où plusieurs évaluations de requêtes sont demandées (intra-query processing). Dans ce manuscrit, nous laissons de coté les politiques de coordination de l'évaluation.

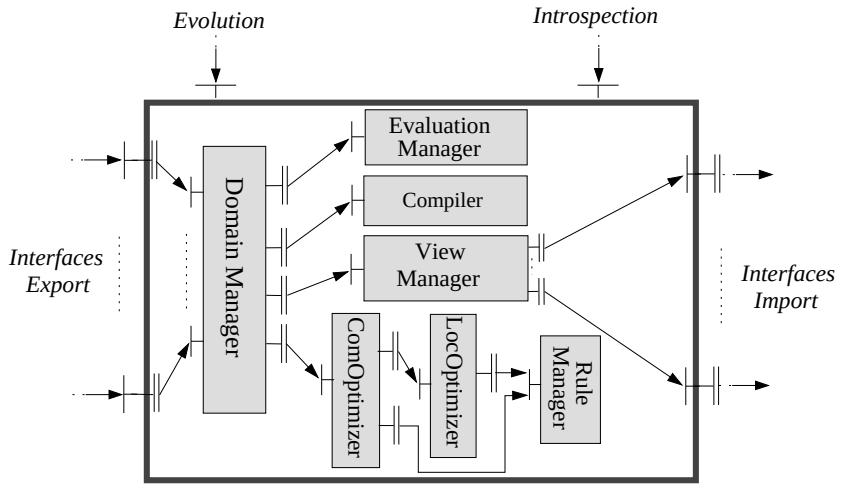


FIG. 3.7 – Architecture d'un domaine de données

3.4 Interfaces d'exportation/importation et de contrôle

Les interactions avec les domaines de données se font à travers des interfaces. Comme illustré par la figure 3.7, il existe deux sortes d'interfaces : les interfaces d'importation ou d'exportation spécifiées par l'interface `ImportExport` et les interfaces de contrôle où l'on retrouve l'interface `Introspection` et l'interface `Evolution`. Ces interfaces sont spécifiées indépendamment de l'implantation des sous-composants d'un domaine.

Tout d'abord, les vues d'un domaine de données sont spécifiées par l'interface `IView`,[1] représentée par la figure 3.8. La méthode `getQueryTree` permet de récupérer la requête désignée

[1] La lettre `I` du nom de l'interface `IView` signifie "Integrated".

```
public interface IView{

    // Récupérer le nom de la vue
    String getName() ;

    // Récupérer la requête correspondant à la vue
    QueryTree getQueryTree() ;

    // Récupérer le type de la vue
    IType getIType() ;
  ...
```

FIG. 3.8 – L'interface `IView`

par l'interface `QueryTree`, associée à la vue (voir chapitre 5). Le nom de la vue ainsi que son type sont respectivement représentés par les deux méthodes `getName` et `getIType`. Notons que cette spécification n'offre pas de fonctions de mise à jour des vues. Ces fonctions font partie de l'interface de contrôle `Evolution` du domaine.

3.4.1 Interface ImportExport

L'interface `ImportExport` permet aux domaines de données d'exporter ou d'importer des vues ainsi que les fonctionnalités de traitement des requêtes. Chacune de ces interfaces représente une ou plusieurs vues importées ou exportées dans un domaine. Dans un domaine de données, lorsque cette interface représente des vues exportées, elle est appelée interface *export*, et lorsqu'elle représente une vue importée elle est appelée interface *import*. La composition de domaines se traduit par l'établissement des liaisons entre les domaines : les interfaces *import* d'un domaine (le domaine qui importe les vues) sont reliées aux interfaces *export* d'un autre domaine (le domaine qui exporte les vues), comme l'illustre l'exemple de la figure 3.5. On peut dire que les interfaces *import* jouent le rôle d'interfaces clientes appellant les interfaces *export* qui jouent le rôle d'interfaces serveurs. Les domaines composés utilisent donc ces interfaces, qui sont toutes les deux de type `ImportExport`, pour interagir notamment lors du traitement de requêtes réparti entre plusieurs domaines. Les interfaces *export* permettent aussi aux applications d'interroger les domaines de données.

La spécification de l'interface `ImportExport` est représentée par la figure 3.9 qui permet :

- *L'exportation du contenu* : l'exportation du contenu se fait à travers les vues. La méthode `getViews` permet de retourner l'ensemble des vues importées ou exportées par cette interface.

- *L'évaluation des requêtes* : la méthode `evaluateQuery` permet d'évaluer des requêtes qui portent sur le contenu du domaine de données. Elle prend en entrée un arbre de requête et des paramètres et retourne le résultat de la requête qui est une collection de type `IType`. Les paramètres sont utiles pour les requêtes paramétrées. Inversement à la méthode `evaluateQuery`, la méthode `closeEvaluation` permet d'arrêter l'évaluation de la requête courante sur cette interface. Ces requêtes sont exprimées en utilisant le canevas

```
Interface ImportExport{

    // Récupérer l'ensemble des vues importées/exportées
    IView[] getViews();

    // Évaluer une requête
    IType evaluate(QueryTree query, Hashmap parameters)
                                    throws EvaluationException;

    // Stopper l'évaluation
     void closeEvaluation ();

     // Récupérer l'espace d'optimisation logique
     QueryTree[] lOptimize(QueryTree query)
                                    throws OptimizationException;

     // Récupérer l'espace d'optimisation physique
     EvaluableQueryTree[] pOptimize(QueryTree query)
                                    throws OptimizationException;

     // Exporter les méta données
     Object getMetaInfo(String topic)
                                    throws IException;
```

FIG. 3.9 – L'interface `ImportExport`

d'expression de requêtes présenté dans le chapitre 5.

- *L'optimisation des requêtes* : Un domaine de données exporte deux types d'optimisation : l'espace de recherche logique et l'espace de recherche physique d'une requête. L'espace logique est exprimé par la méthode `lOptimize` qui prend en entrée un arbre de requête et retourne un ensemble de plans logiques associés à cette requête ; l'espace physique est exprimé par la méthode `pOptimize` qui renvoie l'ensemble des plans physiques d'une requête donnée. Ces fonctionnalités sont utiles dans le cas des domaines de données primitifs ayant des capacités de traitement limitées. Nous détaillerons ces aspects dans le chapitre 6.

- *Exportation des méta-données* : Les domaines de données peuvent aussi exporter des méta-données à travers la méthode `getMetaInfo`, notamment les statistiques et des formules de coût qui sont utiles pour l'optimisation des requêtes.

Les requêtes à optimiser ou à évaluer à travers une interface *export* portent sur les vues exportées du domaine. Les feuilles de l'arbre de requêtes encapsulent alors les vues exportées du domaine. Notons que ceci permet de considérer aussi le cas où les requêtes portent sur les vues importées. Pour cela, il suffit d'exporter les vues importées sans aucune tranformation à travers des vues exportées. Selon le degré d'autonomie, certains domaines permettent d'accéder à leur contenu ainsi qu'aux méthodes d'optimisation, auquel cas ils sont considérés comme des "*boîtes blanches*", et d'autres non, auquel cas ils sont considérés comme des "*boîtes noires*". Dans le deuxième cas, le domaine de données n'exporte pas les méthodes d'optimisation et ne permet pas d'accéder à l'arbre de requêtes des vues qu'il exporte. Nous verrons dans le chapitre 6 que ces deux cas de figure sont pris en compte différemment lors du traitement de requête.

3.4.2 Interfaces de contrôle

Comme déjà évoqué dans la première section, les interfaces de contrôle sont importantes pour l'exploitation et l'administration des architectures à base de domaines de données. Dans l'architecture du domaine de données qui est représentée par la figure 3.7, nous n'avons pas explicité les liaisons entre les interfaces de contrôle d'un domaine et ses sous-composants. On considère que les interfaces de contrôle peuvent interagir avec tous les sous-composants du domaine. Nous parlons de ces interactions dans la spécification de chacune de ces interfaces.

- **1 - Interface d'introspection**

L'interface `Introspection` représentée par la figure 3.10 permet de consulter les vues importées et exportées d'un domaine de données. Les fonctionnalités de cette interface sont fournies par le composant `View Manager`. Cette consultation peut se faire de deux manières différentes.

- La première consiste à récupérer la liste des vues importées et exportées à l'aide respectivement des deux méthodes `getImportedViews` et `getExportedViews`.

- La deuxième manière consiste à découvrir le contenu d'un domaine en utilisant les expressions de chemins régulières. Un domaine de données est alors considéré comme un arbre où les arcs représentent les noms des vues exportées, comme sur la figure 3.2. Ces expressions portent sur les noms des vues exportées ainsi que sur la structure (ou le type `IType`) de ces vues. Dans la spécification de l'interface `Introspection`, la méthode `resolve` permet

de récupérer tous les chemins possibles qui s'apparient structurellement avec l'expression de chemins régulière représentée par l'interface `RegExp`. Ces chemins sont représentés par leur type, `IType`, qui est le type des valeurs pointées par l'expression régulière. Dans une expression de chemin régulière, on utilise les caractères classiques des expressions régulières tels que " ? ", " * " ," | " pour naviguer à travers le contenu d'un domaine. Nous reviendrons dans les détails dans le chapitre 5 sur ces expressions, où nous montrerons comment les exprimer et les utiliser.

La découverte et la consultation du contenu d'un domaine sont comparables à la notion de Data Guide [65] introduite dans les systèmes de médiation des données semi-structurées. Un Data Guide remplace la notion de schéma dans les systèmes d'intégration de données structurées. Il permet de représenter tous les chemins possibles que l'on peut avoir dans un système. Comme les DataGuides, la découverte de la structure permet aux utilisateurs ne connaissant pas le contenu d'un domaine de pouvoir poser des requêtes plus ciblées, en consultant au préalable la structure du domaine. Aussi, l'introspection est un mécanisme utile pour des besoins d'administration permettant d'observer l'évolution du contenu d'un domaine ainsi que les dépendances entre les domaines.

```
interface Introspection {

    // Récupérer toutes les vues importées
    IView[] getImportedViews() ;

    // Récupérer toutes les vues exportées
    IView[] getExportedViews() ;

    // Résoudre une expression de chemin régulier
    IType resolve(RegExp path)
                        throws PathExpressionException ;
```

FIG. 3.10 – L'interface `Introspection`

– 2 - Interface d'évolution

L'interface `Evolution`, représentée par la figure 3.11, permet de contrôler l'évolution du contenu d'un domaine de données. On distingue deux types d'évolution.

- Le premier type d'évolution porte sur la mise à jour de la définition des vues exportées. Cette mise à jour peut être le changement du nom de la vue ou le changement de la requête correspondant à une vue exportée. Dans la spécification de l'interface d'évolution, ces deux opérations sont respectivement représentées par les deux méthodes `renameExportedView` et `updateExportedViewQuery`.

- Le deuxième type d'évolution permet d'ajouter, de supprimer des vues exportées et des vues importées. L'ajout d'une vue exportée se fait par la méthode `addExportedView`,

qui permet de créer une nouvelle vue exportée à partir d'une définition comportant le nom et l'arbre de requêtes de la vue. De la même façon, il est aussi possible de supprimer une vue exportée par la méthode `removeExportedView`. Les deux méthodes qui permettent respectivement d'ajouter ou de supprimer une vue importée sont les deux méthodes `bindImportedView` et `unbindImportedView`. L'ajout d'une vue importée implique l'ajout d'un objet de type `ImportExport`, passé en paramètre, qui sert de proxy pour accéder au domaine de données d'origine de la vue importée. La méthode de suppression d'une vue importée `unbindImportedView` retourne toutes les vues exportées du domaine qui dépendent d'elle.

L'évolution d'un domaine peut avoir des répercussions sur le contenu local d'un domaine mais aussi sur les dépendances entre les différents domaines de données d'un système. En effet, la mise à jour de l'arbre de requêtes d'une vue exportée d'un domaine peut ne pas entraîner de changement dans l'autre domaine qui importe cette vue. Par contre, l'ajout et la suppression des vues importées et/ou exportées ont des conséquences sur les dépendances entre les domaines de données, et cela peut entraîner le changement de la topologie de l'architecture entre les différents domaines d'un système.

Le maintien de la cohérence entre les vues importées et exportées dans un domaine ou entre deux domaines dépendants (un domaine qui importe des vues de l'autre domaine), suite à l'évolution, doit être assurée par l'utilisateur ou l'administrateur du système d'intégration. Elle se traduit par une séquence d'appels aux méthodes fournies par l'interface d'évolution qui permettent une évolution graduelle du contenu d'un domaine ainsi que les dépendances entre les domaines. Cette approche suit le même principe que les approches d'évolution des schémas par transformation, comme typiquement dans [110]. Dans cette approche, où le schéma est représenté sous forme d'un graphe, l'évolution d'un schéma est exprimée par une séquence de transformations sur les éléments du graphe. Ces séquences consistent à ajouter ou supprimer des noeuds, déplacer des noeuds, renommer les arcs du graphe, etc.

Comme pour l'interface d'exportation, un domaine de données peut ne pas offrir de fonction de contrôle ou en offrir seulement quelques-unes. Par exemple, on peut avoir un domaine qui permet seulement l'introspection de son contenu mais pas sa mise à jour.

3.5 Conclusion

Ce chapitre a présenté le canevas de domaine de données à la base de la construction de systèmes d'intégration de données. Un domaine de données permet l'importation de données vers les autres domaines de données et offre des fonctions de contrôle pour introspecter et faire évoluer son contenu. La composition de domaines de données permet la définition d'un système d'intégration de données en autorisant le partage des ressources - données et traitement - entre les différents domaines.

Ce principe architectural de composition suit le principe d'une architecture de médiation. Les domaines de données primitifs sont l'équivalent des adaptateurs et les domaines de données non primitifs sont des médiateurs. Cette architecture peut être considérée comme une spécification à composant logiciel d'une architecture de médiation minimale selon l'approche GAV. Les requêtes associées aux vues exportées ainsi que les requêtes qui permettent d'interroger les domaines sont exprimées indépendamment d'un langage de requêtes par le canevas d'expression

```
interface Evolution {

    // Ajouter une nouvelle vue exportée
    IView addExportedView(QueryTree query, String name)
                                          thorws TakfaException;

    // Supprimer une vue exportée
    void removeExportedView(IView view);

    // Renommer une vue exportée
    void renameExportedView(IView view, String newName)
                                          throws TakfaException;

    // Mettre à jour la définition de la vue exportée
    void updateExportedViewQuery(IView view, QueryTree newQuery);

    // délier une vue importée
    IView[] unbindImportedView(IView view);

    // Lier de nouvelles vues importées
    void bindImportedView(QueryTree query, ImportExport import);

}
```

FIG. 3.11 – L'interface `Evolution`

de requêtes présenté dans chapitre 5. Un domaine de données peut donc être interrogé par différents langages de requêtes : ceci constitue une différence majeure avec les systèmes de médiation classiques.

Dans ce sens, une architecture à base de domaines montre l'avantage d'utiliser les composants logiciels dans la construction des systèmes d'intégration de données. Elle ne résout donc pas tous les problèmes de l'intégration de données mais permet de maîtriser et de simplifier la construction des systèmes en offrant une architecture exploitable et administrable.

Le chapitre suivant évoque la représentation des données dans les systèmes d'intégration à base de domaines.

Chapitre 4

Modèle de données pour l'intégration

L'intégration est un processus et non un état. Cela signifie qu'il y a évolution, mouvement et transformation. Ce processus peut ne pas être linéaire mais suivre un rythme saccadé.

Emile Durkheim " Les règles de la méthode sociologique "

4.1 Introduction

L'un des objectifs des intergiciels d'intégration de données est l'adaptabilité aux modèles de données des sources. Les wrappers dans les systèmes de médiation transforment les données des sources dans le modèle de données des médiateurs considéré comme le modèle pivot. Dans notre cas, ce sont les domaines de données qui forment une couche intergicielle à composants, entre les applications d'intégration et les sources de données. La représentation des données dans les domaines respecte les règles d'un modèle qui n'est pas forcément celui des sources de données, ni celui des applications gestionnaires des schémas, mais un modèle intermédiaire. Ce modèle permet :

- De représenter les données décrites dans les sources en tenant compte de l'hétérogénéité structurelle. Les domaines de données primitifs sont responsables de la conversion des données des sources dans ce modèle. La transformation des données de ce modèle vers le modèle de données des applications est effectuée par les applications d'intégration elles-mêmes. La figure 4.1 illustre les trois niveaux de modèles de données d'un système d'intégration.

- D'échanger des données entre les domaines de données. Dans une architecture d'intégration, on peut avoir des domaines de données qui ne contiennent que des données structurées, des domaines qui ne contiennent que des données semi-structurées et des domaines de données qui soutiennent à la fois des données structurées et des données semi-structurées. Ces différents domaines doivent échanger les données d'une manière uniforme.

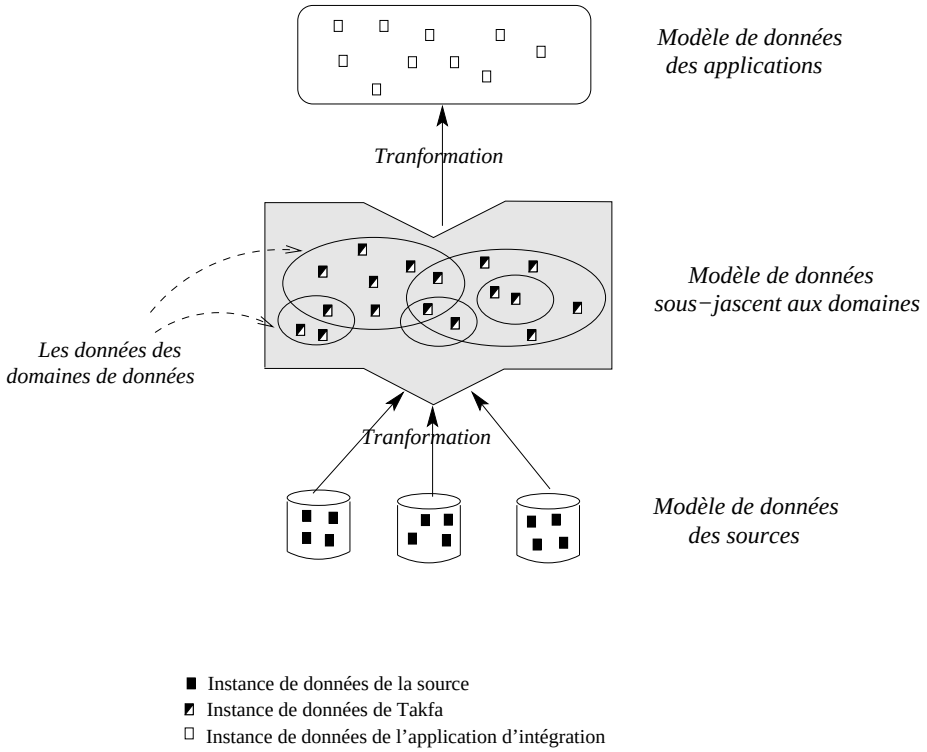


FIG. 4.1 – Modèle de données.

Ce chapitre décrit le modèle de données sous-jacent aux domaines de données. La section 4.3 est consacrée à la présentation du système de types de notre modèle. La section 4.4 présente l'utilisation du typage dans l'inférence de type pour l'intégration de données à travers les opérateurs algébriques. Pour montrer l'adaptabilité de notre modèle de données aux modèles des sources, la section 4.5 montre des cas d'utilisation de ce modèle pour importer des données de sources structurées et semi-structurées. Le mécanisme de nommage qui permet la gestion des références est présenté dans la section 4.6. Enfin, avant de conclure, la section 4.7 présente une analyse de notre modèle de données.

4.2 Propriétés

L'intégration de données se base sur les vues qui sont construites à l'aide d'arbres de requêtes à partir des vues importées. La récupération des données d'un domaine de données se fait par l'évaluation de ces arbres de requêtes. Pour une évaluation efficace, notre modèle de données doit avoir un système de types qui permet d'effectuer des vérifications des types statiquement. Ceci permet, d'une part, de vérifier la concordance des types dans une expression de requêtes, et d'autre part, d'éviter les vérifications des types dynamiques, i.e au moment de l'évaluation des requêtes, qui s'avèrent très coûteuses. Aussi, le fait d'avoir un système de types permet la compilation statique des requêtes pour la génération du code optimisé des requêtes. Notre modèle de données permet des coercisions (ou des conversions) entre les différents types de données, notamment les conversions entre les types primitifs, comme par exemple la conversion d'un type entier vers une chaîne de caractères, ou encore la conversion d'un type composé vers une chaîne

de caractères. Ceci permet de donner une flexibilité au modèle de données qui est utile dans l'intégration de données, notamment en ce qui concerne le problème de conflits de typage.

4.3 Types de données

Le système de types que nous proposons dans cette section est tiré, en partie, des systèmes proposés dans les travaux autour du typage des données semi-structurées, notamment [35, 114]. Nous considérons donc les types séquence, union et les types primitifs avec les expressions régulières de types. Ces différents types de données sont représentés par le diagramme de la figure 4.1, où chacun des types est spécifié par une classe ou une interface correspondante. La figure 4.2 représente la grammaire qui permet de générer ces différents types.

```
Type := Sequence (Type nom_1, ..., Type nom_n)
        Choice (IType nom_1; ...; IType nom_n)
        (Type)*
        (Type)+
        (Type)?
        (Type)!
        Primitif
        Name

Primitif ::= char, byte, int, string, ...
```

FIG. 4.2 – Les expressions de types

Tout d'abord, notre modèle de données permet de représenter à la fois la structure des données - i.e le schéma - et les instances des données elles-mêmes. Les différents types de données sont :

- Les types primitifs.
- Le type composé tuple.
- Le type composé union.
- Les types collection.
- Le type référence.

La sémantique du sous-typage de notre système de type est un sous-typage par substitution :

`IType1 < : IType2` $\Rightarrow$ Le type `IType1` peut être substitué par `IType2`.

C'est l'équivalent du polymorphisme d'inclusion dans la classification des systèmes de typage faite dans [37]. Ce qui veut dire aussi que les données représentées par la structure de données `IType1` peuvent aussi être représentées par la structure `IType2`. `IType1` peut donc être substitué par `IType2` dans le sens où les opérations applicables à `IType2`, notamment les opérateurs d'une expression de requêtes, sont aussi applicables à `IType1`.

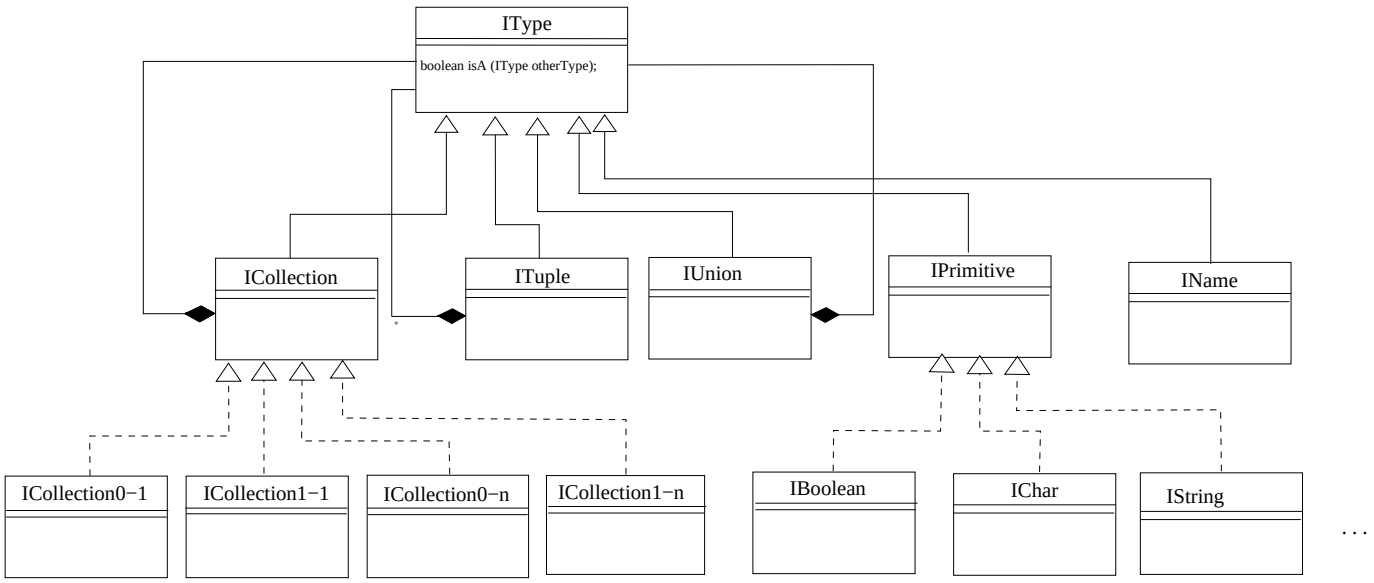


FIG. 4.3 – Le diagramme des types de données

Tout type est décrit par l'interface `IType` dont la spécification est donnée dans la figure 4.4. La méthode `isA(IType otherIType)` permet de tester si le type représenté par ce `IType` est un sous-type du type `otherIType`, qui consiste aussi à vérifier si les données contenues par un type de données sont structurellement incluses dans l'autre type `otherIType`. Dans le cas des types primitifs, cette relation est figée (voir section 4.3.1), et dans le cas des types composés `ITuple, IUnion, ICollection`, cette relation est calculée ou inférée à partir des types qui composent ces types.

Chaque type offre des possibilités de conversion vers un autre type. Les méthodes `toXXX` (e.g : `toBoolean, toChar, toString,` etc.) sur un type de données permettent de le convertir vers le type `XXX`, i.e `Boolean, Char, String,` etc.

4.3.1 Les types primitifs

Ce sont les types qui représentent des valeurs primitives : boolean, char, byte, short, int, long, float, double, etc. A chaque type de données primitif est associée une classe correspondante implantant l'interface `IPrimitif : IChar, IShort, IInt, ILong, IFloat, IDouble,` etc.
Le tableau 4.1 représente les règles de conversion entre les différents types de données primitifs dans le cas du langage de programmation Java.

Par exemple, le type `IByte` est un format convertible à `IInt, ILong, IFloat, IDouble.` C'est à dire que :

```
IByte < : IInt
IByte < : ILong
IByte < : IFloat
IByte < : IDouble
```

```
Interface IType{

    // Test si ce type est inclus dans le type otherIType
    boolean isA(IType otherIType) ;

    // Récupérer la valeur sous
    // forme d'un boolean
    boolean toBoolean()throws TypingException ;

    // Récupérer la valeur sous
    // forme d'un char
    char toChar()throws TypingException ;

    // Récupérer la valeur sous
    // forme d'une string
    String toString()throws TypingException ;
  ...
}
```

FIG. 4.4 – Le type IType

T1 <: T2	IBOOLEAN	ICHAR	IBYTE	ISHORT	IINT	ILONG	IFLOAT	IDOUBLE
IBOOLEAN	x							
IChar		x			x	x	x	x
IBYTE			x	x	x	x	x	x
ISHORT				x	x	x	x	x
IINT					x	x	x	x
ILONG						x	x	x
IFLOAT							x	x
IDOUBLE								x

TAB. 4.1 – Règles de conversions entre les types primitifs

Ceci permet, lors d'une expression de calcul sur des types primitifs dans une expression de requêtes, de connaître le type du résultat. Par exemple, le type du résultat de l'addition d'un Byte et d'un entier est de type entier (`IByte < : IInt)`.

4.3.2 Le type ICollection

Le type `ICollection` représente une collection typée (ou un stream) de valeurs d'un type donné `IType`. Selon le nombre d'occurrences de ce type dans cette collection, on distingue quatre types de `ICollection` comme le montre la figure 4.3.

1. *Des collections* (`IType`) ? : Ce type de collections, représenté par la classe `ICollection0-1`, contient une ou zéro occurrence d'un type `IType`. La présence d'une valeur de type `IType` dans cette collection est optionnelle. Lorsqu'une valeur n'existe pas dans une collection, elle est représentée par la valeur nulle.

2. *Des collections* (`IType`) ! : Ce type de collections, représenté par la classe `ICollection1-1`, contient un et un seul élément de type `IType`. En d'autres termes, on peut dire que cette collection n'est jamais égale à null ou que la présence de la valeur est requise.

3. *Des collections (*`IType`)* : Ce type de collections, représenté par la classe `ICollection0-n`, peut contenir de zéro à n valeurs de type `IType`. Ce type de collections peut aussi être le type du résultat d'une requête ou d'une vue.

4. *Des collections* (`IType`)+ : Ce type de collections, représenté par la classe `ICollection1-n`, contient une ou plusieurs valeurs de type `IType`. Le contenu de ce type de collections n'est jamais vide. Ce type peut aussi représenter le type d'une vue.

Cette spécialisation des différents types de collections permet de représenter les contraintes sur les cardinalités que l'on retrouve à la fois dans les bases de données relationnelles, les bases de données objets et les bases de données XML, à travers les DTDs ou XML Schema [180]. Aussi, ces types représentent implicitement des méta-données sur les valeurs nulles (ou la nullité des données).

Dans la spécification de l'interface `ICollection`, représentée par la figure 4.5, la méthode `GetCollectedIType()` retourne le type des données contenues dans cette collection. Ce type est aussi caractérisé par la façon dont on parcourt ses éléments. Plusieurs algorithmes de parcours peuvent être implantés. Ces algorithmes dépendent du type encapsulé par cette collection et peuvent aller d'un simple itérateur (méthode `getIterator`) à des numérateurs plus spécialisés. Par exemple, si cette collection contient des valeurs de type `ITuple` (voir la section suivante), il est plus simple de parcourir cette collection tuple par tuple en convertissant cette collection en une collection de tuples par la méthode `toTupleCollection`.

Des règles de sous-typage relatives aux expressions régulières de types qui se basent sur le nombre d'occurrence des collections (voir annexe 9.1.3.3) peuvent être appliquées :

```
(IType)? < : (IType)*
```

```
Interface ICollection extends IType{

    //permet d'avoir le type de données
    // représentées dans par la collection
    IType getCollectedIType() ;

   //Récupérer un itérateur sur la collection
    Iterator GetIterator()throws TypingException ;

    // Récupérer le contenu de la collection
    // sous forme d'une collection de tuples
    TupleCollection toTupleCollection()throws TypingException ;
  ...
}
```

FIG. 4.5 – Le type `ICollection`

```
(IType)! < : (IType)+
(IType)+ < : (IType)*
(IType)! < : (IType)?
```

4.3.3 Le type IUnion

Le type `IUnion` représente un choix ou une alternative entre les types qui le composent. Chaque type `IType` de la liste peut avoir éventuellement un nom (ou un label). La méthode `getIUnionFields` de l'interface `IUnion`, représentée par la figure 4.6, permet de récupérer la liste de ces types ainsi que leurs noms respectifs. Chaque couple (type, nom) est représenté par un objet de type `Field`.

```
Interface IUnion extends IType{

    // Récupère les a structure éléments de ce type
    Field[] getIUnionFields() ;
 ...
}
```

FIG. 4.6 – Le type `IUnion`

La sémantique de ce type est la même que celle du choix que l'on retrouve dans la définition des structures des données XML, dénotée par "|" dans les DTDs. Aussi, ce type permet d'exprimer les collections de données d'un super-type dans le cas des bases de données objets.

Beaucoup de règles sont applicables à ce type, notamment les propriétés de commutativité et d'associativité :

```
IUnion (... ; IType1 ; IType1 ; ...) < : IUnion (... ; IType1 ; ...)
```

```
IUnion (... ; IUnion (...) ; ...) < : IUnion (... ; ... ; ...)
IUnion (... ; IType1 ; IType2 ; ...) < : IUnion (... ; IType2 ; IType1 ; ...)
```

La première règle stipule que les duplicata des types qui composent le type `IUnion` peuvent être supprimés. Les deux autres règles montrent l'associativité et la commutativité de ce type.

Les deux règles ci-dessous montrent que les singletons peuvent être réduits en `IUnion`.

```
IType < : IUnion (IType ; ...)
ITuple (IType nom) < : IUnion (IType nom ; ...)
```

4.3.4 Le type ITuple

A l'inverse du type `IUnion`, qui représente une alternative entre plusieurs types, ce type représente une séquence d'un nombre fixe de valeurs d'un type donné. A chaque type est associé éventuellement un nom (ou un label). La méthode `getTupleFields` de l'interface `ITuple`, représentée par la figure 4.7, permet de récupérer la liste des champs qui décrit la structure de ce type (objets `Field`).

```
Interface ITuple extends IType{

    // Récupérer la structure du Tuple
    Field[] getTupleFields() ;

    // Récupérer la valeur de l'attribut de rang rank
    // sous forme d'un boolean
    boolean toBoolean (int rank) throws TypingException ;

    // Récupérer la valeur de l'attribut de rang rank
    // sous forme d'un caractaire
    boolean toBoolean (int rank) throws TypingException ;
 ...
}
```

FIG. 4.7 – Le type `ITuple`

L'avantage structurel que présente ce type de données est la possibilité d'accéder aux valeurs de ces attributs à travers des index. Par exemple, la méthode `toString(int rank)` permet de récupérer la valeur de l'élément de rang `rank` de ce tuple sous forme d'une chaîne de caractères.

Les règles de sous-typage relatives aux caractéristiques du type tuple sont les suivantes :

```
ITuple (... ; IType1 nom ; ...) < : ITuple (... ; IType1 ; ...)
ITuple (IType1) < : IType1
ITuple (... ; IType1 ; ...) < : IUnion (... ; IType1 ; ...)*
```

La première règle stipule que les noms des attributs sont optionnels et peuvent être ignorés. La deuxième règle permet de réduire des tuples singletons à un seul attribut non labélisé en cet

attribut lui-même. La dernière règle permet de réécrire une séquence de types en une collection d'unions de chacun des types qui compose la séquence. Cette règle, que l'on retrouve dans l'ensemble des travaux autour du typage de données semi-structurées [79, 35], est particulièrement utile pour inférer le type de l'union algébrique de deux collections.

4.3.5 Le type IName

Ce type de données représente des identificateurs ou des références des objets `IType` afin de partager les données. Dans la section 4.5, nous verrons comment utiliser ce type dans l'importation des données des sources, et dans la section 4.6, nous proposons un mécanisme de nommage qui permet la gestion de ces noms.

4.4 Sous-typage et intégration de données

La composition des différents types ci-dessus permet de représenter des collections d'objets complexes sous forme de collections d'unions de tuples. Notre système de types contient des types classiques `IUnion` et `ITuple`, que l'on retrouve dans les modèles de données objets et relationnels et des expressions régulières de types. Le sous-typage permet de vérifier si un type est structurellement inclus dans un autre. Ceci permet, entre autres, de savoir si deux vues, ou encore le contenu de deux domaines de données, sont structurellement incluses l'une dans l'autre. Pour les types primitifs, le tableau 4.1 peut être utilisé. Pour les types composés, en plus des règles de sous-typage et de réduction présentées dans les sections précédentes, la vérification porte sur les noms des attributs et leurs types qui composent ces types. Pour donner une petite idée, en voici un exemple.

Soient les deux tuples suivants :

```
ITuple 1 = ITuple((IString nom), (IChar sex)!)
ITuple2 = ITuple((IString nom), (IString sex)?)
```

La relation de sous-typage `ITuple1 < : ITuple2` est vraie parce que les noms des deux attributs des deux tuples sont égaux et `(IChar sex)! < : (IString sex)?`.

Une autre utilisation du sous-typage à laquelle nous nous intéressons particulièrement est l'inférence de types des vues exportées à partir des vues importées. L'inférence de type consiste à construire un nouveau type à partir d'autres types dans une expression d'arbre de requêtes.

Exemple : Soit la construction d'une vue exportée à partir d'une union algébrique de deux vues importées provenant de deux sources de données. Les deux vues importées sont des types `IType1` et `IType2` suivants :

```
IType1 = ITuple((IString nom), (IString Tel))*

IType2 = ITuple((IString nom), (IString e-mail))*
```

Le type résultant de la vue exportée, ou le type du résultat, inféré à partir des deux types `IType1` et `IType2` est :

```
IType3 < : ITuple ((IString nom), IUnion ((IString Tel); (IString e-mail))*
ITYpe3 < : IUnion (ITuple ((IString nom), (IString e-mail)?), ITuple((IString nom),
(IString tel)))*
```

Notons que `IType3` est une sorte de généralisation des deux types `IType1` et `IType2`. Nous avons donc créé un nouveau type de données `IType3` à partir des deux types `IType1` et `IType2`. Dans le chapitre suivant, nous appliquons ce principe sur les opérateurs algébriques en généralisant leurs définitions.

4.5 Importation des données des sources

Les données des sources de données sont tranformées dans notre modèle de données par les domaines de données primitifs. Dans cette section, nous montrons comment représenter les données des sources dans notre modèle de données. Nous prenons les trois cas de sources de données : sources de données relationnelles, sources de données objets et sources de données XML.

– *Cas d'une source relationnelle :*

Les données des sources de données relationnelles sont importées sous forme de collections de tuples. Les vues exportées des domaines de données primitifs, qui représentent les bases de données relationnelles, sont de type (`ITuple(...)`)* (`ICollection0-n(ITuple(...))`). Les attributs peuvent être de types `ICollection0-1` ou `ICollection1-1`, si l'on dispose de méta-informations sur leur nullité (s'ils peuvent être égaux à null ou pas). Cette importation est directe et ne nécessite pas de transformation particulière de données.

– *Cas d'une source de données à objets :*

Pour les bases de données objets, le type des vues exportées par les domaines primitifs est aussi de type collections de tuples : `ICollection0-n(ITuple(....))`. Les attributs d'un tuple sont les attributs des classes correspondantes, qui peuvent être de deux sortes :

- Soit de types `ICollection0-1`, `ICollection1-1` sur des types primitifs dans le cas des attributs primitifs mono-valués et de types `ICollection0-n`, `ICollection1-n` dans le cas des attributs de types primitifs de types multivalués.

- Soit de types `ICollection0-1`, `ICollection1-1`, `ICollection0-n`, `ICollection1-n` sur un type `IName` dans le cas d'une relation entité-association entre les classes d'objets. Les méta-données sur les cardinalités sont implicitement prises en compte par les différents types des collections. Ces types correspondent respectivement aux cardinalités : `[1-0]`, `[1-1]`, `[1-n]`, `[n-n]`.

Exemple :

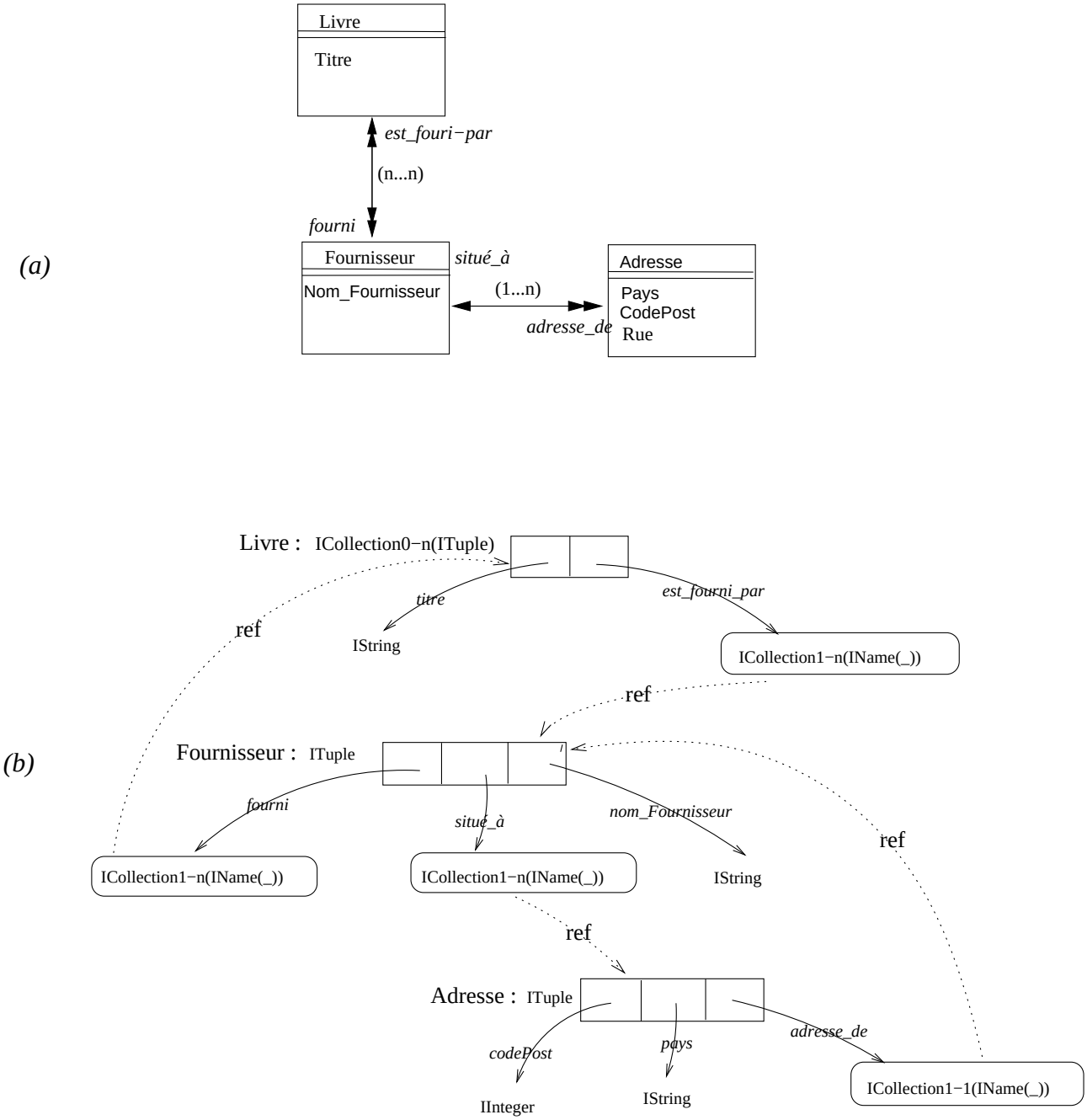


FIG. 4.8 – Exemple d'importation d'une source objet

La partie (a) de la figure 4.8 représente un schéma d'une source de données objet qui contient des livres et des fournisseurs. Un *Livre* peut être distribué par plusieurs fournisseurs et un fournisseur peut fournir un ou plusieurs livres (relation n...n). Un fournisseur peut avoir une ou plusieurs adresses de ses différents sites et inversement, chaque objet de type *Adresse* représente une adresse d'un fournisseur (relation 1..n). La partie (b) de la figure représente le type de la vue qui importe les instances d'objets de la classe *Livre* avec tous ses attributs dans notre modèle

de données.

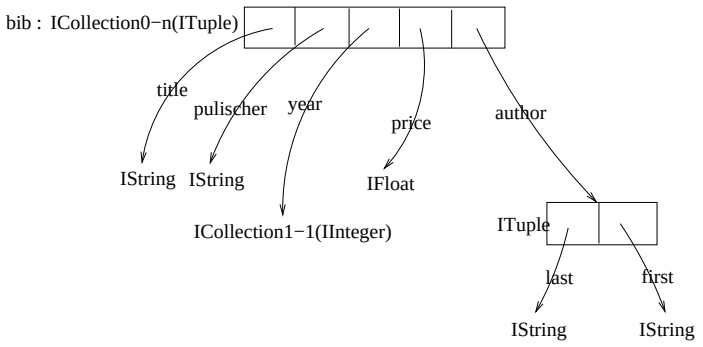
Notons que notre modèle permet de représenter la relation d'héritage. Une super-classe C de deux sous-classes C1, C2 peut être importée sous forme du type `IUnion` entre les collections de tuples représentant les extensions des classes C1 et C2. Le type de la vue qui importe C dans notre modèle est :
`ICollection0-n(IUnion((ITupleC1(...),ITupleC2(...)))`.

– *Cas d'une source XML :*

Les sources de données XML sont représentées dans notre modèle en utilisant tous les types décrits dans notre système de type. Le type `ITuple` permet de représenter les séquences associées à un élément, le type `IUnion` est utilisé pour représenter le choix entre deux structures, dénoté par le caractère "|" dans les DTD. Les mentions représentées par les caractères des expressions régulières " ?", "+", "*" sont respectivement représentées par les types de collections `ICollection0-1`, `ICollection1-n`, `ICollection0-n`. Le type `ICollection1-1` est utilisé pour représenter les attributs d'un document ayant la mention "*Required*" dans une DTD. Les noms des attributs d'un `ITuple` représentent les noms des attributs ou les noms des éléments. Les attributs ID et IDREF sont représentés par des `IName` et l'attribut IDREFS est représenté par un type `ICollection0-n (IName(_))` pour désigner une référence multivaluée.

```
< !ELEMENT  bib (book)*>
< !ELEMENT book (title, author+, publisher, price)>
< !ATTLIST book year  CDATA #REQUIRED >
< !ELEMENT  author (last, first) >
< !ELEMENT  title  (#PCDATA) >
< !ELEMENT  last  (#PCDATA) >
< !ELEMENT  first  (#PCDATA) >
< !ELEMENT  publisher  (#PCDATA) >
< !ELEMENT  price  (#PCDATA) >
```

(a)



(b)

FIG. 4.9 – Exemple d'importation d'une source XML

Notre modèle de données ne prend en compte que les deux types d'items *Element* et *Attribute* par rapport au modèle de données associé à XQuery [176] (voir section annexe 9.1.3.2). Tous les autres types de noeuds sont considérés comme des éléments et sont représentés sous forme de `ITuple` dans notre modèle. Dans [29], cette transformation est appelée *atomisation* d'un document. Notre modèle permet donc de représenter d'une manière atomique un document XML. Inversement, le passage d'une représentation atomique vers une représentation dans le modèle de données XML, en considérant tous les types de noeuds, se fait par la création des différents noeuds à partir d'une collection de tuples. Les applications de mapping des documents XML vers les bases de données relationnelles se basent sur ce même principe.

Exemple :

La partie (a) de la figure 4.9 représente une DTD d'une source de données XML qui contient des livres avec leurs auteurs et leurs publishers. La partie (b) de la figure représente le type de la vue qui importe les livres et les informations associées.

4.6 Gestion des références

Les instances de données des sources sont généralement importées sous forme de collections de tuples. Une source de données peut avoir plusieurs collections de tuples qui partagent les données à travers les `IName`. Pour la gestion de ces références, on utilise les deux concepts de nommage, nom et contexte de nommage, comme dans la spécification ODP(Object Distributed Processing) [84, 85].

Un *contexte de nommage* est responsable de la gestion des noms `IName`. Un *nom* désigne un identificateur ou une référence valide seulement dans son contexte de nommage. Sur la figure 4.10, l'interface `IName` offre une méthode principale `getNamingContext` qui retourne le contexte de nommage gérant ce nom. Cela signifie que, pour accéder à l'objet référencé par un nom, il faut passer par son contexte de nommage pour décoder la référence. Dans notre cas, on associe à chaque type `ICollection0-n(ITuple(_))` un contexte de nommage qui permet de créer des identificateurs et des références `IName` sur les instances des tuples. Comme spécifié par l'interface `INamingContext`, représenté par la figure 4.10, un contexte de nommage exhibe trois opérations principales sur les noms que sont : `export`, `resolve` et `unexport`.

```
Interface IName extends IType{

    // Récupérer le contexte de nommage associé à ce nom
     INamingContext getNamingContext() ;
 ...
}

Interface INamingContext {

    // Récupérer le type représenté par ce type référence
    IType export(Object obj) ;

    // Résoudre un nom dans ce contexte de nommage
    Object resolve (IName nom) ;

    // Supprimer un nom de ce contexte de nommage
    void unexport(IName nom) ;
  ...
}
```

FIG. 4.10 – Les interfaces de nommage : `IName` et `INamingContext`

- L'opération `export` permet de créer des objets `IName` par un contexte de nommage. Selon la sémantique des noms gérés par ce contexte de nommage, la création des noms peut être faite à partir :

  - De clés primaires/étrangères dans le cas des bases de données relationnelles, des identificateurs d'objets dans le cas des sources de données objets, ou à partir des ID, IDREF et IDREFS dans le cas des sources de données XML. Ceci permet d'assurer l'unicité des noms.

  - D'un autre nom `IName`. Un contexte de nommage permet aussi d'associer un nom `IName2` à partir d'un autre nom `IName1`, valide dans un autre contexte de nommage d'une autre collection de tuples. Ceci se traduit par la création d'une référence `IName2` dans une collection de tuples vers une instance de tuples, représentée par un nom `IName2` dans une autre collection de tuples. Cette opération peut être répétée plusieurs fois pour former une chaîne de liaisons entre les différents contextes de nommage.

  - De plusieurs autres `IName`. Ce mode de création est utile lorsqu'on a besoin d'associer des noms uniques aux tuples à partir de plusieurs autres noms. Dans l'intégration de données, ceci est connu sous le nom de fusion d'objets lorsqu'il est question de créer un objet qui intègre plusieurs autres objets [136].

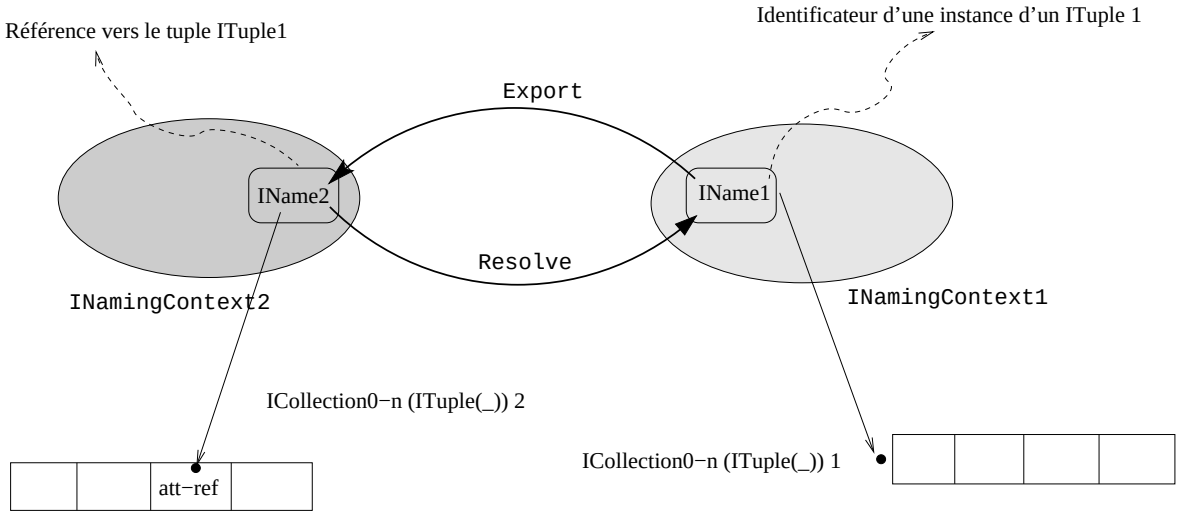


FIG. 4.11 – Noms et Contexte de Nommage : Gestion des références

- L'opération `resolve` sur un nom `IName` est l'inverse de l'opération `export`. Elle permet de retrouver l'instance `ITuple` associée à ce `IName` dans un contexte de nommage. Dans le cas d'une chaîne de nommage, cette opération retourne un IName. L'opération `resolve` est répétée plusieurs fois sur les contextes de nommage de la chaîne de nommage jusqu'à atteindre la valeur de ce nom, i.e, une instance d'un `ITuple`.

- L'opération `unexport` permet de détruire un non `IName` dans ce contexte de nommage.

La figure 4.11 illustre les deux opérations `export` et `resolve` sur deux contextes de nommage de deux collections de tuples. Chaque attribut *att-ref* d'un tuple de la deuxième collection référence un tuple de la première collection. Le contexte de nommage `INamingContext1` permet de générer des noms aux tuples de la première collection de tuples. Le contexte de nommage `INamingContext2` permet de générer les références associées à l'attribut *att-ref* vers les tuples de la première collection. Ce nom `IName2` représente la valeur d'un attribut d'un tuple de la deuxième collection et qui référence une instance d'un tuple dans la première collection identifiée par le nom `IName1`.

Comme déjà évoqué, un nom `IName` peut représenter à la fois une référence et un identificateur . Dans le cas où il est géré par le dernier contexte de nommage de la chaîne de nommage, il est considéré comme un identificateur d'un tuple - comme `IName1` - et dans les autres cas, il représente une référence (comme `IName2`). Ce mécanisme de nommage est le même que celui utilisé dans l'ORB Jonathan [53] et dans le canevas de persistance d'objets JORM [55, 125], présenté dans la section 7.1.4.

4.7 Discussion et conclusion

Dans ce chapitre, nous avons présenté le modèle de données utilisé par les domaines de données. Les données des sources, qu'ellent soient structurées ou semi-structurées, sont transformées dans ce modèle qui permet aussi de représenter la structure des vues dans les domaines. Ce modèle n'est pas vraiment un nouveau modèle, vu qu'on retrouve des concepts de différents modèles de données, relationnel, objet et semi-structuré. Il permet de représenter les données des sources sous forme de collections. Il utilise des concepts objets comme les identificateurs et les références, le concept de séquence pour représenter le tuple, que l'on retrouve dans les bases de données relationnelles et objets, et, enfin, les expressions régulières de types des données semi-structurées. La composition des différents types permet de décrire des structures de données complexes afin de représenter des collections d'objets complexes. Ceci suffit pour représenter les données importées à travers les différents types de sources de données.

Le typage statique des données semi-structurées a fait l'objet de plusieurs travaux de la part de la communauté langage et de la part de la communauté base de données (voir annexe 9.1), notamment dans le systèmes d'intégration YAT [42]. Les principes et les règles de typage que nous avons adoptés dans notre système sont les mêmes que tous ces travaux autour du typage statique et de l'inférence des types, et plus particulièrement le système présenté dans [114] où l'on retrouve le type tuple. Dans YAT, le modèle de données est défini à trois niveaux de généricité : schéma, système de type et instances de données. Le système de types est le même que le nôtre. Cependant, la différence réside principalement dans le fait que les instances de données dans YAT sont représentées sous forme d'un graphe d'objets, typiquement des objets OEM (voir section annexe 9.1.3.1 pour des détails sur OEM), alors que dans notre modèle elles sont représentées par des collections de tuples. Les valeurs des champs inexistants (de type `(IType) ?`) sont représentées par la valeur null alors que dans un graphe d'objets ces valeurs ne sont même pas représentées.

Notons que la notion de valeur optionnelle existe aussi dans le cas des bases de données relationnelles. On peut donc exprimer une séquence par un tuple tout en représentant les valeurs qui n'existent pas par les valeurs nulles. Le traitement des valeurs nulles a fait l'objet de plusieurs

recherches, que se soit dans le domaine des bases de données ou dans le domaine des langages de programmation [77]. Dans les bases de données [191, 98, 70], le traitement consiste souvent à traiter la sémantique de la valeur nulle (valeur inapplicable ou valeur inconnue, etc.). Dans notre cas, le traitement de la sémantique des valeurs nulles est délégué aux applications d'intégration, car leur interprétation dépend du domaine d'application sous-jacent. Cependant, nous verrons dans le chapitre suivant que notre canevas d'expression de requêtes prend en compte les valeurs nulles. Pour le type tuple, des travaux récents ont montré qu'il est possible d'ajouter ce type dans les langages de programmation [171, 172] sans altérer les performances, ce qui va permettre de pallier le problème *Impedance mismatch*. Aussi, le fait d'utiliser le type tuple nous permet de réutiliser le patrimoine des techniques de traitement de requêtes héritées des bases de données relationnelles, notamment le traitement en pipeline, comme nous le verrons dans le chapitre 6. En contrepartie, ce modèle ne permet pas de prendre en compte les données unstructurées tel que le modèle OEM, car c'est un modèle typé.

Dans le chapitre suivant, nous présentons le canevas d'expression de requêtes où nous montrons réellement la flexibilité du système de types pour l'intégration de données.

Chapitre 5

Canevas d'expression des requêtes

Le langage travestit la pensée. Et notamment de telle sorte que d'après la forme extérieure du vêtement l'on ne peut conclure à la forme de la pensée travestie; pour la raison que la forme extérieure du vêtement vise à tout autre chose qu'à permettre de reconnaître la forme du corp.

Ludwig Wittgenstein " Tractus logico–philosophicus "

5.1 Introduction

L'expression de requêtes, indépendamment des langages de requêtes, a été étudiée dans les travaux de recherche qui s'inscrivent pour la construction des optimiseurs de requêtes extensibles pour les SGBD relationnels Volcano [69], OPT+ [93] ou les SGBD objet EROC [113] et TIGUKAT [134]. Dans la majorité de ces travaux, les requêtes SQL ou OQL sont représentées sous forme d'un arbre algébrique, suivant l'algèbre relationnelle ou les algèbres sur les objets. Cette représentation facilite l'application des règles d'équivalence et de réécriture pour l'optimisation des requêtes. L'approche canevas dans la représentation des arbres de requêtes a été particulièrement utilisée dans EROC et TIGUKAT. Les requêtes sont représentées sous forme d'un simple arbre d'opérateurs où chaque opérateur algébrique est représenté par des classes abstraites.

Pour les applications considérées, le contexte d'utilisation des requêtes est plus large, s'inscrivant dans le cadre d'intégration de données. En effet, les requêtes sont utilisées pour deux besoins :

- définir les vues dans les domaines de données. Ceci concerne les requêtes des vues primitives dans les domaines de données primitifs et les vues intégrées dans les domaines de données non primitifs. Dans ce deuxième cas, les opérateurs de ces requêtes jouent le rôle d'opérateurs d'intégration, qui permettent de construire l'arbre de requête des vues exportées à partir des vues importées. L'expression de ces requêtes prend en compte des aspects de l'intégration de données, notamment les conflits de nommage et les conflits de mesures (voir section 2.3) entre les vues importées dans la construction du type des vues exportées.

- Interroger et introspecter le contenu d'un domaine de données par les applications. Le

formalisme d'expression de requêtes est adaptable aux différents langages des applications d'intégration. L'adaptabilité s'entend principalement dans le sens où il capture l'expressivité des langages de requêtes en termes de types d'opérateurs.

Ce chapitre propose un canevas d'expression de requêtes pour décrire et représenter des requêtes, indépendamment d'un langage de requête particulier. Comme dans les langages fonctionnels, l'expression de requêtes se fait d'une manière uniforme par la composition d'expressions pouvant jouer le rôle d'opérateurs ou d'opérandes. Le canevas distingue trois niveaux d'expressions : les expressions de calculs, les expressions de chemins et les expressions d'arbre d'opérateurs algébriques. Le fait de distinguer ces différents types d'expressions permet d'avoir un formalisme d'expression de requêtes extensible. Ainsi, notre approche n'est pas de citer tous les opérateurs possibles mais plutôt de citer tous les types (interfaces) d'opérateurs possibles que l'on peut trouver dans une expression de requête. Adapter notre formalisme de requête revient à implanter un type d'opérateur, notamment un opérateur de calcul, un opérateur algébrique, un opérateur de conversion d'une unité de mesure ou un agrégat, etc.

Dans la première partie de ce chapitre, nous commençons par donner un aperçu des algèbres et opérateurs de base supportés par le canevas. Nous montrons que les définitions des opérateurs algébriques classiques peuvent être généralisées pour notre modèle de données, notamment pour l'intégration de données. Dans les sections 5.4 à 5.6, nous présentons respectivement les trois niveaux d'expression de notre canevas que sont les expressions de calcul, les expressions de chemin et les expressions d'arbre. Enfin, la section 5.7 conclut et résume les propriétés et les avantages du canevas.

5.2 Opérateurs et algèbres

5.2.1 Opérateurs de base

Notre modèle de données hybride permet la description aussi bien de données relationnelles que de données objets et semi-structurées. De la même façon, le canevas d'expression de requêtes supporte plusieurs opérateurs, aussi bien de l'algèbre relationnel que des algèbres sur les objets. On distingue les opérateurs suivants :

- Les opérateurs de restriction : Sélection et Jointure.

- Les opérateurs ensemblistes : Union, Différence, Intersection et Produit Cartésien.

- Les opérateurs de restructuration des objets complexes : Nest, UnNest. Ces opérateurs, introduits dans [60, 87] pour le modèle relationnel étendu (Nested Relationnal), puis généralisés dans les algèbres sur les objets [3, 155], permettent respectivement de grouper et de dégrouper des collections de données.

- Les opérateurs de navigation : ces opérateurs servent à construire des expressions de chemins pour naviguer à travers des structures complexes qui sont défines dans notre modèle de données par des `IType`. Ceci permet un accès associatif aux structures de données à travers des noms. Pour une navigation flexible, et vu que notre modèle de données est hiérarchique et peut avoir une profondeur arbitraire, nous utilisons des expressions régulières pour l'expression des chemins, comme dans les langages de requêtes semi-structurées

YATL [43], LOREL [111], XML-QL [49],Quilt [39] et XQuery [176] .

- Des opérateurs `d'`agrégation : sum, avg etc.

- Des opérateurs de calcul pour l'expression des prédicats associés aux opérateurs algébriques.

- Des opérateurs propres à l'intégration, tels que des opérateurs de renommage et de conversion.

- Des opérateurs d'accès aux sources de données. Ces opérateurs dépendent du type de la source de données cible et de leur mode d'accès.

A un certain niveau, les opérateurs algébriques tels que la sélection ou la projection, portent la même sémantique dans l'expression de requêtes quel que soit le modèle de données sous-jacent. Cependant, la différence réside dans leur définition en terme des types des opérandes d'entrées et de type des résultats retournés par ces opérateurs :

- *Le type des opérandes* : ces types représentent les conditions d'application d'un opérateur donné. Par exemple, dans l'algèbre relationnelle, les relations opérandes des opérateurs ensemblistes Union, Différence et Intersection sont du même type. Dans les algèbres sur les objets, et typiquement l'algèbre d'Encore [155, ?], il y a peu de restrictions sur les types des collections d'entrées, dans la mesure où l'on se base sur les identificateurs d'objets et non pas sur les valeurs des attributs. Par exemple, l'application de l'opérateur Différence revient à tester l'appartenance d'un objet à une collection en comparant les identités d'objets.

- *Le type du résultat* : dans l'algèbre relationnelle, le résultat de l'opération de jointure est une relation. Dans l'algèbre d'Encore, l'opération de jointure retourne une collection d'objets qui n'est pas forcément une collection d'objets d'une classe donnée.

Il faut souligner aussi que, même entre les algèbres sur les objets, les opérateurs n'ont pas exactement les mêmes définitions. Pour des détails, une comparaison entre quatre algèbres sur les objets dont l'algèbre d'Encore est présentée dans [45].

Dans la section suivante, nous donnons les définitions générales des opérateurs algébriques appliqués à notre modèle de données, sans contraintes sur les types d'entrée et les types des résultats. Cette généralisation permet, en particulier, aux opérateurs algébriques de jouer le rôle d'opérateurs d'intégration de données lors de la construction des requêtes associées aux vues intégrées.

5.2.2 Généralisation des opérateurs pour l'intégration des données

Chaque opérateur algébrique est considéré comme une fonction qui prend en entrée des collections d'objets de type `IType` et retourne une autre collection de type `IType`. Dans ce qui suit, nous considérons les deux types de collections de notre modèle de données dans les définitions des opérateurs algébriques, à savoir :

1. Des collection de tuples : `ICollection`i`-n(ituple)`, $i \in \{0, 1\}$.

ou

2. Une collection d'une union de tuples : `ICollection`i`-n (IUnion(ITuple-1, ... , ITuple-n))`, $i \in \{0, 1\}$.

Ces deux types de collections sont représentatives de collections d'objets complexes que l'on peut décrire dans notre modèle de données, notamment dans l'importation des données des sources structurées et semi-structurées (voir section 4.5). Les attributs à projeter, ainsi que les attributs présents dans une expression d'un prédicat, sont désignés par des expressions de chemins, présentées dans la section 5.5.

5.2.2.1 Sélection

$Selection$: $IType1 \rightarrow IType2$

Le cas où le type d'entrée `IType1` est une collection de tuples représente la définition de la sélection dans l'algèbre relationnelle. Le type de sortie `IType2` est alors identique au type `IType1`.

Dans le cas où `IType1` est une collection d'une union de tuples, le type `IType2` est aussi le type `IType1`. Les instances de tuples résultats de la sélection peuvent être des instances de chacun des tuples qui compose la collection.

5.2.2.2 Projection

$Projection$: $IType1 \rightarrow IType2$

Cet opérateur permet de projeter un ensemble d'attributs désignés par des expressions de chemins. Ces attributs forment le type de la collection résultat de l'opération.

Le cas où `IType1` est une collection de tuples représente la définition de la projection dans l'algèbre relationnelle. Le type `IType2` est alors aussi une collection de tuples, constituée des attributs projetés.

Dans le cas où `IType1` est une collection d'une union de tuples, `IType2` est aussi une collection d'une union de tuples. Chaque tuple contient des attributs projetés de l'un des tuples de la collection d'union. Dans certains cas, le type du résultat peut être réduit suivant notre système de type.

Exemple :

soit : `IType1` = `ICollection0-n(IUnion(ITuple1,ITuple2))`.

Si l'on veut projeter sur un attribut *nom* du tuple `ITuple1` et sur un attribut *nom* du tuple `ITuple2` alors le type du résultat `IType2` est :

`ICollection0-n (IUnion(ITuple (IType nom), ITuple(IType nom)))`. Ce type peut être réduit à :

`ICollection`0`-n (IUnion (ITuple (IType nom))<: ICollection0-n (ITuple(IType nom))`.

5.2.2.3 Jointure

L'opérateur de jointure prend en entrée deux collections `IType1` et `IType2` et retourne une autre collection `IType3`. On suppose que les attributs de `IType1` et de `IType2` sont distincts.

Jointure : $IType2$, $IType2 \rightarrow IType3$

Dans le cas où `IType1` et `IType2` sont tous les deux des collections de tuples, alors `IType3` est une collection de tuples dont les attributs appartiennent aux tuples de `IType1` et `IType2`.

Dans le cas où `IType1` ou `IType2` est une collection d'unions de tuples, les règles de distributivité du type `IUnion` sur la concaténation peuvent être appliquées pour réduire le type du résultat.

Exemple :

Supposons que l'on veuille croiser les deux collections suivantes :

`IType1 = ICollection0-n(IUnion(ITuple1,ITuple2))` et

`IType2 = ICollection0-n(ITuple3)`.

Le prédicat de jointure est entre `ITuple1` et `ITuple3` ou/et `ITuple2` et `ITuple3`.

Le type du résultat est :

`IType3 = ICollection0-n(IUnion(ITuple1-3 , ITuple2-3))` tel que :

le tuple `ITuple1-3` contient des attributs provenant de `ITuple1` et de `ITuple2` et le tuple `ITuple2-3` contient des attributs du tuple `ITuple2` et `ITuple3`.

Notons que le même raisonnement peut être appliqué à l'opérateur produit cartésien.

5.2.2.4 Union

L'opérateur d'union prend en entrée deux collections `IType` et retourne une nouvelle collection `IType` qui contient les données des deux collections d'entrées.

Union : $IType1$, $IType2 \rightarrow IType3$

Comme nous l'avons déjà montré dans la section 4.3.3, l'union de deux collections est une collection décrite par le type `IUnion`. On suppose que `IType1` = `ICollection0-n(IType11)` et `ICollection0-n(IType22))` alors `IType3 = ICollection0-n(IUnion (IType11, IType22)`. Suivant les règles de réduction associées au type `IUnion`, `IType3` peut être réduit dans certains cas. Notons que le cas où `IType1` et `IType2` représentent tous les deux des collections de tuples et où `IType11 = IType22` couvre le cas de la définition de l'union dans l'algèbre relationnelle, `IType3` est alors réduit à `ICollection0-n (IType11)`.

Exemple :

Soit à faire l'union de deux collections contenant des livres :

`IType1 = ICollection0-n(ITuple(IString titre, IString auteur))` et

`IType2 = ICollection0-n(ITuple(IString titre, IString auteur, IFloat price))`.

Le type `IType3` union des deux peut être écrit sous la forme suivante :

`IType3 = ICollection0-n(ITuple(IString titre, IString auteur, ICollection0-1(IFloat) price))`

Tels que : `IType1 < : IType3` et `IType2 < : IType3`

5.2.3 Discussion

Dans cette section, nous avons exploité le système de types du modèle de données pour réduire et inférer le type construit par les opérateurs algébriques, sans contrainte sur les type des collections opérandes. Nous avons montré que les définitions des opérateurs dans l'algèbre relationnelle sont couvertes par les définitions générales décrites dans la section précédente. Ces opérateurs sont considérés à la fois comme des opérateurs d'interrogation et comme des opérateurs d'intégration, dans le sens où l'on crée de nouvelles structures de données.

L'utilisation des opérateurs pour l'intégration de données selon l'approche GAV n'est pas nouvelle dans les systèmes d'intégration de données. Comme déjà évoqué dans la section 2.3.2, les vues intégrées orientées objets, sont décrites par un ensemble d'opérateurs d'intégration, comme les opérateurs de généralisation, de spécialisation, etc. Ces opérateurs permettent de créer de nouveaux types d'objets (c'est-à-dire de nouvelles classes) à partir d'autres types (c'est-à-dire d'autres classes ou interfaces). Sachant que notre modèle de données est plus simple que le modèle objet, l'opération algébrique d'union entre deux `IType` peut aussi être considérée comme une généralisation des deux types en entrée. Notons qu'à la différence d'autres approches, nous utilisons les mêmes opérateurs algébriques pour l'interrogation et pour l'intégration, en se basant sur la flexibilité du typage de notre modèle de données.

Même si nous avons généralisé les définitions des opérateurs algébriques, en pratique, cela ne veut pas dire que nous avons une seule implémentation pour chaque opérateur algébrique. En effet, dans l'expression d'une requête d'interrogation d'un domaine de données qui ne contient que des données relationnelles, on peut exiger que les opérateurs algébriques soient conformes aux définitions de l'algèbre relationnelle. On peut aussi personnaliser le type du résultat d'un opérateur donné suivant les besoins. Dans ces cas, on ne tient pas compte des définitions générales et l'on n'exploite pas la flexibilité du typage. On peut donc avoir plusieurs implémentations pour chaque type d'opérateurs algébriques.

Dans notre canevas d'expression de requêtes, nous ne fixons ni la définition ni le mode d'application des opérateurs. L'expression des requêtes se fait par un canevas générique qui permet de supporter différents types d'opérateurs, indépendamment d'une algèbre particulière.

5.3 Expressions

Une expression de notre canevas est une expression primitive, comme, par exemple, une constante, une feuille de l'arbre de requête qui représente une source de données, ou une expression composite composée d'autres expressions telle qu'un opérateur de calcul ou un opérateur algébrique. Comme illustré par le diagramme de la figure 5.1, on distingue trois types d'expressions :

- *Des expressions de calcul* : Les opérateurs formant une expression de calcul (interface `CExpression`) opèrent sur des valeurs, des paramètres et des expressions de chemins.

- *Des expressions de chemins* : Une expression de chemin (interface `PathExpression`) permet de naviguer à travers un type `IType` d'une expression.

- *Des expressions de l'arbre de requête* : Ces expressions permettent de décrire les arbres de requêtes représentés par l'interface `QueryTree`. Ces opérateurs sont des expressions com-

posées des expressions de calculs et des expressions de chemins.

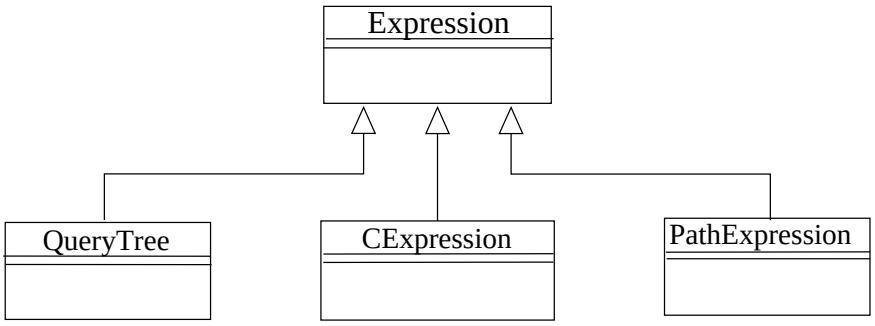


FIG. 5.1 – Les expressions du canevas

```
public interface Expression {

    // Récupérer le type de l'expression
    IType getIType()Throws ExpressionTypingException;

    // Vérifier et compiler l'expression
    IType validateExpression() throws MalformedExpressionException;

    // Vérifier si deux expressions sont équivalentes
    boolean isEquivalent(Expression exp);

}
```

FIG. 5.2 – L'interface `Expression`

Chacune de ces expressions étend l'interface générique `Expression` (voir figure 5.2). Cette interface montre que :

- Les expressions sont typées. A chaque expression est associé un type `IType` qui représente le type de la valeur retournée par l'expression. La méthode `getIType` permet de récupérer le type de l'expression. Ce type dépend des opérateurs et des opérandes qui constituent l'expression. Il peut être donné au préalable, dans le cas des expressions primitives, et inféré ou calculé, dans le cas des expressions non primitives, au moment de la compilation de l'expression.

- Les expressions sont compilables. Le rôle de la compilation est double :

  - Dans un premier temps, elle permet une vérification sémantique et syntaxique de l'expression. Ceci consiste globalement à vérifier la concordance des types des opérandes par rapport aux définitions des opérateurs. Cette vérification concerne aussi bien de simples opérateurs arithmétiques que les opérateurs algébriques. L'appel de la méthode `validateExpression` sur une expression composite déclenche des appels récursifs sur

les méthodes `validateExpression` d'autres expressions qui la composent. Dans le cas où la vérification échoue, une exception est levée.

- Dans un deuxième temps, le type global de l'expression ainsi que les types des sous-expressions qui représentent les types des résultats intermédiaires sont inférés. Ceci revient à créer des objets `IType` correspondant à chacune des expressions. Ces types représentent les structures de données des tampons utilisés, notamment les résultats intermédiaires des requêtes lors de l'évaluation. Notons que la compilation doit être effectuée à chaque mise à jour qui consiste à ajouter ou à enlever des sous-expressions de l'expression globale.

- Chaque expression peut être comparée à une autre expression. La méthode `isEquivalent` retourne vrai si deux expressions sont équivalentes. Cette fonction est utilisée par l'optimiseur de requêtes pour la recherche du plan dans l'espace de recherche.

5.4 Expressions de calcul

Les expressions de calcul permettent d'exprimer des prédicats associés aux opérateurs algébriques, aux opérateurs d'agrégation et aux opérateurs de calcul propres à l'intégration, tels les opérateurs de conversion des unités de mesures.

Une expression de calcul est composée d'expressions opérandes et opérateurs. La figure 5.4 illustre le diagramme qui catégorise les différents types d'opérateurs et d'opérandes dans une expression de calcul. De telles expressions sont simplement représentées sous forme d'un arbre où les feuilles désignent les opérandes, et les noeuds désignent les opérateurs de calcul. Les expressions de calcul sont spécifiées par l'interface `CExpression` (voir figure 5.5). Cette interface hérite de l'interface `Expression` et offre une seule méthode `evaluate` pour calculer le résultat de l'expression. Elle prend comme paramètre d'entrée une liste de valeurs `ParameterOperand[]`, qui sont affectées aux paramètres qui composent l'expression. Le résultat de l'évaluation est systématiquement écrit dans l'objet représentant le type de l'expression retourné par la méthode `getIType` de l'interface `Expression`.

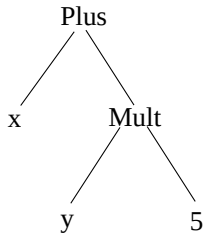


FIG. 5.3 – Représentation d'une expression de calcul sous forme d'un arbre

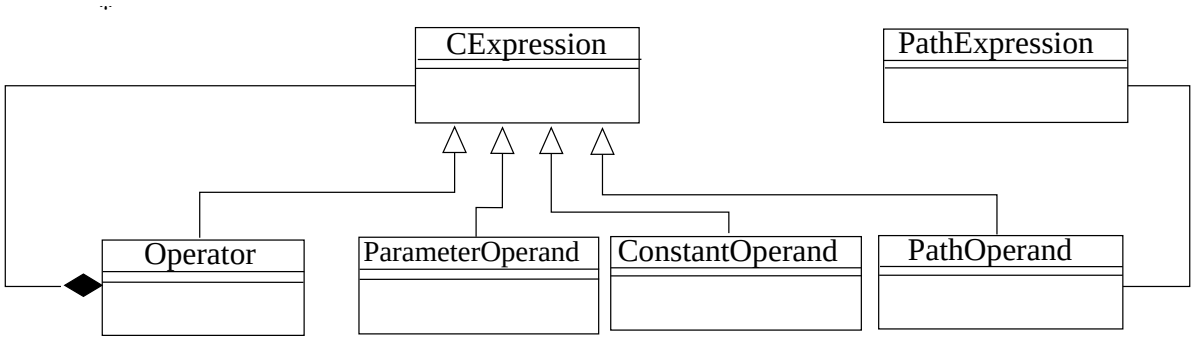


FIG. 5.4 – Le diagrammes des classes des expressions de calcul

```
public interface CExpression {

    // Évaluer l'expression de calcul
    void evaluate(ParameterOperand[] values);
                                    throws ExpressionException;
}
```

FIG. 5.5 – L'interface `Expression`

5.4.1 Les opérandes

On distingue trois types d'opérandes :

- *Les constantes* : Les constantes d'une expression de calcul sont représentées par l'interface `ConstantOperand`. Elles représentent une valeur d'un type `IType`. Les méthodes de l'interface `IType toXXX` sont utilisées pour lire la valeur de la constante lors de l'appel de la méthode `evaluate`. Sur la figure 5.3, cette opérande représente la constante 5.

- *Les paramètres* : Les paramètres sont représentés par l'interface `ParameterOperand`. Cette interface offre des méthodes `setXXX` (`setBoolean`, `setInteger`, etc.) pour affecter de nouvelles valeurs au paramètre lors de l'appel de la méthode `evaluate`. Sur la figure 5.3, les deux paramètres x et y sont représentés par des `ParameterOperand`.

- *Les opérandes représentées par des expressions de chemin :* A chacune de ces opérandes, désignées par l'interface `PathOperand`, est associée une expression de chemin qui désigne sa valeur et son type. Nous reviendrons dans les détails sur les expressions de chemins dans la section 5.5.

5.4.2 Les opérateurs de calcul

Les opérateurs de calcul sont des expressions composées d'un (opérateur unaire), de deux (opérateur binaire) ou de plusieurs sous-expressions. Comme spécifié par l'interface `COperator` (voir figure 5.6), chaque opérateur a un nom et offre des méthodes qui permettent de récupérer et de mettre à jour les expressions qui le composent, notamment les deux méthodes `getCExpression` qui retournent l'expression fils de rang i, et la méthode `setCExpression` qui met à jour l'expression fils de rang i de cette expression. Les méthodes de mise à jour sont notamment utilisées

par l'optimiseur lors de la réécriture des expressions.

```
public interface COperator extends CExpression{

    // Récupérer le nom de l'opérateur
    string getName() ;

    // Récupérer l'expression fils de rang i
    CExpression getCExpression(int i) ;

    // Mettre à jour l'expression fils de rang i
    void setCExpression(CExpression exp, int i) ;
  ...
}
```

FIG. 5.6 – L'interface `COperator`

Il existe plusieurs types d'opérateurs de calcul :

- Des opérateurs arithmétiques binaires tels que l'addition, la soustraction, etc. Ils opèrent sur les types primitifs. Le type `IType` du résultat de ces opérateurs est inféré lors de l'appel à la méthode `validateExpression` à partir des types des sous-expressions. Le tableau 4.1, qui décrit les règles de conversion entre les types primitifs, est utilisé pour déterminer le type englobant. Par exemple, l'opérateur d'addition entre une valeur de type `IInteger` et une autre valeur de type `ILong` retourne une valeur de type `ILong`. Dans tous les cas, pour qu'un type T englobe les types T1, ...Tn, il faut que Ti $<:$ T , $\forall i \in \{1, ..n\}$.

- Des opérateurs logiques (and, or, etc), qui opèrent sur les types `IBoolean`. Le type du résultat de cet opérateur est toujours un `IBoolean`.

- Des comparateurs tels que ">","<", etc. Le type du résultat est toujours un `IBoolean`.

- Des fonctions mathématiques de calcul, telles que la valeur absolue et la racine carrée, etc.

- Les opérateurs sur les collections, tels que `memberOf` qui est un opérateur binaire, permettent de tester l'appartenance d'une valeur à une collection.

- L'opérateur unaire `isNull`, qui permet de tester si une valeur donnée est égale à nulle ou pas. Cet opérateur est utile pour le traitement des valeurs nulles. Notons que cet opérateur est utile seulement si l'opérande concerné est de type `ICollection0-1` ou `ICollection0-n`.

- Des opérateurs d'agrégation, qui portent généralement sur des opérandes de types collection. Ces opérandes peuvent être représentés par des `PathOperand` qui représentent notamment des valeurs d'une collection obtenues à travers l'opérateur `Nest`.

- D'autres opérateurs contribuent directement à l'intégration de données, typiquement les opérateurs de conversions d'unités de mesure. Dans l'exemple que nous avons présenté dans la section 3.2.3, les prix des livres de la source S2 sont exprimés en dollars et les prix des livres de la source S3 sont exprimés en euros. Pour pouvoir comparer les prix de deux

livres, il est indispensable de convertir la valeur des deux attributs qui représentent les prix en une seule unité de mesure. L'expression de calcul du prédicat doit être composée d'un opérateur de conversion de l'euro vers le dollar ou inversement.

5.5 Expressions de chemins

Les expressions de chemins permettent de naviguer ou d'interroger une structure de données représentée par un `IType` en s'appuyant sur les noms des champs de la structure. Pour donner plus de flexibilité dans la navigation, nous utilisons le mécanisme d'expressions régulières présent dans les langages de requêtes semi-structurées [43, 111, 49, 39, 176].

Une expression de chemin est définie par une expression régulière et une structure de données auxquelles cette expression est appliquée. *Une expression régulière* représente une fonction de recherche de chemins. Elle est décrite par une composition d'opérateurs de navigation. Ces opérateurs représentent les caractères d'expressions régulières, comme par exemple "*", " ?", etc. Le diagramme de la figure 5.7 illustre les relations entre les expressions de chemins représentées par l'interface `PathExpression`, et les expressions régulières représentées par l'interface `RegExp`.

Dans la spécification de l'interface `PathExpression` (figure 5.8), chaque expression de chemin est attachée à une expression d'arbre de requête (un noeud `QueryTree`). Le type de l'expression d'arbre représente la structure liée à l'expression de chemin qui peut être récupérée par la méthode `getIType` sur l'interface `QueryTree`. La méthode `getRegExp` permet de récupérer l'expression régulière associée à l'expression de chemin.

L'interface `PathExpression` offre également d'autres fonctionnalités pour la comparaison avec d'autres expressions de chemins. Par exemple, la méthode `isSubPath` permet de tester si le chemin représenté par une expression est un sous-chemin d'une autre expression, ou encore une méthode qui permet d'extraire le sous-chemin commun à deux expressions de chemins. Rappelons qu'un chemin c1 est un sous-chemin de c2 si c2 est un préfixe de c1. Ces méthodes sont utilisées lors de l'optimisation de requêtes afin d'identifier et de factoriser les sous-chemins dans une expression de requêtes, comme dans les techniques d'optimisation de chemin et les expressions régulières dans les requêtes [40].

Lors de la compilation d'une expression de chemin (méthode `validateExpression` de l'interface Expression), le type `IType` de l'expression est le type du chemin ou des chemins représentés par cette expression. Ce type est calculé en appelant la méthode `resolve`, offerte sur l'interface `RegExp` de l'expression régulière associée à cette expression de chemin. Les expressions régulières sont présentées dans la section suivante.

5.5.1 Les expressions régulières

Les expressions régulières sont utilisées pour décrire les noms des attributs et pour exprimer la navigation d'un champ à un autre d'une manière flexible.

Dans le premier cas, elles sont considérées exactement comme dans les expressions régulières sur les chaînes de caractères. Par exemple, l'expression *("A"|"a")dress(s) ?*, regroupe tous les noms : *adress, Adress, adresses, Adresses.* Dans un système d'intégration de données, ce mode d'expression permet de prendre en compte la diversité des modes de nommage. En effet, l'expression *("A"|"a")dress(s) ?* regroupe des noms sémantiquement égaux eu égard à la langue ou

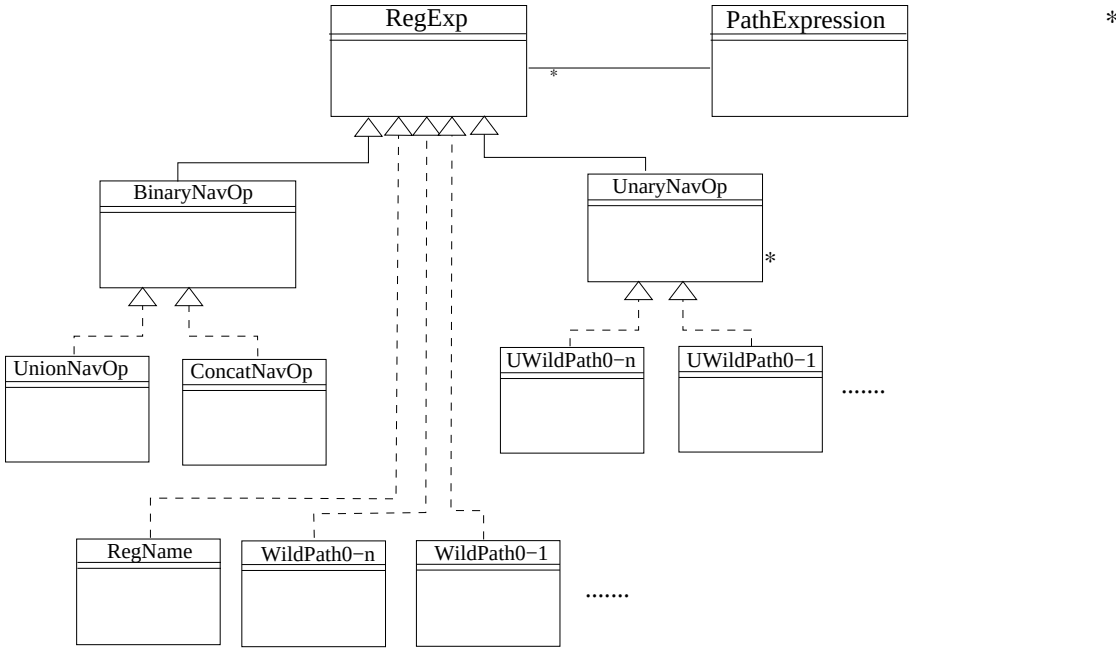


FIG. 5.7 – Diagramme d'expression de chemins

```
public interface PathExpression extends Expression {

    // Récupérer l'expression d'arbre associée
    QueryTree getQueryTree() ;

    // Récupérer l'expression régulière associée
    RegExp getRegExp() ;

    // Tester si cette expression est une sous
    // expression d'une autre expression de chemin
    boolean isSubPath (PathExpression expression) ;
  ...
}
```

FIG. 5.8 – L'interface `PathExpression`

à la syntaxe.

Dans le deuxième cas, les caractères d'expressions régulières (appelés aussi *jokers*) tels que " ?", "*", ,"|", sont considérés comme des opérateurs de navigation. Par exemple, l'expression *"*.last"* permet de sélectionner tous les chemins d'une longueur arbitraire, tels que le nom du champ d'arrivée est *last*. De même, les caractères " ?", " !" désignent respectivement un chemin de zéro ou une seule étape et un chemin d'une seule étape, et ce quels que soient les noms des champs. Un autre type d'opérateurs est la fermeture de *Kleene*. Cet opérateur consiste à appliquer récursivement la même opération n fois. Par exemple, l'expression *(last)** permet de sélectionner tous les chemins *last*, *last.last*, *last.last.last*, etc.

Nous décrivons une expression régulière de chemins comme une composition d'opérateurs de navigation. Elle peut être représentée sous forme d'un arbre où les feuilles représentent des opérateurs de navigation non composés et les noeuds représentent des opérateurs composés (voir figure 5.10) :

- *Les opérateurs non composés* : Ces opérateurs représentent les expressions primitives d'opérateurs de navigation. Parmi ces opérateurs, on distingue :

  - Les deux opérateurs `WildPath0-n` et `WildPath0-1`, où l'opérateur `WildPath0-n` (l'équivalent du caractère "*") représente un chemin arbitraire à n étapes (comme dans "*.last"). Quant à l'opérateur `WildPath0-1` (l'équivalent du caractère " ?"), il représente une expression d'un chemin arbitraire de zéro ou une seule étape (comme dans " ?.last").

  - L'opérateur `RegName` représente une expression régulière sur un nom d'un champ d'une structure. Par exemple, l'expression *("A"|"a")dress(s) ?* est représentée par un opérateur `RegName`.

- *Les opérateurs composés* : Ces types d'opérateurs sont composés d'autres expressions régulières. On distingue les opérateurs de navigation binaires et les opérateurs de navigation unaires :

  - L'interface `BinaryNavOp` spécifie les opérateurs qui composent deux expressions de chemins, comme par exemple l'opérateur ".", représenté par l'interface `ConcatNavOp` et l'opérateur d'union "|", représenté par l'interface `UnionNavOp`. Ces deux opérateurs permettent la concaténation et l'union entre deux opérateurs de navigation, respectivement *(RegExp1"." RegExp2*) et *(RegExp1 "|" RegExp2)*.

  - L'interface `UnaryNavOp` spécifie les opérateurs composés d'un seul opérateur de navigation. Dans cette catégorie, on retrouve les opérateurs d'expressions régulières, comme par exemple les deux opérateurs décrits par les deux interfaces `UWildPathOp0-1` et `UWildPathOp0-1`, qui permettent d'exprimer respectivement les expressions de chemin de type *(RegExp)** et *(RegExp) ?*.

Tous ces opérateurs sont spécifiés par l'interface `RegExp` (voir figure 5.9). Ils offrent une méthode principale `resolve`, qui retrouve tous les chemins qui s'apparient avec cette expression dans une structure donnée `IType`. Dans le cas où l'expression régulière s'apparente avec plusieurs chemins, la méthode `resolve` retourne un type `IUnion` dont chacun des types `IType` qui le compose représente un type de chemins. Ces `IType` appartiennent à la hiérarchie des types de la structure et représentent le type de l'élément pointé par le chemin.

```
public interface RegExp{

    // Récupère tous les chemins qui s'apparient avec l'expression
    IType resolve(IType structure) ;
  ...
}
```

FIG. 5.9 – L'interface `RegExp`

Le fait d'exprimer les expressions de chemins sous forme d'une composition d'opérateurs de navigation permet d'avoir une évaluation récursive de la méthode `resolve`. L'implantation de la méthode `resolve` dépend de la sémantique de chaque opérateur. Par exemple, l'opérateur de concaténation `ConcatNavOp` fait d'abors appel à la méthode `resolve` de gauche pour utiliser le résultat retourné par celle-ci comme paramètre d'entrée pour appeler la méthode `resolve` de l'opérateur de droite. L'opérateur `UnionNavOp` fait appel aux deux opérateurs de gauche et de droite pour faire l'union des chemins résultants. Notons que le fait d'avoir des références dans notre modèle de données peut générer des cycles de chemins. Ce cas doit être pris en compte dans l'implantation de la méthode `resolve`.

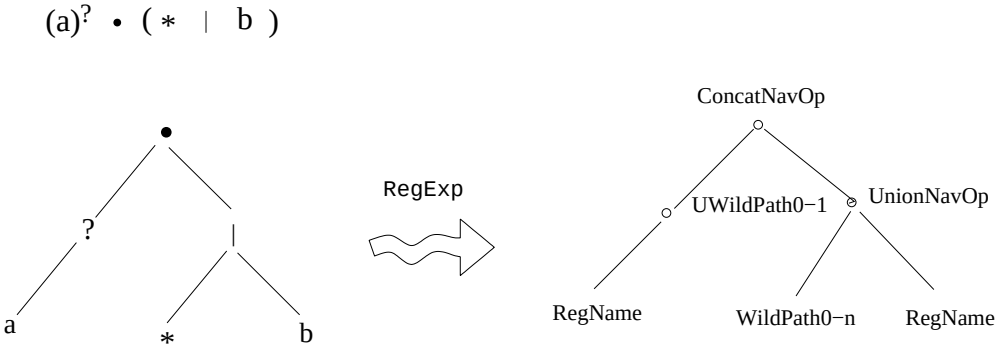


FIG. 5.10 – Exemple de représentation d'une expression régulière de chemin

La représentation arborescente d'une expression régulière peut aussi être vue comme une représentation de l'automate correspondant à l'expression d'opérateurs associée et une expression de chemin. L'avantage de notre représentation est son extensibilité, dans le sens où l'on peut ajouter de nouveaux opérateurs de navigation. On peut par exemple ajouter des opérateurs pour naviguer du fils au père dans la hiérarchie des types, ou alors des opérateurs qui permettent une navigation conditionnelle sur les types (sélectionner seulement les attributs d'un type bien défini), comme dans le langage XPath [181]. L'ajout d'un type d'opérateur revient principalement à implanter la méthode `resolve` de l'interface `RegExp`.

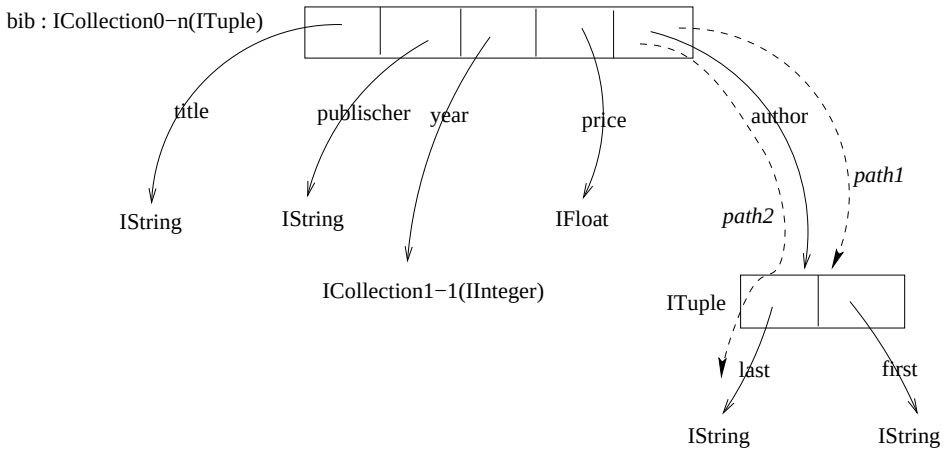


FIG. 5.11 – Exemple de navigation

Exemple :

Dans la figure 5.11, nous reprenons l'exemple de la section 4.5 qui représente le type de la vue importée d'une source de données semi-structurées contenant des livres et des auteurs. Soient deux expressions de chemins *author* et **.last*. Dans le fragment de code ci-dessous, ces deux expressions sont représentées par les deux variables `pathExp1` et `PathExp2`.

```
IType book := new ICollection0-nImpl(_); //
RegExp pathExp1 := new RegPathImp("author");
RegExp pathExp2 := new ConcatPathOpImp(new WildPath0-nImp,
                                       new RegNameImp("last"));
IType path1 := pathExp1.resolve (book);
IType path2 := pathExp2.resolve (book);
```

La variable `path1`, qui contient le résultat de l'opération de résolution de l'expression de chemin *author*, représente un seul chemin dont le type final de navigation est `ITuple`, composé des deux attributs *last* et *first*. La variable `path2` contient un seul chemin, *author.last*, dont le type final de navigation est `IString`, représentant les noms des auteurs.

5.6 Expressions d'arbres de requêtes

Les requêtes sont des expressions de type `QueryTree` qui sont représentées sous forme d'arbres d'opérateurs algébriques. Comme le montre la figure 5.12, un `QueryTree` peut désigner un noeud de l'arbre, représenté par l'interface `QueryNode`, où une feuille est représentée par l'interface `QueryLeaf`. Les noeuds sont des expressions composites composées d'autres expressions d'arbres. Elles représentent les opérateurs algébriques tels que la sélection, la projection, etc. Les feuilles de l'arbre représentent des sous-requêtes qui peuvent être primitives ou non primitives selon qu'elles représentent les arbres des vues primitives ou non.

Dans une expression d'arbre `QueryTree`, le type `IType` de la racine de l'arbre représente le type du résultat final de la requête. Le type des noeuds représente le type des résultats intermédiaires. Selon la définition de l'opérateur algébrique, qui est représenté par un `QueryNode`, le type peut être inféré à partir des types des noeuds fils si l'on utilise les définitions générales des

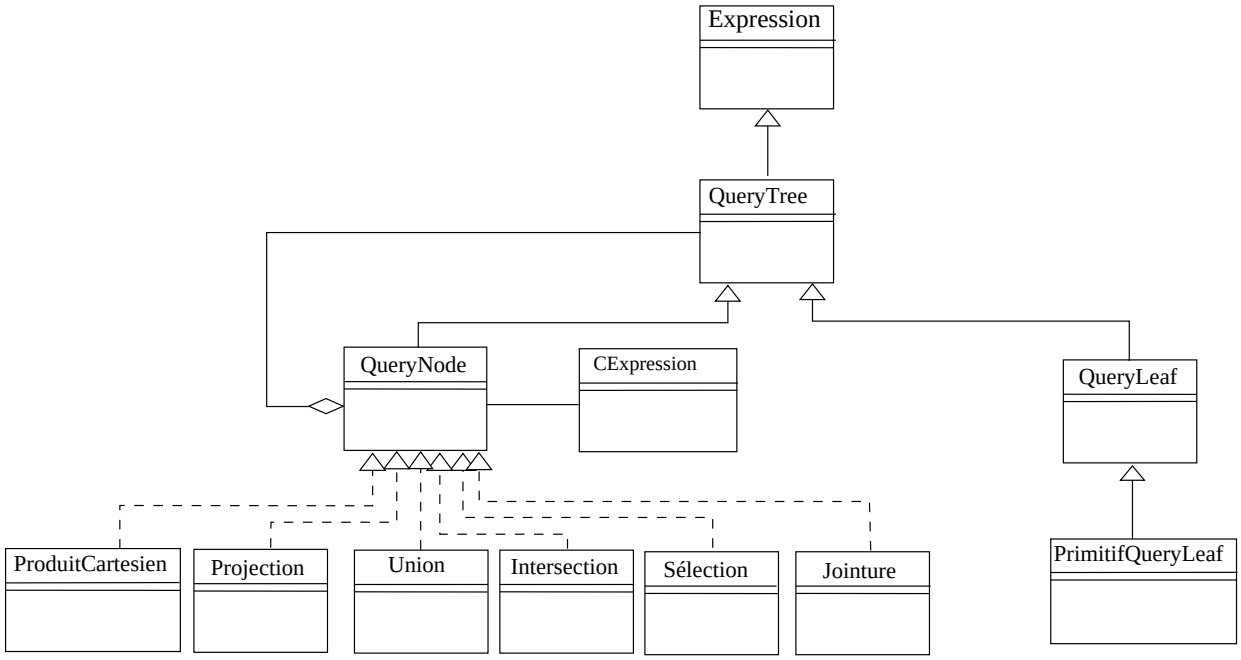


FIG. 5.12 – Diagramme d'expressions d'arbre de requêtes

```
public interface QueryTree extends Expression{

    // Récupérer les propriétés de l'opérateur
    OperationProperties getOperationProperties();

    // Récupérer le nom associé à l'opérateur
    String getName();


}
```

FIG. 5.13 – L'interface `QueryTree`

opérateurs présentés dans la section 5.2.2. Dans le cas d'une feuille primitive, ce type représente la structure des données exportées des sources de données et aussi le type des vues primitives.

```
public interface OperationProperties{

    // Récupérer le domaine de données de l'opérateur
    ImportExport getDataDomain() ;

    // Mettre à jour le domaine de l'opérateur
    void setDataDomain(ImportExport dataDomain) ;

    // Récupérer le comportement de l'opérateur face aux
    // valeurs dupliquées
    boolean isDistinct() ;

    // Mettre à jour le comportement de l'opérateur face
    // valeurs dupliquées
    void setDistinct(boolean distinct) ;

    // Récupérer la liste des attributs qui guident l'ordre"OrderBy"
    PathExpression[] getOrderBy() ;

    // Mettre à jour la liste des attributs qui guident l'ordre
    //"OrderBy"
    void setOrderBy(PathExpression[] fields) ;

    // Fixer le mode d'ordonnancement
    void setOrderBy(int upOrDown) ;
  ..

}
```

FIG. 5.14 – L'interface `OperationProperties`

Dans la spécification de l'interface `QueryTree` (figure 5.13), à chaque expression d'arbre sont associés un nom et une liste de propriétés. Le nom peut être utilisé pour nommer élément lors de la création de nouveaux types de données, ou encore comme alias lors de l'écriture de l'expression de requêtes dans le langage cible d'une source dans les domaines de données primitifs.

Les propriétés spécifiées par l'interface `OperationProperties` (voir figure 5.14) sont résumées dans le tableau de la figure 5.15. Ces propriétés sont :

- *Lieu d'exécution* : Cette propriété désigne le domaine de données (méthode `getDataDomain`) où sera exécuté l'opérateur. Lors de la déclaration de la requête, le lieu d'exécution de chaque `QueryTree` est par défaut le domaine de données courant. L'optimiseur de requêtes peut ensuite décider de changer la répartition de l'arbre de requête entre plusieurs domaines de données, en mettant à jour cette propriété (par la méthode `setDataDomain`). Le lieu d'exécution est représenté par une interface d'exportation `ImportExport`, qui permet d'interagir avec le domaine en question.

- *Valeurs dupliquées* : Cette propriété décrit le comportement de l'opérateur vis à vis des valeurs dupliquées. Elle stipule si cet opérateur doit éliminer ou non les valeurs dupliquées. Le rôle de cette propriété est double : (i) elle permet d'exprimer la clause "*distinct*", que l'on retrouve dans les langages de requête dans le noeud racine de l'arbre; (ii) elle est exploitée par l'optimiseur de requête en sélectionnant les noeuds de l'arbre qui doivent filtrer les valeurs dupliquées. Dans la spécification de l'interface `OperationProperties`, la méthode `getDistinct` retourne vrai si l'opérateur filtre les valeurs dupliquées et faux sinon. La mise à jour de cette propriété est effectuée par la méthode `setDistinct`.

- *Ordre des données* : Cette propriété décrit le comportement de l'opérateur vis à vis de l'ordre des valeurs. Aussi, cette propriété a deux rôles. D'un coté, elle permet de prendre en compte la clause "*orderBy*" que l'on retrouve dans les langages de requêtes. Cette propriété concerne donc l'opérateur racine de l'arbre. D'un autre coté, elle joue un rôle important dans l'optimisation des requêtes, notamment pour les algorithmes d'évaluation qui se basent sur l'ordre des valeurs (voir chapitre suivant). La méthode `setOrderBy` permet de spécifier l'ordre dans lequel les valeurs doivent être ordonnées. Cet ordre est décrit par une liste d'attributs où chacun est désigné par une expression de chemin (`PathExpression`). La méthode `setOrderBy` permet de spécifier si l'ordre est effectué selon les valeurs croissantes ou décroissantes.

Propriétés	Description
Lieu d'exécution	Désigne le lieu d'éxecution de l'opérateur
Valeurs dupliquées	Décrit le comportement de l'opérateur vis à vis des valeurs dupliquées
Ordre des valeurs	Décrit le comportement de l'opérateur vis à vis de l'ordre des valeurs

FIG. 5.15 – Propriétés des opérateurs algébriques

5.6.1 Les noeuds de l'arbre

Selon l'opération algébrique représentée par un noeud et comme l'illustre l'interface `QueryNode` (voir figure 5.16), un noeud est caractérisé par :

- Une expression de calcul représentant un prédicat. Ceci concerne typiquement les opérations de jointure et de sélection. Les deux méthodes `setPredicate` et `getPredicate` permettent respectivement d'assigner ou de récupérer le prédicat associé. Un prédicat est représenté par une expression de calcul `CExpression` de type `IBoolean`, composée d'opérandes `PathOperand` qui pointent sur les attributs du type `IType` des `QueryTree` fils. L'association d'un prédicat à un `QueryNode` entraîne l'ajout les `QueryTree` fils contenus dans les expressions de chemin des prédicats aux fils du `QueryNode` courant.

- Une composition de `QueryTree` qui représentent les sous-arbres de ce `QueryNode`. La méthode `getQueryTreeChilds` retourne la liste de ces fils.

L'interface `QueryNode` offre aussi des méthodes génériques de construction et de mise à jour graduelle de l'arbre de requêtes. Ces opérations permettent, d'un coté, de faciliter l'implantation des opérateurs algébriques, et d'un autre coté, de simplifier l'implémentation des règles de réécriture pour l'optimisation des expressions de requêtes. Le principe est de construire et de mettre à jour le type `IType` d'un opérateur algébrique `QueryNode` à partir d'autres `QueryTree`

```
public interface QueryNode extends QueryTree {

    // Récupérer l'expression du prédicat
    CExpression getPredicate() ;

    // Assigner un prédicat à cette opérateur
    void setPredicate(CExpression predicate)
                                    throws IException;

    // Récupérer la liste des fils
    QueryTree[] getQueryTreeChilds() ;

    // Ajouter un attribut au type du noeud à partir des attributs
    // des noeuds fils. La valeur de cet attribut est le même que
    // celui des attributs des noeuds fils.
    void addPropagatedField(String name,
                            PathExpression[] anc)
                                        throws IException ;

    // Ajouter Un attribut au type du noeud. La valeur de cet
    // attribut est définie par une expression de calcul sur
    // les attributs fils.
    void addCalculatedField(String name,
            CExpression exp) throws IException ;

    // Mettre à jour un attribut propagé
    void updatePropagatedField(String name,
                            PathExpression[] anc)
                                        throws IException ;

    // Mettre à jour un attribut calculé
    void updateCalculatedField(String name,
                            CExpression exp)
                                        throws IException ;
 ...
}
```

FIG. 5.16 – L'interface `QueryNode`

fils. Deux sortes d'attributs sont utilisés : les attributs propagés et les attributs calculés :

- *Les attributs propagés :* Ces attributs sont directement propagés à partir des attributs des noeud fils. La méthode `addPropagatedField` sur un opérateur `QueryNode` permet d'ajouter un nouvel attribut au type `ITuple` qui compose le type de cet opérateur. Ce type d'attribut est créé à partir :

  - D'une liste d'un ou plusieurs attributs fils du noeud courant, où chacun est désigné par une expression de chemin. Le cas d'une liste de plusieurs attributs implique que ces attributs aient la même sémantique, comme dans l'opération de jointure. Notons que l'ajout de ce type d'attribut entraîne l'ajout des `QueryTree` fils qui composent ces expressions de chemins.

  - D'un nom qui peut être l'un des noms des attributs des noeuds fils ou un nouveau nom. L'utilisation d'un nouveau nom est particulièrement utile pour pallier les conflits de nommage entre les noeuds fils. Par exemple, on peut unifier le nom de plusieurs attributs ayant le même sens en un seul. Ce cas peut se présenter, typiquement, lors de la construction de l'arbre de requête d'une vue exportée dans un domaine de données non primitif à partir d'autres vues importées d'autres domaines de données présentant des conflits de nommage.

  Les attributs propagés sont utilisés dans l'implémentation de la majorité des opérateurs algébriques. La méthode `addPropagatedField` peut alors être utilisée pour l'implantation de ces opérateurs. Par exemple, l'opération de projection consiste simplement à faire des appels d'ajout d'attributs projetés. La mise à jour de ces attributs est effectuée par la méthode `updatePropagatedField` qui met à jour systématiquement la liste des fils du noeud courant.

- *Les attributs calculés :* ce type d'attributs est calculé à partir d'autres attributs des noeuds fils. La méthode `addCalculatedField` permet d'ajouter ce type d'attribut au type `IType` d'un `QueryNode`. Ces attributs sont créés à partir :

  - D'une expression de calcul `CExpression`. Cette expression de calcul est composée d'opérandes d'expressions de chemins, `PathExpression`, qui pointent sur des attributs des noeuds fils. Lors de l'ajout de cet attribut, les noeuds fils correspondants sont alors automatiquement ajoutés à la liste des noeuds fils du noeud courant.

  - D'un nom, qui peut être l'un des noms des attributs figurant dans l'expression de calcul, ou un nouveau nom. Comme dans les attributs propagés, ceci peut aussi permettre de pallier le problème de conflits de nommage.

  Les attributs calculés peuvent être utilisés par les opérateurs d'agrégation et les opérateurs de calcul propres à l'intégration, comme les opérateurs de conversion d'unités de mesures. La mise à jour des attributs calculés est effectuée par la méthode `updateCalculatedField`, qui est susceptible de transformer l'arbre de requête.

Exemple :

La figure 5.17 illustre un exemple de construction d'un `QueryNode` QT3 à partir de deux autres `QueryTree` QT1 et QT2, indépendamment du type des opérations algébriques représentées par ces `QueryTree`. L'attribut *nom* du noeud QT3 est un attribut propagé à partir de l'attribut *name* du noeud QT1. On remarque que cet attribut porte un nouveau nom. L'attribut *prix* du noeud QT3 est calculé à partir de l'attribut *price* du noeud QT2. La valeur de cet attribut est la conversion de la valeur de l'attribut *price* du dollar vers l'euro, qui est l'unité de mesure de l'attribut *prix*. On remarque aussi que l'attribut *price* est renommé en *prix*. Tout cela est résumé par le fragment de code ci-dessous :

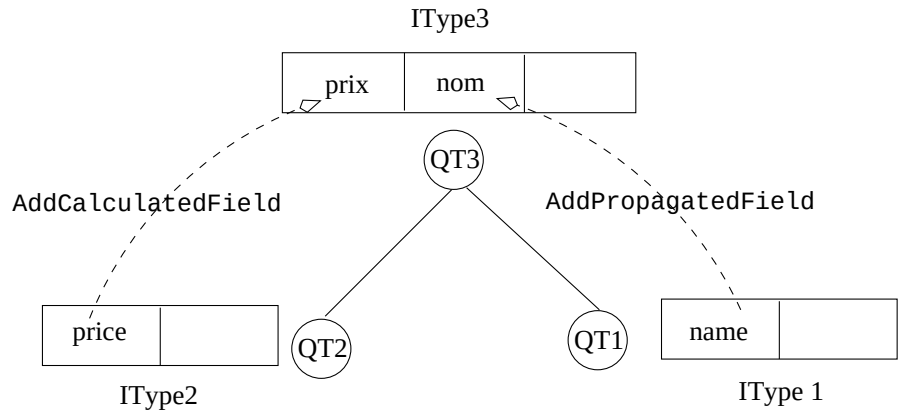


FIG. 5.17 – Exemple d'attributs projetés et d'attributs calculés

```
// Création de QT3
QueryNode QT3 = new QueryNodeImp();

// Création d'expressions de chemins
PathExpression field2 = new PathExpression (QT2, new RegNameImp("price"));
PathExpression field1 = new PathExpression(QT1, new RegNameImp("name"));

Création de l'expression de calcul
CExpression convert = new DollarToEuroOperator(field2);

Ajout de l'attribut calculé et de l'attribut propagé
QT3.addCalculatedField ("prix", convert);
QT3.addPropagatedField ("nom", field1);
```

Un exemple plus complet de la création d'une expression de requête est présenté dans le chapitre validation.

5.6.2 Les feuilles de l'arbre

Les feuilles d'un arbre représentent des sous-requêtes qui désignent des vues. La méthode `getIView` de l'interface `QueryLeaf` (voir figure 5.18) permet de récupérer la vue encapsulée par la feuille. Cette vue peut être primitive ou non primitive. Dans le cas non primitif, la feuille peut représenter une vue importée ou une vue exportée d'un domaine de données non primitif : elle représente une vue exportée dans l'expression de requête pour interroger un domaine de

données, et représente une vue importée dans l'expression de requête d'une vue exportée dans un domaine de données non primitif.

```
public interface QueryLeaf extends QueryTree {

    // Récupérer la vue importée que contient cette feuille
    IView getIView() ;
  ...
}
```

FIG. 5.18 – L'interface `QueryLeaf`

```
public interface PrimtiveQueryLeaf extends QueryLeaf {

    // Récupérer les métadonnées sur le source de données sous-jacente
    DataSourceMetaData getDataSourceMetaData() ;

    // Récupérer la requête sur la source de données
     String getDataSourceQuery() ;
  ...
}
```

FIG. 5.19 – L'interface `PrimtifQueryLeaf`

Dans le cas des vues primitives, les feuilles sont représentées par l'interface `PrimitiveQueryLeaf` (voir figure 5.19). Cette feuille encapsule une expression de requête qui porte directement sur une source de données dans un domaine de données primitif. Ce type de feuilles est aussi considéré comme un opérateur d'accès aux sources. Les deux méthodes `getDataSourceMetaData` et `getDataSourceQuery` permettent respectivement de décrire la source de données et de récupérer la requête représentée par cette feuille dans le langage de requête cible de la source.

Selon le mode d'accès aux sources et le type de la source de données, on peut avoir plusieurs types de feuilles (plusieurs spécialisations de l'interface `PrimitiveQueryLeaf`) : des feuilles créées directement à partir d'une requête écrite dans le langage de la source sous-jacente, ou des feuilles sous forme d'opérateurs d'accès sur les entités du schéma de la source de données sous-jacente. Dans le dernier cas, ces opérateurs sont internes aux domaines de données primitifs et dépendent du type de la source, comme par exemple :

- L'opérateur qui représente une table dans le cas d'une base de données relationnelle.

- L'opérateur *extent(Classe)* qui permet d'accéder à l'extension d'une classe d'objets dans les bases de données objets.

- Pour les sources de données semi-structurées, cet opérateur désigne un noeud ou plusieurs noeuds du graphe de données. Il n'existe pas de convention sur ce type d'opérateur. Cependant, il peut s'agir de l'opérateur algébrique *bind* défini dans l'algèbre XML YAL [41], ou l'opérateur *XScan* [189, 188], ou encore l'opérateur *XSource* [122]. Tous ces opérateurs,

même ayant des noms différents, portent exactement la même sémantique. Ils permettent de lier plusieurs variables d'une requête semi-structurée aux noeuds correspondants de la source de données. Ces noeuds sont désignés par des expressions régulières de chemins décrites dans XPath. Le type du résultat de cet opérateur est représenté sous forme d'un tuple où les noms des attributs représentent les variables, et le type des attributs représente le type des noeuds liés à la variable. Cet opérateur peut être exprimé par notre canevas en utilisant les expressions de chemins réguliers et le type tuple représente le résultat de l'opérateur. Il faut souligner aussi que ce type d'opérateur permet de convertir les données semi-structurées (un graphe d'objet) en une collection de tuples, qui justement font partie de notre modèle de données.

5.7 Conclusion

Dans ce chapitre, nous avons présenté le canevas d'expression de requêtes. Il fournit un formalisme de représentation de requêtes, qui permet de décrire d'une manière uniforme des requêtes d'interrogation du contenu de domaines de données, et les requêtes de définitions des vues intégrées.

Les requêtes sont exprimées indépendamment d'un langage de requêtes et d'une algèbre particuliers. On distingue trois types d'expressions : les expressions de calcul, les expressions de chemins et les expressions d'arbre de requêtes. Une requête est alors décrite par une composition d'expressions, comme dans les langages fonctionnels. En se basant sur la flexibilité du typage du modèle de données, le canevas permet une compilation statique des expressions. Ceci permet de vérifier la concordance des types d'expressions composée d'une requête, et aussi d'inférer statiquement les types des résultats des différentes expressions.

Chapitre 6

Canevas d'optimisation de requêtes

" L'optimisation prématurée est la source de tous les maux ".

Dijkstra

6.1 Introduction

Dans ce chapitre, nous abordons la problématique d'optimisation de requêtes dans les architectures d'intégration de données à base de domaines. Dans le chapitre 3, nous avons montré que ces architectures, qui sont construites par la composition de domaines de données, suivent le principe de l'architecture de médiation. Par conséquent, tous les aspects et les techniques d'optimisation dans ces systèmes, qu'ils soient centralisés ou répartis, peuvent être appliqués dans notre contexte.

Notre approche consiste à aborder le problème du point de vue architectural : nous proposons un canevas adaptable pour l'optimisation de requêtes. Cette adaptabilité couvre les points suivants :

- *Adaptabilité aux stratégies de recherche.* Le canevas permet de personnaliser et de construire différentes stratégies de recherche de plans, aussi bien de type *bottom-up (stratégies génératives)* ou de type *top-down (stratégies transformatives).* Ceci concerne aussi bien les stratégies de recherche locales au sein d'un seul domaine de données que les stratégies collaboratives entre les domaines qu'on appelle stratégies composites.

- *Prise en compte des capacités des sources.* La composition des domaines concerne aussi les domaines de données primitifs représentant des sources avec des capacités d'évaluation limitées. L'architecture du canevas d'optimisation prend en compte ce cas, en permettant aux domaines non primitifs de collaborer avec ces domaines primitifs, pendant le processus de recherche, en vue de négocier un plan suivant les contraintes d'évaluation.

- *Indépendance vis à vis de modèles de coûts.* Le canevas d'optimisation considère le modèle de coût comme un service non fonctionnel. C'est-à-dire que les stratégies de recherche sont indépendantes d'un modèle de coût particulier. Le canevas suppose donc un composant

qui offre des fonctions de calcul de coûts utilisées par les stratégies de recherche.

Nous commençons dans ce chapitre par énumérer les différentes étapes du traitement de requêtes dans un système d'intégration à base de domaines à travers la section 6.2. Les principes de base du canevas d'optimisation font l'objet de la section 6.3. Les sections 6.4 et 6.5 montrent comment représenter les plans d'évaluation ainsi que les règles. La spécification et le fonctionnement des principaux composants du canevas d'optimisation sont présentés dans la section 6.6. Dans la section 6.7, nous analysons l'adaptabilité du canevas. Nous montrons alors que le canevas supporte les deux stratégies de recherches types proposées dans les systèmes de médiation GARLIC et DISCO.

6.2 Phases de traitement

Rappelons que l'interfaçage des applications d'intégration avec les domaines de données d'un système d'intégration peut toujours être représenté par la configuration suivante : une application d'intégration avec un seul domaine de données (voir section 3.2.3). Dans ce qui suit, nous supposons donc que les applications d'intégration projettent leurs données sur un seul domaine de données.

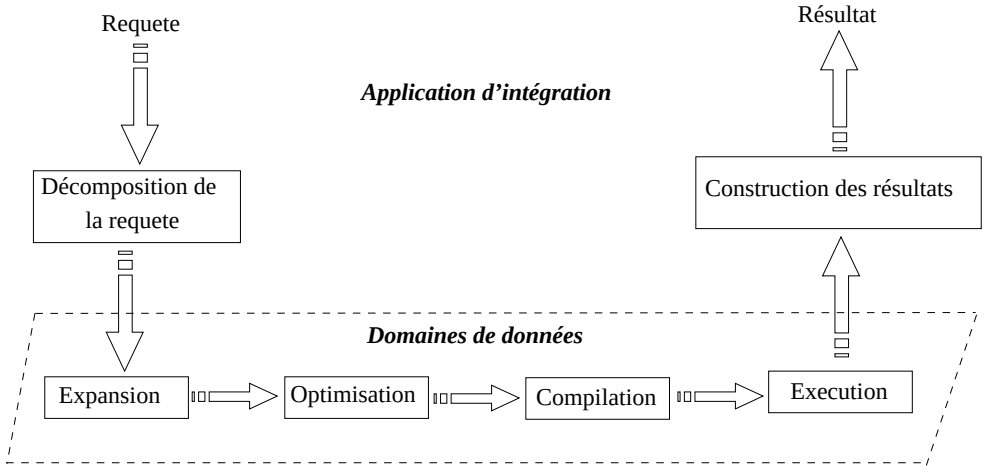


FIG. 6.1 – Phases de traitement de requêtes : vue générale

La figure 6.1 illustre le cycle de vie du traitement d'une requête entre les applications d'intégration et les domaines de données. Les différentes phases sont : la décomposition, l'expansion, l'optimisation, la compilation, l'exécution puis la construction du résultat. Les phases de décomposition et de construction des résultats sont prises en charge par les applications. Elles sont effectuées selon les règles de mapping entre le schéma global et les vues exportées des domaines. Le résultat de la phase de décomposition est donc un `QueryTree` dont les feuilles représentent les vues exportées par le domaine de données.

Dans ce qui suit, nous nous intéressons seulement aux étapes de traitement effectuées par les domaines de données, à savoir l'expansion des vues, l'optimisation, la compilation et l'exécution. Sur la figure 6.2, nous reprenons l'architecture d'un domaine de données qui montre ses composants. Nous illustrons les phases de traitement par un scénario global (ou un algorithme général)

qui montre les différentes interactions entre les composants d'un domaine. Ce scénario est décrit par le diagramme de la figure 6.3. Le composant `Domain Manager` intercepte une requête `QueryTree` introduite par l'interface `ImportExport` du domaine de donnée, puis coordonne les différentes tâches de traitement avec les composants correspondants du domaine.

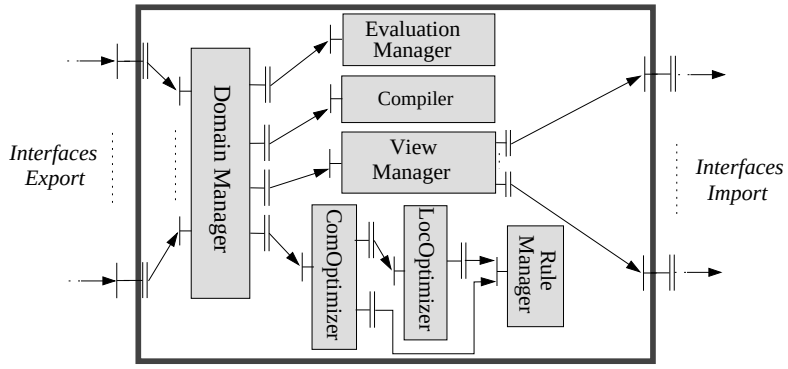


FIG. 6.2 – Architecture d'un domaine de données

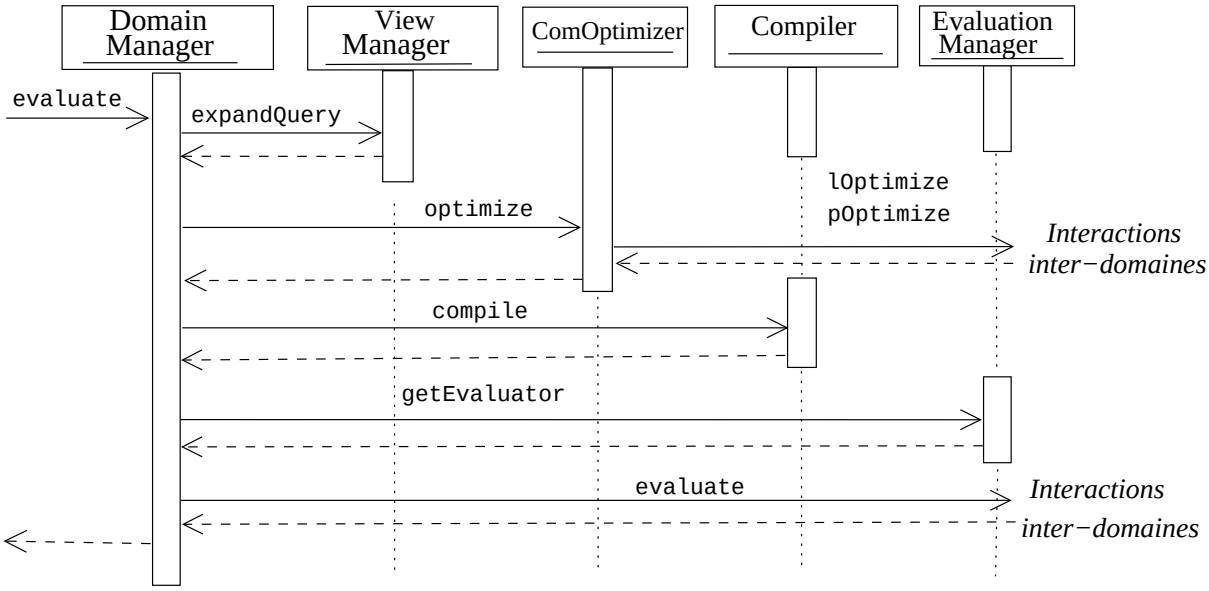


FIG. 6.3 – Diagramme de traitement de requêtes

6.2.1 Réecriture des vues

L'objectif de cette phase est d'éliminer les indirections entre les domaines de données dues à la composition des vues importées et exportées dans les expressions d'arbres de requêtes. Ceci est l'équivalent de l'étape de la réécriture des requêtes en fonction des vues dans les approches GAV (appelé aussi dépliage où expansion des vues comme présenté dans la section 2.3.1). On

distingue deux types de dépliage :

1. ***Dépliage des vues dans un domaine.*** Les feuilles de l'arbre de requête représentent les vues exportées par ce domaine. Cet arbre est réécrit en fonction des vues importées par le domaine, en remplaçant chaque feuille de l'arbre de requête par l'arbre représentant la définition des vues exportées. Le résultat de cette étape est une expression d'arbre de requêtes où les feuilles représentent des feuilles primitives, dans le cas d'un domaine de données primitif, ou des vues importées dans le cas d'un domaine de données non primitif.

2. ***Dépliage des vues entre domaines.*** Cette étape concerne les domaines de données non primitifs. Elle prend en entrée l'arbre de requête résultat de la phase précédente pour remplacer les feuilles qui représentent les vues importées d'un autre domaine par son expression d'arbre de requêtes. Cette phase n'est pas toujours possible et dépend du degré d'autonomie des domaines de données exportant ces vues. Deux cas sont envisageables. Dans le premier cas, les domaines ne donnent pas accès aux informations sur les vues qu'ils exportent (ils sont considérés comme des "boîtes noires"). Les feuilles de l'arbre sont alors perçues comme des feuilles primitives et leur traitement est entièrement délégué à leurs domaines respectifs. Dans le deuxième cas, les domaines de données exportant sont des "boîtes blanches", dans le sens où ils permettent l'accès aux informations sur les vues qu'ils exportent. Dans ce cas, l'arbre de requête est étendu en remplaçant les feuilles de l'arbre des vues importées depuis des domaines "boîtes blanches" par leurs arbres correspondants.

Si l'on se rapporte à la figure 6.3, le composant `Domain Manager` interagit avec le le composant `View Manager` pour étendre la requête à travers la méthode `expandQuery`. Le `View Manager` utilise les informations qu'il détient sur les définitions des vues importées et exportées pour effectuer cette expansion. En prenant en compte le degré d'autonomie des domaines de données concernés par la requête (domaines "boîtes blanches" et domaines "boîtes noires"), il retourne une expression d'arbre `(QueryTree)`expansée où la propriété des opérateurs `QueryNode` " *lieu d'exécution*" (voir section 5.6) est par défaut le domaine de données courant. Les feuilles de l'arbre représentent des vues primitives (des feuilles de type `PQueryLeaf)`des domaines de données concernés par le traitement de la requête.

Exemple :

La figure 6.4 illustre un exemple d'expansion de vues dans une configuration à trois domaines de données D1, D2 et D3 dont D1 est primitif et D2 et D3 non primitifs. Les dépendances entre ces trois domaines sont : D3 importe une vue *v1'* de D2 et une vue *v2* de D1, et le domaine D2 importe la vue *v1* de D1. La requête QT dans le domaine de données D3, qui est une jointure entre *v1'* et *v2*, est réécrite par dépliage de la vue *v1'*. Le domaine de données D3 interagit directement avec le domaine de données primitif D1 (partie (b) de la figure 6.4).

Les deux types de dépliage de vues sus-citées sont effectuées récursivement d'un domaine de données à un autre jusqu'à atteindre les domaines de données primitifs ou des domaines de données de type "boîte noire". On remarque que, si tous les domaines de données non primitifs sont "des boîtes blanches", l'expansion des vues est totale et toutes les feuilles des arbres de requêtes résultant sont des feuilles primitives. Le domaine de données non primitif initiateur de la requête interagit alors directement avec les domaines de données primitifs.

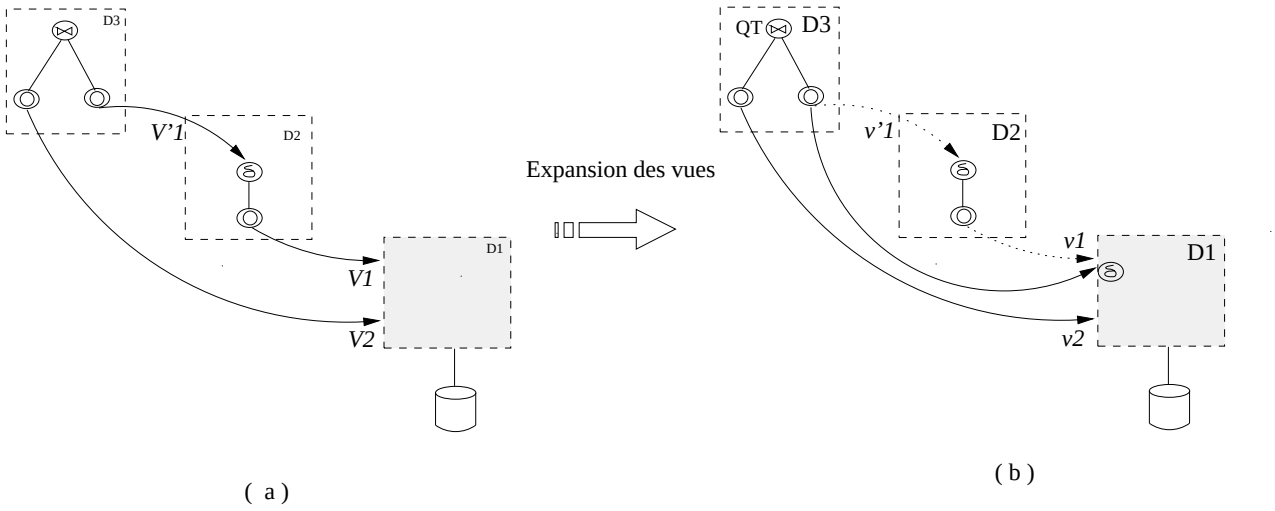


FIG. 6.4 – Exemple d'expansion des vues

6.2.2 Optimisation

Cettte phase du traitement a pour but d'optimiser les expressions de l'arbre de requêtes déplié résultat de la phase précédente. Nous décomposons cette phase de la manière suivante.

- ***Préparation de l'optimisation.*** Cette phase précède la phase de recherche de plan par l'optimiseur. En plus de détecter d'éventuelles incohérences ou contradictions dans les expressions de requêtes, cette phase permet de normaliser et de simplifier les expressions.

  - *La normalisation des expressions* concerne les expressions de chemins `pathExpression` et les expressions d'arbre `QueryTree`. Pour les premières, la normalisation consiste à résoudre des expressions régulières et à les remplacer par les chemins correspondants (en appelant la méthode `resolve` des `RegExp`). Pendant cette étape, des règles de simplification des expressions régulières basées sur les propriétés des opérateurs de navigation sont appliquées, comme par exemple :

    $* \,.\, * = * \,;\; * \,.\, ? = * \,;\; * \,.\, + = + \,;\; + \,.\, +$; etc.
    $(path) * \,.\, (path)* = (path)* \,;\; (path) * \,.\, (path)? = (path)* \,;\; (path) * \,.\, (path)+ = (path)+;$
    $(path) + \,.\, (path)* = (path)+$; etc

    Le processus de normalisation des expressions d'arbres consiste à transformer des `QueryTree` vers d'autres `QueryTree` normalisés appelés arbres de requête logique. Notons que cette étape n'est pas obligatoire dans le cas où le canevas d'expression de requêtes est utilisé avec des opérateurs algébriques classiques, i.e lorsque les expressions de requêtes sont déjà normalisées. Le résultat de la phase de normalisation est une union de plusieurs arbres logiques.

  - *La simplification* correspond à la phase de simplification des expressions. Elle consiste à éliminer notamment les opérateurs inutiles ou redondants ou à grouper certains opérateurs.

- *Recherche du plan :* Cette étape correspond aux stratégies de recherche dans l'optimisation de requêtes. L'objectif est de trouver un plan optimal d'une expression d'un arbre de requête. Ceci consiste à rechercher le meilleur ordonnancement (le moins coûteux) des opérateurs de l'expression de requêtes, à la fois à l'intérieur d'un domaine de données et entre les domaines de données. Cette recherche prend en compte les propriétés liées à l'optimisation classique, notamment les coûts des formules, les statistiques ainsi que les capacités d'évaluation des sources.

Comme montré dans la figure 6.3, le `Domain Manager` fait appel au composant `ComOptimizer` à travers la méthode `optimize` pour effectuer cette tâche d'optimisation. La préparation de l'optimisation est effectuée à l'initialisation de la méthode `optimize` qui déclenche le processus de recherche d'un plan d'évaluation. Cette recherche est alors effectuée avec les composants `LocOptimizer` et `Rule Manager` du domaine, en collaboration avec les autres domaines de données concernés. Nous détaillons à partir de la section 6.3 le canevas d'optimisation qui permet de prendre en compte tous ces aspects de l'optimisation.

6.2.3 Compilation

La phase de compilation permet d'optimiser le code du plan résultat de la phase précédente. Elle vise essentiellement à éliminer les vérifications de type dynamique, comme dans [109]. En effet, si l'approche canevas permet de construire des expressions de calcul extensible, leur évaluation peut être coûteuse. Nous avons vu dans le chapitre précédent que chaque opérateur et chaque opérande de l'arbre d'expression de calcul est représenté par un objet de type `Expression`. L'évaluation naïve de telles expressions se fait par appels récursifs de la méthode `evaluate` sur chacun des opérateurs et opérandes de l'arbre. A chaque appel, la bonne méthode `getXXX` du type `IType` des opérateurs ou des opérandes de l'arbre est choisie suivant les types inférés. Ceci nécessite un test sur le type des expressions opérandes et/ou opérateurs à chaque itération.

La compilation permet de passer d'une représentation arborescente sous forme d'objets à une simple séquence d'instructions. Elle est effectuée par le composant `Compiler` d'un domaine de données à travers la méthode `compile` qui retourne le même plan d'évaluation mais où le code des expressions de calculs est optimisé. Comme illustré par la figure 6.5, pour chaque expression de calcul d'une expression de requête, la compilation est effectuée selon les étapes suivantes :

- L'expression est vérifiée et les types des résultats intermédiaires sont inférés. Ce travail est effectué par la méthode `validateExpression` de l'interface `Expression` elle-même (voir section 5.2).

- A partir de l'expression arborescente de l'expression de calcul et à partir des types, une classe de type `CExpression` est créée. Cette classe représente tous les objets de l'arbre d'expression de calcul. Des outils de transformation et de génération de code sont utilisés pour générer le code correspondant aux méthodes de la nouvelle classe.

- Enfin, la nouvelle classe est chargée dans l'environnement d'exécution. Cette nouvelle expression est affectée au noeud correspondant du plan d'évaluation.

Ce processus est répété pour chaque expression de calcul présente dans l'expression du plan. Cette phase est effectuée par le composant `Compiler` du domaine de données, qui offre une seule

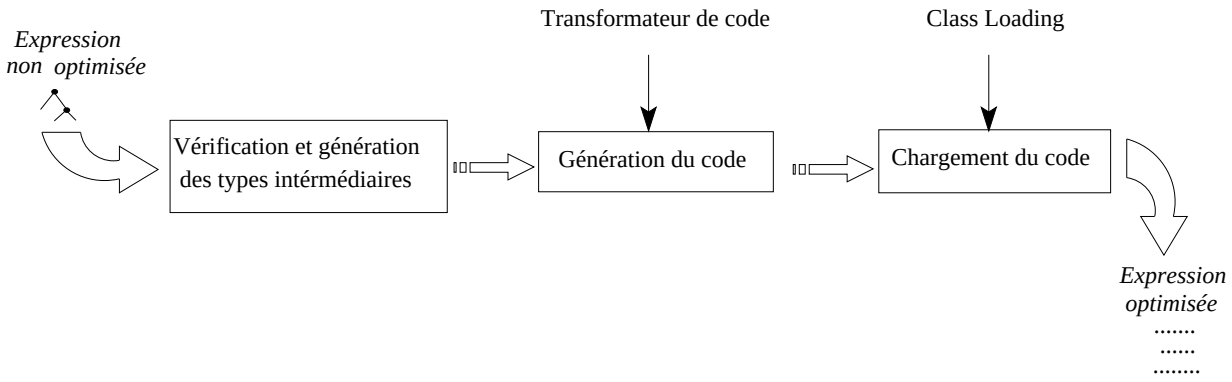


FIG. 6.5 – Compilation des expressions de calcul

méthode `compile` prenant en entrée une expression d'arbre - `EvaluableQuery` - et retroune une expression d'arbre `(EvaluableQuery)`. Notons que cette étape n'est pas obligatoire et ne change en rien l'ordonnancement des opérateurs du plan d'évaluation. Notons aussi que cette phase est possible parce que notre système de types permet une vérification statique des types.

La mise en oeuvre de cette phase est inhérente à l'environnement d'exécution sous-jacent. Par exemple, dans le cas du langage de programmation Java, la deuxième étape du processus d'optimisation du code peut être effectuée en utilisant des outils de manipulation de byte code tels que BCEL [46] Jabyce [106] et ASM [32] qui permettent de créer la pile d'instruction correspondant à l'expression. La troisième étape est effectuée en utilisant les fonctionalités offertes par la machine virtuelle Java, *ClassLoader*, qui consiste à charger dynamiquement la nouvelle classe créée dans l'environnement d'exécution.

6.2.4 Exécution

L'exécution des requêtes constitue la partie *run-time* du système d'intégration. Pour chaque requête à exécuter, le composant `Domain Manager` interagit avec le composant `Evaluation Manager` pour récupérer une instance d'un évaluateur qui est de type `Evaluation` (voir figure 6.6) avant de lancer l'évaluation. La séparation du composant `Domain Manager` du composant `Evaluation Manager` permet d'envisager des politiques évaluations de requêtes parallèles (parallel intra query processing). Le `Domain Manager` est alors responsable de la gestion du pool d'évaluateurs au cours d'exécution dans un domaine de données.

La technique d'évaluation adoptée est l'évaluation par itération [66]. L'évaluation de la requête se fait par la méthode `evaluate` prenant en entrée un plan d'évaluation `(EvaluableQuery)` et retourne une collection de tuples de type `ICollection0-n`, considérée aussi comme un itérateur sur des tuples (voir section 4.3). Cette évaluation se fait en deux étapes :

- Tout d'abord, répartir le plan d'évaluation sur les domaines données concernés en s'appuyant sur la propriété qui identifie le "lieu d'exécution" des opérateurs. Cette information est incluse dans les propriétés de chaque opérateur du plan qu'on peut récupérer par la méthode `getDataDomain` de l'interface `OperationProperties`. Ensuite, chaque itérateur d'un opérateur physique père est lié aux itérateurs des opérateurs fils. Pour les domaines

de données primitifs, le plan est traduit directement dans le langage cible de la source et les connexions nécessaires sont créées. Cette étape est l'équivalent de la primitive "*open*" dans l'évaluation par itération [66].

– Puis, lancer l'exécution effective de la requête. Cette étape se traduit par des appels sur les méthodes `evaluate` de chaque opérateur physique de la partie de l'arbre exécuté localement, et par la méthode `evaluate` de l'interface `ImportExport` pour récupérer les données à partir d'autres domaines de données.

```
Public interface Evaluation {

     // Evaluation de la requête query
     ICollection evaluate(EvaluableQuery query)
                                  throws IException;

     // Stopper l'évaluation de la requête
     void closeEvaluation();

}
```

FIG. 6.6 – L'interface `Evaluation`

L'interface `Evaluation` offre aussi la méthode `closeEvaluation` dont le rôle est l'arrêt de l'évaluation. Cette fonction est l'équivalent de la primitive "*close*" dans [66]. A l'inverse de l'opération d'initialisation, l'arrêt de l'évaluation consiste à libérer les ressources utilisées pour l'exécution des requêtes, notamment les connections, les tampons et les threads. Ceci concerne les ressources du domaine courant ainsi que celles des autres domaines concernés par le traitement. Les méthodes `closeEvaluation` de l'interface `ImportExport` sur ces domaines sont appelées pour arrêter l'évaluation dans tous les domaines de données et libérer les ressources, notamment les connexions vers les sources de données dans le cas des domaines de données primitifs.

6.3 Principes du canevas d'optimisation

Le canevas d'optimisation de requêtes est adaptable. Il permet de construire et de supporter plusieurs stratégies de recherche, tout en prenant en compte les capacités des sources, indépendamment d'un modèle de coût. Pour asseoir cette adaptabilité, le canevas se base sur deux ingrédients : l'utilisation des règles et la séparation de l'espace d'optimisation logique de l'espace d'optimisation physique.

6.3.1 Utilisation des règles

L'utilisation de règles de réécriture (ou de règles de transformation) dans la construction des optimiseurs a largement été identifiée comme un mécanisme très flexible [140, 103, 93]. D'une manière générale, une règle exprime une étape élémentaire de transformation des requêtes selon une sémantique d'optimisation. Le principe de recherche de plan à base de règles consiste à transformer graduellement un arbre de requête en un plan d'évaluation, en appliquant les règles. Le contrôle de l'ordre d'application des règles est effectué par la stratégie de recherche de l'optimiseur. La stratégie de recherche est le noyau de chaque optimiseur : la performance d'un

optimiseur se mesure par la performance de sa stratégie de recherche.

Nous utilisons les règles pour exprimer tous les types d'optimisation que nous avons présentés dans la section 6.2.2. La normalisation, la simplification, l'optimisation par les heuristiques, les propriétés des opérateurs algébriques (comme la commutativité et l'associativité), etc. sont décrites par des règles. Toutes ces règles permettent de transformer une expression en une autre expression.

6.3.2 Séparation de l'espace logique de l'espace physique

La séparation entre optimisation logique et optimisation physique permet de construire plusieurs stratégies de recherche [113, 103, 59, 134, 93]. Rappelons que l'optimisation logique porte seulement sur l'ordonnancement des opérateurs logiques d'expressions de requêtes, indépendamment des algorithmes d'évaluation associés. Quant à l'optimisation physique, elle porte sur l'ordonnancement des algorithmes d'évaluation associés aux opérateurs logiques. L'arbre représenté par la partie gauche (a) de la figure 6.7 est l'expression d'un arbre de requête : il est aussi appelé plan logique. L'arbre représenté par la partie (b) de la figure 6.7 est un exemple d'un plan d'évaluation de l'arbre de gauche : il est aussi appelé plan physique. Les deux noeuds *IndexSelect* et *NLJ* (*NestedLoopJoin*) du plan d'évaluation représentent respectivement deux algorithmes des deux opérateurs algébriques Select et Join de l'expression de (a). Le passage d'un arbre logique vers un arbre physique se fait en remplaçant chaque opérateur de l'arbre par une implantation. Chaque plan logique peut avoir plusieurs plans physiques. D'une manière générale, une expression de requêtes peut avoir plusieurs plans logiques (qui constituent l'espace logique d'une requête) et plusieurs plans physiques (qui constituent l'espace physique de la requête). Ces deux espaces représentent l'espace de recherche d'une requête. Le travail d'une stratégie de recherche consiste à combiner les plans logiques et les plans physiques des expressions de l'arbre pour retrouver le plan physique "optimal".

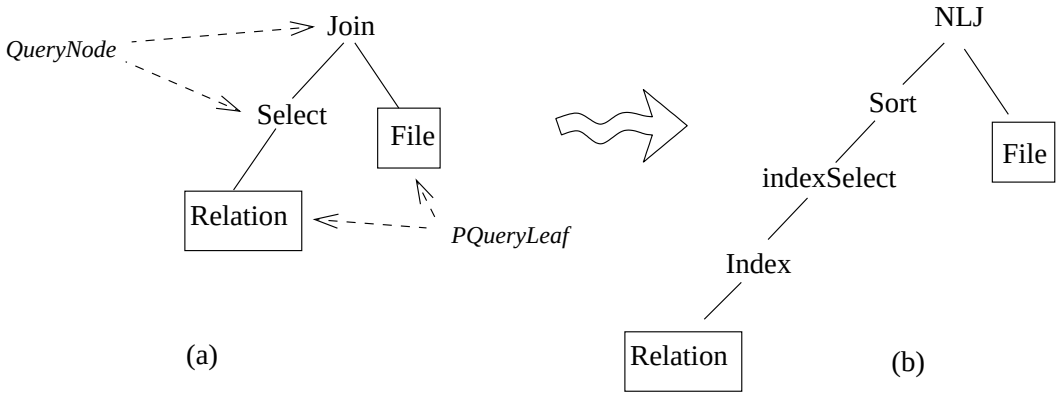


FIG. 6.7 – Optimisation logique vs optimisation physique

6.4 Plan d'évaluation

Un plan d'évaluation de requêtes est représenté par un arbre d'opérateurs physiques, où les noeuds représentent des algorithmes d'évaluation des opérateurs algébriques et les feuilles représentent des requêtes primitives exprimées dans le langage de requêtes cible de la source de

données. Un opérateur algébrique peut avoir plusieurs algorithmes d'évaluation : par exemple, l'opération de jointure peut être exécutée par plusieurs algorithmes, notamment MergeJoin, BindJoin, NestedLoopsJoin, etc, comme illustré par la figure 6.8.

Les opérateurs physiques sont spécifiés par l'interface `EvaluableQuery` (voir figure 6.9) qui étend l'interface `QueryNode`. On définit un opérateur physique par un ensemble de fonctionnalités et un ensemble de propriétés :

- *Les fonctionnalités* : Les deux fonctionnalités essentielles d'un opérateur physique sont l'évaluation et le calcul du coût de son évaluation. La méthode `evaluate` permet de lancer l'évaluation de cet opérateur à partir d'une liste de paramètres. La méthode `computeCost` permet de calculer le coût de l'évaluation effectuée par cet opérateur. Ce coût dépend de l'implantation de l'opérateur et des coûts des opérateurs fils du plan (des sous-plans). Ce calcul fait appel aux formules du modèle de coût du système.

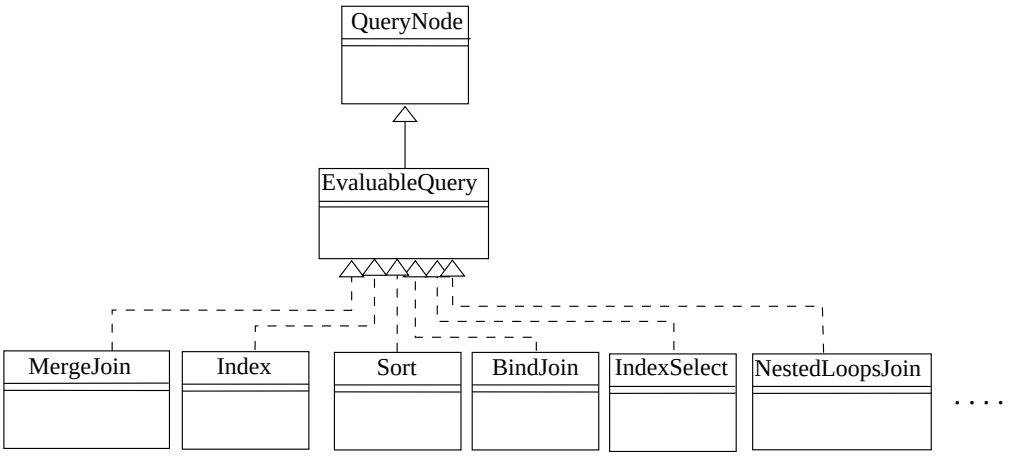


FIG. 6.8 – Algorithmes d'évaluation des opérateurs algébriques

- *Les propriétés* : Les opérateurs physiques sont caractérisés par deux types de propriétés, à savoir les propriétés offertes et les propriétés requises. Les propriétés offertes sont décrites par l'interface `OperationProperties` associée à l'opérateur logique, que nous avons décrite dans la section 5.6. Typiquement, ces informations permettent de savoir si les données délivrées par cet opérateur sont ordonnées, et dans quel ordre, s'il y a des valeurs dupliquées parmi les valeurs retournées, le domaine de données dans lequel cet opérateur sera exécuté, etc. L'interface `OperationProperties` peut être enrichie par d'autres informations relatives aux statistiques (comme par exemple le nombre de tuples) représentées par cet opérateur. Remarquons que ce type de propriétés n'est pas affecté à l'opérateur comme les autres propriétés, mais est calculé en faisant appel aux fonctions de modèles de coût du système.

  Les propriétés requises par un opérateur physique permettent de savoir quelles sont les propriétés que doivent offrir les opérateurs fils de celui-ci dans un plan. Par exemple, pour que l'algorithme de jointure *NestedloopJoin* soit valide dans un plan, il faut que les collections de données de ses entrées soient ordonnées suivant les attributs de jointure. D'autres

algorithmes tels que l'algorithme de l'opération de sélection *indexSelect* exigent que les collections de données soient indexées. Ces propriétés font partie de la définition et des caractéristiques de l'opérateur physique utilisé par l'optimiseur. Nous utilisons la même interface `OperationProperties` pour décrire ces propriétés.

```
Public interface EvaluableQuery extends QueryNode {

    // Calculer le coût relatif à ce noeud de l'arbre
    float computeCost() ;

    // Évaluer ce plan de requête
    void evaluate(Parameters[] param)throws EvaluationException ;

     // Récupérer les propriétés requises par cet algorithme
    OperationProperties getRequiredProperties() ;
}
```

FIG. 6.9 – L'interface `EvaluableQuery`

6.5 Règles

Les règles de réécriture sont spécifiées par l'interface `RewriteRule`, représentée par la figure 6.10. Chaque règle est définie par :

- Sa condition d'applicabilité sur une expression donnée, représentée par la méthode `canBeAppliedTo` qui prend en entrée une expression de type `Expression` et retourne un boolean. Cette expression peut être : une expression de chemin (`PathExpression`), une expression de calcul (`CExpression`), une expression d'arbre de requête (`QueryTree`) ou une expression du plan d'évaluation (`EvaluableQuery`). Si la valeur retournée est vraie alors la règle représentée par cet objet est applicable à cette expression. Les conditions d'applicabilité d'une règle portent d'abord sur le type de l'expression concernée par la règle, la structure de l'expression (la composition de l'expression) et les propriétés associées aux opérateurs logiques et physiques (`OperationProperties` et `RequieredProperties`).

- Une méthode `rewrite` qui représente le corps de la règle. Cette méthode prend en entrée une expression de type `Expression`, et retourne une ou plusieurs autres expressions qui sont aussi de type `Expression`. Ces expressions peuvent être des expressions de calcul, des expressions de chemin, des expressions d'arbre logique de requêtes ou des expression d'arbre physique. Dans la section suivante, nous classifions ces règles selon les types d'expressions en entrée et en sortie.

Selon la signature de la méthode `rewrite` de l'interface `RewriteRule`, on distingue trois types de règles :

```
Public interface RewriteRule {

    // Transformer un arbre de requêtes en plusieurs arbres
     Expression[] rewrite (Expression exp, Object parameters) ;

    // Tester si cette règle est applicable à cet opérateur de l'arbre
    boolean canBeAppliedTo(Expression exp) ;

}
```

FIG. 6.10 – L'interface `RewriteRule`

1. *Des règles de transformation d'expressions logiques vers d'autres expressions logiques.* L'expression transformée par ces règles est exactement du même type que l' expression retournée. Dans ce lot de règles, on retrouve les règles de simplification des expressions, des règles d'équivalence entre les opérateurs (commutativité, associativité, etc) ainsi que les règles basées sur les heuristiques.

   Les règles de réécriture des expressions d'arbres dans le langage cible de la source de données font aussi partie de ces règles. Ces règles sont propres aux domaines de données primitifs qui dépendent des sources. Par exemple, l'optimiseur d'un domaine de données primitif sur une source de données relationnelle contient une règle qui permet de transformer toute expression d'arbre `QueryTree` en une expression `QueryTree,` constituée seulement d'une feuille de type `PrimitiveQueryLeaf`. Cette feuille représente la requête SQL correspondant à l'expression d'arbre. En d'autres termes, ce type de règles permet le passage d'une représentation de requête dans notre canevas vers une représentation de requête dans le langage des sources.

2. *Des règles de transformation d'expressions d'arbre logique vers une expression d'arbre physique.* Ce type de règles permet de transformer une expression d'arbre logique (`QueryTree`) en une expression d'arbre physique (`EvaluableQuery`). Chacune des règles porte sur un opérateur bien déterminé : jointure, sélection, etc. La réécriture effectuée par ce type de règles consiste à remplacer les opérateurs logiques par les opérateurs physiques. Le nombre de règles de ce type est égal au nombre d'opérateurs physiques d'une algèbre. Si l'on se réfère à la figure 6.8, nous avons trois règles de réécriture qui portent sur l'opération de jointure représentant les trois algorithmes associés : *NestedLoopJoin*, *BindJoin* et *MergeJoin*.

3. *Des règles de transformation d'expressions d'arbres physiques vers d'autres expressions d'arbres physiques.* Ces règles concernent les expressions de types `EvaluableQuery`. Le rôle de ces règles est essentiellement d'insérer de nouveaux noeuds d'opérateurs physiques dans le plan pour que les propriétés requises par un opérateur physique donné soient vérifiées. Un exemple typique de ce genre de règles est la règle de réécriture qui insère l'opérateur *sort* pour ordonner les données, et la règle qui permet d'insérer l'opérateur d'indexation de données *index* dans un plan. Les conditions d'applicabilité de ces règles portent essentiellement sur les propriétés requises par les opérateurs : par exemple, pour appliquer la

règle de réécriture associée à l'opérateur *sort* à un opérateur donné du plan, il faut que la propriété d'ordre requise par cet opérateur soit fournie par les opérateurs fils. L'opérateur *sort* est alors inséré comme fils de cet opérateur pour que la propriété d'ordre soit vérifiée et que le plan devienne cohérent. Notons que ces opérateurs ne font pas forcément partie de l'algèbre de requête (i.e des opérateurs du canevas d'expressions de requêtes).

Ces trois types de règles suffisent pour exprimer les règles utilisées dans les stratégies de recherches bottom-up, top-down ou hybrides. Les règles des stratégies top-down permettent de remanier les expressions d'arbres, comme "pousser les sélections". Quant aux règles associées aux stratégies bottom-up, elles permettent d'étendre l'arbre en insérant à la racine d'expressions d'arbres de nouveaux opérateurs : à chaque type d'extension est associée une règle de réécriture, notamment une règle pour étendre la jointure, une règle pour étendre la sélection, etc. La spécification de l'interface `RewriteRule` suffit pour représenter ces deux mécanismes de réécriture. Pour plus de détails, nous vous invitons à vous référer à [93, 103] où le même principe est utilisé.

Notons que les stratégies de recherche utilisées par les wrappers en présence des capacités de calcul limitées sont généralement des stratégies génératives [73, 169] (voir section 6.2.2). Nous utilisons aussi des règles d'expansion pour exprimer les capacités des sources. Par exemple, si une source ne sait faire que la sélection sur un attribut donné, alors la règle d'expansion de la sélection doit se limiter à cet attribut : l'expression retournée par la méthode `rewrite` de cette règle est une sélection sur cet attribut. Ou encore, si une source ne sait pas faire de sélection alors il n'y aura même pas de règle d'expansion des sélections. Le processus de recherche consiste à rechercher le plan suivant le principe des stratégies génératives, sauf qu'au lieu d'appliquer les règles d'expansion classique (sans contraintes), on applique les règles d'expansion, qui expriment aussi les capacités des sources. L'arbre récupéré par ces expansions est comparé à l'expression d'arbre à optimiser. Le résultat final de l'optimisation est arbre où la partie supérieure représente éventuellement les opérations de calcul non prises en compte par la source.

6.6 Composants du canevas

Nous avons divisé les processus de recherche en deux catégories : le processus d'optimisation locale et le processus d'optimisation globale. Le processus d'optimisation locale est représenté par le composant `LocOptimizer`, responsable du contrôle de la stratégie de recherche dans un seul domaine indépendamment des autres. Le processus d'optimisation globale est représenté par le composant `ComOptimizer`. Il représente une stratégie de recherche composite associée à une composition de domaines. Cette stratégie considère à la fois la stratégie de recherche locale et les stratégies de recherche des domaines primitifs. Ces deux composants interagissent avec le composant `Rule Manager` qui est responsable de la gestion des règles.

6.6.1 Composant de gestion des règles

Le `Rule Manager` représente toutes les règles d'optimisation sus-citées. Il offre une seule interface `RuleManager` (voir figure `6.11`) avec une seule méthode `match` dont le rôle est de retrouver l'ensemble des règles applicables à une expression donnée `exp`. Pour cela, la méthode `canBeAppliedTo` de l'interface `RewriteRule` des règles peut être utilisée pour identifier la liste des règles applicables. Le paramètre `context` de cette méthode est optionnel. Il permet de restreindre la recherche à un groupe de règles suivant un partitionnement donné. Un exemple de

partitionnement naïf serait, selon la classification de règles que nous avons dressée dans la section précédente, de distinguer les règles de simplification et de normalisation des autres règles.

```
Public interface RuleManager {

    // Récupérer les règles applicables à une expression
    // dans un contexte
     RewriteRule[] match (Expression exp, Object context) ;

}
```

FIG. 6.11 – L'interface `RuleManager`

6.6.2 Composant d'optimisation locale

L'optimisation locale d'une requête est effectuée par un seul domaine de données indépendamment des autres. Elle concerne les deux types de domaines : primitifs et non primitifs. Ceci comprend les stratégies de recherches génératives et transformatives classiques que l'on retrouve dans les SGBD. La figure 6.12 représente les deux composants responsables de cette optimisation : le composant `LocOptimizer` et le composant `Rule Manager`. Le composant `LocOptimizer` interagit avec le composant `Rule Manager` via l'interface `RuleManager`.

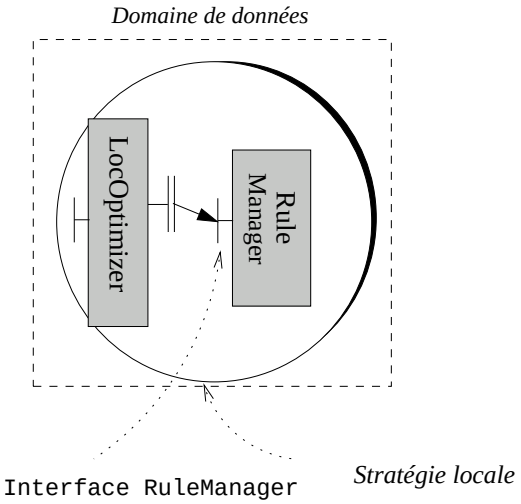


FIG. 6.12 – Composants des stratégies locales

Comme illustré par le diagramme 6.13, nous résumons l'algorithme général d'interaction de `LocOptmizer` et `Rule Manager` par :

1. Le `LocOptmizer` fait appel au gestionnaire de règles `Rule Manager` pour récupérer les règles applicables à une expression `exp` via l'interface `RuleManager`.

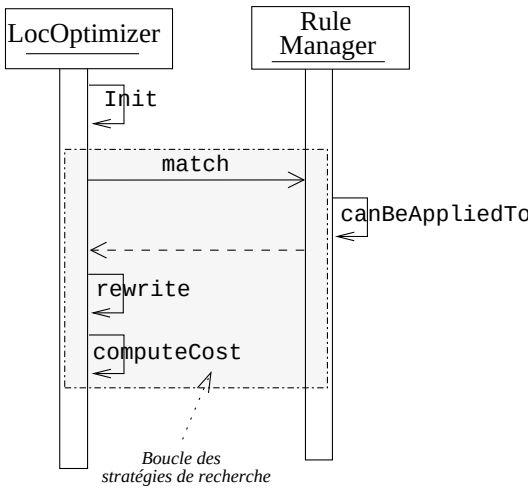


FIG. 6.13 – Interactions des stratégies recherches locales

2. Les règles retournées sont appliquées à cette expression. L'ordre d'application de ces règles dépend de la stratégie de recherche qu'incarne `LocOptmizer`.

3. Les plans retournés pendant la phase de réécriture (2) sont triés pour ne retenir que les expressions les plus intéressantes. La stratégie de recherche peut faire appel à la méthode `computeCost` de l'interface `EvaluableQuery` pour évaluer les coûts des différents plans, afin de retenir les plans les moins coûteux.

Ces trois séquences sont répétées jusqu'à l'obtention du plan. Notons que cet algorithme s'applique aussi dans une configuration où nous avons un domaine de données non primitif avec des domaines de données primitifs dont les capacités se limitent à une simple opération d'itération (ou un scan). L'optimisation est alors effectuée entièrement au niveau du domaine non primitif. Dans la section 6.7.3, nous verrons que ces composants supportent plusieurs stratégies de recherche.

6.6.3 Composant d'optimisation composite

Les stratégies d'optimisation composite représentent le processus de recherche d'un plan global d'une composition de domaines. Dans cette composition, on considère une configuration générale où l'on a un domaine de données non primitif lié avec des domaines de données primitifs ayant des capacités d'évaluation limitées. Cette configuration peut être le résultat de la phase de dépliage qui est effectuée par le `View Manager` (voir section 6.4). Le composant `ComOptimizer` du domaine non primitif s'appuie sur sa stratégie locale, représentée par le composant `LocOptimizer`, ainsi que sur les stratégies de recherche des domaines primitifs pour la recherche du plan. L'interaction avec les domaines primitifs est effectuée via les interfaces `ImportExport` correspondantes.

La figure 6.14 illustre une configuration à trois domaines de données dont deux domaines primitifs D2 et D3 à capacités limitées et un domaine non primitif D3. Pour optimiser une requête posée sur l'interface `ImportExport` de D1, le `ComOptimizer` de D1 interagit à la fois localement avec les composants `Rule Manager` et `LocOptimizer` et avec les deux autres domaines suivant

une stratégie de recherche donnée. Il fait appel au `Rule Manager` au moment de l'initialisation de la recherche afin d'appliquer les règles de normalisation et de simplification des expressions. Les deux domaines primitifs D2 et D3 incarnent des stratégies locales qui prennent en compte les capacités de traitement des sources. Comme déjà évoqué, les domaines de données primitifs peuvent utiliser des stratégies de type bottom-up pour énumérer localement les plans suivant les capacités des sources.

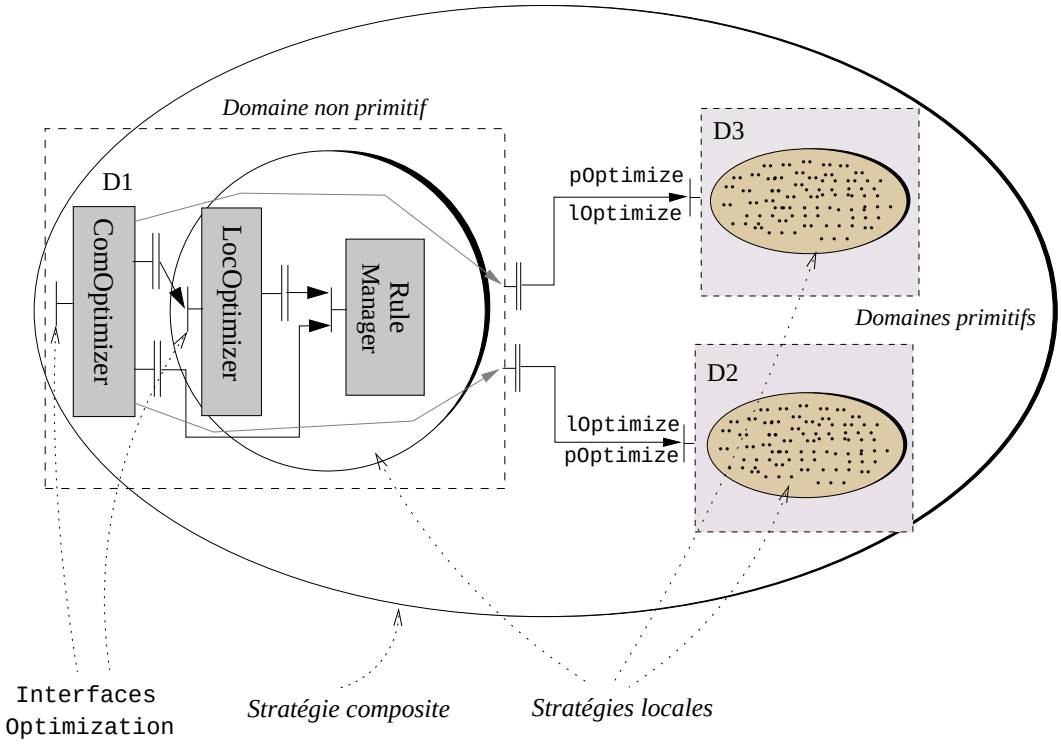


FIG. 6.14 – Stratégie de recherche composite

Dans ce cas aussi, nous nous sommes basés sur le principe de la séparation de l'optimisation logique de l'optimisation physique. En effet, les domaines de données primitifs exportent le traitement lié à l'optimisation logique et à l'optimisation physique via l'interface `ImportExport`. Dans l'architecture d'un domaine de données, ces fonctions sont représentées par l'interface `Optimisation` du composant `ComOptimizer`. Cette interface est la même que celle fournie par le composant `LocOptimizer` pour le composant `ComOptimizer` (voir figure 6.15). Comme représenté par la figure 6.15, cette interface permet de :

- Retrouver les plans logiques d'une expression d'arbre de requêtes par la méthode `lOptimize`. Dans le cas d'un domaine de données primitif, les expressions retournées sont décorées pour distinguer la partie de l'arbre prise en compte par le domaine de la partie non prise en compte. Les opérateurs de l'arbre pris en compte sont spécifiés en mettant à jour la propriété qui décrit leur "*lieu d'exécution*" en utilisant la méthode `setDataDomain` de l'interface `OperationProperties`.

- Retrouver les plans physiques d'une expression d'arbre de requêtes par la méthode `pOptimize`. Comme pour les plans logiques, les expressions d'arbres physiques sont décorées en utili-

sant la méthode `setDataDomain` pour distinguer la partie du plan prise en compte par la source et la partie non prise en compte.

- Rechercher le plan physique optimisé d'une expression d'arbre par la méthode `optimize`. Cette méthode retourne un seul plan optimal `EvaluableQuery`.

Ces fonctions sont exportées notamment par les domaines de données primitifs sans imposer une quelconque implantation. Le domaine de données primitif n'est pas obligé d'utiliser les composants `ComOptimizer` et `LocOptimizer` pour implanter les trois méthodes ci-dessus (principe du canevas : séparation de l'interface de son implantation). Par exemple, si un domaine de données primitif ne sait faire qu'un simple accès aux données de la source (comme les fichiers), alors l'implantation de la méthode `lOptimize` est basique : elle retourne la même expression `QueryTree` en fixant la propriété *"lieu d'exécution"* des feuilles (`PrimitiveQueryLeaf`) sur le domaine primitif courant. Le même raisonnement s'applique à la méthode `pOptimize`.

```
Public interface Optimization {

     // Récupérer les plans logiques d'une requête
     QueryTree[] lOptimize(QueryTree query)
                                   throws IException;

     // Récupérer les plans physique d'une requête
     EvaluableQuery[] pOptimize(QueryTree query)
                                   throws IException;

     // Récupérer le plan physique optimisé
     EvaluableQuery optimize(QueryTree query)
                                   throws IException;

}
```

FIG. 6.15 – L'interface `Optimization`

Comme illustré par la figure 6.14, le principe général de recherche des stratégies de recherche composites consiste à :

- Tout d'abord, lorsque le `ComOptimizer` reçoit une requête `QueryTree`, optimiser par le `Domain Manager`, en faisant appel à `Rule Manager` via l'interface `RuleManager` pour récupérer et appliquer les règles de simplification et éventuellement de normalisation. Cette étape correspond à la phase de préparation de l'optimisation que nous avons présentée dans la section 6.2.2.

- Puis, selon la stratégie composite incarnée par `ComOptimizer` du domaine D1, interagir alternativement une ou plusieurs fois avec les domaines de données primitifs D2 et D3 via les interfaces `ImportExport` correspondantes et avec le composant `LocOptimizer` de D1 via l'interface `Optimization`. Il utilise les deux méthodes `pOptimize` et/ou `lOptimize` de l'interface `ImportExport` pour déduire les parties de l'arbre prises en compte par ces

domaines. Le reste de l'arbre est exécuté au niveau du domaine non primitif selon une stratégie d'optimisation locale en faisant appel aux services de `LocOptimizer`.

Dans la section suivante, nous donnons des exemples d'interactions associées à deux types de stratégies de recherches composites.

6.7 Adaptabilité du canevas

Le canevas d'optimisation de requêtes est comparable à une lignée de travaux qui s'inscrivent autour de la construction des optimiseurs extensibles pour les SGBD : Starbust [72, 141], Exodus [68] et son successeur Volcano [69], Genesis [20], EROC [113], [103], [59]], TIGUKAT [134] OPT++ [93]. Ces travaux sont venus pallier la complexité du développement des optimiseurs dans les SGBD. Comme pour notre canevas d'optimisation, l'objectif de ces travaux n'est pas de proposer de nouvelles techniques d'optimisation mais plutôt des architectures d'optimiseurs permettant de supporter plusieurs techniques ou stratégies d'optimisation. Les premières solutions, notamment Starbust Volcano et Exodus, sont exclusivement basées sur le mécanisme de règles de réécriture pour construire des optimiseurs qui supportent plus au moins un nombre fixe de stratégies de recherches. Plus tard, d'autres travaux se rapprochant plus de notre travail, sont apparus avec l'avènement de la programmation orientée-objet (notamment C++), [103], EROC et OPT++. Ces travaux se basent à la fois sur le mécanisme de règles et sur les concepts de canevas logiciel[1] pour proposer des optimiseurs extensibles pouvant supporter plusieurs stratégies de recherche dans un contexte SGBD relationnel ou objet.

Notre canevas s'inscrit dans un contexte plus large d'optimisation de requêtes dans les systèmes d'intégration de données. Les points d'adaptabilité qu'offre ce canevas sont décrites dans les sous-sections suivantes.

6.7.1 Prise en compte de nouveaux cas d'optimisation

L'utilisation des règles apporte une extensibilité du canevas dans le sens où de nouvelles règles peuvent être supprimées ou ajoutées pour prendre en compte de nouveaux cas d'optimisation. Aussi, le fait de séparer le composant `Rule Manager` des deux composants `LocOptimizer` et `ComOptimizer` conforte cette flexibilité.

6.7.2 Adaptabilité aux opérateurs algébriques

Comme le canevas d'expression de requêtes est extensible en terme d'opérateurs algébriques, le canevas d'optimisation doit pouvoir prendre en compte de nouveaux opérateurs algébriques. En effet, l'architecture du canevas d'optimisation ne suppose pas une algèbre de requête particulière. Comme illustré par la figure 6.16, l'ajout d'un nouvel opérateur dans le canevas d'expression entraîne : (i) l'ajout d'un ou plusieurs opérateurs physiques correspondants en incluant leurs propriétés, (ii) l'ajout de règles de réécriture de type expressions logiques vers expressions logiques qui décrivent les règles de commutativité et d'associativité, etc. associées ainsi que les règles d'équivalence avec les autres opérateurs algébriques, (iii) l'ajout de nouvelles règles de type expressions logiques vers expressions physiques qui permettent d'associer l'opérateur logique à ces opérateurs physiques.

[1] Dans ces travaux, on ne mentionne pas explicitement le concept de canevas mais on parle de programmation orientée-objet (des classes d'objets).

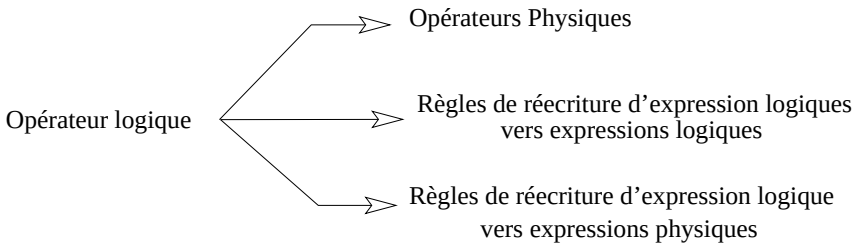


FIG. 6.16 – Adaptabilité aux algèbres

6.7.3 Adaptabilité aux stratégies de recherche

Le canevas offre un minimum d'interfaces qui couvrent les concepts de l'optimisation à base de règles et permet de construire des stratégies de recherche locales aux domaines de données que l'on retrouve dans les systèmes [72, 141, 68, 69, 93] : stratégies génératives, transformatives et hybrides comme dans les travaux sus-cités. Le principe de recherche des plans suit le processus *match select apply* [59, 67]. La clause *match* consiste à rechercher les règles applicables à une expression, la clause *select* consiste à sélectionner une liste parmi toutes ces règles et la clause *apply* consiste à appliquer ces règles. La clause *match* est offerte par le composant `Rule Manager` indépendamment d'une stratégie de recherche et les deux autres sont représentées par le composant `LocOptimizer`. La spécialisation d'une stratégie de recherche locale revient à personnaliser ces deux composants :

- Pour le composant `Rule Manager`, il s'agit de spécifier les différentes règles associées à la stratégie et d'organiser ces règles selon des critères. Dans tous les cas, ce composant comporte au moins des règles de transformation d'arbres logiques vers des arbres physiques et des règles de transformation d'arbres logiques vers d'autres arbres logiques qui décrivent les propriétés de commutativité et d'associativité entre les opérateurs. Les autres règles dépendent de la stratégie représentée par le `LocOptimizer`. Par exemple, si l'on veut implanter une stratégie bottom-up, les règles correspondront aux règles d'expansion des opérateurs : règle d'expansion des sélections, règle d'expansion des jointures, etc. Dans le cas d'une stratégie top-down, ces règles correspondront aux règles qui permettent de pousser les sélections, de faire monter les unions, d'éviter (ou de réécrire) les produits cartésiens.

- Pour le composant `LocOptimizer`, il s'agit d'implanter l'algorithme propre de la stratégie de recherche représentée par la méthode `optimize` de l'interface `Optimization` qui s'appuie sur la méthode `match` qu'offre le `RuleManager`.

Les deux autres méthodes de l'interface `Optimization` de `LocOptimizer` sont simplement implantées en appliquant respectivement les règles de type 1 et les règles de type 2 et 3 que nous avons présentées dans la section 6.5. Ces règles sont bien entendu récupérées à partir du `Rule Manager`.

6.7.4 Adaptabilité aux stratégies composites

La valeur ajoutée de notre canevas est son adaptabilité aux stratégies de recherche composites en prenant en compte les capacités d'évaluation des sources. Les domaines de données primitifs exportent leurs capacités logiques et leurs capacités physiques. On distingue deux ap-

proches d'optimisation : une proposée dans le système de médiation GARLIC et une dans le système de médiation DISCO, qui suit aussi les mêmes principes que celles proposées dans les systèmes YAT, TSIMMIS et Information Manifold. Nous avons choisi ces deux stratégies car elles sont représentatives des stratégies classiques bottom-up et top-down (voir section 2.5.10).

Comme déjà évoqué dans la section d'analyse des différents systèmes 2.5.10, les deux approches d'optmisation DISCO et GARLIC présentent toutes les deux des avantages et des inconvénients. L'avantage de l'approche DISCO est que l'optimisation est effectuée localement au niveau du médiateur, ce qui peut réduire le temps d'optimisation dans le cas où les wrappers et les médiateurs sont répartis. L'inconvénient de cette approche est que les médiateurs et les wrappers sont contraints de partager le même formalisme de description et qu'il est donc difficile de prendre en compte de nouveaux cas de description de capacités ; ce qui réduit l'adaptabilité. Dans GARLIC, on n'impose pas de formalisme de description des capacités des sources. Les médiateurs interagissent directement avec les wrappers via les méthodes `accessPlan` et `joinAccess`, indépendamment d'un formalisme de description particulier. On retrouve aussi dans ce cas le principe du canevas, dans le sens où les deux méthodes cachent la représentation des capacités. Ceci permet de couvrir tous les cas de description des capacités. En contrepartie, le coût de l'optimisation peut être important dans le cas où le wrapper et le médiateur sont répartis, car les interactions se traduisent par des appels de méthodes à distance.

L'approche que nous avons adoptée dans notre canevas est une approche hybride. En effet, du point de vue description des capacités des sources, notre approche est similaire à celle de GARLIC car nous utilisons aussi l'approche canevas pour ne pas imposer de formalisme de description. La similitude avec l'approche DISCO vient du fait que nous exportons l'espace logique des requêtes en plus de l'espace physique.

6.7.5 Validation par deux optimiseurs existants

Dans ce qui suit, nous montrons que notre canevas supporte les deux stratégies de recherche composites : DISCO et GARLIC. Dans les diagrammes qui illustrent les interactions des composants qu'engendrent ces deux stratégies, nous considérons l'exemple de la composition des trois domaines D1, D2 et D3 de la figure 6.14.

6.7.5.1 Cas de la stratégie DISCO

L'algorithme correspondant à la stratégie de recherche adoptée dans DISCO est illustré par le diagramme de la figure 6.17. L'itération des plans par le processus de recherche se fait en quatre phases :

1. *Génération de l'espace logique :* Le travail effectué pendant cette phase consiste à sélectionner un plan logique préliminaire parmi l'espace de recherche (les plans logiques) correspondant à l'expression de requête globale. Pour cela, le composant `ComOptimizer` de D1 appelle le `LocOptimizer` par la méthode `lOptimize` qui lui retourne une liste (ou une collection) de plans logiques.

2. *Pousser des sous-arbres logiques vers les sources* : Au début de cette phase, le `ComOptimizer` de D1 sélectionne un plan préliminaire parmi l'espace de recherche généré pendant la phase

précédente. Puis, il essaie de pousser des sous-arbres de cet arbre vers les domaines primitifs. Le choix des sous-arbres est effectué de manière à déléguer un maximum de traitement vers les sources. Il interagit donc avec les domaines D2 et D3 via leurs interfaces `ImportExport` en invoquant la méthode `lOptimize` qui retourne des plans où l'on distingue la partie de l'arbre prise en charge par la source de la partie qui sera exécutée par D1.

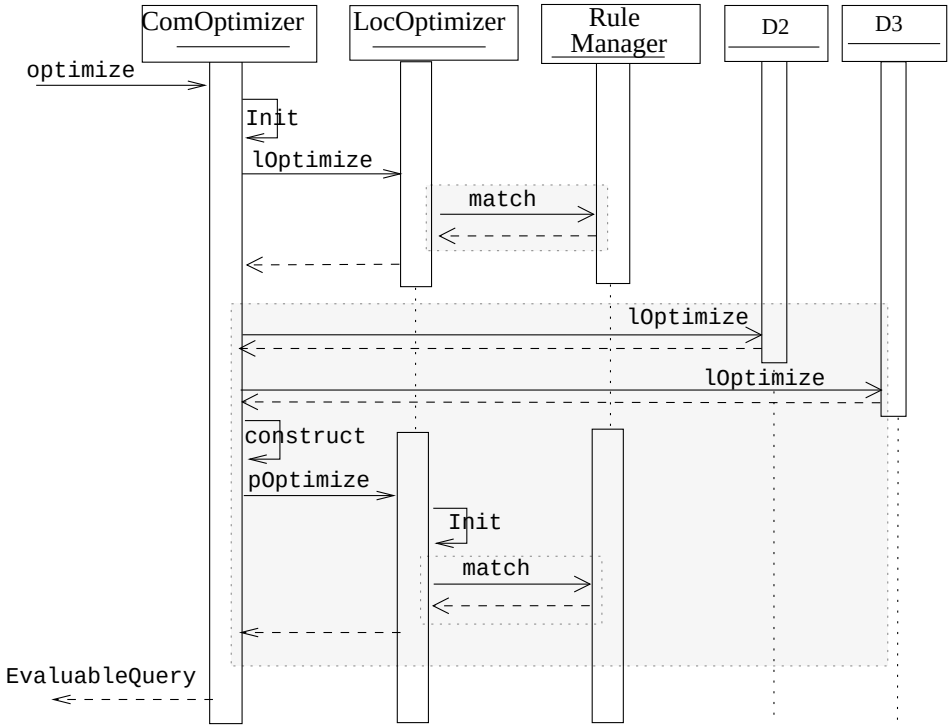


FIG. 6.17 – Stratégie de recherche dans Disco

3. *Construire le plan logique final :* Le `ComOptimizer` construit les plans logiques de l'arbre préliminaire sur la base des résultats retournés par la méthode `lOptimize` des deux domaines D1 et D2, en distinguant les parties de l'arbre exécutées dans D1 des parties exécutées dans D2 et D3.

4. *Génération du plan physique* : Durant cette phase, les plans physiques associés aux plans logiques résultats de la phase précédentes sont générés en considérant la partie de l'arbre exécuté dans D1. Le composant `ComOptimizer` fait alors appel localement à la méthode `pOptimize` du composant D1 qui lui interagit avec le `Rule Manager` pour appliquer les règles appropriées.

Les trois dernières étapes sont répétées jusqu'à l'obtention du bon plan.

6.7.5.2 Cas de la stratégie GARLIC

La stratégie de recherche dans GARLIC est de type *bottom-up*. Elle consiste à construire graduellement le plan depuis les feuilles jusqu'à la racine par expansion. Comme illustré par le diagramme 6.18, cette stratégie se traduit dans notre architecture par les trois phases suivantes :

1. *Pousser les sélections et les projections vers les sources* : Cette phase est l'équivalent de l'expansion des sélections dans les stratégies bottom-up. Elle consiste à pousser les sélections vers les domaines primitifs pour identifier les sélections et les projections de l'arbre que peuvent prendre en charge D2 et D3. Durant la phase d'initialisation, le `ComOptimizer` construit, à partir de la requête globale, deux parties de l'arbre, dont une pour D2 et l'autre pour D3, telles que ces arbres sont composés seulement des opérations unaires de sélection et de projection. Ces deux arbres sont ensuite poussés vers D2 et D3 en appelant les méthodes `pOptimize` offertes par leurs interfaces `ImportExport`. Les plans retournés sont annotés de manière à distinguer la partie du traitement effectuée par D1 et D2 des parties déléguées à D1.

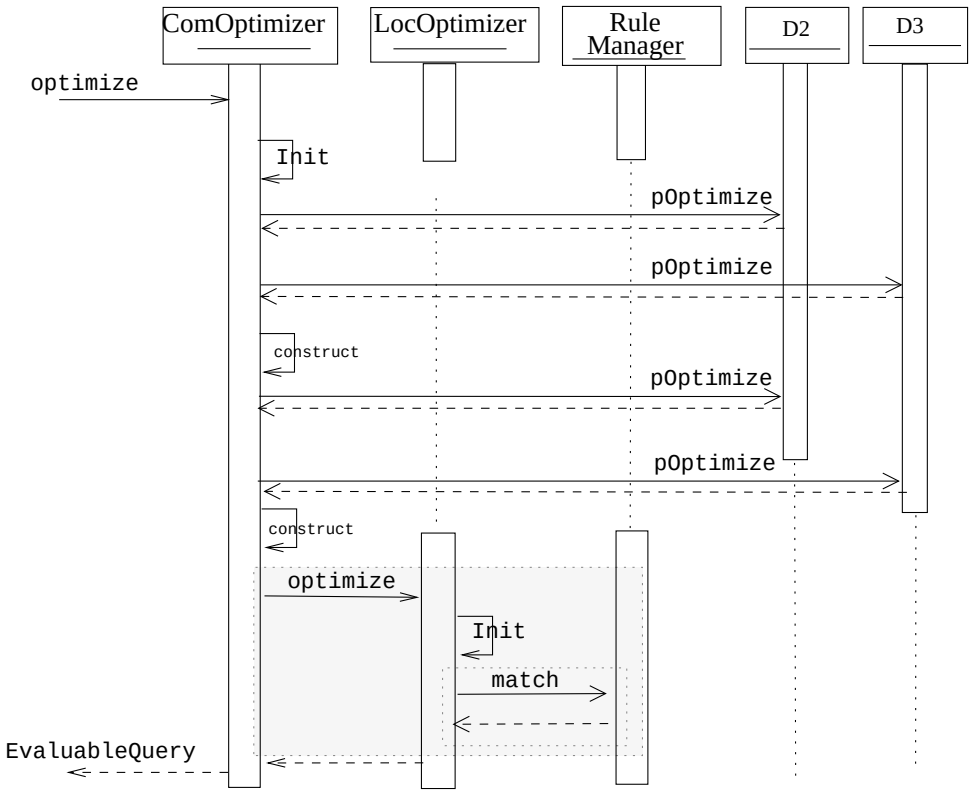


FIG. 6.18 – Stratégie de recherche dans Garlic

2. *Pousser les opérations de jointure vers les sources* : Cette phase est équivalente à l'expansion des jointures dans la stratégie bottom-up. Sur la base des plans physiques retournés dans la phase précédente, le `ComOptimizer` ajoute les opérations de jointure au-dessus des plans retournés pendant la phase précédente. Ensuite, il demande à D1 et D2 les plans physiques associés aux deux arbres à travers la méthode `pOptimize`. Comme dans la phase

précédente, les plans retournés sont bien sur annotés de manière à distinguer la partie du traitement effectuée par D1 et D2 et les parties déléguées à D1.

3. *Construire le plan final* : Après avoir poussé les sélections et les jointures de la requête vers D2 et D1, le `ComOptimizer` utilise les résultats de la phase précédente pour construire les parties des plans exécutés dans D1. Ces plans sont soumis à `LocOptimizer` par la méthode `optimize` qui retourne les plans physiques associés, avant que le `ComOptimizer` ne sélectionne et ne retienne le bon plan.

6.8 Conclusion

Le traitement des requêtes dans une composition de domaines de données s'appuie sur les techniques d'optimisation de requêtes dans les systèmes de médiation. Les composants d'un domaine permettent l'expansion des vues, l'optimisation, la compilation et l'évaluation des requêtes. L'expansion, qui permet d'éliminer les indirections d'une composition des domaines, se fait en se basant sur les définitions des vues importées/exportées par le gestionnaire des vues. La compilation est effectuée par le compilateur du domaine, dont le rôle est d'éliminer les vérifications de types dynamiques par la génération de code. L'exécution des requêtes se fait suivant une méthode classique par itération.

Nous avons présenté un canevas adaptable d'optimisation de requêtes qui prend en compte tous les aspects de l'optimisation dans une architecture à domaines. Ce canevas se base sur un mécanisme à base de règles, et sur la séparation de l'espace logique de l'espace physique des requêtes. Dans son architecture, nous avons également séparé les composants responsables du contrôle des stratégies d'optimisation locales et les composants responsables du contrôle des stratégies d'optimisation composites. Nous avons alors montré que toutes ces séparations permettent de construire diverses stratégies de recherche, aussi bien locales que composites.

Chapitre 7

Implantation et validation

C'est sur son utilité que la vérité fonde sa valeur
et ses droits; elle peut être quelquefois désagréable
à quelques individus et contraire à leurs intérêts,
mais elle sera toujours utile à l'espèce humaine ...
L'utilité est donc la pierre de touche des systèmes,
des opinions et des actions des hommes.

Paul–Henri "Baron d'Holbach"

7.1 Validation 1 : Persistance d'objets Java

Nous avons réalisé une première implantation en Java appelée MEDOR (Middleware Enabling Distributed Object Requests) [126] des canevas d'expression et d'optimisation de requêtes. MEDOR a été développé dans le cadre du consortium ObjetWeb par le laboratoire AMS de la Recherche et Développement de France Télécom, et constitue le socle logiciel pour la validation des travaux de cette thèse.

MEDOR s'inscrit dans un cadre plus large au sein d'ObjectWeb, qui regroupe plusieurs projets en vue de constituer un canevas de persistance. En particulier, un tel canevas doit permettre d'implanter différentes "personnalités", chacune réalisant l'implantation de spécifications de standards de persistance d'objets Java, tels que JDO et EJB-CMP pour lesquels une telle personnalité a été effectivement implantée.

7.1.1 JDO

La spécification JDO (Java Data Objects) de Sun [162] offre une API qui permet de faire persister d'une manière transparente l'état des objets Java. Le modèle objet de JDO est le modèle objet du langage de programmation Java. Cette spécification n'explicite pas le support de persistance à utiliser. Cependant, les produits implantant cette spécification utilisent les bases de données relationnelles comme support. L'exercice consiste donc à faire le mapping entre un schéma objet et un schéma relationnel (mapping objet-relationnel).

Le type de persistance offerte par JDO est la persistance par atteignabilité[1]. Le composant

[1] Traduction de "Reachability"

de base fourni par JDO, *Persistence Manager*, offre les fonctions de base pour faire persister, mettre à jour et accéder aux objets persistants en permettant la gestion des démarcations des transactions. Un objet persistant est accessible à travers son identificateur, par itération sur l'extension de sa classe correspondante (extent) ou en utilisant des requêtes.

L'expression de requêtes se fait à travers un langage de requêtes dédié JDO QL. Même si JDO QL opère les objets, ce langage est moins puissant que le langage promu par ODMG OQL. En effet, il permet simplement de sélectionner des objets à partir des collections ou à partir d'une extension d'une classe selon des filtres, mais ne supporte pas toutes les opérations algébriques, telles que le produit cartésien. Aussi, il faut noter qu'à l'encontre des requêtes OQL, les requêtes sont exprimées d'une manière programmatique et non pas sous forme d'une chaîne de caractères (string).

7.1.2 EJB-CMP

La spécification EJB-CMP (Entreprise Java Beans - Container Managed Persistence) de Sun [161] fait partie de la spécification plus large J2EE, Java 2 Enterprise Edition, et porte sur un service de persistance pour les containers EJB. Ces containers permettent de gérer les entités beans *(entity bean)* persistantes qui peuvent être accédées directement à partir d'une application Java en utilisant les protocoles RMI/IIOP, ou indirectement à partir des serveurs Web via HTTP en invoquant des servlet, JSP ou des Web services.

Les entités beans persistantes sont organisées sous forme d'un schéma objet. Le modèle de données inhérent est un modèle objet supporté de type entité-association dans le sens où les entités sont simplement un ensemble d'objets avec des associations (l'équivalent des *relationships* dans le modèle ODMG) pouvant supporter les cardinalités : (1..1), (1..n), (n..1) et (n..n). Au déploiement, à chaque entité CMP est associé un comportement, notamment des méthodes *get/set* sur leurs attributs (l'interface *remote interface*), la création et la recherche des beans (l'interface *remote home interface*).

Pour l'accès associatif aux entités bean, le langage EJB QL a été spécifié. C'est un langage proche de SQL92 avec des possibilités de navigation à travers le schéma des entités bean. Les requêtes EJB QL sont définies pour les méthodes *finder* et *select* associées à une entité bean dans le descripteur de déploiement.

7.1.3 Canevas de persistance : approche

L'approche générale de la persistance dans ObjectWeb se base aussi sur le principe de la séparation de préoccupations. Deux intergiciels sont concernés :

- JORM pour la gestion des entrées/sorties (des *setters* et des *getters*) des objets persistants sur les sources de données.
- MEDOR pour la gestion des accès associatifs aux objets persistants ainsi que tous les aspects liés au traitement de requêtes, notamment l'implantation des deux langages de requêtes JDO QL et EJB QL.

L'interaction entre ces deux intergiciels, et leur utilisation dans les deux cadres JDO et EJB-CMP, sont illustrées par la figure 7.1. Cette figure montre les trois couches considérées dans notre intergiciel d'intégration (voir figure 3.1), où l'on distingue la couche source de données, la couche intergicielle composée de MEDOR et JORM et la couche supérieure représentant JDO et EJB-CMP ; elles sont considérées comme des applications d'intégration. Afin de mieux comprendre comment les travaux de cette thèse ont pu être validés dans ce contexte, il est nécessaire de présenter plus en détails l'intergiciel JORM.

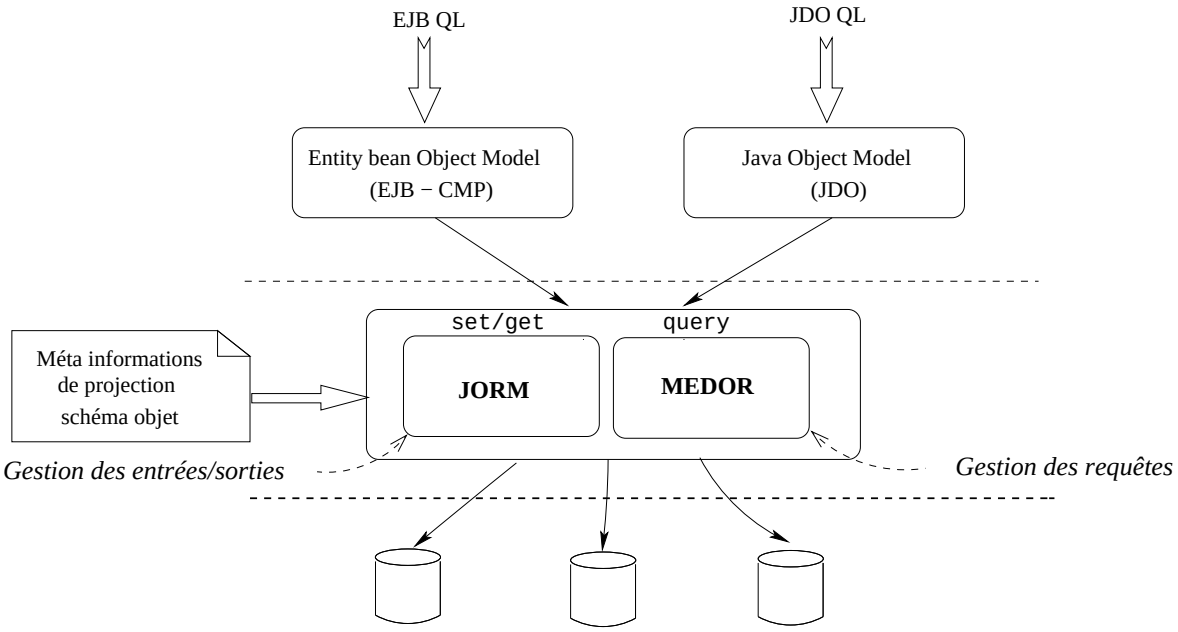


FIG. 7.1 – Approche de persistance d'objets Java

7.1.4 JORM

JORM (Java Object Repository Mapping) [55, 125] est un intergiciel de persistance d'objets, également développé par le laboratoire AMS de la Recherche et Développement de France Télécom.

JORM traite des entrées/sorties entre le support de stockage et les objets en mémoire, et utilise des identifiants d'objets persistants. Pour cela, JORM s'appuie sur des objets de liaison qui sont des objets d'interposition représentant les instances du support de stockage (*Data Store Instances* - DSI). Le contenu persistant des objets mémoire est projeté sur les objets de liaison, qui sont eux-mêmes projetés sur le support de stockage selon les méta-informations associées. Les données persistantes des objets mémoire sont typiquement les attributs (champs) des objets. Les objets de liaison suivent un modèle objet. Les interactions entre les objets de liaison et les objets en mémoire se font par des objets, appelés Instances Mémoire (*Memory Instances, MI*), qui contiennent les attributs persistants. Les concepts manipulés par JORM sont les suivants.

Data Store Instance (DSI)

Une DSI est une entité stockée dans une source de données. Une telle entité est identifiée par un nom persistant dans JORM. A cette fin, les deux concepts de nom et de contexte de nommage empruntés à RM-ODP sont utilisés (voir section 4.6). Une DSI peut être un n-uplet dans une table relationnelle, un objet d'une classe dans une base de données objet, ou un fichier dans un système de fichiers.

Instances Mémoire

Une Instance Mémoire (*Memory Instance, MI*) est un objet qui contient les variables des objets mémoire dont la persistance doit être assurée. Elle coopère avec un objet de liaison afin de charger/stocker ses variables depuis/dans la source de données. Du point de vue de JORM, la MI est un accesseur (voir section 7.1.4.1). Une telle instance peut être, par exemple, l'objet *PersistenceCapable* d'une implantation de JDO.

Gestionnaire d'instance mémoire

Un gestionnaire d'instances mémoire (*Memory Instance Manager - MIM*) est la couche logicielle (couche applicative représentant une application d'intégration) qui gère les instances mémoires correspondant aux DSIs. Cette couche offre en général un niveau élevé de transparence concernant la gestion de la projection (c'est-à-dire qu'elle cache les actions de chargement et de stockage). Elle définit également le cycle de vie des MIs. Des exemples de MIMs incluent les implantations de JDO, du Persistent State Service de CORBA ou d'EJB CMP. La séparation entre les MIs et les objets de liaison offre une liberté de choix pour l'implantation de cycles de vie divers et variés, et peuvent être réalisés de manières différentes. L'utilisateur du canevas de persistance JORM doit simplement implanter les MIs et les liens entre les MIs et les objets de liaisons, qui sont eux gérés par JORM.

7.1.4.1 Gestion des entrées/sorties : objets d'interposition et accesseurs

Le principe architectural de JORM consiste donc à interposer des objets de liaison entre les instances mémoire et les entités sur le support de stockage. Ces objets de liaison offrent des fonctions de synchronisation typées entre la MI et la DSI, les deux fonctions principales étant *read* pour la lecture (chargement en mémoire) et *write* pour l'écriture (stockage dans le support). Ces fonctions sont typées dans la mesure où elles offrent une structure particulière pour chaque classe d'objets persistants, ainsi que la manière de projeter cette structure sur les DSIs associées. Ainsi, un objet de liaison est l'objet Java qui assure la projection lors des opérations d'entrée/sortie.

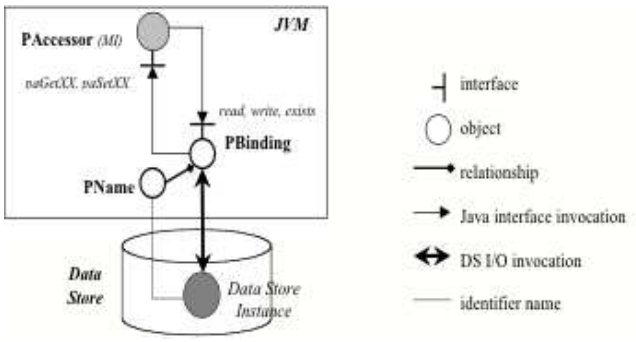


FIG. 7.2 – Objets de liaisons pour la persistance d'objets Java

La figure 7.2 illustre la chaîne de persistance entre la DSI et la MI à travers trois interfaces : `PBinding`, `PAccessor` et `PName`. Avant de pouvoir effectuer les synchronisations, un objet de liaison, représenté par l'interface `PBinding`, doit être associé à un identifiant persistant, représenté par l'interface `PName`. L'identifiant est un object Java qui désigne la DSI à laquelle l'objet de liaison est lié. De manière symétrique, l'objet de liaison doit être associé avec un objet accesseur, représenté par l'interface `PAccessor`, afin d'avoir accès aux variables (ou champs) contenant l'état de la MI à laquelle il est lié. Une fois ces deux associations réalisées, la chaîne de persistance est opérationnelle. Afin de rendre un objet persistant, la méthode `write` est appelée sur l'objet `PBinding`; l'objet PAccesseur est utilisé pour lire depuis la mémoire les valeurs à stocker dans le support de persistance, en appelant les méthodes `paGetXXX`. Inversement, pour lire un objet persistant depuis le support de stockage, la méthode `read` est appelée sur l'objet PBinding; l'objet `PAccessor` est alors utilisé pour écrire en mémoire les valeurs obtenues depuis le support de stockage, en appelant les méthodes `paSetXXX`.

L'utilisateur du canevas de persistance JORM a la responsabilité de fournir l'implantation de l'interface `PAccessor`, dans la mesure où la gestion des objets en mémoire n'est pas du ressort de JORM lui-même. L'interface `PAccessor` est composée de méthodes set et get spécifiques aux attributs de l'objet (les méthodes `paSetXXX` et `paGetXXX` pour chaque attribut nommé `XXX`). La méthode `paGetXXX` retourne la valeur de l'attribut nommé `XXX`. Une table de traduction entre les types objet du modèle et leurs équivalents Java détermine le type de cette valeur (par exemple un *int* Java). De manière symétrique, la méthode `paSetXXX` réalise l'action d'assigner une valeur typée à l'attribut `XXX` de l'instance mémoire correspondante.

7.1.4.2 Les méta-informations

Les méta-informations décrivent principalement les règles de mapping entre un schéma objet et les sources de données. Cette méta-information décrit la projection structurelle entre les objets du modèle JORM et ceux du modèle des sources de données sous-jacentes. Cette méta-information contient la définition des projections pour chaque classe et chaque type de support de stockage dans lequel les objets de cette classe peuvent être stockés. L'information de projection dépend entièrement du type de la source de données de stockage. Dans le cas des bases de données relationnelles, plusieurs règles de projection peuvent être utilisées, notamment pour prendre en compte l'héritage, les références entre classes (soit dans une table, ou en tant que référence inverse dans la table de la classe référencée), et les classes génériques (avec ou sans table de jointure supplémentaire). Ces méta-informations permettent de générer les objets d'interposition pour la gestion des entrées/sorties selon les règles de projection et les autres méta-données décrivant le mode de gestion des références.

7.1.5 MEDOR

Le modèle de données utilisé par MEDOR est un modèle de donné typé structuré qui est basé sur les tuples. C'est donc un sous-ensemble du modèle présenté au chapitre 4. Ce modèle prend également en compte les objets du modèle de données de JORM, et en particulier les noms persistants. MEDOR comprend les implantations suivantes :

- Le canevas d'expression de requêtes a été entièrement implanté, et contient les expressions de calcul (des opérateurs `CExpression`), les expressions de chemin simples de type "a.b.c" (des opérateurs `ConcatNavOp`) permettant la navigation à travers les références entre ob-

jets JORM, et les expressions d'arbre de requête. Plusieurs types de feuilles primitives, (des opérateurs `PQueryLeaf`) ont été implantés, en particulier pour les classes JORM, les bases de données relationnelles et les données en mémoire.

- Le canevas d'optimisation comprend un ensemble de règles permettant la transformation d'arbres de requêtes. Un premier ensemble de règles est constitué de règles d'optimisation classiques permettant par exemple de pousser les sélections vers le bas ou de supprimer des noeuds ne contenant aucun filtre. Un autre ensemble de règles concerne la réécriture de feuilles portant sur des classes JORM en feuilles portant sur des données du support de stockage. L'utilisateur de MEDOR peut ensuite combiner programmatiquement l'enchaînement de ces règles pour constituer sa propre stratégie d'optimisation.

- Implantation d'un évaluateur de requêtes. Cet évaluateur délègue l'évaluation des requêtes des feuilles aux supports de stockage correspondants, et réalise en mémoire le reste de l'évaluation.

Nous détaillons maintenant certains points qui mettent en avant l'adaptabilité de l'approche suivie dans cette thèse et plus particulièrement les deux canevas d'expression et d'optimisation de requêtes.

7.1.5.1 Expression de requêtes

Des feuilles d'arbres pour les extensions de classes JORM ont été implantées, appelées *extents*. Un extent représente conceptuellement tous les objets d'une classe persistante, indépendamment de la manière dont elle peut être projetée sur une source de données. Le type `IType` de cet extent est `ICollection0-n (ITuple)`, dont les attributs sont les attributs de la classe persistante, ainsi qu'un attribut contenant le `PName` de la classe persistante. Ce `PName` peut être manipulé comme tout autre attribut (il peut être projeté, ou encore faire partie d'une expression dans le filtre de la requête).

Exemple :

Soit un schéma objet représentant des producteurs et des fournisseurs : des objets *Product* et des objets *Supplier*. La classe *Product* est décrite par deux attributs *name* et *price* et un autre attribut *supplier* de type référence sur les objets *Supplier*. La classe *Supplier* est décrite par un attribut *supplierName*. Ces deux classes sont respectivement projetées sur une base de données relationnelle composée de deux tables : *SE_Product* et *SE_Supplier*. L'attribut référence *supplier* de la classe *Product* est représenté par la valeur de la clé primaire *Sname* de la table *SE_Supplier*.

Soit la requête "retrouver les identifiants des objets *Product* (i.e PName) dont les prix sont compris entre 100 et 200 et qui sont fournis par un fournisseur "*MySupplier*". En utilisant le canevas d'expression de requêtes, l'expression de cette requête est représentée par la figure 7.3.

```
// Création des deux feuilles sur Product et Supplier
ClassExtent pExtent = new ClassExtent ("invoice.Product");
ClassExtent sExtent = new ClassExtent ("invoice.Supplier");


// Création de la racine de l'arbre
QueryNode myQuery = new JoinProject();


// Projection des noms de Product
myQuery.addPropagatedField(pExtent, new RegExp("PName"));


// Création d'un prédicat
CExpression filter = new And(
   new And (
      new Greater(new PathExpressionImpl(pExtent, new RegPathImpl("price")),
         new VariableOperand (100)),
      new Lower(new PathExpressionImpl(pExtent, new RegPathImpl("price")),
         new VariableOperand (200)),
      ),
   new And(
      new Equal (new PathExpressionImpl(sExtent, new
RegPathImpl("supplierName")),
          new VariableOperand ("mySupplier")),
      new Equal (new PathExpressionImpl(sExtent, new RegPathImpl("PName")),
      new PathExpressionImpl(pExtent, new RegPathImpl("supplier"))
   )
);


// Affectation du prédicat au noeud
myQuery.setPredicate(filter);
```

FIG. 7.3 – Exemple de création d'une requête

7.1.5.2 Optimisation de requêtes

L'approche suivie dans l'utilisation de JORM et MEDOR en termes d'optimisation de requêtes est la suivante. L'évaluation de requêtes est déléguée, dans la mesure du possible, au support de stockage :

- Règles de réecriture

1. Une règle de réécriture transforme les extents en feuilles d'arbre correspondant au support. Cette règle s'appuie sur la méta-information de projection JORM. Le `PName`, qui est un attribut de l'extent, est transformé en attribut calculé à partir des attributs correspondants de la feuille du support (par exemple les clés primaires dans le cas d'une projection sur une base relationnelle).

2. Ensuite, une règle de réécriture modifie la structure de l'arbre afin de rapprocher les feuilles correspondant au même support. Dans notre cas d'application, nous avons utilisé une source de données relationnelle pour la projection. Ces sources de données ont des capacités d'évaluation supérieures aux besoins des deux langages EJB QL et JDO QL. Pour simplifier l'implantation de cette règle et aussi pour des raisons de performance, cette règle ne tient pas compte des capacités d'évaluation de la source sous-jacente.

3. Enfin, une dernière règle réduit les feuilles du même support ainsi regroupées en une seule feuille, ceci afin de déléguer l'évaluation de la portion de l'arbre correspondant au support de stockage. Ainsi, dans le cas relationnel, une règle permet de transformer un sous-arbre contenant plusieurs extents portant sur une même base de données en une seule feuille relationnelle contenant une unique requête SQL. On notera que, même pour le même type de source de données, des adaptateurs peuvent être nécessaires pour prendre en compte les différences de dialecte. Ceci a été nécessaire pour les bases de données relationnelles.

Par rapport au canevas d'optimisation de requêtes présenté dans le chapitre 6, l'implantation MEDOR comprend seulement une stratégie d'optimisation locale et non composite, en considérant que les sources ont toutes les capacités d'évaluation. Comme le cycle de traitement de requêtes dans un système à base de domaines (voir section 6.2), la dernière phase de traitement est la construction des résultats. Dans notre cas, ceci consiste à créer des `PName` à partir des collections des tuples. Le type `PName` est l'équivalent du `IName` et les noms et les contextes de nommage sont utilisés pour la gestion des références (voir section 4.6.

- Prefetching

Le canevas composé de JORM et MEDOR contient une optimisation importante, qui consiste à pré-charger (*prefetch*) des attributs d'objets au moment de l'évaluation de la requête. L'utilisation type de JORM et MEDOR est la suivante : l'utilisateur soumet tout d'abord une requête, dont la réponse est constituée de noms persistants d'objets qui respectent une condition particulière. Ensuite, l'utilisateur interagit avec JORM pour charger les objets correspondants en mémoire. Un séquencement naïf consisterait à effectuer un premier accès au support de stockage afin de retourner les noms persistants, puis autant d'accès au support qu'il y a d'objets à charger en mémoire, afin de remonter la valeur de leurs attributs. L'optimisation de préchargement

consiste à remonter, lors de la phase d'évaluation de la requête, suffisamment d'information (c'est à dire les attributs des objets, et pas seulement leur nom persistant), afin de pouvoir charger la MI sans avoir à effectuer d'accès supplémentaire au support de stockage. Notons que ce type d'optimisation n'est pas explicité dans le canevas d'optimisation de requêtes présenté dans le chapitre 6. Elle est représentée par un opérateur physique spécial de matérialisation des résultats intermédiaires.

Exemple :

En considérant l'exemple de la section précédente, la figure 7.4 illustre les trois grande étapes de réecriture appliquées à la requête représentée par la figure . Ces étapes représentées par les règles *règles 1*, *règle 2* et *règle 3* correpondent aux trois étapes de réécriture sus-citées.

7.1.6 Synthèse

JORM et MEDOR ont été couplés à d'autres composants intergiciels de persistance afin de prendre en compte les aspects liés aux caches, à la concurrence et à la gestion des transactions [55, 63]. Cette intégration a montré que l'approche suivie dans cette thèse a effectivement permis une adaptabilité à d'autres aspects techniques.

En ce qui concerne JDO et EJB-CMP, MEDOR et JORM sont utilisés respectivement dans l'implantation des systèmes Speedo [127] et JOnAS [128]. La validation en termes de support de stockage a été effectuée sur un support relationnel, en centralisé (une seule base de données). En partant de leurs propres descripteurs, les deux systèmes Speedo et JOnAS génèrent le modèle objet utilisé par JORM et MEDOR, ainsi que les méta-informations de projection correspondantes. Dans les deux cas, les classes des objets de liaison sont générées. Les architectures des deux systèmes sont différentes, ce qui illustre effectivement la flexibilité de l'approche. Cette flexibilité a aussi permis de prendre en compte la spécification des deux stantards de persistance, notamment en termes de langage de requêtes. En effet, par exemple en termes de langage de requêtes, l'adaptabilité du canevas d'expression de requêtes nous a permis de suivre facilement l'évolution de la spécification des langages de requêtes des deux standards de persistance.

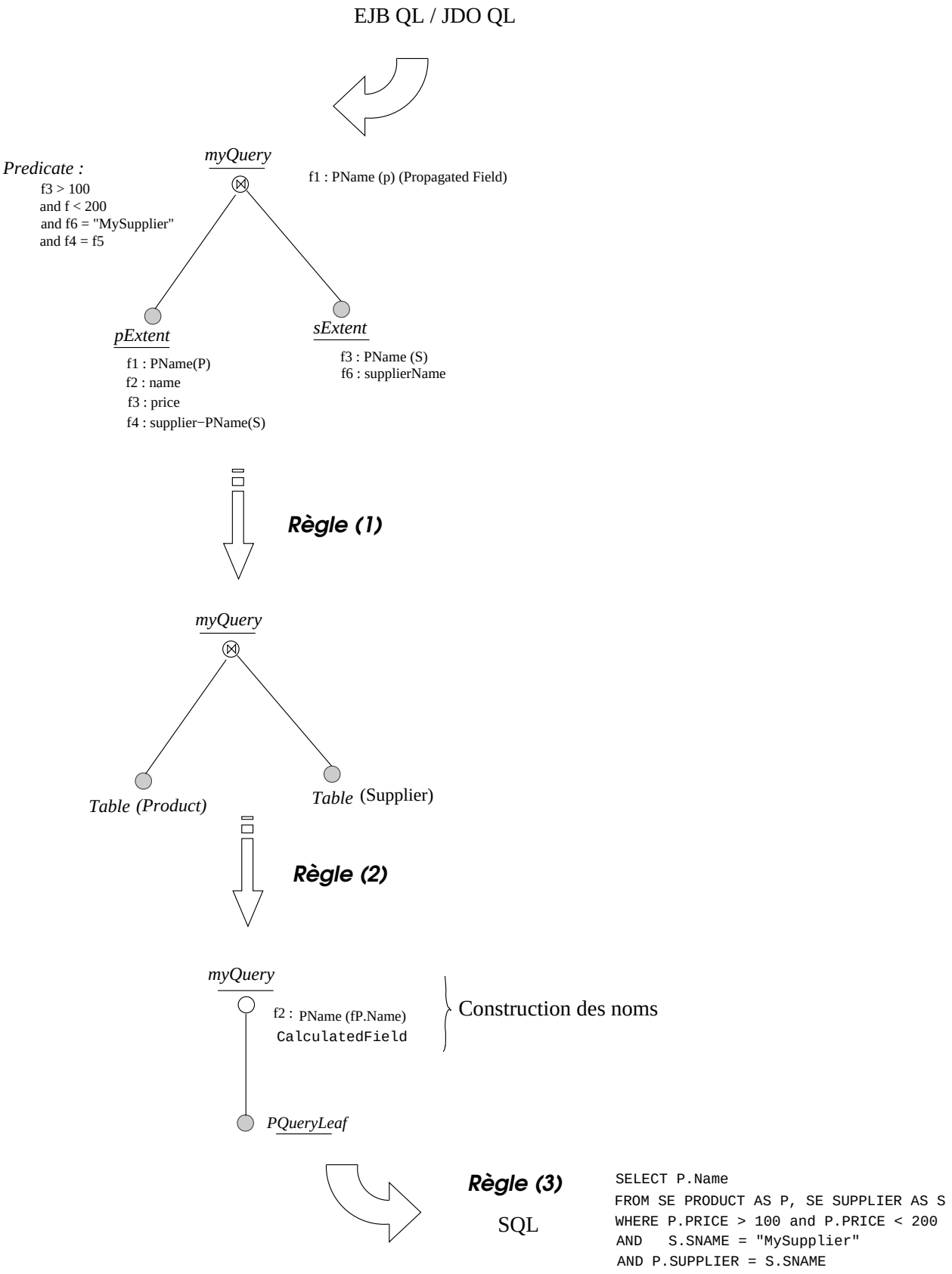


FIG. 7.4 – Etapes de réecriture de requêtes dans MEDOR

7.2 Validation 2 : requêtes sur les systèmes réflexifs à composants

L'approche d'intégration par la composition de domaines se base sur les concepts et les principes des architectures à composants. Dans cette section, nous procédons inversement : nous utilisons les domaines de données et les canevas associés pour l'interrogation et l'intégration de données dans les architectures à composants à large échelle.

Les données et les services de grandes compagnies tendent à devenir de plus en plus hétérogènes et répartis. Ces systèmes d'informations et réseaux incluent des équipements réseaux tels que les routeurs IP et les switch ATM, des serveurs tels que les serveurs d'authentification, des serveurs Web ou des serveurs d'application et des systèmes de base données. La complexité de ces systèmes se mesure par les propriétés suivantes : (i) *répartis,* dans le sens où leurs éléments sont interconnectés à travers le réseau, (ii) *dynamiques*, dans le sens où des éléments peuvent être ajoutés ou supprimés et (iii) *hétérogènes* car leurs fonctions et leurs capacités sont différentes. Cette répartition, dynamicité et hétérogénéité fait que ces systèmes sont difficiles à développer et déployer, i.e, à installer, contrôler ou mettre à jour par les différents acteurs qui interagissent avec de tels systèmes, notamment les administrateurs, les intégrateurs d'équipement réseaux et les développeurs.

Maîtriser la complexité de ces systèmes, tel est l'objectif général des travaux autour des architectures des composants logiciels et plus précisément des modèles à composants réflexifs. Nous supposons donc que ces systèmes sont développés en utilisant un modèle à composants réflexifs, comme typiquement le modèle de composants Fractal. Les différentes ressources et les serveurs qui composent le système sont réifiés par des composants d'une manière uniforme, comme nous le montre la figure 7.5.

Les différentes tâches de déploiement que nous avons citées portent sur des composants particuliers du système tels que les ressources. Il s'avère donc nécessaire de localiser et d'interroger ces systèmes selon des critères. Pour ce faire, nous considérons un composant *réflexif* comme une source de *méta-données*. Il s'agit donc d'introspecter la totalité du système en se basant sur les fonctions de réflexion.

7.2.1 Modèle de composants réflexif : Fractal

Le modèle de composants Fractal a été élaboré dans le cadre du consortium ObjectWeb par le laboratoire AMS de la Recherche et Développement de France Telecom et le projet SARDES du laboratoire LSR à l'INRIA. L'objectif du modèle de composants Fractal[31, 30] est d'implanter, de déployer et de gérer des systèmes complexes. La philosophie de Fractal est aussi basée sur le principe de la séparation des préoccupations. Comme le montre la figure 7.6, un composant Fractal comporte deux parties : la partie *contrôleur,* appelée aussi *membrane,* qui représente les aspects non fonctionnels et la partie *contenu* qui représente les aspects fonctionnels. Les propriétés originales du modèle Fractal sont :

- *Identité* : Comme dans les langages orientés-objet, un composant Fractal a un identificateur unique qui permet sa manipulation d'une manière non ambigüe pendant son cycle de vie.

- *Encapsulation* : Les interactions avec les composants sont réalisées à travers des points

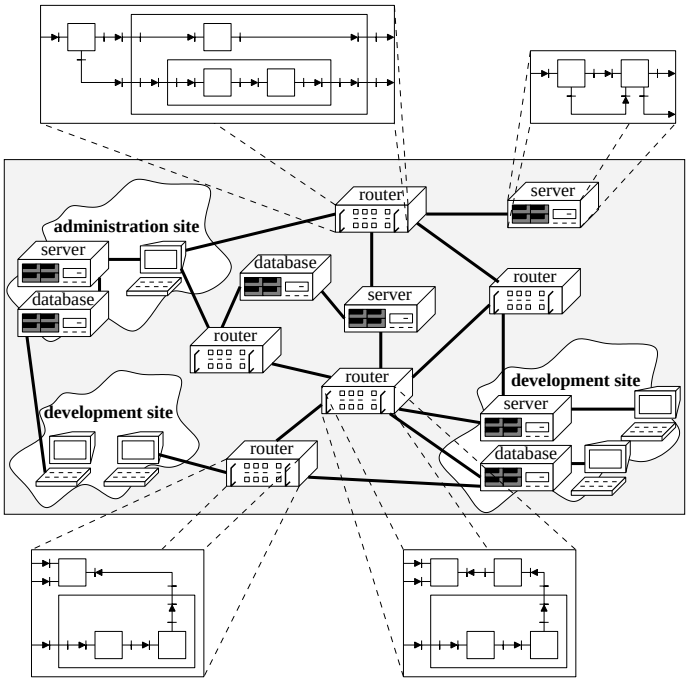


FIG. 7.5 – Système réflexif à composants

d'accès bien définis que sont les interfaces. Une *liaison* (*Binding*) est un canal de communication entre deux interfaces. Ces liaisons sont arbitraires et peuvent être centralisées ou réparties.

– *Composition récursive* : Un composant peut être récursivement composé de sous-composants. Il est alors appelé un composant *composite*. Cette récursion se termine par des composants appelés composants primitifs. Les composants primitifs ne contiennent pas de sous-composants et encapsulent directement des entités du langage sous-jacent, et typiquement des classes d'objets. La figure 7.6 illustre un composant composite avec deux sous-composants primitifs.

– *Dynamicité et contrôle* : Un composant Fractal est une entité présente à l'exécution. Il peut être manipulé pendant son exécution grâce au mécanisme de *contrôle*. Le contrôle permet d'ajouter, d'écarter, de démarrer, de stopper, de lier, de délier ou de mettre à jour l'état des composants. Ces primitives de contrôle sont effectuées à travers la membrane d'un composant en utilisant les interfaces de contrôle : `BindingController` pour la gestion des liaisons et `ContentController` pour la gestion des sous-composants d'un composite.

Un composant Fractal exhibe deux types d'interfaces pour interagir avec d'autres composants : les *interfaces clientes* et les *interfaces serveurs*. La figure 7.6 montre un composant composite avec deux interfaces client et une interface serveur. Les interfaces clientes spécifient les fonctionnalités requises par le composant et les interfaces serveur représentent les fonctionnalités offertes par celui-ci. La composition de deux composants se concrétise par l'établissement d'une liaison entre l'interface cliente d'un composant et l'interface serveur de l'autre. Ces deux interfaces doivent être du même type. Les interactions sont initiées par l'interface cliente en appelant par

exemple une méthode sur l'interface serveur.

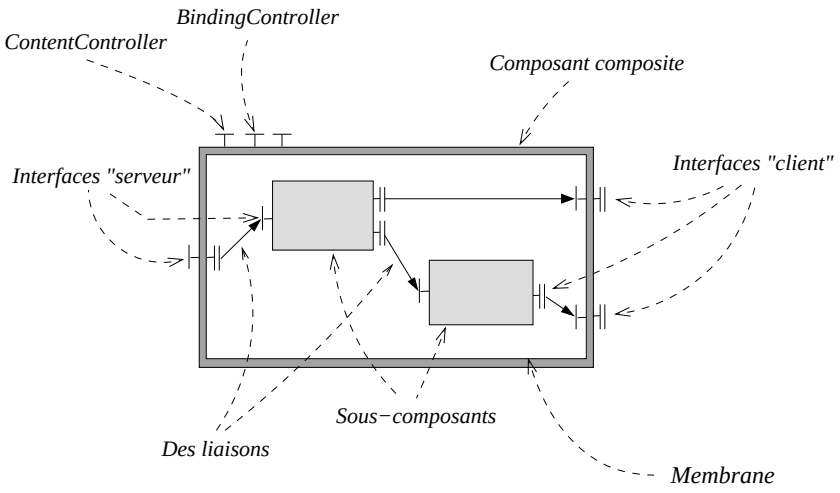


FIG. 7.6 – Le modèle de composant Fractal

Méta-données des composants Fractal : Un composant Fractal est considéré comme une source de méta-données. Les informations contenues par un tel composant au run-time sont : l'identificateur de ce composant, l'état de son cycle de vie (started, stopped, etc), les identificateurs de ses sous-composants (si c'est un composite), les composants liés à celui-ci et ses interfaces fonctionnelles et de contrôle.

Les composants Fractal sont spécifiés et instanciés programmatiquement en utilisant l'API Fractal ou en utilisant un langage de description d'architecture ADL, qui permet de spécifier statiquement les *templates de composants*. Ce langage utilisant XML comme langage de description, décrit les interfaces des composants (noms et signatures), les sous-composants identifiés par des noms et les liaisons entre les différents composants, les valeurs initiales (états) des composants et les implantations (par exemple des noms des classes Java). D'autres informations peuvent être attachées à chaque élément de l'ADL. Les définitions des templates sont identifiées par des noms uniques et peuvent être spécialisées par héritage dans le but de les factoriser, et donc de pouvoir les réutiliser. Plus généralement, les templates de composants dans l'ADL sont aussi considérées comme des sources de méta-données. A l'égard des composants en cours d'exécution, les informations contenues dans les templates de composants sont statiques.

Un gisement (repository ou une source) de templates de composants Fractal contient à la fois les informations décrites selon l'ADL et le code utile pour l'implantation des composants primitifs. Il contient aussi d'autres informations non structurées comme la Javadoc relative à la signature des interfaces.

Les modèles de composants réflexifs comme Fractal offrent des vues sur leurs contenus d'une manière dynamique ou statique. Cependant, ces vues restent à l'échelle d'un seul composant. Il serait intéressant d'offrir un système qui permette de construire des vues en considérant la totalité du système. Ces vues sont construites en se basant sur ce que l'on a appelé *introspection associative* , i.e, l'interrogation de tout le système en utilisant l'introspection comme mécanisme d'accès aux méta-données.

7.2.2 Scénarios

Dans les deux scénarios ci-dessous, nous considérons non seulement l'équipe des développeurs responsable du développement et de l'assemblage de composants réutilisables, mais aussi d'autres acteurs ayant besoin de manipuler les systèmes en cours d'exécution, tels que les administrateurs des systèmes. Cette section présente deux scénarios qui montrent la portée et le besoin de l'introspection associative dans les systèmes réflexifs.

Étude de cas : on considère une architecture simplifiée d'un routeur IP, comme une composition de composants Fractal illustrée par la figure 7.7. Le composant *router C2* est lié au composant *protocol stack* à travers deux interfaces fournies par le composant *C1* : l'interface de contrôle Internet et l'interface de connexion TCP/IP. Dans cette configuration, *C1* est un composant composite qui contient deux composants : *C11* qui implante le protocole ICMP et fournit l'interface de contrôle Internet et le composant *C12* qui implante la pile de protocole TCP/IP et fournit l'interface de connection TCP/IP qui, elle aussi, est fournie par *C1*. *C12* est à son tour un composant composite contenant deux composants : *C122* qui implante le protocole TCP/IP et le composant de filtrage *C122*. Les deux composants *C121* et *C122* sont reliés l'un à l'autre et permettent d'offrir l'interface de connexion TCP/IP. *C121* est considéré comme un proxy de *C122* dans la mesure où il filtre les connexions ouvertes par *C122*. *C11* et *C12* sont reliés aux composants des protocoles de bas niveaux dans le but d'émettre et de recevoir des paquets sur le réseau. On considère que la majorité des routeurs du système ont la même architecture logicielle.

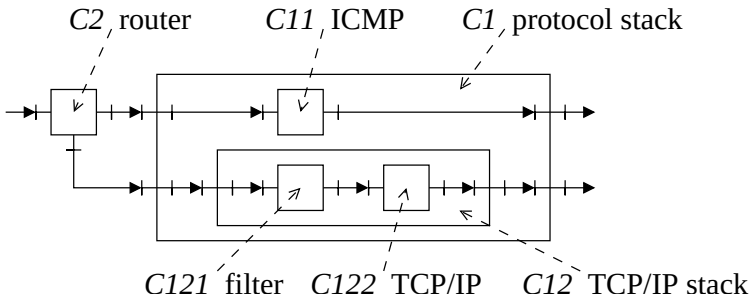


FIG. 7.7 – Exemple d'une architecture d'un routeur

1. **Scénario 1 : Identifier les composants en cours d'exécution qui fournissent le service routeur**

   L'administrateur a besoin d'interagir avec chacun des routeurs du système pour collecter des informations sur les statistiques du trafic des paquets. Chaque composant *router* d'un routeur (voir figure 7.7) permet d'accéder à ces informations à travers son interface fournie. L'administrateur doit donc rechercher et identifier l'ensemble des composants, déjà déployés dans le réseau, qui offrent l'interface *router*. En utilisant les identificateurs de ces composants, il interagit avec chacun de ces composants routeur pour collecter les informations désirées. Ce service de requêtes est proche du service de courtage dans les systèmes à objets tel qu'il est spécifié dans les services de courtage dans ODP [86], le service de courtage dans CORBA ou encore dans les protocoles de gestion des architectures tels que JMX (Java Management eXtension) [163]. Dans le contexte de courtage des composants hétérogènes stockés dans un repository, [82] a comparé le courtage dans les systèmes à

composants au courtage dans les systèmes à objets pour proposer une architecture d'un système de trading dédié à la recherche de composants. Cependant, dans notre contexte, le service de requêtes doit offrir des fonctionnalités de courtage qui permettent d'interroger des systèmes à composants dynamiquement, i.e, en cours d'exécution.

2. **Scénario 2 : Remplacer le composant de filtrage de paquets**
   Suite à une attaque de déni de service, l'administrateur responsable de la sécurité du réseau a décidé de mettre à jour la politique de sécurité. Il doit donc remplacer tous les composants de filtrage des composants routeurs du réseau. Pour se faire, il doit effectuer les trois tâches suivantes : (i) identifier les composants de filtrage à remplacer, (ii) puis, retrouver depuis le repository les composants implantant ce type de composants, (iii) et enfin, instancier le composant pour effectuer effectivement le remplacement.

   (a) **Identifier des composants** : Les composants de filtrage ne sont pas seulement utilisés dans les routeurs, mais aussi dans les serveurs d'application et les serveurs Web du système, qui eux ne sont pas concernés par la politique de sécurité du réseau. Dans l'identification des composants de filtrage, l'administrateur doit donc limiter sa recherche aux composants de filtrage appartenant aux routeurs. Pour se faire, il doit pouvoir décrire de manière non ambigüe ce type de composant comme "un composant primitif lié à une interface fournie par un composant composite qui à son tour est contenu dans un composant composite qui est relié au composant router". Ce type de requête prend en compte les relations de compositions, typiquement la composition hiérarchique, et les liaisons dans le modèle de composants Fractal.
   Cette fonctionnalité peut être utilisée pour d'autres besoins, par exemple dans le cas où l'administrateur détecte qu'un composant composite stack TCP/IP produit des erreurs ou encore consomme beaucoup de ressources (telle que la mémoire), sans pouvoir identifier lequel de ses sous-composants est responsable de ces erreurs. En effet, dans ce cas, l'administrateur doit identifier ce type de composite (selon les relations de composition) avant de le remplacer par un autre.

   (b) **Rechercher les composants depuis des gisements de stockage** : Dans le scénario précédent, afin de remplacer les composants Fractal, l'administrateur utilise les interfaces de réflexion offertes par les composants pour arrêter, instancier les nouveaux composants, mettre à jour les liaisons entre les différents composants et enfin initialiser les nouveaux composants. Il faut dire que d'une manière générale, ce problème de reconfiguration est complexe car beaucoup de composants sont dotés d'un état, et ne peuvent donc pas être facilement remplacés dynamiquement. Cependant, le problème de reconfiguration n'est pas à la portée du service de requêtes que nous proposons, et nous considérons que les paquets de filtrage n'ont pas d'état.
   Le problème qui demeure est la construction des requêtes qui portent sur le repository en se basant sur les méta-informations offertes par les interfaces d'introspection des composants à remplacer, identifiés lors de la phase précédente. Le système de recherche de composants doit permettre l'expression de requêtes à la fois sur les systèmes qui sont en cours d'exécution et sur les sources contenant des implantations (template) des composants.

7.2.3 Recherche et stockage des composants : état de l'art

Comme l'un des objectifs du développement orienté-composants est la réutilisabilité, la recherche des composants logiciels est le centre d'intérêt de plusieurs travaux de recherche. Le problème généralement adressé dans ces travaux est la recherche de composants selon des critères définissant des besoins à partir des librairies ou d'un repository de composants. Dans cette section, nous classifions ces recherches dans deux catégories. La première catégorie regroupe des travaux généraux, notamment des études théoriques abondant le problème de *comparaison de composants*[2]. Ces travaux considèrent un seul modèle de composant homogène avec un minimum de spécifications. La deuxième catégorie représente les travaux qui se sont intéressés à la *recherche et au stockage des composants* à partir des gisements de composants. Dans ce cas, on se base sur les spécifications et les descriptions externes des composants, en plus des informations utilisées dans les techniques de comparaison de composants, comme dans la première catégorie de travaux.

7.2.3.1 Comparaison de composants

La *comparaison des composants* concerne plusieurs niveaux de spécification de composants : la signature, la sémantique et le comportement des composants. Typiquement, ces spécifications sont décrites à travers les langages de description des architectures (ADL). La comparaison des signatures [150, 192, 83] utilise le typage comme critère de comparaison des composants. La signature d'un composant est définie par l'ensemble des signatures de ses interfaces. Les signatures de ses interfaces sont à leur tour décrites par l'ensemble des signatures de leurs opérations. Par exemple, la signature d'une interface peut correspondre à la signature d'une interface Java, comme c'est le cas dans le modèle de composant Fractal pour Java. La comparaison des signatures des composants se fait donc par comparaison récursive des signatures de leurs interfaces respectives. Plus exactement, deux composants sont égaux du point de vue signature s'ils ont exactement les mêmes signatures d'interfaces, et récursivement, les signatures des opérations qu'elles offrent. Il existe un autre type de comparaison appelé comparaison relaxée[3] qui est plus tolérant que le précédent. En effet, dans ce type de comparaison, par exemple, deux signatures de méthodes peuvent être égales même si elles retournent des types différents.

La *comparaison sémantique et comportementale* [76, 89, 193], appelée comparaison des spécifications, se base sur des descriptions plus abstraites des composants et de leurs interfaces. Ce type de spécifications formelles inclut par exemple les assertions (pré et post conditions et invariants) ou des modèles de logique temporelle [146].

7.2.3.2 Recherche et stockage des composants

L'objectif principal des travaux pour la recherche des composants à partir des gisements est la construction des systèmes par assemblage des composants existants. Ces travaux prennent en compte l'hétérogénéité des composants. Cette hétérogénéité concerne aussi bien l'hétérogénéité des modèles de composants dans laquelle ils sont développés que l'hétérogénéité des spécifications et l'hétérogénéité des domaines d'application considérés (banque, médecine, etc). Ces composants sont souvent désignés par COTS-C (Commercial Off The Shelf Components). Deux questions majeures sont alors posées : (i) comment représenter les composants dans les gisements ? (ii) comment exprimer et traiter les requêtes posées ?
Ces deux questions relèvent du problème d'intégration de données d'une manière générale. En effet, une représentation des composants qui prend en compte l'hétérogénéité des modèles de

[2] Traduction de "*component matching*"
[3] Traduction de *"relaxed matching"*

composants et de spécification eu égard à leurs domaines d'application est nécessaire comme dans les systèmes de médiation. Plusieurs classifications de la représentation des informations des composants ont été proposées [160, 117, 96]. La majorité de celles-ci distingue : des représentations selon *des mots clés (keyword sets)*, des représentations selon des *facettes (facets)* et des représentations selon des *specifications en langage naturel.*

Représentation selon des mots clés : Des mots clés sont définis pour chaque composant selon un ensemble de mots clés relatifs aux domaines d'application [118]. Dans ce type de système, les requêtes sont aussi exprimées simplement par des mots clés. Le pouvoir d'expression et de sémantique de ces requêtes est donc très limité.

Représentation selon des facettes : La représentation des composants par des facettes sont des représentations plus élaborées selon un schéma (ou une ontologie) bien déterminé [139, 194, 115, 26]. Une facette peut décrire des caractéristiques fonctionnelles des composants, des caractéristiques de qualité de service, les relations qui existent entre les différents composants, des caractéristiques relatives aux domaines d'application sous-jacents, etc. Les requêtes sont exprimées selon le modèle de données utilisé pour la description du schéma décrivant les différentes facettes, tel que SQL. Cependant, il est difficile d'intégrer les différentes facettes et de prendre en compte l'évolution des domaines d'application sous-jacents. Des systèmes de médiation des composants sont apparus pour prendre en compte les aspects d'intégration et de recherche de composants [26, 27].

Représentation selon des spécifications en langage naturel : Dans cette variété de systèmes, les requêtes sont directement exprimées en langage naturel [160, 64]. Ces requêtes sont riches en sémantique et sont très pratiques vis-à-vis des utilisateurs. Cependant, les requêtes en langage naturel demandent l'utilisation de techniques de manipulation complexes, telles que des analyses morpho-lexicales et des ressources lexicales comme les dictionnaires et les thésaurus.

7.2.3.3 Analyse

Les travaux de recherche sur la recherche des composants que nous avons décrits dans la section ci-dessus considèrent des spécifications de composants à plusieurs niveaux : structurel, sémantique, comportemental, descriptions ontologiques, descriptions en langage naturel, etc. Notons que tous ces niveaux de description sont complémentaires. Notre service de requêtes doit donc pouvoir permettre de considérer et de combiner ces différentes représentations afin d'améliorer le pouvoir d'expression des requêtes. Par exemple, un de nos besoins concrets est de combiner les spécifications ADL des composants stockés dans des fichiers et les spécifications en langage naturel correspondant à la Javadoc. Plus encore, puisque dans notre contexte, nous considérons le modèle de composant Fractal comme le seul modèle utilisé, notre service de requêtes peut donc utiliser aussi les techniques de comparaison de composants dans le traitement des requêtes.

En considérant à la fois les systèmes en cours d'exécution et des implantations de composants dans un repository, la portée de notre système est plus large que les systèmes existants dont l'objectif est de rechercher des composants à partir des gisements pour les réutiliser par assemblage. Ces systèmes ne prennent pas en compte la recherche de composants dans des systèmes qui tournent, besoin que nous avons motivé à travers les scénarios dans le contexte de l'administration et de la supervision des systèmes à large échelle. Plus précisément, ces scénarios

ont montré le besoin d'un système de recherche largement réparti qui permet de construire des vues sur le système. Les requêtes associées à ces vues doivent pouvoir prendre en compte les caractéristiques fonctionnelles des composants, les relations de composition, etc.

7.2.4 Proposition : un service de requêtes général

Le service de requêtes que nous proposons dans cette section peut être utilisé par les différents acteurs, à savoir les équipes de développement, les administrateurs, etc. L'objectif de ce service est de poser des requêtes pour collecter les informations à partir du système introspecté et à partir des gisements de stockage de templates de composants. En considérant le système introspecté et le gisement de stockage comme des sources de données, le problème consiste alors à interroger ces deux sources de données d'une manière uniforme, qui est un des problèmes pris en compte par les systèmes d'intégration de données.

7.2.5 Architecture

La figure 7.8 représente l'architecture du service de requêtes. Cette architecture est définie par une hiérarchie de composants logiciels répartis : *Query Service*, *Repository Query Service* et le composant *System Query service*.

Le composant Query Service est le point d'entrée pour l'interrogation utilisé par les différents acteurs. Ce composant joue le rôle de composant intégrateur en permettant d'homogénéiser l'accès au système introspecté et au gisement des composants par délégation à deux autres composants : Repository Query service et System Query Service. Le composant Repository Query service permet de rechercher les composants stockés dans des sources : ce composant est donc fonctionnellement comparable aux travaux existants dans le domaine de la recherche et du stockage des composants sus-cités. Le System Query Service interagit avec les composants Local System Query Service, qui sont localement directement liés aux interfaces d'introspection des composants du système pour interroger leurs méta-données. Le Local System Query *Service* peut récursivement introspecter les liaisons et le contenu des différents composants (voir section 7.2.1) en utilisant les interfaces d'introspection Fractal, en commençant à partir du composant auquel il est lié. Ce composant couvre donc toutes les requêtes portant sur le sous-système représenté par le composant introspecté. Notons que le placement, le choix ainsi que le nombre de composants Local System Query Service dépend du nombre de sites introspectés et de la sémantique des différents sites.

L'architecture de notre service suit l'architecture des systèmes d'intégration. En effet, tous les composants de ce service peuvent être considérés comme des spécialisations de domaines de données. Les composants Query Service et Repository Query Service sont des types de domaines de données non primitifs et les composants Local Query Service sont des types de domaines de données primitifs. Il est aussi important de souligner que cette architecture prend en compte les propriétés de répartition et de dynamicité des systèmes : les composants Local System Query Service peuvent être dynamiquement déployés pour s'adapter à l'évolution du système.

7.2.6 Expression des requêtes

Comme nous l'avons décrit à travers les scénarios précédents, les requêtes doivent pouvoir exprimer les requêtes du style "rechercher les composants d'un gisement qui sont similaires à un composant donné récupéré à partir du système en cours d'exécution". Dans les sections suivantes, nous montrons que les requêtes introspectives sur les composants Fractal sont exprimées à la

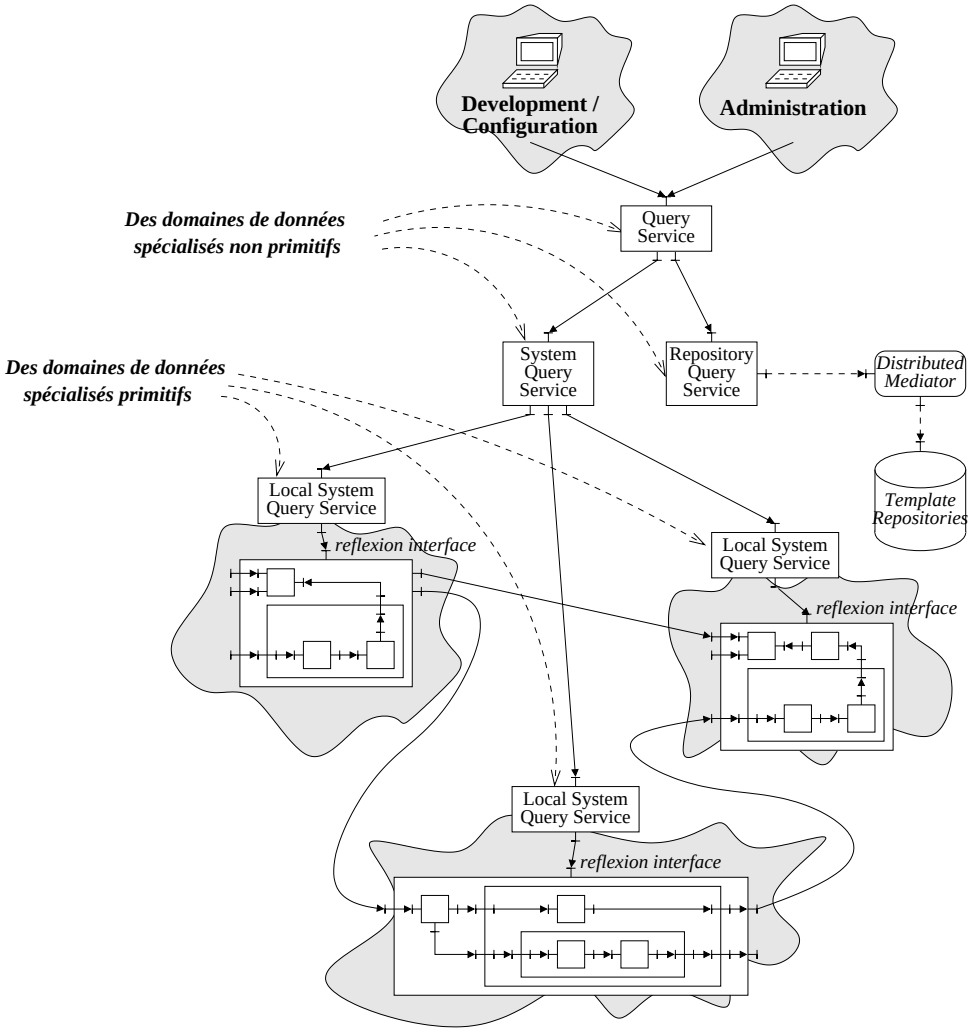


FIG. 7.8 – Architecture Générale

manière des requêtes dans les langages pour données semi-structurées. Comme les gisements de template représentent aussi les informations structurelles, i.e, les relations de composition et de liaison mais avec plus d'informations détaillées (voir section 7.2.1), les requêtes qui portent sur ces templates de composants sont décrites comme les requêtes dans les requêtes semi-structurées. Les sections suivantes montrent le type de requêtes exprimées au niveau de chacun des composants de notre service de requêtes, accompagnées d'exemples illustratifs.

7.2.6.1 System Query Service

Les liaisons et les relations de composition entre les composants Fractal peuvent être représentées par un graphe orienté labélisé. Par exemple, la figure 7.9 illustre partiellement un graphe décrivant une configuration de composition relative à l'architecture du composant routeur, représenté par la figure 7.7. Les noeuds représentent les éléments architecturaux, i.e, les composants, les liaisons, les interfaces, les signatures d'interfaces, etc. Les feuilles représentent des valeurs primitives, tels que les noms des interfaces et les méthodes ainsi que les types des arguments de chacune des méthodes. Cette représentation de données est la même que celle du modèle de données semi-structurées OEM [137] (voir section 9.1.3.1) : comme les composants réflexifs, les données semi-structurées s'auto-décrivent. Exprimer les requêtes d'introspection associatives sur les composants revient donc à exprimer des requêtes sur les données semi-structurées. Les relations de dépendance (topologie) et les fonctionnalités des composants peuvent être décrites par des expressions de chemins sur le graphe de données, avec éventuellement des expressions régulières et des prédicats, exactement comme dans les langages de requêtes semi-structurées [43, 111, 49, 39, 176]. Par exemple, la requête qui permet de rechercher les identificateurs des composants ICMP des routeurs se traduit par "rechercher le composant C qui offre une interface de nom "icmp", qui est contenue dans un composant composite qui lui est lié à un autre composant qui offre une interface de nom "router"". L'expression de cette requête est :

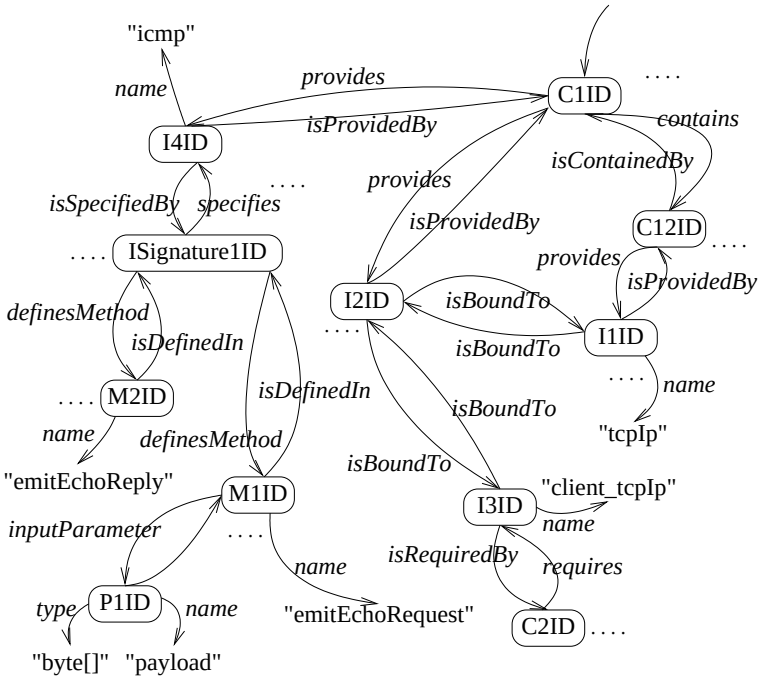


FIG. 7.9 – Exemple d'une configuration de composition Fractal

```
select C
where *.C[provides.name="icmp"]
.isContainedIn.provides.isBoundTo
.isRequiredBy.provides.name="router"
```

Dans l'expression de requête ci-dessus, on peut dire que le type de comparaison utilisé est de type relaxé car les critères de sélection se contentent seulement des noms des interfaces. Cependant, il est possible d'affiner la comparaison en considérant aussi la signature des interfaces, etc.

7.2.6.2 Repository Query Service

Comme dans les systèmes d'intégration de données, les gisements de templates de composants peuvent avoir différents modèles de données et schémas dont la sémantique dépend aussi du domaine d'application, i.e, des composants dédiés pour les applications réseaux, des composants dédiés aux applications relatives aux banques, etc. Nous utilisons les domaines de données comme médiateurs pour intégrer ces différentes descriptions de composants, comme dans [27, 26]. Le composant Repository Query Service a comme rôle de transformer les requêtes déléguées par le composant System Query Service dans le langage de requêtes des médiateurs. Notons que si ces médiateurs sont directement implantés en utilisant les domaines de données sans ajouter une couche langage, alors ce composant ne fait que déléguer les requêtes vers ces domaines de données.

Un exemple de requête type prise en charge par ce composant est : " rechercher les templates de composants qui offrent une interface de nom "icmp" ainsi que toutes les templates qui les étendent". Notons que cette requête utilise la relation " extends" entre les composants Fractal d'un gisement d'ADL et qu'elle n'est plus valable une fois les composants instanciés et déployés dans le système en cours d'exécution. L'expression de cette requête est :

```
select T
where *.T.(extends)*
.provides.name="icmp"
```

Comme autre exemple, considérons des spécifications en langage naturel qui consistent à associer à chaque élément architectural (templates, signature d'interfaces, liaisons, etc.) du gisement un attribut de description. Ces spécifications sont typiquement extraites de la Javadoc. L'expression de requête suivante permet de répondre au besoin "les templates qui fournissent des interfaces dont la documentation de la signature contient un mot qui est sémantiquement proche du mot "creates" quelque soit le langage dans lequel cette documentation est écrite, en utilisant l'opérateur relatif au langage naturel "`containsWordCloseTo`".

```
select T
where containsWordCloseTo("creates",
*.T.provides.isSpecifiedBy)
```

Cette requête peut retourner les templates fournissant des interfaces contenant le mot "created" par analyse morphologique, qui permet de comparer les différentes formes de mot "create".

Comme ce mot est synonyme des mots "instanciate", "créer" et "instancier" en langue française, cette requête retourne aussi les templates fournissant des interfaces telle que la documentation contient ces mots. Bien sûr, la mise en oeuvre de l'analyse morphologique nécessite beaucoup de ressources tels que les thérausus pour améliorer la qualité de réponse des requêtes.

7.2.6.3 Query Service

Les requêtes posées sur le composant Query Service par les développeurs et les administrateurs sont des requêtes globales qui sont décomposées en sous-requêtes déléguées aux composants System Query service et Repository Query Service, comme dans un système de médiation à base de domaines (voir section 6.2).

La comparaison des signatures est directement exprimée à travers les opérateurs de calcul classiques de requêtes tels que : "<" et "=", comme illustré par les exemples de requêtes ci-dessus. Pour les requêtes en langage naturel, des opérateurs comme `containsWordCloseTo` doivent être implantés.

La comparaison sémantique et comportementale entre composants ne peut pas être prise en compte par de simples opérateurs de calcul mais nécessite plus de travail. En effet, pour vérifier si deux spécifications de comportement sont égales, le médiateur récupère d'abord toutes les informations à partir des gisements avant d'effectuer la comparaison localement.

Les opérateurs de comparaison de substitutionnalité [83] ("< :") sont des opérateurs de haut niveau qui se déclinent en un ensemble d'opérateurs de calcul classiques.

La prise en compte de ces différents opérateurs est possible en étendant notre canevas d'expression de requêtes.

7.2.6.4 Traitement de requêtes

Les sous-requêtes déléguées aux domaines de données représentant les gisements de données sont traités en se basant sur les algorithmes classiques de traitement de requêtes relatifs aux domaines de données. Cependant, les requêtes posées sur les System Query Service nécessitent des algorithmes plus adaptés car les composants Local System Query Service sont répartis avec des liaisons entre les différents sous-systèmes. En effet, l'introspection se fait récursivement sur les composants en parcourant les liaisons et les contenus des composants. Comme déjà évoqué dans la section 7.2.6.1, on peut dire que ce parcours porte sur un graphe de données semi-structurées, et par conséquent cette problématique est similaire à la problématique de la récursion structurelle [34]. On peut alors utiliser l'algorithme d'optimisation et d'évaluation des requêtes sur les bases de données semi-structurées réparties proposé dans [159], qui relève de la récursion structurelle. Plus précisément, l'optimisation de requêtes dans notre cas signifie que le nombre d'interactions entre les composants System Query Service et Local Query Service pour l'optimisation des requêtes doit être constant et que la taille des données résultats transférées doit seulement dépendre de la taille des résultats et du nombre total de composants introspectés, et non pas de la taille (le nombre de composants) de tout le système.

7.2.7 Synthèse

Dans cette deuxième validation, nous avons proposé une architecture d'un service de requêtes sur les architectures à composants réflexifs. Nous avons aussi montré que cette architecture peut être implantée en utilisant les domaines de données et ses différents canevas. Des spécialisations de domaines de données permettent d'implanter les différents composants du système ,i.e, Query Service, Repository Query Service, System Query Service et Local System Query Service. Les requêtes utilisées par ce système permettent différents niveaux de comparaison entre les composants, notamment aux niveaux signature ou type, comportemental et sémantique et au niveau langage naturel. Le fait que notre canevas d'expression de requêtes soit extensible permet de prendre en compte les différents opérateurs qui expriment ces niveaux de comparaison.

Par ailleurs, cette proposition motive aussi le besoin des architectures à composants de services de requêtes permettant d'introspecter l'état du système. Les architectures à composants doivent prendre en compte l'intégration de tels services. On pense que ce type de requêtes peut aussi être utilisé pour des tâches de configuration ou de reconfiguration, en exploitant cette fois-ci les interfaces d'intercession au lieu des interfaces d'introspection. Cependant, ce service ne constitue qu'une brique parmi d'autres briques utiles pour le déploiement, l'administration, la gestion des services, etc.

7.3 Conclusion

En plus de l'implantation MEDOR, ce chapitre montre deux contextes d'utilisation de notre intergiciel. La première validation montre surtout l'adaptabilité des deux canevas d'expression et d'optimisation de requêtes en permettant d'implanter les deux langages de requêtes des deux standards de persistance Java JDO QL et EJB QL. Cette validation peut être perçue comme une implantation d'un domaine de données primitif qui en plus des aspects de traitement de requêtes, prend aussi en charge les aspects de mise à jour et de cohérence des données.

La portée et l'adaptabilité de notre intergiciel d'intégration de données est démontré dans l'utilisation des domaines de données ainsi que les canevas associés pour la recherche des composants. Nous avons proposée un système de recherche des composants à partir des gisements de composants et des systèmes en cours d'exécution. Pour l'implantation de ce système, il faut au préalable implanter les domaines de données.

Nous envisageons d'utiliser l'implantation Java de Fractal Julia [129] pour l'implantation des domaines de données. Un domaine de données est directement modélisable sous-forme d'un composants Fractal. Les interfaces *import* et *export* des domaines de données sont respectivement l'équivalent des interfaces *cliente* et *serveur* et qui sont de type `ImportExport`. Le choix de Fractal est aussi motivé par ces deux propriétés de *dynamicité* et de *partage* ainsi que par son ADL :

- *Dynamicité* : Comme déjà décrit dans la section 7.2.1, les composants Fractal sont contrôlable dynamiquement grâce aux interfaces de contrôle. Dans un système d'intégration à base de domaines de données, ceci se traduit par le pouvoir d'ajout dynamique de nouvelles vues intégrées ou le déploiement de nouveaux domaines de données. Ces domaines peuvent être primitifs, dans le cas où l'on rajoute de nouvelles sources, ou encore non primitifs. Ceci permet la manipulation de la topologie de l'architecture et la création de liens dynamiques

entre les composants du système d'intégration. Les interfaces de contrôle offertes par les composants Fractal permettent l'implantation des interfaces de contrôle des domaines de données, et plus particulièrement l'interface d'évolution.

- *Partage :* Cette propriété permet, à deux ou plusieurs composants, le partage des ressources qui sont aussi réifiées par des composants, dans un système. Ces ressources peuvent être physiques, comme le CPU et la mémoire, ou virtuelles telles que les données. Dans le dernier cas, la ressource "données" peut être sous forme de bases de données. La notion de partage existe naturellement dans le canevas de domaines de données. En effet, deux domaines de données partagent un même espace de données à travers un autre domaines de données.

- *ADL* : L'utilisation de Fractal pour le développement des domaines de données permet d'automatiser la génération des systèmes d'intégration de données à base de domaines en s'appuyant sur l'ADL. On peut alors spécifier un langage de description des domaines au dessus de l' ADL Fractal. Ce langage va permettre de spécifier notamment les différents domaines de données de l'architecture du système d'intégration, les vues intégrées initiales partagées entre ces domaines ainsi que leurs structures (types). Dans la spécification des domaines de données, un type de wrapper lui est associé (JDBC, ODBC, etc) ainsi que les vues primitives importées.

Chapitre 8

Conclusion

La vérité de demain se nourrit par l'erreur d'hier.

Antoine de Saint–Exupéry

Dans un système d'intégration de données, on distingue l'architecture logicielle du système, i.e, les composants logiciels le constituant, des aspects de l'intégration de données, tels que le modèle de données, le langage de requêtes, l'optimisation des requêtes. L'adaptabilité[1] d'un système d'intégration de données se mesure par la capacité de son architecture à supporter divers modèles, langages et techniques relatifs aux différents aspects de l'intégration. En se plaçant à l'intersection des deux domaines, les architectures logicielles et les systèmes d'intégration de données, nous avons revisité les aspects des systèmes d'intégration de données et nous avons proposé un intergiciel d'intégration de données adaptable sous forme de canevas logiciels. Cet intergiciel ne résout pas tous les problèmes de l'intégration de données ; cependant, il offre des fonctionnalités minimales pour la définition et la construction d'un système d'intégration sous ses différents aspects. Comme dans beaucoup d'autres travaux autour des systèmes de bases de données adaptables [63, 51, 55], nous avons aussi constaté qu'une analyse fine de l'intégration de données et des systèmes d'intégration de données a été indispensable pour asseoir l'adaptabilité de l'intergiciel. Cette analyse n'est pas toujours facile vue la disparité des travaux de recherche dans ce domaine. Les aspects de l'adaptabilité de notre intergiciel sont véhiculés à travers les différents canavas qui constituent le coeur de notre travail. La section suivante résume le bilan de ce travail.

8.1 Bilan de la thèse

L'un des résultats les plus importants de cette thèse est la proposition d'une nouvelle approche pour la construction des systèmes d'intégration de données basée sur le paradigme de la composition logicielle [15, 14]. Les données des sources et les données intégrées sont représentées par une abstraction logicielle qui est un domaine de données. Un système d'intégration de données est défini par une composition de domaines de données. Une composition de domaines est décrite par le mécanisme d'importation et d'exportation des vues entre les différents domaines

[1] On peut aussi dire que notre intergiciel est capable d'avoir plusieurs personalités.

selon l'approche GAV. Dans un sens, nous avons donc ramené la problématique d'intégration de données à une problématique de composition logicielle et montré que les architectures à composants permettent effectivement de maîtriser la complexité des systèmes d'intégration de données. Dans la spécification des interfaces et des sous-composants d'un domaine de données, nous avons été guidés par le fait qu'il faut être adaptable aux différents aspects d'intégration de données : modèle de données sous-jacent, langages de requêtes et techniques d'optimisation de requêtes.

Le modèle de données est un modèle de données commun qui permet de représenter les données des sources et qui est utilisé comme modèle d'échange entre domaines de données. Une analyse des modèles de données classiques, relationnels, objets et semi-structurés, utilisés dans les systèmes d'intégration de données, nous a permis de constater que l'approche des travaux récents autour des systèmes de types pour données semi-structurées [79, 78, 35, 114] est très proche des travaux sur la construction des types dans les systèmes d'intégration de données à objets [108, 99, 101]. En effet, ces deux types de travaux utilisent l'inférence de type pour produire de nouveaux types d'objets à partir d'une composition d'autres types. Nous avons alors proposé un modèle de données hybride qui regroupe les concepts des différents modèles de données, avec un système de type flexible qui permet l'inférence de nouveaux types. Ce modèle prend aussi en compte les mécanismes indispensables dans un modèle de données d'intégration, notamment l'identification, la gestion des références et l'agrégation ou la construction des objets complexes.

Les interactions avec les domaines de données et la composition des domaines ne dépendent pas d'un langage de requête particulier. Conformément à notre objectif, les domaines de données offrent des interfaces d'interrogation génériques. Nous avons proposé un canevas d'expression de requêtes adaptable, qui permet aussi bien de considérer d'une manière uniforme des requêtes simples "à la SQL" que des requêtes complexes avec des expressions de chemins "à la XQuery". Une requête est décrite par un arbre de compositions d'expressions. En examinant les différents langages, nous avons spécifié trois types d'expressions : les expressions de calculs, les expressions algébriques et les expressions de chemins, suffisantes pour la description des requêtes. En plus de l'interrogation, ce canevas est utilisé pour la définition de nouvelles vues exportées d'un domaine (de nouveaux types d'objets) à partir d'autres vues importées d'autres domaines en exploitant la flexibilité du typage du modèle de données. Ce canevas prend également en compte la compilation statique des expressions. Ceci permet de vérifier la concordance des types d'expressions composés d'une requête, et aussi d'inférer statiquement les types des résultats des différentes expressions. Un autre avantage de cette approche canevas est l'extensibilité du pouvoir d'expression des requêtes. Ceci permet de considérer de nouveaux types d'opérateurs d'intégration pour la prise en compte de nouveaux cas d'intégration (des unités de mesure par exemple) qui dépendent généralement du domaine d'application cible.

La composition des domaines de données se traduit par le partage des ressources (données et traitement) entre les différents domaines. Cette composition, qui peut former diverses topologies, peut inclure des domaines de données non primitifs et des domaines primitifs avec des capacités d'évaluation et des restrictions d'accès différentes. Dans l'analyse des techniques d'optimisation, nous avons remarqué que chaque système d'intégration utilise différents formalismes de description des capacités, différentes stratégies de recherche, avec des approches de modélisation de coût différentes également. L'objectif visé par le canevas d'optimisation de requêtes est de prendre en compte les différents paramètres et/ou contraintes de l'optimisation de requêtes et de pouvoir supporter diverses stratégies de recherche. Cet objectif est le même que celui visé par une lignée de travaux de recherche pour la construction d'optimiseurs extensibles pour les SGBD [72, 141, 68, 69, 21, 113, 103, 59, 134, 93]. Comme dans ces travaux, nous avons alors utilisé

les deux ingrédients que sont les règles et la séparation de l'espace logique de l'espace physique. Dans la spécification des interfaces d'un domaine de données, celui-ci exhibe des méthodes qui permettent d'exporter l'espace physique et l'espace logique d'une requête. Dans l'architecture du canevas d'optimisation, nous avons séparé les composants responsables du contrôle des stratégies d'optimisation locales des composants responsables du contrôle des stratégies d'optimisation composites, i.e celles relatives à une composition de domaines. Nous avons alors montré que ces séparations permettent de construire diverses stratégies de recherche (bottom-up et top-down), aussi bien locales que composites. Pour augmenter encore l'adaptabilité de ce canevas, celui-ci est indépendant d'un formalisme de description de capacités des sources et d'un modèle de coût.

Les bénéfices et les avantages effectifs de notre vision d'un integiciel d'intégration de données adaptable a été démontré à travers deux validations dans deux contextes différents.
Une implantation du canevas d'expression de requêtes et du canevas d'optimisation de requêtes a été réalisée dans le projet ObjectWeb MEDOR [126]. MEDOR est utilisé au côté d'autres projets, notamment JORM [125], pour l'implantation des deux standards de persistance d'objets Java JDO et EJB CMP. Ceci a permis d'un coté, d'affiner l'architecture des deux canevas d'expression et d'optimisation de requêtes, et d'un autre coté de démontrer l'adaptabilité de ces canavas [13]. Pour le canevas d'expression de requêtes, son adaptabilité est validée à travers l'implantation des deux langages de requêtes JDO QL et EJB QL. Plusieurs opérateurs de calcul et des opérateurs de navigation ont alors été implantés pour capturer l'expressivité des deux langages. Pour le canevas d'optimisation de requêtes, plusieurs règles de transformation génériques, notamment celles exprimant les heuristiques, des règles de mapping des schémas objets vers les schémas des sources, ainsi que des règles dédiées à la transformation des requêtes dans le langage de sources ont été implantées. Le développement de ce dernier type de règles nous a obligé à prendre en compte les différentes variantes du langage SQL utilisées par chaque produit de base de données. Ce point renforce notre point de vue qui est l'indépendance vis à vis des syntaxes.

Dans un domaine d'application autre que les bases de données, nous avons proposé un système d'introspection et de recherche de composants dans les architectures réflexives à large échelle. Le canevas d'expression de requêtes est alors utilisé pour exprimer les liaisons de dépendance entre les composants d'un système. Ces requêtes se rapprochent des requêtes à la XQuery. Il est alors possible de poser des requêtes qui permettent de récupérer des composants selon les caractéristiques internes aux composants ou selon leurs dépendances avec d'autres composants. Même si ce système n'a pas été implanté, il est considéré comme un résultat important par la communauté des architectures logicielles [16]. D'un autre coté, ceci montre la portée de notre approche et de notre intergiciel.

8.2 Perspectives

Les résultats obtenus à travers ce travail nous laissent entrevoir plusieurs pistes de recherche. Ces dernières sont présentées à travers les sections suivantes.

8.2.1 C'est pas fini ... Approfondir la validation

Nous envisageons d'implanter les domaines de données pour compléter la validation des différents canevas de notre intergiciel. Comme déjà évoqué dans le chapitre précedent, les domaines de données sont directement modélisables sur des composants Fractal [31, 30]. Les propriétés de dynamicité et les fonctionalités de contrôle et de déploiement offertes par ce modèle de composants nous laisse dire que l'implantation Java Julia [129] de ce modèle est idéale pour

l'implantation des domaines. Pour cela, on pourrait prolonger la validation commencée dans le cadre de la gestion de la persistance ObjectWeb.

8.2.2 Généralisation de l'approche

Tout d'abord, nous pensons que la notion de domaine peut être généralisée par rapport aux deux aspects suivants :

- *Modèle de données :* Même si nous avons proposé un modèle de données générique sous-jacent au domaine, la définition et la spécification de celui-ci peuvent être appliquées à n'importe quel modèle de données. Il est donc envisageable d'avoir des domaines avec des modèles de données respectant exactement les modèles objets, XML, etc.

- *Intégration des schémas* : Dans la spécification actuelle des domaines de données, le mapping entre les vues importées et les vues exportées suit l'approche GAV. Dans la mesure où, que ce soit dans l'approche GAV ou dans LAV, le mapping est réalisé par des opérateurs, l'approche d'intégration LAV peut être supportée par les domaines de données. Ceci nous permet ainsi de construire des systèmes d'intégration de données hybrides, dans le sens où certaines compositions de domaines d'un système suivent l'approche LAV, et d'autres suivent l'approche GAV. Le choix de la répartition de ces approches sur les domaines du système peut être motivé par le fait que certaines compositions contiennent des données évoluant constamment, et par conséquent adoptent une approche LAV, et d'autres, n'évoluant pas, utilisent GAV.

8.2.3 Extension du canevas d'optimisation

Les travaux des dernières années dans le domaine d'optimisation de requêtes réparties en général, et dans les systèmes d'intégration de données en particulier, ont démontré que les techniques d'optimisation de requêtes traditionnelles ne sont plus efficaces dans les environnements répartis [190]. Ceci est dû à l'instabilité des ressources de l'environnement sous-jacent, et à la difficulté, voire à l'impossibilité d'avoir un modèle de coût générique stable. Ces travaux ont débouché sur de nouvelles techniques d'évaluation dynamiques qui englobent les deux phases d'optimisation et d'exécution de requêtes. Il serait donc intéressant d'étendre le canevas d'optimisation de requêtes de manière à supporter l'évaluation dynamique des requêtes. A cet effet, nous pensons que QBF (Query Framework Broker) [175], résultat d'un travail de thèse au sein de l'équipe NODS [123], peut être utilisé.

8.2.4 Génération des systèmes d'intégration à base de domaines de données

L'un des objectifs derrière l'utilisation des composants logiciels est de pouvoir générer et déployer les systèmes à partir des langages de description d'architecture (ADL). Il serait donc intéressant d'exploiter cette fonctionnalité pour spécifier un langage de description des architectures des systèmes d'intégration de données à base de domaines. Notons que plusieurs approches permettent la génération des médiateurs à partir des langages de description, comme ODL dans le projet INEEL [135], ou en utilisant des langages plus sophistiqués comme K2MDL [48], qui décrivent l'intégration en combinant la syntaxe d'ODL avec le langage OQL pour spécifier les transformations. L'avantage qu'offre un tel langage est qu'il va permettre de générer à la fois

les domaines de données et les liaisons entre ces différents domaines. De plus, le code généré est sous forme de composant logiciel (non pas de simples objets comme dans d'autres approches) permettant ainsi le contrôle et l'administration des domaines dynamiquement.

8.2.5 Autres contextes d'utilisation

8.2.5.1 Utilisation des domaines pour la collecte et l'intégration de données dans les environnements dynamiques

L'une des conséquences de l'expansion des systèmes informatiques est l'apparition de nouvelles sources de données sous différentes formes (capteurs, terminaux mobiles, compteurs, etc.) qui peuvent évoluer dans des environnements dynamiques (mobiles). Une application intéressante serait d'utiliser les domaines de données pour la collecte et l'intégration de données à partir de ces sources. L'exercice consiste alors à placer (déployer) les domaines de données dynamiquement vis-à-vis de ces sources. Les interfaces de contrôle et d'évolution qu'offrent les domaines de données devraient permettre de placer (déployer) les domaines de données et de faire évoluer la topologie de l'architecture selon la mobilité des sources et les besoins des applications d'intégration. Pour prendre en compte les deux types de sources, actives comme certains capteurs (de type push) et passives (de type pull) comme les compteurs, des spécialisations des interfaces `ImportExport` (actives et passives) des domaines de données peuvent être implantées. Dans le cas des sources actives, les domaines de données vont jouer le rôle de filtres et les interfaces d'evolution offertes par les domaines vont permettre des mises à jour dynamiques de ces filtres.

8.2.5.2 Intégration des données des systèmes autonomes

L'un des objectifs majeurs des travaux de recherche autour de l'architecture logicielle est de maîtriser la complexité des systèmes en construisant des architectures ouvertes et adaptables. L'une des tendances d'actualité les plus prometteuses dans ce domaine est *"l'autonomic computing"*. L'objectif visé par cet axe de recherche est de rendre réflexifs les systèmes dans le sens où ils s'auto-protègent, s'auto-reconfigurent, etc, d'une manière autonome. Le principe de fonctionnement de ces systèmes est qu'ils sont dotés de capacités "sensorielles", permettant d'agir suite à une observation ou un changement dans le système ou l'environnement dans lequel ils évoluent. Nous considérons ces changement comme des méta-données de l'état du système. Comme dans la deuxième validation que nous avons proposée, les composants de ces systèmes sont considérés comme des sources de données. Ces systèmes sont contraints d'intégrer un nombre important de données relatives à l'état actuel du système, mais aussi à l'historique des états précédents. Nous pensons donc que le système d'introspection que nous avons proposé à travers la validation peut être affiné et utilisé pour l'intégration de données des systèmes autonomes.

Chapitre 9

Annexes

9.1 Modèles de données & typage

L'objectif de cette section est de montrer les différences et les similitudes entre les trois modèles de données classiques : le modèle relationnel, le modèle objet et les modèles semi-structurés. Dans la dernière section, nous présentons aussi un modèle hybride qui combine les concepts des modèles de données semi-structurées et du modèle objet.

9.1.1 Le modèle de données relationnel

Les données d'une base de données relationnelle sont représentées sous la forme d'une table structurelle. Le modèle de type est plat, dans le sens où un tuple contient des attributs de types atomiques. On dit que les valeurs que peut prendre un attribut sont des *valeurs d'un domaine*, comme le domaine d'entiers. Les valeurs que peut prendre un attribut sont appelées valeurs d'un *domaine*. Un attribut peut aussi ne pas avoir de valeur. Ce cas peut être interprété en disant que la valeur de l'attribut est *inconnue* ou que la valeur est *inapplicable* pour cet attribut dans ce tuple. Une valeur inconnue est souvent représentée par la valeur spéciale *null*.

9.1.2 Les modèle de données objet

Le modèle de données objet, concurrent directe du modèle relationnel, fournit un pouvoir d'expression très puissant pour modéliser le monde en prenant en compte implicitement les contraintes d'intégrité référentielles. Bien que le modèle objet ODMG (Object Database Management Group) [56] soit désigné comme le standard des bases de données objets, il n'existe pas de modèle objet universel accepté par toute la communauté.

Le modèle ODMG se base sur les concepts objets définis dans la norme OMG [166] et, de plus, introduit des primitives de modélisation propres aux bases de données, comme la description des relations entre les objets. C'est un modèle indépendant d'un langage de programmation particulier grâce à son langage de description du schéma ODL (l'équivalent d'un IDL) compilable. Son système de types repose sur les deux primitives de modélisation : les *objets* et les *littéraux*. La différence entre les deux est qu'un objet a un identificateur alors qu'un littéral n'en a pas. Les objets sont donc *mutables* alors que les littéraux ne le sont pas car ils représentent des valeurs. Toutes les valeurs des objets sont typées conformément aux schémas décrits par le langage ODL (Object Définition Langage). Les types d'objets sont des *extent* du type *class,* quant aux valeurs des littéraux, elles peuvent être de type *primitif* (int, real,etc.), de type *collection* ou de type *struct*. Le type *interface* est une spécification qui définit le comportement abstrait

d'un type objet à travers un ensemble de *méthodes*. Le type *class* encapsule le comportement (les méthodes) et l'état de ce type d'objets. Cet état est défini par les *attributs* et les relations (*relationships*) avec d'autres classes d'objets. La relation *relationship* permet d'établir des relations de contenance entre les objets en construisant des objets complexes. Cette relation exige la définition des associations entre les objets dans les deux sens. C'est-à-dire que, si une classe c1 contient une relation vers une classe c2, alors c2 doit avoir une relation vers c1 (à chaque clause *relationship* est associée une clause *inverse* dans le schéma ODL). Dans les langages orientés-objets, cette relation peut être seulement dans un seul sens dans la mesure où un objet peut avoir des références vers d'autres objets et pas inversement.

Dans le domaine de persistance des objets des langages de programmation, il existe d'autres standards de persistance d'objets, comme dans le cas des containers de persistance transparente des objets Java, à savoir JDO [162](Java Data Object) et EJB CMP [161](Container Managed Persistence), et le modèle de persistance CORBA PSS [124](Persistent State Service). Les modèles de données de JDO et EJB CMP sont des modèles objets supportés par le modèle objet du langage Java, alors que le modèle objet de PSS est un modèle de l'OMG qui est donc indépendant d'un langage de programmation.

9.1.3 Les modèles semi-structurés

Les données semi-structurées sont des données qui s'auto-décrivent. Une source de données semi-structurées contient à la fois les données et la structure de ces données d'une manière non séparée. Cette caractéristique implique que ces données ne sont pas dotées d'un système de types fort, ce qui bouscule l'ordre établi dans la vision des modèles de données structurées. Les principaux aspects des données semi-structurées sont [1, 61] :

- *La structure est irrégulière :* Des données représentant une même information peuvent avoir des formats et des types différents dans une même source de données. Par exemple, une adresse peut être représentée par un élément composé du numéro de la rue, du nom de la rue et du code postal dans certaines parties de la source, et sous forme d'une chaîne de caractères dans d'autres parties.

- *Le schéma n'est pas toujours donné à l'avance :* Contrairement aux bases de données structurées, la notion de schéma peut être antérieure ou postérieure à l'existance des données. La notion d'extension et d'intention, que l'on retrouve dans les bases de données traditionnelles, n'a plus beaucoup de sens. Le schéma peut être alors relativement large du fait qu'il évolue avec les données. Cette évolution n'est pas très contraignante, contrairement aux bases de données conventionnelles, dans le sens où le schéma n'est pas prescriptif mais plutôt descriptif.

- *Le type des éléments est "éclectique" :* La structure des données peut varier suivant des points de vues ou suivant les besoins du processus d'acquisition des données. Cette caractéristique vient du fait que les données semi-structurées permettent avant tout d'échanger des informations. De ce fait, une information donnée peut changer de structure durant son cycle de vie lors de l'échange.

S'il y a un consensus sur les modèles de données structurées objets et relationnelles, les modèles de données semi-structurées restent très immatures. Ceci est sans doute dû à leur grande flexibilité et à l'absence de règles précises pour la construction et la description des données semi-structurées, qui constitue justement l'objectif de cette approche.

9.1.3.1 Le modèle OEM

Le modèle OEM (Object Exchange Model) est un modèle semi-structuré, initialement introduit pour l'échange de données dans le système de médiation TSIMMIS [62], puis utilisé dans le système de gestion de bases de données semi-structurées LORE (Lighweight Object REpository) [9]. Ce modèle, aussi appelé modèle d'échange *d'informations*, permet de représenter simplement les collections de données provenant de plusieurs sources, indépendamment du modèle de données des applications sous-jacentes et indépendamment du modèle de stockage.

Comme le montre la figure 9.1, un objet OEM est décrit par un quadruplet

Étiquette	OID	Type	Valeur

FIG. 9.1 – La structure d'un objet OEM

tel que :
Étiquette : Est une chaîne de caractères permettant de décrire l'objet représenté.
OID : Permet d'identifier tout objet OEM d'une manière unique. Un identificateur peut être référencé par une ou plusieurs références locales ou réparties pour le partage des objets.
Type : C'est le type de la valeur de l'objet. Un type peut être *atomique* (int, char, float, string etc.) ou un ensemble (set) ou une liste (list) d'OIDs.
Valeur : Représente la valeur de l'objet qui peut être une valeur atomique ou un objet complexe de type list ou set. Les objets contenus dans un objet complexe peuvent être ordonnés ou non.

OEM est une représentation sous forme de collections d'objets de données semi-structurées. Il utilise deux concepts du modèle objet : les *identificateurs* d'objet et le type d'*objet complexe* avec un typage flexible. La démarche d'utiliser OEM pour représenter les données semi-structurées est une approche minimaliste, dans le sens où OEM est la représentation la plus simple et la plus naturelle des données semi-structurées. En effet, c'est une représentation sous forme d'un graphe labélisé où les noeuds représentent les identificateurs d'objets, et les arcs, les étiquettes qui dénotent la liste des objets contenus dans l'objet père. Par exemple, la source de données semi-structurées de la figure 9.3 est représentée par le graphe OEM de la figure 9.2.

Ce modèle a servi de base pour d'autres systèmes tels que UnQl [33] et Strudel [58], dont le modèle sous-jacent n'est qu'une variante du modèle OEM. Ce modèle a été enrichi dans d'autres systèmes d'intégration et particulièrement dans le système YAT *[42],* en introduisant quelques notions de typages sur les données semi-structurées.

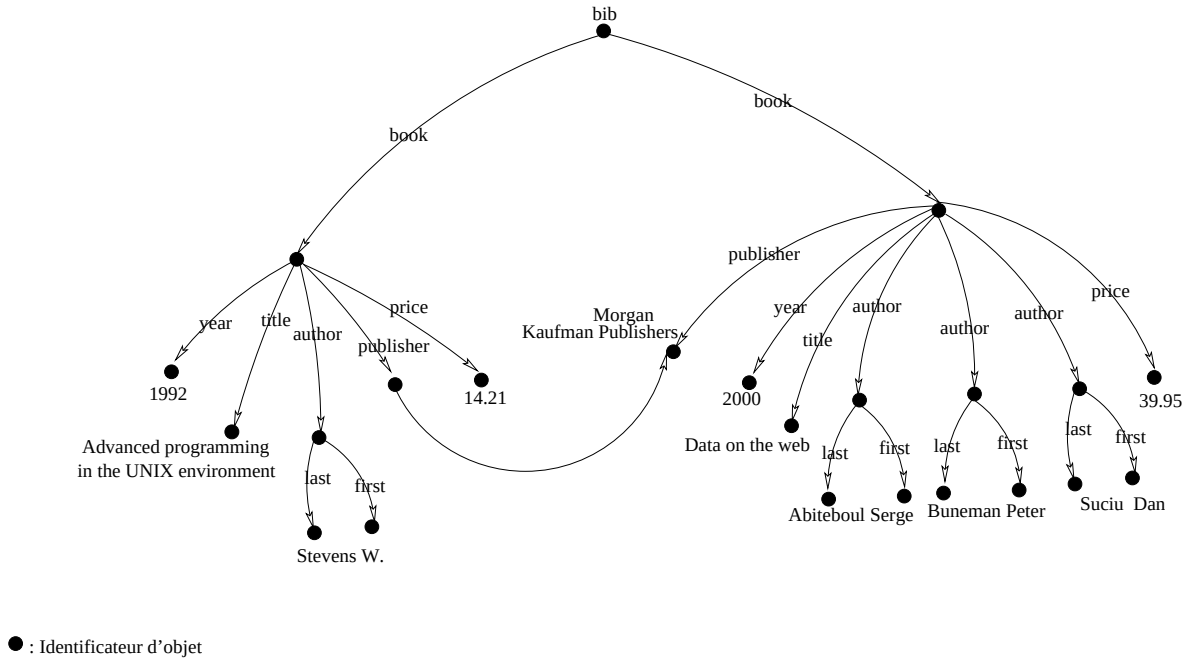


FIG. 9.2 – Un graphe d'objets OEM

```
<bib>

<book year= "1992">
    <title>Advanced Programming in the Unix Environment</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price>70.21</price>
 </book>

<book year"2000">
    <title>Data on the Web</title>
    <author><last>Abiteboul</last><first>Serge</first></author>
    <author><last>Buneman</last><first>Peter</first></author>
    <author><last>Suciu</last><first>Dan</first></author>
    <publisher>Morgan Kaufmann Publishers</publisher>
    <price>39.95</price>
 </book>

</bib>
```

FIG. 9.3 – Un exemple de données semistructurées XML

9.1.3.2 Modèles de données XML

XML est un langage textuel balisé pour l'échange d'informations à travers le Web. Un document XML peut être conforme ou non à une description de structure. Cette structure peut être décrite par une DTD (Document Type Definition) ou par le langage XML Schema [180]. La DTD (voir la figure 9.4) est la première structure introduite pour la description des documents XML, et toutes les autres structures utilisées par la suite conservent le même principe. Une DTD permet de déclarer les deux entités de la structure que sont les éléments et les attributs présents dans un document, avec une grammaire qui spécifie le formatage et l'ordre d'apparence de ces entités. Les entités sont accompagnées par des méta-données sur les cardinalités (le nombre d'occurrences) en utilisant les caractères " *","+"," ?", qui ont le même sens que dans les expressions régulières. Le langage XML Schema, successeur des DTD, permet une description plus élaborée et plus fine des structures XML (pour plus de détails, voir la spécification [180]).

```
<!ELEMENT bib (book* )>
<!ELEMENT book (title, author+, publisher, price )>
<!ATTLIST book year CDATA #REQUIRED >
<!ELEMENT author (last, first )>
<!ELEMENT title (#PCDATA )>
<!ELEMENT last (#PCDATA )>
<!ELEMENT first (#PCDATA )>
<!ELEMENT publisher (#PCDATA )>
<!ELEMENT price (#PCDATA )>
```

FIG. 9.4 – Un exemple d'une DTD des document bib

Il n'existe pas réellement un seul modèle de données XML, mais plusieurs. Plusieurs standards autour de l'utilisation de XML sont apparus à travers des communautés du consortium W3C tels que DOM [182], XPath [181], infoset [183], XQuery [176], où chacun spécifie une variante de modèle de données pour une utilisation particulière. La différence entre ces modèles porte principalement sur les types de noeuds XML à considérer (*attributs*, *comment*, *text*, ...). Cette différence dépend à la fois de la façon dont on veut manipuler les données XML et de la nature de l'application cible. Par exemple, le modèle DOM (Document Object Model) a été introduit pour manipuler des documents XML dans les navigateurs Web. DOM n'est pas typé et est simplement une API objet qui permet de représenter et de manipuler un document à travers un langage de programmation. Il considère tous les noeuds comme des objets, et gère les références entre les noeuds pères et les noeuds fils. Plus tard, le besoin de définir des langages de requêtes sur les données XML a conduit à élaborer d'autres modèles, notamment à l'issue de la spécification du langage d'adressage XPath et du langage de requêtes XQuery. Tous les modèles ont tendance à converger vers le modèle défini dans XQuery, qui englobe les modèles antérieurs, notamment ceux définis dans XPath et XML Schéma. Notons que ce modèle est toujours en discussion et en évolution dans le cadre du consortium W3C. Il repose sur les deux types *séquence* et *item*.

- Le type *sequence* représente une collection ordonnée de 0 ou n items.
- Un *item* peut être de type node ou de type atomique.
- Une *valeur atomique* est une instance des types primitifs. Les types primitifs supportés regroupent tous les types définis dans le langage XML Schema (integer, string, decimals,etc.).
- Le type *node* englobe les sept types de noeuds que l'on retrouve dans les documents XML, à savoir : *attribute()*, *comment()*, *document-node()*, *element()*, *namespace()*, *processing - instruction()* et *text()*. Un noeud *node* est caractérisé par un identificateur unique au sein d'un document et par un nom.

Ce modèle de données est conforme à un système de types. La section suivante présente le sens et la portée du typage dans les données semi-structurées, notamment le système de types relatif au modèle de données XQuery.

9.1.3.3 Typage des données semi-structurées

Les modèles de données structurées et semi-structurées sont diamétralement différents : le premier se soucie de la maîtrise de la cohérence des données et le deuxième concerne plutôt l'interopérabilité. Cette différence est traduite par des systèmes de typage différents.

La notion de type dans les données semi-structurées est différente de celle des données structurées. Cela est dû au fait que dans les données semi-structurées, la structure est noyée dans les données. En effet, dans les bases de données traditionnelles, les types de données sont préalablement définis à travers un schéma pour ensuite peupler efficacement la base de données. Dans les bases de données semi-structurées, les types de données peuvent être inférés et extraits à partir d'une source de données semi-structurées déjà peuplée [7, 119]. Ces types peuvent être récupérés à travers des requêtes. Par exemple, on peut avoir un type de données qui représente les livres écrits par Serge Abiteboul ou un autre type représentant les livres écrits par Serge Abiteboul avant l'année 2002. Un type de données semi-structurées peut donc apparaître à plusieurs niveaux d'une source semi-structurée, et de ce fait, peut appartenir à plusieurs types de noeuds : c'est comme si l'on avait un objet avec plusieurs rôles dans le modèle objet ! Le typage des données semi-structurées s'est avèré nécessaire, essentiellement pour interroger et manipuler uniformément les données, non pour peupler d'une manière efficace et fiable une base de données. Ce typage permet la vérification statique des types, utile pour le traitement des requêtes.
Très récemment, le typage des données semi-structurées est un domaine de recherche très actif et controversé. On distingue deux communautés de recherche en la matière : la communauté base de données dans le cadre de la standardisation du langage XQuery, et la communauté de recherche dans le domaine des langages de programmation.
Contrairement aux bases de données objets, qui ont été influencées par l'apparition des langages de programmation orientés-objets, ce sont les données semi-structurées, et particulièrement XML, qui ont influencé l'apparition des langages de programmation orientés XML tels que XDuce [79] et Scala [131] (voir figure 9.5). Ce sont des langages fonctionnels qui permettent de traiter nativement les données semi-structurées au lieu d'utiliser des API tel que SAX [151] ou encore DOM [182], qui s'avèrent très coûteuses en terme d'espace mémoire. Ce traitement concerne particulièrement la prise en compte native des fonctionnalités des langages de requêtes XPath et XQuery, et ce à deux nivaux :

- La définition d'un système de type permet de manipuler nativement les données XML. Ceci permet de pallier le problème *d'impedance mismatch*[1] en supportant nativement le modèle de données XML. Ces systèmes sont basés sur les expressions régulières initialement utilisées dans les DTDs comme décrit dans la section suivante.

[1] On parle *d'impedance mismach* lorsqu'il y a un recouvrement entre le système de types d'une base de données et le système de types du langage de programmation utilisé pour manipuler les données de la base. Ce problème a été identifié entre les bases de données relationnelles et les langages de programmation orientés-objet à la fin des années 90. Il dénote aussi, en quelque sorte, la différence entre la vision de la communauté des bases de données et la communauté des langages de programmation.

- L'extension des mécanismes de patterns matching classiques dans les langages de programmation vers des patterns matching sur des expressions régulières. Ces expressions régulières représentent des types de données. Ceci permet de prendre en charge les expressions régulières de chemins utilisées dans les standards de requêtes XQuery et XPath.

Les expressions régulières de types

Les différents systèmes de types définis, soit dans la description de la sémantique formelle de XQuery [184], soit dans les langages de programmation orientée-XML [79, 131, 78, 35], sont basés sur les expressions régulières. On parle alors d'*expressions de types réguliers*. Ces types permettent de décrire des ensembles sous forme de structures suivant le principe *nom de l'entité et son contenu*.
Par exemple, le type

```
type := author (first[String], last[String])
```

décrit les valeurs du label book qui contient une séquence (ou une concaténation) de labels `first` et `last` qui contiennent des valeurs de type string.
Un type d'expression rationnel peut avoir un nom comme le montre l'exemple ci-dessous. Ce nom permet de composer en plusieurs plusieurs types en utilisant les caractères d'expressions régulières "*" , " ?" et '+" pour créer d'autres types :

```
authorType = author (first[String], last[String])
bookType = book (title [String], authorType+, publisher [String], price [Float])
```

Dans cet exemple, le type de nom `bookType` est une séquence du type `titre` qui contient une valeur de type String, d'un ou de plusieurs types `auteurs` et d'un type `publisher` qui contient une valeur de type String et du label `price` de type float.
Pour couvrir la possibilité de choisir entre deux types, comme l'équivalent du symbole "|" dans les DTDs, le type union est utilisé pour un choix entre deux ou plusieurs types au lieu d'une séquence. Dans l'exemple ci-dessous, le type `authorType'` est une séquence du type `author` suivi du type `email` qui contient une `string` ou le type `tel` de type String :

```
authorType' = author (name[String],(email[String] | tel[String]))
```

Des formes de sous-typage ont été définies pour les types d'expressions régulières en se basant sur les relations de contenance entre les symboles "+", "*" , " ?".

Par exemple, pour un type donné T, le type $T?$ est un sous-type du type T. En appliquant cette règle, on peut déduire les relations de sous-typage ci-dessous :

```
(first[String], last[String]) < : (first[String], last[String])?
(first[String], (last[String])?) < : (first[String], last[String], Tel[String] ?)
```

D'autres règles basées sur les relations algébriques permettent de raisonner sur les expressions de types régulières, notamment :
- L'associativité de la concaténation et de l'union.
- La commutativité de l'union.
- La distributivité de l'union sur la concaténation.

Pour plus de détails sur les expressions de types régulières, nous vous invitons à vous référer à [79, 78, 35].

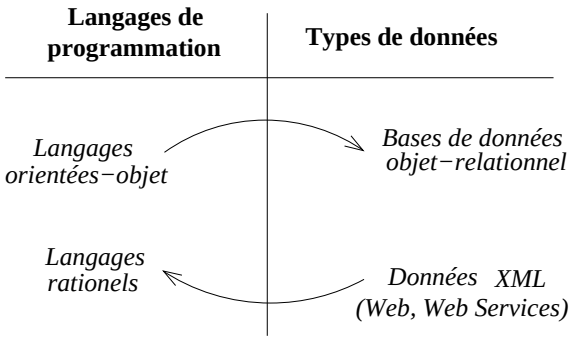


FIG. 9.5 – Langage de programmation vs types de données

Dans le modèle de données XQuery, les expressions de types régulières sont appliquées sur les différents types d'items (voir section). La description formelle du système de type de XQuery n'est pas encore standardisée et reste en cours de discussion[184]. Notons que les recherches autour du typage sont très immatures et parfois controversées entre la communauté des bases de données et la communauté des langages de programmation, et la majorité des résultats que nous avons présentés ici n'existait pas au démarrage de la thèse.

9.1.4 Un modèle hybride

Le système OZONE [102] utilise un modèle de données hybride pour l'intégration des données structurées et semi-istructurées. Ce modèle fait cohabiter à la fois des objets OEM et des objets ODMG, mais pas d'une manière transparente. Un objet OEM peut encapsuler un objet ODMG et inversement, un objet ODMG peut encapsuler ou peut contenir un graphe d'objets OEM.

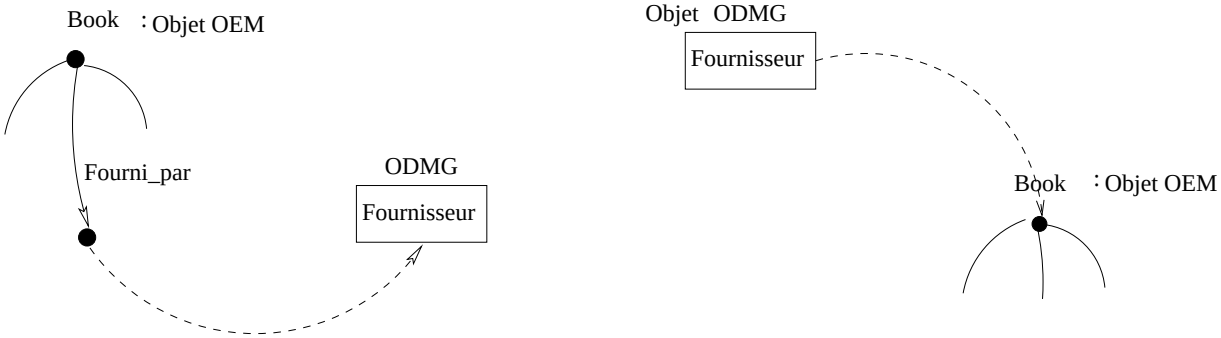


FIG. 9.6 – Le modèle de données dans le système OZONE

Le passage d'un monde structuré vers un monde semi-structuré s'effectue au travers des objets proxy dédiés. Cette approche est intéressante parce qu'elle permet de donner une vision structurée des données semi-structurées et une vision semi-structurée des données structurées. Des objets ODMG peuvent alors être interrogés par le langage semi-structuré Lorel[10], et des objets OEM peuvent à leur tour être interrogés par une requête OQL. Cependant, ce système

n'a pas défini de système de types uniforme, mais propose seulement une solution pour gérer des références des objets OEM vers des objets ODMG et inversement (voir la figure 9.6).

Bibliographie

[1] ABITEBOUL, S. Querying Semi-Structured Data . In *International Conference on Database Theory (ICDT)* (1997).

[2] ABITEBOUL, S. On Views and XML. In *Proceedings of the Symposium on Principles of Database Systems (ACM)* (1999), pp. 1–9.

[3] ABITEBOUL, S., AND BEERI, C. On the Power of the Languages for the Manipulation of Complex Objects. Tech. rep., INRIA - Verso Project, 1993.

[4] ABITEBOUL, S., BENJELLON, O., AND MILO, T. Towards a Flexible Model for Data and Services Integration. In *Proceedins of the International Workshop on Fondations of Models and Languages for Data and Objects* (2001).

[5] ABITEBOUL, S., BENJELLOUN, O., MILO, T., MANOLESCU, I., AND WEBER, R. Active XML : A Data-Centric Perspective on Web Services. In *Proceedings of the Bases de Données Avancées* (2002).

[6] ABITEBOUL, S., AND BONNER, A. Objects and Views. In *Proceedings of the SIGMOD Conference on Management of Data, San Francisco, California* (March 1991).

[7] ABITEBOUL, S., BUNEMAN, P., AND SUCIU, D. *Data on the Web : from Relations to Semistructured Data and XML.* 1-55860-622-X. Morgan Kaufmann Publishers, 2000.

[8] ABITEBOUL, S., CLUET, S., FERRAN, G., AND ROUSSET, M. C. The Xyleme Projet. Tech. rep., INRIA, Gemo Team, November 2001.

[9] ABITEBOUL, S., GOLDMAN, R., QUASS, D., AND WIDOM, J. Lore : A Database Management System for Semistructured Data. *SIGMOD Record 26*, 3 (September 1997), 54–66.

[10] ABITEBOUL, S., QUASS, D., MCHUGH, J., WIDOM, J., AND WIENER, J. L. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries 1*, 1 (1997), 68–88.

[11] ADALI, S., CANDAN, K. S., PAPAKONSTANTINOU, Y., AND SUBRAHMANIAN, V. S. Query Caching and Optimization in Distributed Mediator Systems. In *Proceedings of The International Conference on Management of Data (ACM SIGMOD), Montreal, Canada* (1996), pp. 137–148.

[12] Ahmed, R., Smedt, P. D., Du, W., Kent, W., Ketabchi, M. A., Litwin, W. A., Rafii, A., and Shan, M.-C. The Pegasus Heterogeneous Multidatabase System. *Computer 24*, 12 (1991), 19–27.

[13] Alia, M., Chassande-Barrioz, S., Déchamboux, P., Hamon, C., and Lefebvre, A. A Middleware Framework for Persistence and Querying of Java Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Oslo, Norway* (june 2004).

[14] Alia, M., Collet, C., and Lefebvre, A. Systèmes d'Intégration de Données : une Approche à Composants. In *Proceedings of Journée Composants (JC) and Langages et modèles à objets conference (LMO) and Revue des Sciences et Technologie de l'information (RSTI), Lille, France* (march 2004).

[15] Alia, M., Collet, C., and Lefebvre, A. Takfa : A Component-Based Middleware Data Integration System. In *Proceedings of the Bases de Données Avancées Conference (BDA), Montpellier, France* (October 2004), pp. 309 – 329.

[16] Alia, M., Lenglet, R., Coupaye, T., and Lefebvre, A. Querying Reflexive Component-Based Architectures. In *Proceedings of the EUROMICRO International Conference, Component-Based Software Engineering track (CBSE), Rennes, France* (2004).

[17] Arens, Y., Hull, R., King, R., and al. Reference Architecture for the Intelligent Information Integration. Tech. rep., ARPA, 1995.

[18] Barbosa, A. C. P. *A Middleware for Heterogeneous Data Source Integration Based on Frameworks Composition.* PhD thesis, PUC-Rio University, Brazil, May 2001.

[19] Barbosa, A. C. P., Porto, F. A. M., and Melo, R. N. Configurable Data Integration Middleware System. In *Proceedings of the International Workshop on Information Integration on the Web* (Rio de Janeiro, RJ, Brasil, 2001).

[20] Batory, D. S., Barnett, J. R., Garza, J. F., Smith, K. P., Tsukuda, K., Twichell, C., and Wise, T. E. GENESIS : An Extensible Database Management System. *IEEE Transaction Software Engeneering 14*, 11 (1988), 1711–1730.

[21] Batory, D. S., Leung, T. Y., and Wise, T. E. Implementation Concepts for an Extensible Data Model and Data Language. *ACM Transaction Database Systems 13*, 3 (1988), 231–262.

[22] Beringer, D., Tornabene, C., Jain, P., , and Wiederhold, G. A Language and System for Composing Autonomous, Heterogeneous and Distributed Megamodules. In *International Workshop on Large-Scale Software Composition (DEXA)* (Vienna Austria, August 1998).

[23] Bonnet, P. *Prise en Compte des Sources de Données Indisponibles Dans les Systèmes de Médiation.* PhD thesis, Université de Savoie, 1999.

[24] Bornhovd, C. MIX : A Representation Model for the Integration of Web Based Data. Tech. rep., Darmstat University of Technology, Germany, 1998.

[25] Bouguettaya, A., Benatallah, B., and Elmagarmid, A. Interconnecting Heterogeneous Information Systems. Kluwer Academic Publischers, The Netherlands, 1998.

[26] Braga, R. M., Mattoso, M., and Werner, C. M. The use of mediators for component retrieval in a reuse environment. In *Proceedings of the Workshop on Component-Based Software Engineering Process (TOOLS-30 USA'99)* (Santa Barbara, CA, aug 1999), pp. 542–546.

[27] Braga, R. M., Mattoso, M., and Werner, C. M. The use of mediation and ontology technologies for software component information retrieval. In *Proceedings of the 2001 Symposium on Software Reusability : Putting Software Reuse in Context (SSR2001)* (Toronto, Ontario, Canada, may 2001), pp. 19–28.

[28] Bright, M. W., Hurson, A. R., and Pakzar., S. A Taxonomy and Current issues in Multidatabase Systems. *Computer* (March 1992), 50–60.

[29] Brundage, M. *XQuery : The XML Query Language* . 2004.

[30] Bruneton, E., Coupaye, T., Leclerc, M., Quema, V., and Stefani, J.-B. An Open Component Model and its Support in Java. In *Proceedings of the International Symposium on Component-based Software Engineering ICSE* (Edinburgh, Scotland, may 2004).

[31] Bruneton, E., Coupaye, T., and Stefani, J. Recursive and Dynamic Software Composition with Sharing. In *Proceedings of the International Workshop on Component-Oriented Programming (WCOP), Malaga, Spain* (June 2002).

[32] Bruneton, E., Lenglet, R., and Coupaye, T. ASM : A code Manipulation Tool to Implement Adaptable Systems. In *Proceedings of ASF (ASM SIGOPS) et Journées Composants Conference, France* (2002).

[33] Buneman, P., Davidson, S., Hillebrand, G., and Suciu, D. A Query Language and Optimization Techniques for Unstructured Data. In *International Conference on Management of Data, Montréal, Canada* (1996), pp. 505–516.

[34] Buneman, P., Fernandez, M. F., and Suciu, D. UnQL : a query language and algebra for semistructured data based on structural recursion. *VLDB Journal : Very Large Data Bases 9*, 1 (mar 2000), 76–110.

[35] Buneman, P., and Pierce, B. C. Union types for semistructured data. In *Proceedings of the International Workshop on Database Programming Languages* (2000), Springer-Verlag, pp. 184–207.

[36] Cali, A., Calvanese, D., Giacomo, G. D., and Lenzerini, M. On the Expressive Power of Data Integration Systems. In *Proceedings of the International Conference on Conceptual Modeling* (London, UK, 2002), Springer-Verlag, pp. 338–350.

[37] Cardeli, L., and Wegner, P. On Understanding Types, Data Abstraction and Polymorphysm. *ACM Computing Survey 4*, 17 (December 1985).

[38] Carey, M. J., Florescu, D., Ives, Z. G., Y.Lu, Shanmugasundaram, J., Shekita, E. J., and Subramanian, S. N. . XPERANTO : Publishing Object-Relational Data as XML. In *Proceedings of the WebDB (Informal Proceedings), Dallas, Texas* (2000), pp. 105–110.

[39] Chamberlin, D., Robie, J., and Florescu, D. Quilt : An XML Query Language for Heterogeneous Data Sources. In *Proceedings of WebDB Conference, Dallas, Texas* (2000).

[40] Christophides, V., Cluet, S., and Moerkotte, G. Evaluating Queries with Generalized Path Expressions. In *Proceedings of the international conference on Management of data (ACM SIGMOD)* (Montreal, Quebec, Canada, 1996), ACM Press, pp. 413–422.

[41] Christophides, V., Cluet, S., and Siméon, J. On Wrapping Query Languages and Efficient XML Integration. In *Proceedings of the International Conference on Management of Data (SIGMOD), Seattle* (1998), pp. 177–188.

[42] Cluet, S., Delobel, C., Simeon, J., and Smaga, K. Your Mediators Need Data Conversion ! In *Proceedings of the International Conference on Management of Data, Seattle* (1998), pp. 177–188.

[43] Cluet, S., and Simeon, J. Yatl : A Functional and Declarative Language for XML. Tech. rep., INRIA-Bell Labs, 2000.

[44] Cluet, S., Veltri, P., and Vodislav, D. Views in Large Scale XML Repository. In *Proceedings of the International Conference on Very Large Data Bases (VLDB), Rome, Italy* (2001).

[45] Culik, C., Marck, R., and Shewchun, D. SQL Extensions ans Algebras for Object-Oriented Query Languages. Tech. rep., University of Waterloo, 1992.

[46] Dahm, M., Macneill, C., Manolache, C., Zyl, J. V., Dixon-Peugh, D., and Haase, E. BCEL : Byte Code Engineering Library. http ://jakarta.apache.org/BCEL.

[47] Date, C. *An Introduction To Database System.* Reading Mass, 1986.

[48] Davidson, S., Crabtree, J., Brunk, B., Schug, J., Tannen, V., Overton, C., and Stoeckert, C. K2 Kleisli and GUS : Experiments in Integrated Access to Genomic Data Sources. *IBM Systems Journal 40* (2001), 512–531.

[49] Deutsch, A., Fernandez, M., Florescu, D., Levy, A., and Suciu, D. XML-QL : A Query Language for XML. *Computer Networks, Amsterdam, Netherlands 31*, 11 (1999), 1155–1169.

[50] Dittrich, K. R., and Geppert, A. *Component Database Systems.* Maurgan Kaufmann Publishers, 2001.

[51] Drapeau, S. *RS2.7 : un Canevas Adaptable de Services de Duplication.* PhD thesis, Institut National Polytechnique de Grenoble (INPG), 2003.

[52] Du, W., Krishnamurthy, R., and Shan, M. C. Query Optimization in Heterogeneous Databases Management Systems. In *Proceedings of the International Conference en Very Large Data Bases (VLDB)* (1992), pp. 277–291.

[53] Dumant, B., Horn, F., Tran, F. D., and Stefani, J. B. Jonathan : An Open Distributed Processing Environment in Java. In *Proceedings of the International Conference on Distributed Systems Plateforms and Open Distributed Processing, The Lake District U.K* (September 1998).

[54] Duschka, M., and Genesereth, M. R. Query Planning in Infomaster. In *Proceedings of the Symposium on Applied Computing (ACM), San Jose, Canada* (1997).

[55] Déchamboux, P., Basset, R., and Chassande-Barrioz, S. JORM : A Java Object Repository Mapping Framework. http ://jorm.objectweb.org/doc/index.html, October 2004.

[56] Eastman, J., Jordan, D., Russell, C., Schadow, O., Stanienda, T., Velez, F., Berler, M., Cattell, R. G. G., and Barry, D. K. *The Object Data Standard : ODMG 3.0.* Morgan Kaufmann Publishers, 1999.

[57] Fankhausser, P., Gardarin, G., Lopez, M., Munoz, J., and Tomasic, A. Experiences in Federated Databases : From IRO-DB to MIRO-Web. In *Proceedings of the International Conference on Very Large DataBases (VLDB), New York* (1998).

[58] Fernandez, M., Florescu, D., Kang, J., Levy, A., and Suciu, D. Catching the Boat with Strudel : Experiences with a Web-Site Management System. In *Proceedings of the International Conference on Management of Data (SIGMOD), Seattle, Washington* (1998), ACM Press, pp. 414–425.

[59] Finance, B. *Une Plateforme pour la Génération d'Optimiseurs Extensibles.* PhD thesis, Université Paris 6, 1992.

[60] Fischer, P. C., and Thomas, S. J. Operators for Non-First-Normal-Form Relations. In *Proceedings of computer software and application conference (IEEE)* (1983), pp. 464–475.

[61] Florescu, D., Levy, A. Y., and Mendelzon, A. O. Database Techniques for the World-Wide Web : A Survey. *SIGMOD Record 27*, 3 (1998), 59–74.

[62] Garcia-Molina, H., Papakanstantinou, Y., Quass, D., Rajaraman, A., Sagiv, Y., Ullman, J., Vassalos, V., and Widom, J. The Tsimmis Approach to Mediation : Data Models and Languages. *Journal of Intelligent Information Systems* (1997).

[63] García-Bañuelos, L. *An Adaptable Infrastructure for Customized Persistent Object Management.* PhD thesis, Institut National Polytechnique de Grenoble (INPG), 2003.

[64] Girardi, M. R., and Ibrahim, B. Using English to retrieve software. *The Journal of Systems and Software 30*, 3 (sep 1995), 249–270.

[65] Goldman, R., and Widom, J. Approximative DataGuides. In *Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats, Israel* (January 1999).

[66] Graefe, G. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys 25*, 2.

[67] GRAEFE, G., AND DEWITT, D. *The EXODUS Optimizer Generator.* PhD thesis, University of Winconsin, 1997.

[68] GRAEFE, G., AND DEWITT, D. J. The EXODUS Optimizer Generator. In *Proceedings of the International Conference on Management of Data (SIGMOD)* (1987), ACM.

[69] GRAEFE, G., AND MCKENNA, W. J. The Volcano Optimizer Generator : Extensibility and Efficient Search. In *Proceedings of the International Conference on Data Engineering (ICDE)* (1993).

[70] GUY DE TRÉ, R. D. C., AND PRADE, H. Null Values Revisited in Prospect of Data Integration. In *Semantics of a Networked World : Semantics for Grid Databases, International IFIP Conference, ICSNW 2004* (June 2004), M. Bouzeghoub, C. Goble, V. Kashyap, and al., Eds., Springer-Verlag Heidelberg, pp. 79–90.

[71] HAAS, L., MILLER, R., NISWONGER, B., ROTH, M., SCHWARZ, P., AND WIMMERS, E. L. Transforming Heterogeneous Data with Database Middleware : Beyond Integration. *IEEE Data Engineering Bulletin* (1997).

[72] HAAS, L. M., CHANG, W., LOHMAN, G. M., MCPHERSON, J., WILMS, P. F., LAPIS, G., LINDSAY, B., PIRAHESH, H., CAREY, M. J., AND SHEKITA, E. Starburst Mid-Flight : As the Dust Clears. *Transactions on Knowledge and Data Engineering (IEEE) 2*, 1 (1990), 143–160.

[73] HAAS, L. M., KOSSMANN, D., WIMMERS, E. L., AND YANG, J. Optimizing Queries Across Diverse Data Sources. In *Proceedings of the International Conference on Very Large Data Bases (VLDB), Athens, Greece* (1997), pp. 276–285.

[74] HASSELBRING, W. Information System Integration. *Communications of the ACM 43*, 6 (2000), 33–38.

[75] HEIMBIGNER, D., AND MCLEOD, D. A Federated Architecture for Information Management. *ACM Transaction 3*, 3 (1985), 253–278.

[76] HEMER, D., AND LINDSAY, P. Specification-based retrieval strategies for module reuse. Tech. rep., University of Queensland, Australia, 1999.

[77] HENNEY, K. Null Object : Something for Nothing. In *Proceedings of the Europeen Conference on Pattern Languages of Programms, 2002.*

[78] HOSOYA, H., AND PIERCE, B. Regular Expression Pattern Matching for XML. In *Proceedings of the Symposium on Principles of Programming Languages (ACM SIGPLAN-SIGACT)* (2001), ACM Press, pp. 67–80.

[79] HOSOYA, H., AND PIERCE, B. XDuce : A Statically Typed XML Processing Language. *ACM Transactions on Internet Technology* (2002).

[80] HÜRSCH, W. L., AND LOPES, C. V. Separation of Concerns. Tech. Rep. NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, Massachusetts, USA, 1995.

[81] INRIA, CARAVEL PROJECT. LeSelect : A Middleware System for publishing Data and Services on the Internet. http ://www-caravel.inria.fr/ leselect/.

[82] IRIBARNE, L., TROYA, J. M., AND VALLECILLO, A. Trading for COTS components in open environments. In *Proceedings of the 27th Euromicro Conference 2001 : A Net Odyssey (EUROMICRO'01)* (Warsaw, Poland, sep 2001).

[83] IRIBARNE, L., TROYA, J. M., AND VALLECILLO, A. Selecting software components with multiple interfaces. In *Proceedings of the 28th Euromicro Conference (EUROMICRO'02)* (Dortmund, Germany, sep 2002).

[84] ISO. ITU/ISO Reference Model of Open Distributed Processing -Part 2. Architecture. International Standard ISO/IEC 10746-3, ITU-T Recommendation X.903, 1995.

[85] ISO. ITU/ISO Reference Model of Open Distributed Processing -Part 2. Foudations. International Standard ISO/IEC 10746-2, ITU-T Recommendation X.902, 1995.

[86] ISO/IEC. ODP trading function. ITU-T Draft Rec. X.9tr | ISO/IEC DIS 13235, ISO/IEC, 1995.

[87] JAECHKE, G., AND SCHEK, H. J. Remarks on the algebra of non-first-normal-form relations. In *Proceedings of the Symposium on Principles of Database Systems (ACM SIGACT-SIGMOD)* (Los Angeles, 1982), pp. 124–138.

[88] JARKE, M., LENZERINI, M., VASSILIADIS, P., AND VASSILIOU, Y. *Fundamentals of Data Warehouses.* 2000.

[89] JENG, J.-J., AND CHENG, B. H. C. Specification matching for software reuse : A foundation. In *ACM Symposium on Software Reusability (SIGSOFT)* (Seattle, Washington, 1995), pp. 97–105.

[90] JOHNSON, R. E. Framework = (Components + Patterns) : How Frameworks Compare to Other Object-Oriented Reuse Techniques. In *Communications of the ACM* (october 1997), vol. 40, pp. 39–42.

[91] JOSIFOVSKI, V. *Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration.* PhD thesis, Linkping, Suède, juin 1999.

[92] JOSIFOVSKI, V., KATCHAOUNOV, T., AND RISCH, T. Optimizing Queries in Distributed and Composable Mediators. In *Proceedings of the fourth IEICIS International Conference on Cooperative Information Systems, Edinburgh, Scotland* (1999).

[93] KABRA, N., AND DEWITT, D. J. OPT++ : An Object-Oriented Implementation for Extensible Database Query Optimization. *Proceedings of the International Journal on Very Large Data Bases (VLDB)* (1999).

[94] KATCHAOUNOV, T., JOSIFOVSKI, V., AND RISCH, T. Scalable View Expansion in a Peer Mediator Systems. In *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA)* (2003).

[95] KENT, W. The Breakdown of the Information Model in Multi-database Systems. *SIGMOD RECORD 20*, 4 (December 1991), 10–15.

[96] KHAYATI, O., AND GIRAUDIN, J.-P. Component retrieval systems. In *Proceedings of the Workshop on Reuse in Object-Oriented Information Systems Design (OOIS 2002)* (Montpellier, France, sep 2002).

[97] KIRK, T., LEVY, A., SAVIG, Y., AND SRIVASTAVA, D. The Information Manifold. In *proceedings of the AAAI Spring Symposium on Information Gathering,* (1995).

[98] KOCHAREKAR, R. Nulls in Relational Databases : Revised. *SIGMOD Rec.* (1989), 68–73.

[99] KUNO, H. A., AND RUNDENSTEINER, E. The MultiView OODB View System : Design and Implementation. *Journal of Thery and Practice of Object Systems (TAPOS), special issue on Subjectivity in Object-Oriented Systems* (September 1996).

[100] LACOSTE, M.-A. *Une Machine Virtuelle Répartie pour la Programmation de Processus Mobiles.* PhD thesis, Université Joseph Fourier (UJF) - Grenoble 1 Sciences et Geographie, 2003.

[101] LACROIX, Z., DELOBEL, C., AND BRÈCHE, P. Object Views and Database Restructuring. In *Proceedings of the International Workshop on Database Programming Languages* (Colorado, USA, August 1997), Springer, pp. 80–201.

[102] LAHIRI, T., ABITEBOUL, S., AND WIDOM, J. Ozone : Integrating Structured and Semistructured Data. *Lecture Notes in Computer Science 1949* (2000), 297+.

[103] LANZELOTTE, R. S. G., AND VALDURIEZ, P. Extending the Search Strategy in a Query Optimizer. In *Proceedings of International Conference on Very Large DataBases (VLDB)* (1991).

[104] LAURA M. HAAS, JOHANN CHRISTOPH FREYTAG, G. M. L., AND PIRAHESH, H. Extensible Query Processing in Starburst. In *Proceedings of the International Conference on Management of Data (SIGMOD)* (1989).

[105] LEDOUX, T., BLAY, M., BRUNETON, E., CAROMEL, D., COUPAYE, T., HAGIMONT, D., MENAUD, J.-M., NOYÉ, J., AND RIVEILL, M. Etat de l'Art sur l'Adaptabilité. ARCAD Project, december 2001.

[106] LENGLET, R. *Composition Flexible et Efficace de Transformation de Programmes.* PhD thesis, Institut National Polytechnique de Grenoble (INPG), 2004.

[107] LEVY, A. Answering Queries Using Views : A Survey. *The International Journal on Very Large Data Base (VLDB) 10*, 4 (2001), 270–294.

[108] LIU, L., AND PU, C. An Adaptive Object-Oriented Approach to Integration and Access of Heterogeneous Information Sources. *Distributed Parallel Databases 5*, 2 (1997), 167–205.

[109] LORIE, R., AND WADE, B. The Compilation of High Level Data Language. Tech. rep., RJ 2598, IBM Research, San Jose, CA, 1979.

[110] MCBRIEN, P., AND POULOVASSILIS, A. Schema Evolution in Heterogeneous Database Architectures, A Schema Transformation Approach. In *Proceedings of The International*

Conference on Advanced Information Systems Engineering, Toronto, Ontario, Canada (2002).

[111] McHugh, J., and Widom, J. Query Optimization for XML. In *Proceedings of International Conference on Very Large Data Bases (VLDB)* (1999).

[112] McIver, W., King, R., Osbone, R., and Och, C. The COIL Project : A Common Object Interconection Language to Support Database Integration and Evolution. In *Proceedings of the International Baltic Workshop on Databases and Information Systems* (1998).

[113] McKenna, W. J., Burger, L., Hoang, C., and Truong, M. EROC : A Toolkit for Building NEATO Query Optimizers. In *International Conference on Very Large Data Bases (VLDB)* (1996).

[114] Meijer, E., and Schulte, W. Unifying Tables, Objects and Documents. In *Proceedings of the Conference on Declarative Programming in the Context of Object Oriented Languages (DP-COOL)* (2003).

[115] Meling, R., Montgomery, E. J., Ponnusamy, P. S., Wong, E. B., and Mehandjiska, D. Storing and retrieving software components : a component description manager. In *Proceedings of the 2000 Australian Software Engineering Conference* (Canberra, Australia, apr 2000), pp. 107–117.

[116] Microsoft. The Component Object Model Specification. http ://www.microsoft.com/com/resources/comdocs.asp, 1996.

[117] Mili, H., Valtchev, P., Sciullo, A.-M. D., and Gabrini, P. Automating the indexing and retrieval of reusable software components. In *Proceedings of the 6th International Workshop (NLDB'01)* (Madrid, Spain, 2001), pp. 75–86.

[118] Mili, R., Mili, A., and Mittermeir, R. T. Storing and retrieving software components : A refinement based system. *IEEE Transactions on Software Engineering 23*, 7 (jul 1997), 445–460.

[119] Milo, T., and Suciu, D. Type Inference for Queries on Semistructured Data. In *Proceedings of the symposium on Principles of database systems (ACM SIGMOD-SIGACT)* (1999), ACM Press, pp. 215–226.

[120] Naacke, H. *Modèles de Coût pour Médiateurs de Bases de Données Hétérogènnes.* PhD thesis, Université de Versailles Saint-Quentin-en-Yvelines, 1999.

[121] Naacke, H., Gardarin, G., and Tomasic, A. Leveraging Mediator Cost Models With Heterogeneous Data Sources. In *Proceedings of the International Conference on Data Engineering* (1998).

[122] Ngoc, T. T. D. *Fédération de Données Semi-structurées avec XML.* PhD thesis, Université de Versailles Saint-Quentin-en-Yvelines, 2003.

[123] NODS. Networked Open Database Services. http ://www-lsr.imag.fr/Les.Groupes/STORM/Storm2002/Francais/index.html.

[124] Object Management Group (OMG). Persistent State Service Specification (PSS). http ://www.omg.org/technology/documents/formal/persistent.htm, September 2002.

[125] ObjectWeb Consortium. JORM : Java Object Repository Mapping Framework. http ://jorm.objectweb.org.

[126] ObjectWeb Consortium. MEDOR : Middleware Enabled Distributed Object Request. http ://medor.objectweb.org.

[127] ObjectWeb Consortium. Speedo : Open Implementation of the JDO specification. http ://speedo.objectweb.org/.

[128] ObjectWeb Consortium. JOnAS : Java (TM) Open Application Server. http ://jonas.objectweb.org/.

[129] ObjectWeb Consortium. Fractal : A Component Framework Model. http ://fractal.objectweb.org.

[130] Och, C., King, R., and Osborne, R. Integrating Heterogeneous Data Sources using COIL Mediator Definition Language. In *Proceedings of the SAC international Conference* (March 2000).

[131] Odersky, M., Altherr, P., Cremet, V., Emir, B., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., and Zenger, M. The Scala Language Specification Version 1.0. http ://scala.epfl.ch/.

[132] Orfali, R., D.Harkey, and J.Edwards. *The Essential Client Server Survival Guide.* 1996.

[133] Ozsu, M., and Valduriez, P. *Principles of Distributed Database Systems (2nd edition).* Prentice-Hall, Inc., 1999.

[134] Ozsu, M. T., Munoz, A., and Szafron, D. An Extensible Query Optimizer for an Objectbase Management System. In *Proceedings of The International Conference on Information and Knowledge Management, Baltimore, MD* (November 1995).

[135] Panchapagesan, P., Hui, J., Wiederhold, G., Erickson, S., Dean, L., and Hempstead, A. The INEEL Data Integration Mediation System. In *Proceedings of international ICSC Symposium on Advances in Intelligent Data Analysis (AIDA), Rochester, New York* (June 1999).

[136] Papakonstantinou, Y., Abiteboul, S., and Garcia-Molina, H. Object Fusion in Mediator Systems. In *Proceedings of the International Conference on Very Large Data Bases (VLDB), Bombay, India* (1995).

[137] Papakonstantinou, Y., Garcia-Molina, H., and Widom, J. Object Exchange Across Heterogeneous Information Sources. In *Proceedings of the International Conference on Data Engineering (ICDE)* (1995), IEEE Computer Society, pp. 251–260.

[138] Parnas, D. L. On the Criteria to be Used in Decomposing Systems into Modules. *Communication of the ACM 15*, 12 (1972), 1053–1058.

[139] PIETRO-DIAZ, R., AND FREEMAN, P. Classifying software for reusability. *IEEE Software 4*, 1 (jan 1987), 6–16.

[140] PIRAHESH, H., HELLERSTEIN, J. M., AND HASAN, W. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *The International Conference on Management of Data (SIGMOD)* (1992), ACM Press, pp. 39–48.

[141] PIRAHESH, H., LEUNG, T. Y. C., AND HASAN, W. A Rule Engine For Query Tranformation in Starbust and IBM DB2 c/s DBMS. In *Proceedings of International Conference on data Engineering (ICDE), Birmingham, UK* (1997), pp. 391–400.

[142] POTTINGER, R., AND LEVY, A. Y. A Scalable Algorithm for Answering Queries Using Views. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)* (2000), Morgan Kaufmann Publishers Inc., pp. 484–495.

[143] REZENDE, F. D. F., AND HERGULA, K. The Heterogeneity Problem and Middleware Technology : Experiences with and Performence of Database Gateways. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)* (August 1998), pp. 196 – 157.

[144] RICHINE, K. Distributed Query Scheduling in DIOM. Tech. rep., TR97-03, Computer Science Department, University of Alberta, 1997.

[145] RISCH, T., AND JOSIFOVSKI, V. Distributed Data Integration by Object-oriented Mediator Servers. *Concurrency and Computation : Practice and Experience 13*, 11 (2001), 933–953.

[146] RIVIERRE, N., AND COUPAYE, T. Observing component behaviours with temporal logic. In *Proceedings of the ECOOP Workshop on Correctness of Model-based Software Composition (CMC'02)* (Darmstadt, Germany, jul 2002).

[147] ROSENTHAL, A., RENNER, S., SELIGMAN, L., AND MANOLA, F. Data Integration Needs an Industrial Revolution. In *Proceedings of the Very Large Data Bases Conference (VLDB)* (2001).

[148] ROTH, M., AND SCHWARZ, P. Don't Scrap It, Wrap It ! A Wrapper Architecture for Legacy Data Sources. In *Proceedings of the 23rd Very Large Data Bases Conference (VLDB) , Athens, Greece* (1997).

[149] ROTH, M. T., OZCAN, F., AND HAAS, L. Cost Model Do Matter : Providing Cost Information for Diverse Data Sources in a Federated System. In *Proceedings of the International Conference on Very Large Data Bases (VLDB), Edinburgh, Scotland* (September 1999), pp. 599–610.

[150] RUNCIMAN, C., AND TOYN, I. Retrieving reusable software components by polymorphic type. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture* (London, UK, 1989), ACM Press, pp. 166–173.

[151] RUST, P. M., BRAY, T., MEGGINSON, D., BOSAK, J., AND AL. SAX : A simple API for XML. http ://www.saxproject.org/.

[152] SAMPLE, N., BERINGER, D., MELLOUL, L., AND WIEDERHOLD, G. CLAM : Composition Language for Autonomous Megamodules. In *Proceedings of the International Conference on Coordination Languages and Models* (1999), Springer-Verlag, pp. 291–306.

[153] SARACCO, C. M., AND RIEGER, T. F. Accesing Federated Databases with Application Server Components. Tech. rep., IBM Silicon Valley Labs, San Jose, CA, February 2003.

[154] SHANMUGASUNDARAM, J., KIERNAN, J., SCHEKITA, E., FAN, C., AND FUNDERBURK, J. Querying XML Views of Relational Data. In *Proceedings of the International Conference on Very Large Data Bases (VLDB), Rome, Italy* (2001).

[155] SHAW, G., AND ZDONIK, S. A Query Algebra for Object-oriented Databases. In *Proceedings of International Conference on Data Engineering (ICDE)* (Los Angeles, February 1990), pp. 154–162.

[156] SHETH, A., AND LARSEN, J. Federated Database Systems and Managing Distributed Heterogeneous and Autonomous Databases. *ACM Transactions on database systems 6*, 1 (1990), 140–173.

[157] SHETH, A. P., AND LARSEN., J. A. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys 19*, 4 (September 1990), 183–236.

[158] STEFANI, J.-B. Composants et Reflexivité. In *Proceedings of the Langages et modèles à objets Conference (LMO)* (September 2004).

[159] SUCIU, D. Distributed query evaluation on semistructured data. *In ACM Transaction Database Systems 27*, 1 (2002), 1–62.

[160] SUGUMARAN, V., AND STOREY, V. C. A semantic-based approach to component retrieval. *SIGMIS Database 34*, 3 (2003), 8–24.

[161] SUN MICROSYSTEMS. Java 2 Enterprise Edition Specifications (J2EE). http ://java.sun.com/j2ee.

[162] SUN MICROSYSTEMS. Java Data Objects Specification (JDO). http ://java.sun.com/products/jdo.

[163] SUN MICROSYSTEMS. The Java Management eXtensions (JMX) 1.2 specification.

[164] SUN MICROSYSYEMS. Remote Method Invocation (RMI). www.java.sun.com, 1998.

[165] SZYPERSKI, C., GRUNTZ, D., AND MURER, S. *Component Software Beyond Object-Oriented Programming (Second edition).* ISBN 0-201-74572-0. MIT Press, 2002.

[166] THE OBJECT MANAGEMENT GROUP (OMG). http ://www.omg.org/.

[167] THE OBJECT MANAGEMENT GROUP (OMG). The Common Object Request Broker, Architecture and Specification. Revision 2.1. www.omg.org, 1997.

[168] TODD, J., OCH, C., KING, R., OSBORNE, R., MCIVER, W. J. ., GETRICH, N., AND TEMPLE, B. Building Mediators from Components. In *Proceeding of the International Symposium on Distributed Objects and Applications (DOA)* (1999).

[169] TOMASIC, A., RASCHID, L., AND VALDURIEZ, P. Scaling Access to Heterogeneous Data Sources with Disco. *IEEE Transactions on Knowledge and Data Engineering 10*, 5 (1998), 808–823.

[170] UDDI. Universal Description, Discovery, and Integration of Business for the Web, http ://www.uddi.org.

[171] VAN REEUWIJK, C., AND SIPS, H. Adding Tuples to Java : a Study in Lightweight Data Structures. In *Java Grande/ISCOPE Conference* (November 2002).

[172] VAN REEUWIJK, C., AND SIPS, H. Adding Tuples to Java : a Study in Lightweight Data Structures. *Concurrency and Computation : Practice and Experience* (2005).

[173] VARGUN, A. *Semantic Aspects of Hetrogeneous Databases.* 1999.

[174] VELTRI, P. *Un Systèmes de Vues pour les Données XML du Web : Conception et Implantation.* PhD thesis, Université Paris XI, octobre 2002.

[175] VU, T.-T. *Une Approche pour la Construction d'Evaluateurs Adaptables de Requêtes.* PhD thesis.

[176] W3C. XQuery : An XML Query Language. http ://www.w3.org/TR/xquery.

[177] W3C. Web Services Description Language (WSDL). http ://www.w3.org/TR/wsdl.

[178] W3C. Resource Description Framework (RDF) . http ://www.w3.org/RDF/.

[179] W3C. Simple Object Access Protocol (SOAP). http ://www.w3.org/TR/soap/.

[180] W3C. XML Schema. http ://www.w3c.org/xml/Schema.

[181] W3C. XML Path Language (XPath).

[182] W3C. Document Object Model (DOM). http ://www.w3.org/DOM/.

[183] W3C. XML Information Set (Infoset). http ://www.w3.org/TR/xml-infoset/.

[184] W3C. XQuery 1.0 and XPath 2.0 Formal Semantics. http ://www.w3.org/TR/xquery-semantics/doc-fs-Type.

[185] WIEDERHOLD, G. Mediators in the Architecture of Future Information Systems. In *Readings in Agents.* 1992, pp. 38–49.

[186] WIEDERHOLD, G. Mediation to Deal with Heterogeneous Data Sources. In *Proceedings of the Interoperability in Large-Scale Heterogeneous Systems Conference (interop)* (1999), pp. 1–16.

[187] WIEDERHOLD, G., WEGNER, P., AND CERI, S. Toward megaprogramming : A Paradigm for Component-Based Programming. *Communication of the ACM 35*, 11 (1992), 89–99.

[188] ZACHARY, G. I., HALEVY, A. Y., AND WELD, D. S. An XML Query Engine for Networked-Bound Data. *Journal of the Very Large Data Bases* (2003).

[189] ZACHARY, I. *Efficient Query Processing in Data Integration Systems.* PhD thesis, University of Washington, 2002.

[190] ZACKARI, G. I., FLORESCU, D., FRIEDMAN, M., AND AL. An Adaptavie Query Execution System for Data Integration. In *Proceedings of the International Conference on the Management of Data (SIGMOD)* (1999).

[191] ZANIOLO, C. Database Relations with Null Values. In *Proceedings of the Symposium on Principles of database systems (SIGACT-SIGMOD)* (1982), ACM Press, pp. 27–33.

[192] ZAREMSKI, A. M., AND WING, J. M. Signature matching : a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology 4*, 2 (1995), 146–170.

[193] ZAREMSKI, A. M., AND WING, J. M. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology 6*, 4 (1997), 333–369.

[194] ZHANG, Z. Enhancing component reuse using search techniques. In *Proceedings of the 23rd Conference on Information System Research in Scandinavia (IRIS'23)* (Lingatan, Sweden, aug 2000).

[195] ZHU, Q., AND LARSON, P. A. A Query Scrambling Method for Estimating Local Cost Parameters in a Multidatabase System. In *Proceedings of the International Conference on Data Engineering (ICDE)* (1994), pp. 144–153.

[196] ZHU, Q., AND LARSON, P. A. Solving Local Cost Estimation Problem for Global Query Optimisation in Multidatabase Systems. *Distributed and Parallel Databases 6* (1998), 273–221.