Habilitation à diriger des recherches

Spécialité

INFORMATIQUE

présentée à

L'UNIVERSITÉ PARIS XI

par

Sophie CLUET

Titre :

Bases de Données : Quelques clés pour l'ouverture vers le réseau

Soutenue le 8 mars 1999 devant le jury composé de :

MM.	Claude DELOBEL	Président
	Catriel BEERI	Rapporteurs
	Georges GARDARIN	
	Dave MAIER	
Mme	Marie-Christine ROUSSET	Examinatrice
MM.	Serge ABITEBOUL	Examinateurs
	François BANCILHON	

À Serge,
merci pour tout.

Cette thèse ne saurait être complète sans des remerciements. Cet exercice convenu et traditionnel s'accompagne d'une difficulté supplémentaire : j'estime réellement et j'apprécie sincèrement toutes celles et tous ceux qui m'ont aidée. J'espère que ces quelques expansions adjectivo-nominales traduiront mon plaisir de les avoir rencontrés.

Je remercie

Claude Delobel,
    mon bon maître,
Serge Abiteboul,
    mon beau capiston,
Catriel Beeri, Georges Gardarin, Dave Maier,
    chats fourrés rapportants, augustes et magnanimes,
Marie Christine Rousset,
    Ada olrette,
François Bancilhon,
    coruscant sachem,
Vassilis Christophides,
    bouillonant camaro,
Sihem Amer-Yahia,
    pertinente Almée,
Jérôme Siméon,
    pipo singulier,
Guy Ferran,
    délicieux manitou,
Tova Milo,
    fraline piquante,
Guido Moerkotte,
    fulgurant escogriffe,
Michel Scholl,
    épatant cicérone,
Danny Moreau,
    vaillante flanc-garde,
les Verso d'hier et d'aujourd'hui,
    chaleureuse smala,
Sophie, Jean-Claude, Cassio, Fernando et les O_2 d'avant,
    bath tribu,
Alain, Margot et Gaspard,
    mes indispensables.

Table des matières

Introduction

Après avoir longtemps vécu en autarcie, les systèmes d'information s'ouvrent vers le réseau. L'émergence du Web au début de cette dernière décennie a certainement accentué l'ampleur d'un mouvement que le développement des réseaux, l'accroissement des données numérisées et la diversité des traitements rendaient inéluctable. Aujourd'hui, les systèmes informatiques gèrent, importent, diffusent, échangent, intègrent de gros volumes de données parfois répliquées, souvent dans des formats différents.

Les systèmes de gestion de bases de données (SGBD) permettent la gestion efficace et cohérente de gros volumes de données, interrogées et mises à jour de façon concurrente par plusieurs utilisateurs. Ils sont donc bien évidemment au cœur du problème. Conçus à l'origine comme des systèmes fermés pour des données régulières et des applications de gestion, ils évoluent pour devenir la brique de base de nouvelles applications ouvertes sur le réseau telles que librairies numériques, systèmes Intra/Internet, serveurs de Web, commerce électronique, etc.

Ces applications sont à l'origine d'une mine de nouveaux problèmes pour les bases de données. Cette thèse traite plusieurs de ces problèmes. Elle s'articule autour d'une idée forte et de son corollaire : l'utilisation de langages déclaratifs et leur optimisation. Elle comprend cinq contributions principales.

1. Les langages de requêtes déclaratifs permettent l'accès aux bases de données depuis le réseau. Notre thèse de troisième cycle [14, 32] a porté sur la définition d'un langage de requêtes pour bases de données à objets. Ce langage est depuis devenu le standard de l'"Object Database Management Group" (ODMG [26]) sous le nom d'OQL. Bien adapté aux nouvelles applications puisque conçu sur un modèle objet, OQL est cependant moins bien établi que son concurrent relationnel Ansi-SQL [78]. Notre première contribution [34] est une nouvelle version d'OQL dont nous pensons qu'elle permettra une plus grande diffusion du langage.

2. Pour être utilisable sur des volumes de données importants, un langage déclaratif doit reposer sur un optimiseur solide. Notre deuxième contribution [35, 38, 39, 40] consiste en la conception de techniques d'optimisation pour OQL et leur implantation dans le SGBD O_2 [12].

3. Les documents structurés ou semistructurés forment la grande majorité des données accessibles sur le Web. Notre troisième contribution [2, 3, 4, 6, 28, 29, 30, 33] repose sur l'adaptation des langages et techniques d'optimisation de bases de données pour permettre une meilleure manipulation des documents, qu'ils soient stockés dans un SGBD ou dans des fichiers.

4. De nombreuses applications Web nécessitent le chargement en masse de données d'un système à un autre, avec ou sans restructuration (par exemple, la construction d'un site Web ou de son miroir). Notre quatrième contribution [9] porte sur la conception d'une technique de chargement en masse déclarative et optimisée.

5. Le monde informatique regorge de données aux formats très hétérogènes qu'il est nécessaire d'intégrer pour construire des applications telles que serveurs Intranet ou bibliothèques numériques. La conception d'un système permettant l'intégration dynamique, simple et efficace de données réellement hétérogènes forme notre cinquième contribution [37, 98].

La thèse et la suite de cette introduction sont découpées en cinq parties correspondant à ces différentes contributions.

Le standard OQL

L'accès depuis le réseau aux données contenues dans une base est par nature dynamique, dépendant des besoins d'une application ou des désirs d'un utilisateur. Il est donc nécessaire d'offrir un langage interprétable. Ce langage doit de plus être déclaratif : l'interrogation doit se faire sans connaissance particulière du système sous-jacent et de sa façon de stocker les données. Ceci simplifie la formulation des requêtes sur la base mais surtout permet au système de réorganiser les données sur disque sans gêner les utilisateurs.

Il existe actuellement deux standards d'interrogation des bases de données : ANSI-SQL [79] et ODMG-OQL [26]. SQL est le langage le plus ancien et le mieux implanté. Conçu à l'origine sur le modèle relationnel, ce langage était simple et clairement défini par une algèbre. Ses multiples évolutions, et notamment celle en cours (SQL3 [10]) dont un des objectifs est d'intégrer les objets, ont détruit sa simplicité et SQL n'a maintenant plus de sémantique claire. OQL est un langage plus récent, moins répandu, mais dont l'atout majeur est qu'il est simple et défini formellement sur un modèle objet (lequel englobe les structures relationnelles). Il repose sur une sémantique fonctionnelle qui lui permet d'offrir un pouvoir d'expression supérieur à SQL, tout en ayant une grammaire 10 fois inférieure en taille.

Afin de satisfaire aux besoins des nouvelles applications, les SGBD tendent à évoluer vers l'objet. Dans ce contexte, il est douteux qu'il y ait de la place pour deux standards d'interrogation. Nous pensons qu'OQL est le meilleur. Nous avons donc travaillé, en collaboration avec la société O_2Technology (qui fait maintenant partie du groupe Ardent Software [12]), à sa promotion au sein de la communauté industrielle.

Dans un premier temps, nous avons pensé qu'une collaboration ANSI-ODMG permettrait d'arriver à une solution satisfaisante : OQL gommerait ses différences avec SQL sur les données de type relationnel et SQL3 (qui n'est toujours pas finalisé) adopterait un style OQL sur les données objet. Nous avons travaillé en ce sens avec Jim Melton (éditeur de SQL3) et Jeff Mischkinsky (de la société Sybase). Hélas, comme nous l'expliquons en détail dans le chapitre 1, nous avons réalisé l'impossibilité d'une unification des deux langages qui impliquerait soit une perte sémantique pour OQL (moins de pouvoir d'expression, plus de complexité), soit une refonte totale de SQL.

Ce travail a néanmoins apporté des éléments positifs. Une nouvelle version d'OQL a été conçue qui offre un certain niveau de compatibilité avec SQL mais, aussi, plus de simplicité, certaines erreurs de jeunesse ayant été corrigées. Ces modifications ont certainement contribué à l'accroissement d'intérêt que le langage suscite, qui se traduit par une implication plus forte des industriels non éditeurs de SGBD dans l'ODMG et par des implantations toujours plus nombreuses du langage (citons pour l'exemple Poet [93], Jasmine [42], Java Blend [68] et Apertus [11]). Finalement, nous pensons que le nouvel OQL aura un impact sur la conception de SQL3.

Dans [34], nous retraçons les différentes étapes de ce travail auquel François Bancilhon, Claude Delobel, Guy Ferran, Sophie Gamerman et Cassio Souza dos Santos ont contribué.

Optimisation

Formuler une requête dans un langage déclaratif consiste à exprimer les caractéristiques de l'information que l'on veut obtenir. L'algorithme qui permettra d'extraire cette information de la base n'est pas spécifié. Il ne peut donc y avoir de langage déclaratif sans un optimiseur, logiciel en charge de trouver l'algorithme d'évaluation le plus efficace.

Les accès aux disques restent encore le facteur de coût le plus important dans l'évaluation d'une requête sur une grosse base de données. La plupart des techniques d'optimisation tendent donc à diminuer leur nombre. Les données étant généralement chargées en mémoire centrale par page[1], l'objectif est de maximiser leur exploitation avant qu'elles ne soient remplacées par d'autres.

SQL repose sur la notion d'accès associatif : une requête peut être vue comme un filtre, plus ou moins complexe, sur une ou plusieurs relations. Si, de façon simpliste, on considère qu'une relation est stockée séquentiellement dans un fichier, il est alors relativement aisé de garantir une bonne exploitation des pages chargées en mémoire, par exemple, en concentrant les opérations effectuées sur une relation. Les techniques d'optimisation utilisées pour cela reposent généralement sur l'algèbre relationnelle et un système de réécriture de requêtes basé sur les équivalences qu'elle supporte.

Si OQL propose des accès associatifs, il permet également de naviguer à travers la base en suivant les liens de composition des objets. Cette navigation peut entraîner des accès aléatoires à la mémoire secondaire. C'est pour cela que certains parlent de nature intrinsèquement "non-optimisable". Une autre difficulté tient à la nature complexe des données manipulées qui se traduit par un nombre relativement important de requêtes imbriquées qui, si elles ne sont pas correctement optimisées, engendrent également de nombreux accès aléatoires aux disques.

Les travaux de Jenq, Woelk, Kim et Lee [69] ont prouvé qu'il n'y avait pas d'opposition réelle entre navigation et accès associatifs : par une bonne utilisation des types d'une base objet, les requêtes navigationnelles peuvent être transformées en requêtes associatives. Dans le chapitre 2, nous introduisons des techniques de réécriture algébrique formalisant cette approche et permettant également un traitement original et efficace des requêtes imbriquées. L'algèbre supporte l'intégralité des équivalences relationnelles (ce qui n'est pas le cas de la plupart des algèbres qui ont été définies dans ce contexte [36, 96, 101, 112]) ainsi que des nouvelles, spécifiques au contexte objet. Nous démontrons que la richesse du modèle objet est de fait un élément simplificateur pour l'optimisation des requêtes imbriquées ou de regroupement, deux types de requêtes existant également dans le contexte relationnel. Ceci est dû au fait que, à l'inverse du modèle relationnel, le modèle objet comprend les structures imbriquées.

1. Bloc de données rangées séquentiellement sur un disque, dont la taille est fixe.

Nos travaux sur l'optimisation d'OQL [38, 40, 39, 35] ont été réalisés en collaboration avec Claude Delobel et Guido Moerkotte.

Documents Structurés

Les données actuellement accessibles sur le Web sont pour la plupart textuelles. Leur interrogation repose généralement sur une recherche par mot-clé dont l'inconvénient majeur est qu'elle est souvent trop peu sélective. Si certains systèmes de recherche d'information permettent également une interrogation basée sur la structure des documents, ces facilités sont très limitées et ne permettent pas, par exemple, de trouver les documents dont un des auteurs est l'éditeur de la norme SQL. Les SGBD reposent sur des langages de requêtes offrant justement ce type d'interrogation. Par contre, ils nécessitent une connaissance parfaite de la structure des données interrogées et n'autorise pas les recherches par mot-clé.

Afin de faciliter la programmation d'applications nécessitant l'intégration de données textuelles, nous avons développé des techniques bases de données mariant ces deux types d'interrogation. Notre contribution dans ce domaine est triple. Elle concerne (i) la modélisation des documents, (ii) leur interrogation déclarative et (iii) efficace.

Nous avons développé une technique originale pour intégrer, de façon virtuelle ou réelle, des documents dans une base à objets. Cette technique repose sur l'annotation de grammaires par des programmes bases de données. Une étude plus approfondie des documents SGML [66] [2] nous a également amené à comprendre les limitations du modèle ODMG et l'atout que représenterait son extension aux unions de types.

Nous avons étendu le langage OQL pour permettre une interrogation sans connaissance préalable de la structure des documents. Cette extension, appelée POQL ou OQL-Doc, offre la possibilité de rechercher des documents (ou toute autre donnée de la base) par mot-clé mais également d'interroger de façon homogène structures et données. Comparé à ces prédécesseurs [17, 18, 24, 56, 58, 71, 76], POQL présente un plus grand pouvoir d'expression tout en conservant une sémantique simple.

Enfin, nous avons conçu de nouvelles techniques d'optimisation adaptées au langage et reposant sur une extension de notre algèbre objet. Elles permettent notamment (i) de ne charger que les données nécessaires à l'évaluation d'une requête lorsque les documents interrogés résident à l'extérieur de la base et (ii) l'utilisation d'index plein-texte, éventuellement en combinaison avec des index structurés.

Serge Abiteboul, Vassilis Christophides (dont c'est en partie le sujet de thèse [27]), Tova Milo, Guido Moerkotte, Michel Scholl et Jérôme Siméon ont contribué à ce travail [2, 3, 4, 6, 28, 29, 30, 33] qui a été validé et partiellement supporté par deux projets Esprit R&D : Aquarelle et WIRE. Plusieurs prototypes ont été développés par Vassilis Christophides, Jérôme Siméon et Fanny Wattez. Un des prototypes a fait l'objet d'un transfert vers la société Euroclid.

Chargement en Masse

Les nouvelles applications (telles que bibliothèques numériques, serveurs Intra ou Internet, commerce électronique, entrepôts de données, etc.) nécessitent souvent l'intégration de

2. SGML est un format très populaire dans certaines industries comme l'aéronautique. Il est à la base des deux formats HTML et XML.

données provenant de différents systèmes aux formats potentiellement hétérogènes. Une façon de les concevoir consiste à convertir puis charger toutes les données dans un système centralisateur, généralement un SGBD. Si la taille des données est importante (par exemple, un entrepôt de données ou un catalogue de galerie marchande électronique), le chargement peut nécessiter plusieurs heures, voire jours, et il est donc essentiel de fournir des outils pour optimiser ce processus.

Si la vitesse est un critère important pour un outil de chargement en masse, deux autres critères sont également essentiels : (i) les données migrées doivent être organisées sur le disque de façon à minimiser les temps de réponse des applications futures et (ii) le processus de migration doit être correctement décomposé afin d'éviter des pertes d'effort importantes en cas de défaillance pendant le transfert et une saturation de la mémoire vive.

L'état de l'art dans ce domaine est particulièrement pauvre. Les outils proposés par les éditeurs de SGBD [46, 55, 64, 85, 87, 89, 91] nécessitent un travail de développement considérable ou sont inefficaces. A l'exception de [107, 108] qui s'intéressent principalement au problème de la saturation de mémoire, il n'existe pas de travaux de recherche sur le chargement efficace de données.

Nous avons conçu une méthode qui repose sur une spécification déclarative du chargement et une optimisation algébrique qui prend en compte vitesse, bonne organisation des données et décomposition. Outre l'apport que représente l'ajout de déclarativité à un processus qui jusque là requérait un important travail de programmation, l'originalité de ce travail tient à sa généralité (la méthode est applicable à la majorité des types de données structurées), mais surtout à l'approche algébrique sur laquelle il repose.

Un programme de chargement est décomposé en une séquence d'expressions algébriques, chacune créant une partie de la base cible par effets de bord. Afin d'optimiser un tel programme, il est essentiel de comprendre les liens sémantiques existant entre ses différentes composantes mais également les effets de bord qu'elles sous-tendent. Pour ce faire, nous avons introduit une nouvelle classe d'équivalences algébriques dont les preuves reposent sur des propriétés naturelles des fonctions de mises à jour d'une base : idempotence de la création d'objets (la création d'un objet existant est sans effet) et insensibilité à la valeur nulle (la mise à jour d'une valeur indéfinie n'affecte pas la base).

Ce travail, publié dans [9], a été réalisé en collaboration avec Sihem Amer-Yahia, dont c'est le sujet de thèse [8], et Claude Delobel. L'approche a été validée par un prototype de chargement d'une base relationnelle quelconque dans le système O_2 qui a été développé par Sihem Amer-Yahia et Ségolène Galabert.

Données Hétérogènes

Nous avons vu qu'une façon de construire des applications distribuées consistait à répliquer l'information dans un système unique. Une autre manière d'aborder le problème consiste à conserver l'information répartie sur plusieurs systèmes hétérogènes et à concevoir un système "médiateur" offrant une vue intégrée à l'utilisateur. L'interrogation du système peut alors se faire par navigation (par exemple, depuis le Web), via une API ou au travers de formulaires plus ou moins sophistiqués. Chaque requête est décomposée en plusieurs sous-requêtes sur les différents systèmes, les résultats sont ensuite convertis dans le format du médiateur puis intégrés pour donner la réponse finale. Ce type d'architecture, encore peu utilisé, est maintenant largement préconisé pour la réalisation d'applications distribuées sur un réseau local, telles que les serveurs Intranet. Dans le chapitre 5, nous présentons un système médiateur basé sur (i) un modèle de données à même de comprendre tout type de formats, (ii) un modèle d'opérations permettant aux différents systèmes de préciser les requêtes qu'ils sont capables d'évaluer et (iii) un langage déclaratif pour spécifier intégration et conversion. Ces travaux ont été menés dans le contexte du projet Esprit R&D OPAL et validés par un prototype utilisé dans les applications pilotes des sociétés Bosch, PSA et Fatronik.

La systèmes médiateurs peuvent être regroupés en deux familles. Les systèmes dits structurés [21, 25, 41, 59, 53, 74, 75, 84, 94, 102, 110], reposent sur un modèle très typé pour représenter les données. Leur mise en place nécessite un important travail de programmation : l'interface (également appelée adaptateur) entre chaque système et le médiateur consiste en un programme traduisant les données importées vers le schéma de l'application intégrée. Ces programmes sont longs et parfois difficiles à concevoir (notamment lorsqu'il s'agit d'intégrer des données peu typées comme des documents XML) et non réutilisables (puisque conçus pour une application particulière). Les systèmes semistructurés [50, 92] reposent quant à eux sur une approche sans type. Les données des différents systèmes sont importées sous forme d'un graphe arbitraire et leur intégration est spécifiée à l'aide d'un langage déclaratif. Les adaptateurs sont faciles à implanter (un simple arbre syntaxique peut constituer une exportation valide) et génériques (c.à.d., non liés à une application particulière). Par contre, l'absence de type interdit toute vérification des programmes et ne permet pas d'optimiser efficacement les requêtes de l'utilisateur. De plus, aucune des deux familles de médiateurs ne proposent d'outils déclaratifs pour convertir les données intégrées en un autre format (par exemple, HTML pour une navigation depuis le Web ou des objets JAVA pour permettre l'écriture d'applications plus élaborées). Si certains systèmes offrent une interface, celle-ci est (i) soit fixe et non adaptable aux désirs d'un utilisateur, (ii) soit dédié à un format particulier (HTML pour [50]).

Le modèle de données YAT, que nous présentons dans le chapitre 5, s'inspire d'une première expérience dans la famille des modèles semistructurés [5]. Une information cohérente sur le type des données manipulées nous a rapidement paru essentielle. L'originalité et la force de YAT viennent justement de sa capacité à représenter des données arbitraires mais également (et facultativement) leur type et/ou leur système de types. La connection entre les différents niveaux de représentation est donnée par un mécanisme d'instanciation. Le modèle offre une grande flexibilité, permettant d'englober naturellement aussi bien des données très structurées (comme celles des SGBD) que d'autres qui le sont moins (des documents XML ou HTML par exemple), tout en conservant le maximum d'information sur les types. YAT combine ainsi les avantages des familles structurées et semistructurées.

Le modèle opérationnel de YAT comprend une algèbre et permet aux différents systèmes intégrés de spécifier les opérations qu'ils peuvent évaluer. A la fois extensible et spécialisable, il peut comprendre de façon générique aussi bien des langages très structurés, comme SQL ou OQL, que les primitives plein-texte des systèmes de recherche d'information. YAT est ainsi simple à utiliser (du fait de sa généricité) tout en offrant des possibilités d'optimisation qui vont au-delà de celles des systèmes structurés.

Un langage utilisateur, conçu au-dessus de ce modèle, permet de restructurer, intégrer et interroger les données. Ce langage, appelé YATL, offre notamment à l'utilisateur la possibilité de manipuler différents types de collection (ordonnées ou non, avec ou sans duplicats) et de conserver la correspondance existant entre les données sources et leurs images dans l'application intégrée. Le mécanisme d'instanciation du modèle de données est étendu aux programmes et autorise : (i) leur typage, composition ou combinaison cohérente, et (ii) une programmation simplifiée par raffinement et personnalisation (par exemple, un programme de conversion ODMG-HTML peut être instancié par un schéma ODMG particulier pour ensuite être personnalisé).

Ce travail publié dans [37, 98], a été réalisé avec Jérôme Siméon dont c'est le sujet de thèse [97], Claude Delobel et Vassilis Christophides. Jerôme Siméon a également implanté le prototype qui a été utilisé par nos partenaires industriels.

Chapitre 1 : Le standard OQL

Les systèmes de gestion de bases de données relationnels (SGBDR) sont nés dans les années 70 d'un concept fort : l'indépendance entre niveaux physique et logique. Le niveau physique gère l'organisation des données sur le disque et est en charge de minimiser l'espace nécessaire au stockage et les temps d'accès aux données. Le niveau logique représente l'interface entre le système et le programmeur. Ce dernier voit les données comme de simples tables à deux dimensions qu'il peut interroger et manipuler au travers d'un langage déclaratif, SQL pour la plupart des systèmes. Cette dichotomie logique/physique est rendue possible par un module particulièrement important, l'optimiseur de requêtes, qui cherche la meilleure façon d'évaluer les requêtes du programmeur en fonction de la connaissance qu'il a de l'organisation physique de la base. L'intérêt de cette approche est double : (i) l'accès aux données est simplifié et (ii) une réorganisation des données sur le disque n'invalide pas les programmes existants.

Ces avantages ont hélas une contrepartie. Pour être optimisable, le langage doit rester simple. Notamment, SQL ne permet ni la récursion, ni l'utilisation arbitraire de variables temporaires. En conséquence, les applications relationnelles consistent généralement en des programmes C ou Cobol intégrant des appels à un interprète SQL. Cette programmation à deux langages, dont les systèmes de types sont différents, est difficile.

Au début des années 90, une nouvelle génération de systèmes de gestion de bases de données apparaît : les systèmes à objets (SGBDO)[12, 55, 87, 86, 88, 106]. L'atout principal de ces nouveaux venus est double. D'une part, ils reposent sur un modèle de représentation riche, plus apte à supporter les applications émergentes de l'époque : applications géographiques, conception ou publication assistée par ordinateur, etc. D'autre part, ils proposent des langages de programmation objets, impératifs mais complets (C++, Smalltalk et maintenant Java), ce qui a notamment pour effet d'éliminer la nécessité de gérer le dysfonctionnement structurel engendré par l'immersion d'un langage base de données dans un langage de programmation. A l'origine, seul le système O_2 de la société O_2Technology (maintenant intégrée au groupe Ardent Software [12]) propose, en complément, un véritable langage de requêtes. Ce passage à une programmation impérative signifie-t-il la perte de l'indépendance logique-physique?

La réponse est non, et oui. De fait, l'utilisateur voit un monde de données objet logique qui est très différent des fichiers et enregistrements physiques stockés sur le disque. Le SGBD est toujours responsable d'établir de façon transparente la correspondance entre ces deux mondes. Les données peuvent donc être réorganisées physiquement sans invalider les programmes. Par contre, ce type de programmation interdit soit l'utilisation de structures d'accès secondaires rapides tels que les index, soit leur dynamicité (le fait de pouvoir les ajouter ou supprimer en fonction des besoins en espace ou efficacité). De plus, le temps d'exécution d'un programme varie en fonction des réorganisations et un bon programme peut devenir très inefficace sans que le système puisse intervenir, la richesse et la nature impérative des langages rendant leur optimisation impossible.

C'est pour cela que le système O_2 propose un langage de requêtes déclaratif en complément de son langage de programmation objet. O_2 est issu d'un travail de recherche effectué à l'INRIA, dans le cadre du GIP Altaïr [3], et auquel nous avons participé. En collaboration avec François Bancilhon et Claude Delobel, notre contribution a justement porté sur la conception du langage de requêtes O_2SQL. Nous pensions, à juste titre, qu'il était possible d'avoir le meilleur des deux mondes. (1) Le langage de programmation objet d'O_2 permet l'écriture d'applications complexes, (2) le langage de requêtes représente une alternative pour interroger directement la base ou écrire des programmes ou sous-programmes pouvant être optimisés. Ainsi, nous conservons une véritable indépendance logique-physique, en évitant un dysfonctionnement structurel puisque les deux langages reposent sur le même modèle de données.

Les représentants d'O_2Technology au sein de l'"Object Database Management Group" (ODMG) [4] n'eurent aucun mal à imposer ce point de vue et O_2SQL devint un standard sous le nom de OQL dès 1991. Evidemment, étant le seul candidat industriel, ses chances étaient particulièrement bonnes. Nous préférons penser que ce choix a été guidé par la pertinence du concept et l'élégante simplicité du langage. Quoiqu'il en soit, OQL ne connut pas un grand succès dès le départ et, de fait, jusqu'au milieu de la décennie seul O_2 possédait un interprète/optimiseur. L'émergence des applications distribuées sur le Web a changé cela.

Simple, interprétable, optimisable, un langage de requêtes est un excellent outil pour échanger des données entre systèmes. L'OMG, qui standardise la programmation distribuée par objets, a d'ailleurs reconnu ce fait en ajoutant SQL et OQL à son catalogue de standards. Les deux langages se retrouvent donc en compétition. SQL étant très largement diffusé et soutenu par une industrie puissante a certainement pris de l'avance. A l'origine simple et bien défini, SQL souffre de deux maux. (1) Il a été conçu sur le modèle relationnel, modèle de tables à deux dimensions, et a donc du mal à s'adapter aux structures riches des objets. (2) Il a vécu presque trente ans au sein d'un organisme de standardisation et a donc été de nombreuses fois modifié et étendu, les dernières extensions, non encore officielles, incluant la manipulation des objets. Il a ainsi perdu sa simplicité d'origine et certainement sa sémantique claire. A l'opposé, OQL a été défini dès le départ sur un modèle objet, il est toujours simple et clairement défini. Nous pensons qu'il représente l'avenir.

En collaboration avec O_2Technology, nous avons travaillé à la promotion d'OQL au sein de la communauté industrielle. Pour ce faire, nous avons fait évoluer le langage de façon à offrir plus de convivialité et à faciliter son adoption par des programmeurs SQL. Ce chapitre est un survol bref et informel, on trouvera plus d'informations dans [14, 26] et l'article [34] donné en annexe. Il est organisé comme suit. Tout d'abord, nous présentons le modèle objet à l'aide d'un exemple. Nous introduisons ensuite le langage d'origine avant de présenter les modifications que nous avons apportées. Finalement, nous expliquons pourquoi nous sommes optimistes et évoquons les écueils qui guettent OQL.

---

3. Altaïr était un groupement d'intérêt public fondé par IN2 (la filiale informatique du groupe Intertechnique), BULL, l'INRIA (Institut National de Recherche en Informatique et Automatique) et le LRI (le laboratoire de recherche informatique de l'université de Paris XI, Orsay). Il a été mis en place en Septembre 1986, pour une durée de cinq ans, au terme desquels la société O_2Technology a été créée pour commercialiser le produit développé.

4. Organisme de normalisation pour les SGBDOO dépendant de l'Object Management Group(OMG) et qui s'est constitué en 1991.

1.1 Le modèle ODMG par l'exemple

Le modèle de données ODMG reprend les concepts qui ont été introduits dans [13]. Nous les illustrons ici en utilisant les exemples simples de schéma et de base des figures 1.1 et 1.2.

Classes	
Country	name: String cities: set(City) currency: struct(name: String, rate: Float) **Method** Add_city(City): Country
City	name: String country: Country map: Bitmap hotels: list(Hotel) monuments: set(Monument)
Hotel	name: String address: struct(street: String, city: City) facilities: set(String) category: Integer closing_days: set(String) price: Float **Method** check_price(): Boolean
Monument	name: String address: struct(street: String, city: City) description: Text closing_days: set(String) **Method** is_in(Country): Boolean
Historical_Hotel	**Inherits** Hotel, Monument **Method** check_price(): Boolean

Names	**Types**
Countries	list(Country)
Cities	set(City)
Paris	City
Hotel_of_the_Month	Hotel
Hotels	set(Hotel)
Monuments	set(Monument)

FIG. 1.1: *Un schéma objet*

Une base objet est composée d'objets et de valeurs complexes, regroupés respectivement en classes et types, et accessibles par des points d'entrée. Les classes sont organisées en une hiérarchie sur laquelle repose la notion de sous-typage. Nous expliquons ci-dessous ces différents concepts.

Points d'entrée et persistance : Les point d'entrées de la base sont caractérisés par des noms, une valeur ou un objet étant affecté à chacun. Seuls les objets ou valeurs accessibles depuis un nom, directement ou indirectement, sont persistants. L'ODMG propose aussi un second modèle de persistance que nous ne présentons pas ici (voir [26]).

Countries	=	$\langle$ c1, c2, ...$\rangle$
Cities	=	{ci1, ci2, ... }
Hotels	=	{ h1, h2, ... }
Paris	=	ci1
Hotel_of_the_Month	=	h1

....
c1 : [name: “France”, cities: {ci1, ci2, ...}, currency: [name: “FF”, rate: 1]]
...
ci1 : [name: “Paris”, country: c1, Map: m1, hotels: <h1, h2, ...>]
...
h1 : [name: “Hotel du Pont Neuf”, address: [street: “Malaquais”, city: ci1], ...]

FIG. 1.2: *Une base objet(partielle)*

Types et classes : Les types sont obtenus par application récursive de constructeurs (ensemble/set, liste/list, multi-ensemble/bag, tableau/array, n-uplet/struct) sur des types atomiques (entier/integer, réel/float, caractère/char, booléen/boolean), des types préconstruits (chaîne de caractères/string, Bitmap, etc.) et des classes définies par l'utilisateur. Ainsi, le nom *Countries* est défini comme ayant le type liste d'objets de la classe *Country*.

Une classe est caractérisée par la structure commune (ou type commun) des objets lui appartenant et par l'ensemble des opérations, encore appelées méthodes, qu'elle connaît. Toutes les classes de notre schéma étant de type n-uplet, la figure 1.1 présente directement les attributs de ces n-uplets. Ainsi, la classe *Country* a une structure n-uplet comprenant trois attributs, parmi lesquels *currency* représente un autre n-uplet dont les attributs *name* et *rate* sont atomiques. La méthode *Add_city* permet d'affecter une nouvelle ville aux objets de cette classe. Dans un contexte “pur objet”, les méthodes représentent les seuls moyens d'accès aux objets. En d'autres termes, un objet ne peut être lu ou modifié que par appel de méthode. On parle alors d'encapsulation stricte. OQL ne respecte pas ce principe. La raison principale est que ce langage ne permet pas la modification des données et ne peut donc violer l'intégrité de la base.

Objets et valeurs : Chaque objet possède un état (en général une valeur même si un objet peut en encapsuler un autre) et un identificateur qui est unique et qui ne dépend pas de la valeur de l'objet. Un objet peut donc changer de valeur sans changer d'identité. Les identificateurs permettent de référencer les objets, qui peuvent être ainsi partagés et cycliques. Sur la figure 1.2, les objets **c1** et **h1** partagent l'objet **ci1**, lequel les référence à son tour créant ainsi des cycles. La cohérence des cycles peut être, dans une certaine mesure, gérée par la notion de propriété inverse que nous n'avons pas illustrée ici. Par exemple, il est possible de spécifier que l'attribut *country* de la classe *City* est inverse de l'attribut *cities* de *Country*. Le système garantit alors la cohérence du cycle.

Les valeurs n'ont pas d'identité, elles ne sont donc pas partageables et ne peuvent pas évoluer : changer une valeur consiste à en créer une autre. C'est pour cette raison que l'on désigne aussi les valeurs sous le terme d'“objet immuable”.

Héritage et sous-typage : Une classe (ou sous-classe) peut hériter d'une ou de plusieurs autres classes (ou super-classes). Le concept d'héritage recouvre un partage de structure et de comportement. Par exemple, la classe *Historical_Hotel* hérite des attributs et méthodes des classes *Hotel* et *Monument*. Elle est donc d'un type n-uplet dont les attributs sont, entre autres, *name*, *address*, *facilities*, ..., *description*, *closing_days*. Elle connaît également les méthodes *check_price* (qui vérifie que le prix d'un hotel correspond à sa catégorie) et *is_in* (qui dit si un monument se trouve dans un pays donné). Notons que seule la méthode *check_price* apparaît dans la définition de la nouvelle classe. Ceci correspond à une redéfinition de l'opération (changement de code). Il est également possible de raffiner par sous-typage les paramètres de la méthode. Nous expliquons maintenant le sous-typage.

Les classes sont organisées en hiérarchie[5] dont la racine est une classe pré-définie appelée *Object*. Cette hiérarchie définit une relation de sous-typage explicite (c.a.d., définie par l'utilisateur, le système ne servant que de vérificateur de cohérence). Il existe également un ordre implicite entre les types qui repose sur l'héritage et une notion standard du sous-typage. De façon schématique, un type est sous-type d'un autre si il est plus spécifique (par exemple, un attribut de plus dans un n-uplet, un ensemble défini sur une sous-classe plutôt que sa super-classe). Le sous-typage permet de considérer un objet (resp. une valeur) d'une certaine classe (resp. un certain type) comme appartenant à ses super-classes (resp. super-types). Il est utilisé notamment pour vérifier la cohérence de la relation d'héritage. Le type d'une sous-classe doit être un sous-type de celui de ses super-classes. Reprenons l'exemple de la classe *Historical_hotel*. Par définition, son type est:

```
struct (name: String,
        address: struct(street: String, city: City),
        facilities: set(String),
        category: Integer,
        closing_days: set(String),
        price: Float,
        description: Text)
```

Notons que certains attributs, comme *address*, apparaissaient dans chacune des super-classes. Ces attributs ayant un type commun, cela n'a pas soulevé de conflit. Un conflit apparaît quand les types d'un même attribut n'ont pas de relation de sous-typage et est généralement résolu par un renommage implicite ou explicite des attributs. Si l'attribut *address* avait été défini dans la classe *Hotel* comme étant de type:

```
struct(street: String, city: City, country: Country)
```

la définition aurait également été valide et le type de l'attribut *address* dans *Historical_Hotel* aurait été le plus petit des deux, c'est à dire :

```
struct(street: String, city: City)
```

5. Il s'agit ici d'un abus de langage, les classes étant en fait organisées en semi-treillis.

1.2 OQL originel

Les atouts d'OQL tiennent à sa nature fonctionnelle qui lui donne adéquation avec le modèle objet, simplicité (d'utilisation, de définition et d'implantation) et extensibilité. Les types d'une base objet sont construits de façon fonctionnelle par application récursive de constructeurs sur des types atomiques ou des classes. Les requêtes sont construites de la même façon. A chaque type atomique et à chaque constructeur est associé un ensemble de requêtes (ou opérations) de base. Les classes ajoutent leur méthodes. Les requêtes complexes sont ensuite construites par simple combinaison de requêtes de base, la seule restriction étant qu'une requête doit être bien typée. Nous illustrons brièvement ce principe à l'aide de quelques exemples représentatifs.

Les briques de base de la construction de requêtes sont les opérations élémentaires d'arité nulle.

Exemples 1.1 *Requêtes constantes*
2
"Paris"
Paris
sont des exemples de ce type de requêtes. Elles retournent, respectivement, 2, "Paris" and **ci1** qui est l'objet associé au nom *Paris.* □

Puis, les requêtes sont combinées en fonction de leur type.

Exemples 1.2 *Simples Combinaisons*
2 + 2
"Paris" + "sur Seine"
Paris.name
count(Cities)
sont des exemples de combinaison simples. Notons que la requête de base '+' est surchargée : elle représente l'addition entière dans la première requête, la concaténation de chaînes de caractères dans la seconde, mais aussi l'union d'ensembles ou la concaténation de listes. L'opérateur '.' est utilisé pour extraire un attribut d'une valeur ou d'un objet n-uplet et *count* retourne le nombre d'éléments d'une collection : dans ce cas, un ensemble de villes. □

Le principe de base devrait maintenant être compris. Ajoutons juste un niveau de complexité supplémentaire.

Exemples 1.3 *Combinaisons plus complexes*
Paris.name = "Paris"
Paris.hotels[0]
struct(cityname: Paris.name, countryname: Paris.country.name)
Monument(name:"Tour Eiffel",
address: **struct**(street: "Quai Branly", city: Paris)
La première requête retourne **true**, la seconde extrait le premier élément de la liste des hotels parisiens, la troisième construit un n-uplet et la quatrième un objet de la classe *Monument.* Notez que l'attribut *address* de la troisième requête est obtenu par ce qui est communément appelée une expression de chemin et qui dénote une combinaison d'opérations ".". Finalement, notez que seuls les attributs *name* et *address* de l'objet *Monument* ont été assignés. Les attributs manquants prendront une valeur par défaut (la chaîne vide pour *description*, l'ensemble vide pour *closing_days*). □

La plupart des programmeurs base de données ne reconnaîtront pas là un langage de requêtes. Effectivement, une fonctionnalité essentielle manque à l'appel : les accès associatifs. En OQL, ceux-ci sont donnés par des itérateurs dont la syntaxe n'est pas sans rappeler SQL mais dont nous verrons qu'ils sont en fait différents.

Exemples 1.4 *Accès associatifs*

La première requête sélectionne les hotels français dont les prix sont trop élevés par rapport à leur catégorie. Pour chacun de ces hotels, elle construit un n-uplet avec deux attributs : le nom de l'hotel et une booléenne précisant si l'hotel propose des salles de bains individuelles.

```
select  struct( name: h.name, bathroom: "bath" in h.facilities )
from    h in Hotels
where   h.address.city.country.name = "France" and not(h− >check_price())
```

Les experts noteront quelques différences significatives entre SQL et OQL. Tout d'abord, la clause **Select** ressemble peu à son équivalent SQL. OQL accepte n'importe quelle sous-requête bien typée dans cette clause, SQL régit son utilisation par une grammaire complexe. Le résultat est un plus grand pouvoir d'expression pour OQL (par exemple, la sous-requête **in** n'est pas dans la grammaire SQL alors qu'elle ne pose aucun problème particulier) tout en offrant un apprentissage plus simple (petite grammaire, une seule règle de composition). Finalement, les clauses **From** et **Where** semblent ne différer de SQL que par la syntaxe. De fait, elle sont très différentes puisque, suivant le même principe de combinaison bien typée, elles permettent l'utilisation de n'importe qu'elle requête retournant une collection pour la première et une booléenne pour la seconde (notez l'appel de la méthode booléenne *check_price* dans le **where**).

La requête suivante compare l'hotel qui a été sélectionné comme *Hotel_of_the_Month* (voir la figure 1.2) avec les autres hotels. Elle retourne un n-uplet unique comprenant deux attributs : l'hotel du mois et les hotels qui proposent au moins le même nombre de services pour un prix moins élevé. Elle illustre le fait qu'un itérateur peut être combiné comme n'importe qu'elle requête.

```
struct( hotel:           Hotel_of_the_Month,
        cheaper_hotels:  select  h
                         from    h in Hotels
                         where   count(h.facilities) >=
                                   count(Hotel_of_the_Month).facilities
                                 and h.price < Hotel_of_the_Month.price )
```

Finalement, notre dernière requête retourne, pour chaque ville française dont au moins un monument est ouvert le dimanche, son nom et ses hotels 3 étoiles.

```
select  struct ( city: c.name, 3hotels:  select  h
                                         from    h in c.hotels
                                         where   h.category=3 )
from    co in Countries
        c in co.cities
where   co.name = "France"
        exists m in c.monuments: not ("sunday" in m.closing_days)
```

On peut noter l'imbrication d'itérateur dans la clause **Select** qui associe à chaque ville ses hotels trois étoiles. Notons aussi, dans la clause **where**, l'itérateur de quantification existentiel (**exists**), dont le pendant universel est **forall**. Les itérateurs de quantification ont deux clauses, l'une précisant la collection quantifiée, l'autre le prédicat appliqué. Naturellement, ces clauses peuvent être définies par n'importe quelle requête de type approprié (collection et booléen). La clause **From** contient deux variables, la seconde étant définie par rapport à la première. Ce type de définition dépendante, qui n'existe pas en SQL, permet de naviguer à travers des collections imbriquées et est donc indispensable dans le contexte objet (même si dans le cas présent, la requête aurait pu être formulée différemment). □

Avant de présenter les évolutions du langage, un mot sur le typage des itérateurs. Les collections définissant les variables de la clause **From**, qu'elles soient de type liste, ensemble ou autre, sont systématiquement converties en multi-ensembles et le résultat des **Select-From-Where** précédents est donc de type multi-ensemble. Pour obtenir un ensemble (c.à.d., une suppression des duplicats), l'utilisateur devra invoquer sur le résultat la fonction de conversion **distinct**. S'il souhaite une liste, il devra appliquer un tri (nous en parlerons plus loin). Enfin, s'il sait que la collection retournée ne contient qu'un élément qu'il veut extraire, il utilisera la fonction **element**. Ceci est une illustration du fait que le langage couvre effectivement l'intégralité du système de types objet.

1.3 Evolutions

Cette section présente quelques unes des modifications que nous avons apportées au langage. Notre objectif premier était de rendre le langage compatible avec SQL sur les collections de n-uplets simples (c.a.d., plus ou moins des relations). Le groupe ANSI-SQL a participé à la réflexion qui a précédé ce travail dont l'objectif complémentaire était de rendre le futur SQL3 compatible avec OQL.

De fait, nous avons rapidement réalisé l'impossibilité de cette tâche. Nous illustrons ici pourquoi OQL ne peut être totalement compatible SQL sans perdre son pouvoir d'expression. La raison pour laquelle SQL3 ne peut se conformer à OQL tient à son approche grammaticale du problème, plutôt que fonctionnelle. Intégrer toute la puissance d'expression d'OQL ferait exploser une grammaire déjà très lourde. Quoiqu'il en soit, OQL a emprunté beaucoup à SQL dans cet effort de normalisation. Nous verrons que certains de ces emprunts sont très bénéfiques et d'autres moins. Quant à SQL3, il n'est toujours pas officialisé mais nous pensons que ce travail de réflexion commun influencera certains des choix qui seront faits concernant l'interrogation des objets.

Les modifications sont de trois ordres. Elles portent sur (1) la syntaxe générale des itérateurs, (2) le regroupement et le tri et (3) les valeurs nulles.

Maquillage SQL

La syntaxe SQL est souvent plus simple que celle d'OQL sur les requêtes élémentaires (par exemple, sans imbrication). Nous avons donc ajouté quelques raccourcis syntaxiques au langage. L'exemple suivant montre qu'il est maintenant parfois impossible de distinguer les deux langages.

Exemples 1.5

select	name, address
from	Hotels
where	category=3

Les différences avec le premier OQL sont les suivantes : (i) aucune variable n'est introduite dans la clause **from**, (ii) l'accès aux attributs est direct (par exemple *name* au lieu de *h.name*) et (iii) la clause **select** contient deux sous-requêtes séparées par des virgules qui correspondent à une construction de n-uplet (qui aurait requis une sous-requête **struct** dans la précédente version du langage). Le résultat est donc une collection de n-uplets avec deux attributs : *name* et *address*. Bien sûr, les accès et constructions explicites sont toujours possibles. Cette façon plus concise de formuler une requête est obtenue par simple ajout de raccourcis syntaxiques et ne modifie ni la nature fonctionnelle du langage, ni sa sémantique. Par exemple, considérez la requête précédente avec une clause **select** légèrement différente :

select	name
from	Hotels
where	category=3

La syntaxe est toujours SQL. Cependant, les sémantiques de SQL et OQL sont différentes. La requête retourne un multi-ensemble de chaînes de caractères ; en SQL elle retourne une relation, soit un multi-ensemble de n-uplets. Cette différence ne peut être comblée si l'on veut préserver la sémantique du langage et surtout permettre la construction de structures ODMG arbitraires. □

Les raccourcis syntaxiques que nous venons de présenter gomment certaines différences syntaxiques légères et sont, à notre avis, bénéfiques. Le langage gagne en simplicité d'expression sans perdre son âme. D'autres différences sont plus fondamentales. Certaines, comme nous l'illustrerons avec les requêtes de regroupement, sont impossibles à éliminer. D'autres ont requis l'addition, non pas de raccourcis, mais d'alternatives syntaxiques. A notre avis, ces ajouts sont de très mauvais goût[6], même s'il nous faut les accepter. Dans la mesure où, même sur des requêtes de type relationnel, OQL ne peut adopter un "look" 100% SQL, elles nous paraissent de plus inutiles. Le seul élément positif les concernant est qu'elles ne sont que des alternatives et qu'il est donc possible de les oublier.

Exemples 1.6 La requête suivante est OQL/SQL et retourne le nom et le prix en dollars des hotels de la base.

select	name, Price_in_Dollars: price * 6.20
from	Hotels

Notez l'utilisation du symbole ":" pour nommer le deuxième attribut des n-uplets. A l'évidence, *price* dénote ici le prix d'un hotel. Nous allons voir maintenant, que l'adoption de la syntaxe SQL a des effets pervers, et notamment celui de changer la sémantique des sous-requêtes en fonction de leur position dans une requête. C'est ce type de modification qui nous déplaît et que nous regrettons encore. La requête suivante retourne le prix moyen des hotels de la base.

select	**avg**(price)
from	Hotels

6. Il est important de signaler qu'à cet endroit du document, le "nous" dénote en fait l'auteur. D'autre avis personnels seront donnés par la suite.

La sémantique de *price* a changé : nous appliquons ici la fonction *avg* (qui calcule la moyenne arithmétique) à la collection contenant tous les prix des hotels. En utilisant la construction OQL standard, cette requête est formulée de la façon suivante.

avg (	**select**	price
	from	Hotels)

respectant ainsi la construction fonctionnelle. Dans le nouvel OQL, les deux formes sont admises. □

Tri et Regroupement

Dans la première version du langage, tri et regroupement étaient effectués par des itérateurs particuliers. Dans SQL, ces deux fonctionnalités sont implantées par deux clauses optionnelles du **select-from-where**. Nous avons adopté cette façon de faire qui, de fait, nous plaît plutôt mieux. Le tri ne posant pas de problème particulier, nous ne l'illustrerons pas ici. Le regroupement est un peu plus délicat dans la mesure où il présente une sémantique différente de celle de SQL sur les relations.

Pour comprendre le regroupement, il est nécessaire de bien saisir la sémantique de produit cartésien de la clause **from**. En SQL, les collections contiennent toutes des n-uplets. En conséquence, le résultat d'un produit cartésien est obtenu en concaténant les n-uplets des deux collections sur lequel il est appliqué. Le modèle ODMG propose différents types de collection (collections d'objets, d'entiers, etc.). Nous ne pouvons donc pas appliquer un mécanisme similaire et le résultat de la clause **from** est différent : il consiste en un multi-ensemble de n-uplets avec un attribut par collection entrante.

Exemples 1.7 Considérons par exemple la requête suivante qui retourne les hotels de Caen dont le prix est correct et qui offrent la télévision dans toutes les chambres.

select	**distinct** h
from	h **in** Hotels,
	f **in** h.facilities
where	h->check_price() **and** h.city.name = "Caen" and f = "TV"

La clause **from** retourne un multi-ensemble dont le type est :

```
bag(struct(h: Hotel, f: String))
```

Notons que cette structure est parfaitement adaptée au traitement des autres clauses qui contiennent des opérations appliquées sur chaque élément des collections entrantes dans le but de filtrer ou de construire les éléments résultants.

Considérons maintenant la clause **group by** de regroupement.

select	*
from	h **in** Hotels
where	h->check_price() and h != Hotel_of_the_Month
group by	city: h.address.city, nb_goodies: count(h.facilities)

Notez que, en opposition à SQL, les critères de regroupement sont plus que de simples attributs (ce qui explique pourquoi nous les nommons) et que l'on peut utiliser un astérisque dans la clause **select** pour retourner directement le résultat du regroupement. Ceci est impossible en SQL puisque les collections imbriquées n'existent pas. Le type du résultat de la requête est donné ci-dessous :

```
set(struct(city: City,
          nb_goodies: Integer,
          partition: bag(struct(h: Hotel))))
```

Ceci paraît assez raisonnable : chaque critère est représenté par un attribut, un attribut supplémentaire dénote les partitions résultantes, le type d'une partition est celui de la clause **from**, le résultat est un ensemble puisque le regroupement garantit l'absence de duplicats. La sémantique est celle attendue : nous regroupons tous les hotels satisfaisant les conditions de la clause **where** en fonction de leur ville et de leur nombre de services. Ainsi, à chaque couple distinct (*city, nb_goodies*) sont associés les hotels correspondants.

Dans certains cas, et notamment celui-ci, l'utilisateur préférerait une partition de type collection d'hotels plutôt que de 1-uplets. Mais, (i) un regroupement peut être effectué sur plusieurs collections auquel cas la construction n-uplet est indispensable et (ii) cette construction issue du **from** est nécessaire aux autres clauses. Ainsi, dans l'exemple, nous utilisons la variable h pour comparer chaque hotel à l'hotel du mois.

Voyons maintenant comment appliquer des opérations sur les partitions obtenues. Nous illustrons ceci en redéfinissant uniquement la clause select, les autres restant inchangées.

Compter est simple :

select	city, nb_goodies, nb_hotels: **count**(partition)

Le type du résultat est alors :

```
set(struct(city: City,
           nb_goodies: Integer,
           nb_hotels: Integer))
```

Restructurer les partitions est également facile :

select	city.name, nb_goodies, hotels:	**select** h.name, h.price
		from partition

Ici, l'attribut hotels représente une collection de n-uplets possédant deux attributs: *name* et *price*. Ce type de requêtes n'est pas exprimable en SQL. La raison en est que chaque partition est vue comme une séquence de partitions distinctes, une par attribut. Ainsi, à chaque couple (*city, nb_goodies*) sont associées une collection de noms d'hotels, une collection de prix, etc. La connection existant entre le nom d'un hotel et son prix est perdue. Cette perte de sémantique a cependant un avantage, elle offre un plus grande facilité pour appliquer des opérations agrégats. En OQL, ceci nécessite l'utilisation d'un itérateur. Par exemple :

select	city.name, nb_goodies, avg_price: avg(**select** h.price **from** partition)

□

Valeurs Nulles

A l'origine, le modèle ODMG n'avait pas de valeur indéfinie, ou valeur nulle. Seuls les objets pouvaient être indéfinis (**nil**). Cette lacune tient au fait que le modèle a été conçu pour des langages tels que C++ qui, eux non plus, n'ont pas de valeurs nulles (par exemple, une variable de type entier ne peut avoir que des valeurs entières définies). Elle était la cause d'un type d'erreurs particulier et relativement fréquent dans les requêtes OQL : "accès à l'object **nil**".

Exemples 1.8 Supposez qu'un objet **nil** soit contenu dans l'ensemble Hotels. Considérons à présent la requête suivante :

```
select  h.price
from    h in Hotels
```

Le résultat est une collection de valeurs de type réel. Si ce type ne comprend pas la notion de valeur indéfinie, que doit-on donc faire de la sous-requête h.price lorsque h est égal à **nil**? Nous pourrions l'ignorer, mais ce ne serait pas sémantiquement correct. D'où, l'erreur en cours d'exécution.

A première vue, cette requête ne paraît pas compliquée et l'on pourrait penser qu'une clause **where** contenant la condition *h!=***nil** serait une solution raisonnable. Il suffit de considérer une requête un peu plus complexe pour voir les limites de cette solution. Supposons que la classe *Text* possède une méthode *contains*. La requête suivante retourne les monuments qui se trouvent à Paris ou dont la description contient le mot "liberté".

```
select  m
from    m in Monuments
where   m! = nil and
        ((m.address.city ! = nil and m.address.city.name = "Paris")
           or (m.description ! = nil and m.description->contains("liberté")))
```

Premier inconvénient, la requête est plutôt lourde. De plus, l'ayant écrite avec grande précaution, le programmeur pense qu'il est protégé d'erreurs possibles à l'exécution. Hélas, ce n'est pas le cas. Pour optimiser, il est nécessaire de comprendre comment déplacer les opérations, et particulièrement les prédicats. Pour cela, les prédicats sont généralement mis sous une forme normale, conjonctive (cnf) ou disjonctive (dnf). Les deux sont aussi problématiques l'une que l'autre. Nous montrons pourquoi la requête n'est pas sûre en utilisant une cnf.

```
m != nil and
(m.address.city != nil or m.description != nil) and
(m.address.city != nil or m.description->contains(``liberté'')) and
(m.address.city.name = ``Paris'' or
                 m.description != nil) and
(m.address.city.name = ``Paris'' or
                 m.description->contains(``liberté''))
```

Partant de ce prédicat, l'optimiseur a le choix entre plusieurs options, parmi lesquelles ne rien faire. Ce n'est peut être pas la meilleure solution mais cela ne change rien au problème et permet d'aller droit au but. Ainsi, le premier objet Monument entre en jeu et supposons que : (i) il n'est pas égal à **nil**, (ii) il possède une description qui contient le mot "liberté" mais

(iii) que la ville dans laquelle il est situé est indéterminée. En conséquence, il passe les trois premiers tests de la conjonction mais génère une erreur sur le quatrième. Soit, l'optimiseur n'a pas été intelligent et aurait dû inverser les deux prédicats de la disjonction incriminée. Le problème se déplace alors sur les monuments parisiens possédant une description indéterminée qui passent les quatre premiers tests mais pas le dernier. □

Une première idée pour solutionner ce problème récurrent consistait à introduire des opérateurs **cand** et **cor** : leur sémantique est celle du **and** et **or** mais ils ne peuvent être ni déplacés ni mis sous forme normale. Cela signifiait une perte de déclarativité et d'optimisation importante. Nous avons donc, une fois de plus, emprunté à SQL et introduit des valeurs nulles dans le langage. De façon très schématique, l'accès a une propriété d'un objet indéfini retourne la valeur nulle et tout prédicat sur une valeur nulle retourne faux. La sémantique est celle de SQL avec les désagréments qu'elle comporte (a=b n'est plus équivalent à **not**$(a!=b)$). La requête précédente peut maintenant être formulée de la façon suivante, sans risque d'erreurs.

select	m
from	m **in** Monuments
where	m.address.city.name = "Paris"
	or m.description->contains("liberté")

1.4 OQL versus SQL

Jusqu'à présent, OQL est le seul langage de requêtes à couvrir l'intégralité du modèle objet. De plus, il est clairement défini. Ces raisons nous font penser que, malgré un départ difficile dans sa compétition avec SQL, OQL a toute ses chances. Des éléments semblent indiquer cette orientation positive. Notamment, JavaSoft a choisi d'implanter OQL dans son produit JavaBlend [68] et les ingénieurs qui travaillent sur le projet s'impliquent maintenant fortement au sein de l'ODMG. D'autres sociétés ont également choisi OQL : Apertus [11], Computer Associate (pour son SGBD Jasmine) [42] et Poet [93] qui a été le second à implanter le langage après O_2Technology. Finalement, dans l'effort récemment entrepris pour définir le langage d'interrogation de XML, de nombreuses propositions reposent sur des extensions d'OQL. Tous ces éléments nous rendent très optimistes.

Cependant, le travail des groupes de standardisation peuvent avoir des effets pervers sur un langage et, plus le groupe est important, plus le danger est grand. SQL représente une bonne illustration de ce phénomène. Nous espérons que l'ODMG saura se préserver de ces écueils et qu'OQL conservera ses qualités au cours des évolutions futures.

Chapitre 2 : Optimisation d'OQL

La façon la plus naturelle d'implanter l'interprète d'un langage consiste à se reposer sur sa définition sémantique. Ainsi, OQL ayant une sémantique fonctionnelle, il suffit d'associer un code à chaque fonction de base et itérateur du langage. Cette décomposition en fragments élémentaires rend le travail de mise en œuvre très simple (par expérience, deux ou trois semaines suffisent)[7].

Cette solution est, hélas, peu réaliste dans le contexte des SGBD où il n'est pas exceptionnel de manipuler des gigas octets de données. Elle peut engendrer une explosion de la mémoire centrale et implique des accès aléatoires et inefficaces à la mémoire secondaire, se reposant sur le *swap* du système d'exploitation plutôt que sur l'organisation physique des données. D'où la nécessité d'un optimiseur, logiciel en charge de générer le meilleur plan d'exécution d'une requête en fonction de la taille des données, leur disposition sur le disque et d'éventuelles structures secondaires d'accès rapide (ou index).

Les SGBDR ont ouvert la voie dans ce domaine, les premiers optimiseurs ont été conçus pour le langage SQL. L'accès au disque restant le facteur de coût le plus important, l'idée principale consiste à exploiter au maximum chaque bloc de données lorsqu'il est chargé en mémoire centrale. Si l'on considère, ce qui est souvent le cas, que les éléments d'une collection sont stockés de façon séquentielle sur le disque, ceci revient à optimiser les accès à ces collections. La plupart des optimiseurs s'appuient pour cela sur une approche algébrique. Les requêtes de l'utilisateur sont traduites dans une algèbre, un système de réécriture basé sur des équivalences algébriques permet alors de découvrir les différents plans d'évaluation possibles qui sont jugés en fonction d'un modèle de coût. L'optimiseur travaille suivant une stratégie de recherche plus ou moins complète, qui doit réaliser un bon compromis entre temps d'optimisation et d'évaluation de la requête. Dans ce document, nous nous intéressons à la partie algébrique de ce processus.

Si, comme SQL, OQL repose essentiellement sur le filtrage et la restructuration de collections, il a également une particularité qui rend son optimisation plus délicate : les expressions de chemin, qui permettent de naviguer suivant les liens de composition des objets[8]. Considérez, par exemple, la requête OQL suivante qui retourne les hotels français:

select	h
from	h **in** Hotels
where	h.address.city.country.name = "France"

7. Notons que le fragment le plus élaboré, l'itérateur **select-from-where** peut être codé par de simples boucles imbriquées, éventuellement suivies par un tri/regroupement.

8. Cette caractéristique est inhérente au modèle objet et devra donc être offerte pas SQL3.

A première vue, cette requête porte sur l'ensemble *Hotels* et nécessite une seule lecture en mémoire centrale de cette collection. Cependant, remarquez l'expression de chemin de la clause **where** qui, vraisemblablement, va nécessiter la lecture de toutes les villes et tous les pays de la base. En supposant, raisonnablement, qu'une ville est référencée par différents hotels et que la mémoire ne permet pas de contenir toutes les données accédées, il est possible qu'une même ville ait besoin d'être chargée plusieurs fois. Ces lectures multiples ne peuvent être gérées statiquement puisqu'elles dépendent de la valeur des objets, valeur qui n'est connue qu'à l'évaluation. Ceci explique que la navigation est source potentielle d'inefficacité, ce que certains traduisent par : OQL n'est pas optimisable.

De fait, cette analyse est rapide et fausse. En 1990, [69] propose une technique d'optimisation consistant à transformer les expressions de chemin en requêtes associatives, permettant ainsi de ramener le problème dans un contexte plus traditionnel. De plus, les SGBDOO permettent des regroupements sur disque qui, sur certaines requêtes, rendent la navigation plus efficace que d'autres techniques d'évaluation. Enfin, s'il est indéniable que le modèle objet accroît la complexité du processus d'optimisation, (i) ceci sera également vrai pour le futur SQL3 et (ii) nous verrons qu'OQL permet toutes les optimisations traditionnelles sur les requêtes de type relationnel.

L'objectif de ce chapitre est de démontrer qu'OQL est bien optimisable. Il présente une algèbre et des techniques d'optimisation par réécriture adaptées au langage. Il est organisé de la façon suivante. Dans un premier temps, nous expliquons les difficultés spécifiques à l'optimisation objet. Puis, nous démontrons les capacités d'optimisation du langage à travers quelques exemples de réécriture sur des requêtes plates. Enfin, nous expliquons rapidement pourquoi l'algèbre objet que nous proposons est particulièrement adaptée à l'optimisation de requêtes imbriquées, requêtes existant également dans le contexte relationnel. Plus de détails sur ces différents points peuvent être trouvés dans [38, 39, 40]. Le papier [35] que nous donnons en annexe est le premier papier que nous ayons écrit sur le sujet. Basé sur une algèbre différente de celle que nous utilisons ici (et moins aboutie), il est moins technique que les papiers qui ont suivi et devrait permettre de mieux appréhender la problématique du domaine.

2.1 Sur l'optimisation objet

Vingt années de recherche ont permis l'élaboration de nombreuses techniques pour l'optimisation des requêtes relationnelles. Pour la plupart, ces techniques sont réutilisables dans le contexte objet et, parfois même, plus faciles à mettre en œuvre et à spécifier formellement que dans le cadre relationnel (voir les requêtes imbriquées de la Section 2.3). Dans cette section, nous illustrons par un exemple les problèmes spécifiques à la réécriture de requêtes objet.

Exemple 2.1 La requête repose sur l'existence d'une méthode *avg_price* dans la classe *Hotel* qui calcule le prix moyen des chambres d'un hotel. Elle retourne pour chaque couple (hotel, service de l'hotel), tel que l'hotel est situé en France et a un prix inférieur ou égal à 300, leurs noms et une collection contenant le nom des hotels situés dans la même ville et qui sont moins onéreux bien que proposant le même service.

```
select  h1.name, f, better?:  select  h2.name
                              from    h2 in Hotels
                              where   h2->avg_price() < h1->avg_price()
                                      and f in h2.facilities
                                      and h2.address.city = h1.address.city
from    h1 in Hotels
        f in h1.facilities
where   h1.address.city.country.name = "France" and
        h1->avg_price() <= 300
```

□

Cette requête illustre trois caractéristiques des requêtes objet.

Jointure dépendante et navigation : Dans le chapitre précédent, nous avons évoqué la sémantique de produit cartésien de la clause **from**. De fait, par la prise en compte des prédicats impliquant les différentes collections, ces produits sont généralement transformés en "*jointures*". Examinons maintenant la clause **from** de la requête principale. Elle comprend deux variables, la seconde étant définie par rapport à la première. En d'autres termes, il ne s'agit pas ici de réaliser une jointure simple mais autant de jointures qu'il y a d'éléments dans la première collection. Ce type d'opérations est fréquent dans le contexte objet et permet de naviguer à travers des collections imbriquées. Il est modélisé dans notre algèbre par une opération appelée *D-join* ou jointure dépendante.

Etrangement, cette opération indispensable a longtemps été absente des algèbres dédiées à l'optimisation objet[70, 96, 101, 105, 112]. De surcroît, certaines de ces algèbres proposaient des jointures qui n'étaient soit pas adaptées au modèle ODMG dans toute sa généralité ([105]), soit n'offraient pas la propriété d'associativité essentielle à l'optimisation ([36, 96, 101, 112]). Nous verrons que notre algèbre, qui est une extension de celle de [70], englobe effectivement la puissance d'expression d'OQL tout en offrant les propriétés de l'algèbre relationnelle dont elle s'inspire. Ceci est également le cas maintenant dans d'autres approches basées sur des modèles de représentation des requêtes de type fonctionnel ([48, 54]).

En fonction de l'organisation des données sur le disque, un D-join est peu ou très coûteux. Dans le cas présent, il rentre vraisemblablement dans la première catégorie puisqu'il permet d'accéder à des chaînes de caractères n'appartenant qu'aux objets *Hotel* les référençant. Ce type de composition n'impliquant pas de partage est généralement traduit par un regroupement des données sur disque (c.a.d. les services d'un hotel suivent cet hotel sur le disque). La navigation à travers *h.facilities* ne nécessitera donc aucun accès supplémentaire à la mémoire secondaire. Les jointures dépendantes impliquant des compositions d'objets partageables (par exemple, les villes d'un pays) sont potentiellement plus coûteuses. Cependant, comme nous l'illustrerons par la suite, elles peuvent être réécrites en jointures traditionnelles par l'utilisation d'une technique reposant sur (i) la connaissance que l'on a des types de la base et (ii) des collections particulières appelées *extensions* et qui contiennent tous les objets d'une classe. Cette technique est également applicable aux navigations impliquant des structures plates (comme, par exemple, *h1.address.city.country.name*). Informellement introduite dans [69], elle est, à notre avis, la technique essentielle de l'optimisation objet. Elle permet aux optimiseurs de choisir entre deux modes de calcul complémentaires, l'un et l'autre pouvant se révéler très bons ou très mauvais en fonction de l'organisation des données manipulées. Elle offre ainsi une grande flexibilité dans le choix de stockage des données, flexibilité essentielle à, par exemple, l'intégration de documents. Dans la section 2.2, nous montrons comment la modéliser par des équivalences algébriques.

Méthodes et factorisation : La requête contient un certain nombre d'appels à des méthodes, certains étant dupliqués (*h1->avg_price*). Les méthodes sont problématiques essentiellement parce que l'on ne peut pas analyser leur code. Deux raisons à cela : il est généralement écrit dans un langage autre que le langage de requêtes (par exemple, C++ ou Java) et la redéfinition d'une méthode dans une sous-classe implique une recherche dynamique du code à l'exécution (un ensemble pouvant contenir des objets d'une classe et de ses sous-classes). Si les travaux de [43] s'intéressent à la seconde partie du problème, la première reste sans solution. Ceci a trois conséquences négatives que nous évoquons maintenant. Notons que SQL3 devra faire face aux mêmes difficultés.

- La recherche dynamique de code peut être coûteuse. Dans de nombreux cas, une analyse statique des types de la base permet de l'éviter. Pour les autres cas, l'optimiseur de requêtes se repose sur la couche système du SGBD qui, si elle est bien conçue, rendra cette recherche la plus rapide possible. Dans tous les cas, il est intéressant de factoriser les appels de méthodes dupliqués pour en diminuer le coût. La factorisation est d'ailleurs un élément important dans l'optimisation objet où de nombreuses sous-requêtes sont dupliquées (par exemple, *h1.address.city*). Dans [32], nous avons proposé un algorithme de factorisation systématique qui est implanté dans l'optimiseur d'O_2Technology.

- Le coût d'évaluation d'une méthode est à priori inconnu et peut être très élevé. La stratégie communément pratiquée consistant à effectuer les opérations de sélection (filtrage) le plus tôt possible peut donc se révéler mauvaise si la sélection implique une méthode. En ce qui concerne l'évaluation du coût, plusieurs techniques existent. Par exemple, comme dans [95], le programmeur peut intervenir pour donner une estimation du coût. Egalement, il est possible de mémoriser les temps d'exécution précédents afin d'établir une moyenne. En ce qui concerne la réécriture des sélections, l'optimiseur peut soit reposer sur une stratégie de recherche traditionnelle avec un modèle de coût approprié, soit travailler en deux phases (optimisation traditionnelle puis analyse et remontée des sélections si nécessaire) comme proposé dans [62]. L'optimiseur d'O_2 n'implante aucune de ces stratégies pour le moment. Il repose sur l'heuristique consistant à évaluer les sélections impliquant des méthodes le plus tard possible mais avant les jointures.

- Une méthode peut avoir des effets de bord sur la base. Il est donc possible de générer deux plans d'exécution aux sémantiques différentes. Par exemple, l'exécution d'une sélection avant ou après un appel de méthode peut générer une différence dans l'état de la base. Analyser une méthode pour savoir si elle a ou n'a pas d'effets de bord est indécidable. Il n'y a donc pas de solution idéale à ce problème. Parmi les solutions possibles, nous trouvons : l'interdiction des appels de méthodes dans les requêtes (ce qui serait dommage), la non-optimisation en présence de méthodes (ce qui serait également dommage), le refus des méthodes potentiellement dangereuses, cette qualification étant soit obtenue par analyse statique reposant sur une condition suffisante, soit donnée par l'utilisateur. C'est cette dernière solution, très raisonnable à notre avis, que met en œuvre l'optimiseur d'O_2Technology.

Requêtes imbriquées : La requête comprend un itérateur imbriqué dans la clause **select**, ce qui est en général coûteux. Ce type d'imbrication est à l'évidence nécessaire pour construire des structures complexes et n'existe pas dans le contexte relationnel. Cependant, la position de l'imbrication n'a finalement pas une importance majeure et nous verrons dans la section 2.3 que les techniques que nous proposons s'inspirent de celles proposées pour l'imbrication dans la clause **where** des requêtes SQL [52, 72, 45, 83]. De plus, ce type d'imbrication diffère peu des requêtes SQL avec regroupement. Nous verrons également que notre algèbre, qui manipule des structures imbriquées, se prête beaucoup mieux que l'algèbre relationnelle à ce type de manipulation. Avec [38], nous pensons avoir ouvert le domaine de l'optimisation des requêtes imbriquées objet. D'autres travaux ont suivi où l'on retrouve des idées similaires [48, 100].

2.2 Des requêtes plates

L'objectif de cette section est de démontrer, à l'aide d'un exemple, que l'optimisation de requêtes avec navigation, une fois les bonnes techniques mises en place, diffère finalement peu de l'optimisation relationnelle.

Exemple 2.2 Nous reprenons la requête de la section précédente sans l'imbrication de la clause **select** et avec quelques prédicats supplémentaires, introduits pour les besoins de la discussion.

select	h.name, f, h.address.city
from	h **in** Hotels
	f **in** h.facilities
where	h.address.city.country.name = "France" **and**
	count(h.address.city.hotels) > 5 **and**
	h->avg_price() >= 200 **and** h->avg_price() <= 300

□

Avant d'être donnée à l'optimiseur, une requête subit quelques traitements préliminaires que nous ne détaillerons pas ici : analyse syntaxique et de dépendances, factorisation, typage, mise sous forme normale des prédicats, traduction algébrique. L'optimiseur reçoit alors en entrée une expression algébrique typée et factorisée. L'expression algébrique correspondant à la requête 2.2 est illustrée par la figure 2.1. L'algèbre utilisée est donnée en annexe, certaines équivalences pourront être trouvées dans [35], d'autres dans [38].

- La première opération appliquée ($Hotels[h]$) est un cas particulier d'utilisation de l'opérateur *Map*. Cet opérateur applique une fonction (dans le cas présent, une construction de n-uplet à un attribut h) à chaque élément d'une collection et existe, sous d'autres noms, dans différentes algèbres objet [1, 15, 35, 70, 96]. Ainsi, $Hotels[h]$ construit une collection de n-uplets à partir d'une collection d'objets. Cette opération est fondamentale puisqu'elle nous permet de définir les autres opérateurs de l'algèbre exclusivement sur ce type de collections et d'obtenir ainsi toutes les bonnes propriétés de l'algèbre relationnelle.

- La seconde opération est le *D-join* évoqué dans la section précédente et qui traduit la clause **from** de la requête. Cette opération retourne une collection de n-uplets avec deux attributs, h et f.

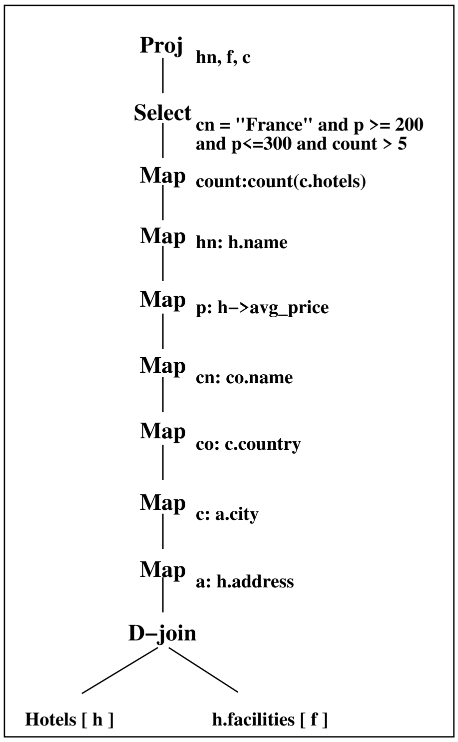

FIG. 2.1: *Traduction algébrique de la requête 2.2*

– Suit une séquence d'opérations *Map*, une par sous-requête élémentaire, qui chacune ajoute un attribut au flot de données (le nom de l'attribut figure à gauche du symbole ":"). Cette découpe systématique, effectuée avant la traduction algébrique, permet une bonne factorisation de toutes les sous-expressions communes (par exemple, *h.address.city* ou *h->avg_price*). Différents *Map* peuvent ensuite être regroupés en un seul. Par la suite, pour des raisons de lisibilité, nous omettrons parfois d'utiliser cet opérateur et ferons directement référence à la fonction qu'il applique. Notons que, dans tous les cas, il ne nécessite pas la construction d'une collection intermédiaire et que, comme la majorité des autres opérateurs, il peut être évalué en *pipeline*.

– L'opération de sélection permet ensuite de ne conserver que les n-uplets satisfaisant les conditions données dans la clause **where**.

– La projection finale permet d'extraire les attributs *hn*, *f* et *c* représentant respectivement le nom d'un hotel, un service qu'il propose et la ville dans lequel il est situé. Elle correspond à un cas particulier de *Map*.

Comme nous l'avons évoqué précédemment, cette expression peut se révéler particulièrement inefficace dans la mesure où les accès aux villes et pays se font de façon incontrôlée. L'expression algébrique en haut de la figure 2.2 propose une réécriture basée sur les extensions (*Ext_xxx* représentant le nom de l'extension de la classe *xxx*)[9].

9. Les *extensions* sont plus ou moins bien maintenues par les différents SGBDOO. Notamment, les systèmes reposant sur la persistance par attachement ne peuvent garantir des extensions exactes. Par contre, il leur est relativement facile de proposer des sur-ensembles de ces extensions, ce qui est suffisant. Une autre possibilité,

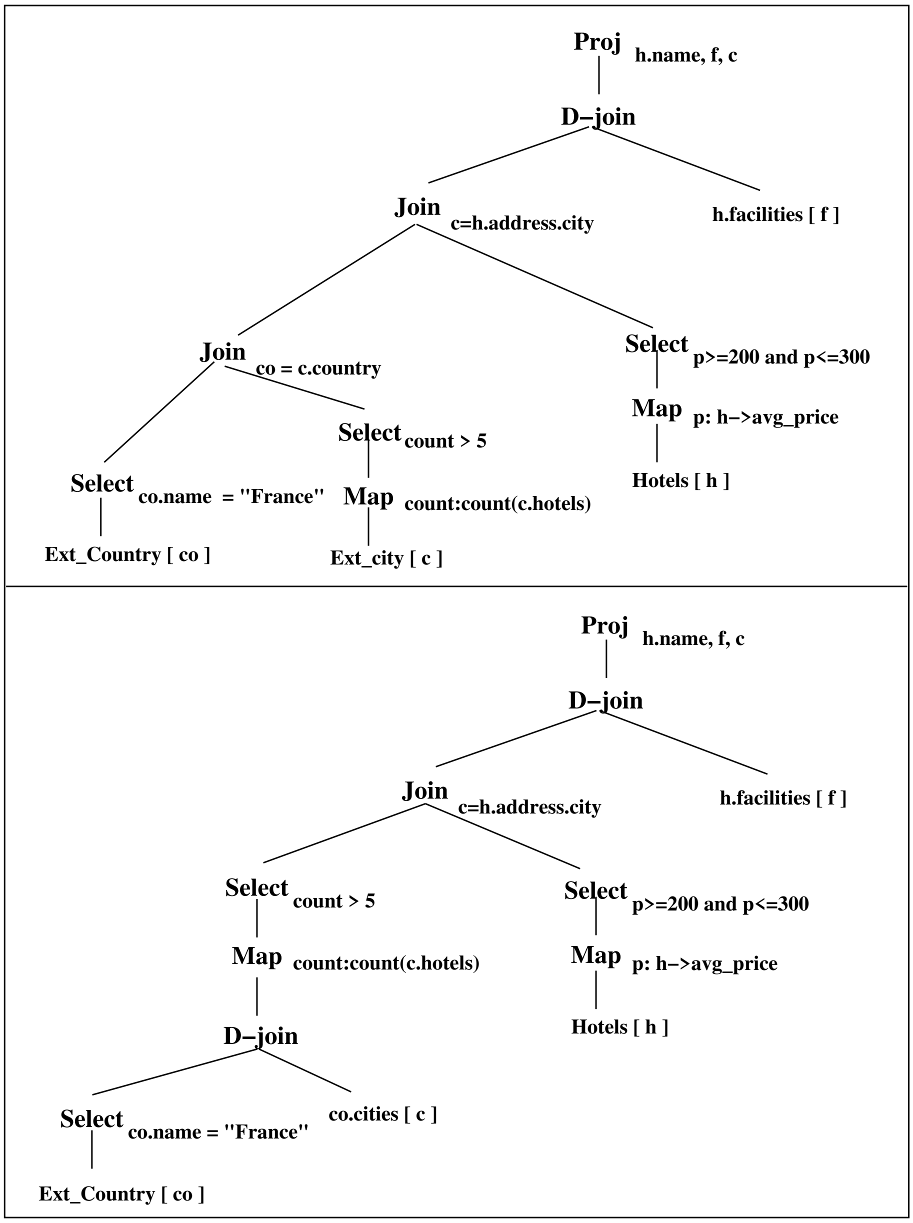

FIG. 2.2: *Deux réécritures de la requête 2.2*

Pour la plupart, les étapes qui nous ont permis d'aboutir à cette expression sont traditionnelles (pousser les sélections, réordonner les jointures, etc). La seule qui ne l'est pas repose sur l'équivalence suivante, qui formalise la technique de [69].

$Map_{att:y.a}(Alg\_Exp) \equiv Alg\_Exp\ Join_{att=y.a}\ Ext\_A[att]$

ssi

le type de $y.a$ est la classe A, il existe une sélection nil-éliminante sur *att* [10].

Il est intéressant de constater que l'expression obtenue ressemble beaucoup à une traduction relationnelle de la requête. Deux jointures réunissent pays, villes et hotels. Comme dans le relationnel, ces jointures peuvent être réordonnées. Les sélections ont été poussées sur les collections avant jointure. Les seules opérations non relationnelles restent les *Map*s qui permettent d'évaluer les opérations autres que les simples extractions d'attributs (par exemple, *count* et *avg_price*) et le *D-join*. Cette réécriture est intéressante si (i) les objets appartenant aux différentes classes sont stockés de façon consécutive dans des fichiers différents et/ou (ii) si il existe des index sur les extensions permettant d'évaluer les sélections (par exemple, un index permettant d'extraire directement le pays dont le nom est *France*). Les jointures pourront alors être évaluées efficacement en utilisant un algorithme de jointure par hachage ou par tri [103].

L'expression algébrique en bas de la figure 2.2 illustre une relation existant entre *D-join* et *Join*. Dans le cas présent, elle repose sur l'existence d'un lien (1,n) inverse *Country.cities : City.country* [11], des équivalences similaires existent ne nécessitant pas ce type de lien. Par exemple, il est toujours possible de transformer une jointure portant sur deux extensions et reposant sur un prédicat d'appartenance, en une jointure dépendante (et vice versa). A notre connaissance, la première utilisation des liens inverses pour l'optimisation a été proposée dans [96].

La première opération de cette nouvelle expression est une jointure dépendante qui permet d'extraire les villes associées au pays dont le nom est "France". Si la politique de regroupement mise en place consiste à stocker villes et pays dans un même fichier et à faire suivre chaque pays de ses villes (ce qui est raisonnable puisqu'une ville n'appartient qu'à un seul pays), cette réécriture est très intéressante. En effet, en supposant l'existence d'un index sur *Ext_country.name*, elle permet d'évaluer l'association entre pays et villes en ne lisant que les pages contenant les villes concernées. Ceci illustre bien le fait que les SGBDOO, par leur politique de placement flexible et les techniques d'optimisation qu'ils offrent, permettent des évaluations particulièrement efficaces. Bien sûr, un travail important reste à faire pour mettre en œuvre l'intégralité des techniques développées, ces systèmes étant encore bien jeunes par rapport à leurs concurrents relationnels. Mais l'avenir est prometteur.

---

proposée dans [51], consiste à permettre à l'utilisateur de spécifier les propriétés intéressantes des objets et valeurs de la base (comme par exemple, *Cities* représente l'extension de la classe *City*) et à utiliser ces informations pour l'optimisation.

10. Cette dernière condition stipule qu'un objet ne fera pas partie du résultat de la requête si il existe une prédicat qui est faux quand l'attribut *att* de l'objet est nil. Elle peut être vérifiée syntaxiquement. Si elle n'est pas satisfaite, un jointure externe (ou outer-join) doit remplacer la jointure traditionnelle.

11. Les SGBDOO ne maintiennent pas tous les liens inverses. Par exemple, O_2 ne le maintient que dans certains cas (utilisation du *binding* C++ ou Java). Si ces liens ne sont pas maintenus par le système, il est toujours possible de se reposer sur des informations données par l'utilisateur comme proposé dans [51].

2.3 Aux requêtes imbriquées

Nous illustrons maintenant l'optimisation de requêtes imbriquées. Afin de faciliter la compréhension des techniques utilisées, nous simplifions la requête 2.1. La requête dans son intégralité peut être traitée de façon similaire[12].

Exemple 2.3 La requête retourne pour chaque hotel dont le prix est inférieur ou égal à 300, son nom et celui des hotels dont le prix est moins élevé que le sien.

select	h1.name, cheaper:	**select**	h2.name
		from	h2 **in** Hotels
		where	h2->avg_price() < h1->avg_price()
from	h1 **in** Hotels		
where	h1->avg_price() <= 300		

□

La représentation algébrique de la requête est illustrée sur la partie gauche de la figure 2.3.

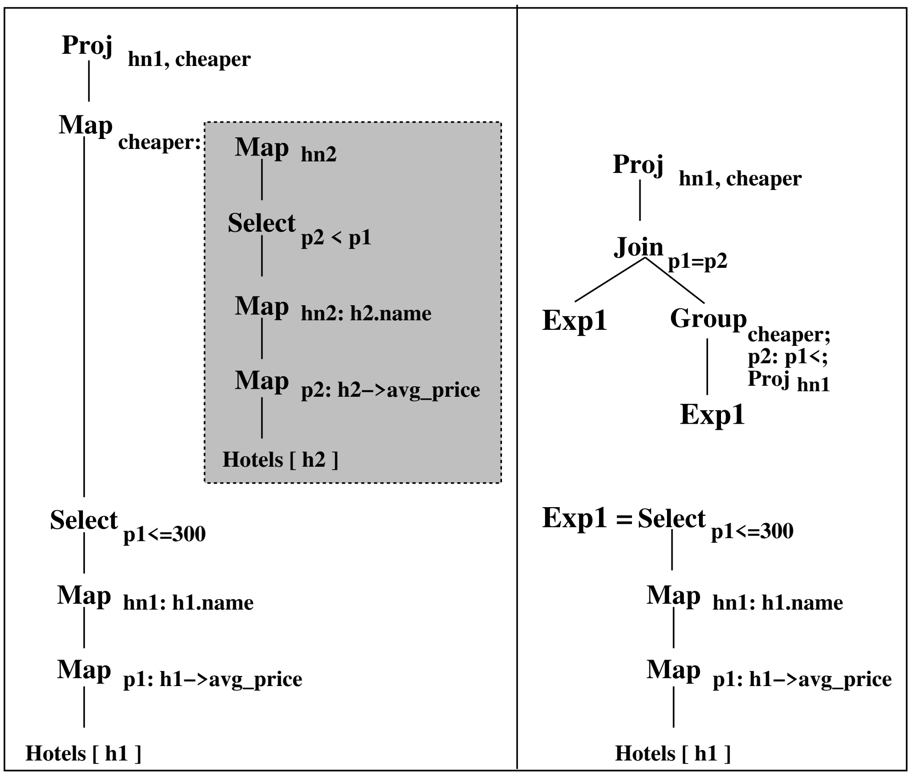

FIG. 2.3: *Traduction algébrique et optimisation de la requête 2.3*

L'imbrication de la clause **select** se traduit par un *Map* dont la fonction est d'appliquer une expression algébrique (partie grisée) à chaque élément de la collection entrante. Ainsi, la requête nécessite de parcourir l'ensemble des hotels autant de fois qu'il y a d'hotels dont le prix est inférieur ou égal à 300. De plus, la méthode *avg_price* sera évaluée de nombreuses et

12. Elle requière simplement la transformation préalable de la partie imbriquée par l'introduction d'un D-join.

inutiles fois sur chaque hotel. Une solution pour éviter ces évaluations multiples, consiste à mémoriser le résultat des applications précédentes (on parle alors de *memoisation*[80]). Dans le cas présent, il est possible de faire encore mieux en réduisant également les parcours de la collection *Hotels*. Ceci est illustré sur la partie droite de la figure 2.3 qui représente la requête optimisée.

La requête est décomposée en deux parties. La première construit l'ensemble des hotels (avec leur nom et prix) satisfaisant le critère de sélection et stocke le résultat dans une variable temporaire **Exp1**. Une analyse statique et assez élémentaire permet d'inférer que cette collection est suffisante pour évaluer requêtes imbricante et imbriquée. **Exp1** est alors utilisée en partie gauche et droite de la jointure figurant dans la deuxième partie. La partie droite de cette jointure traduit la sémantique de regroupement de l'expression imbriquée. Elle comprend une opération **group** dont les trois paramètres sont :

- le nom que prendront les partitions générées par l'opération (*cheaper*),
- le critère de regroupement et son nom ($p2 : p1 <$, le symbole $<$ indiquant que l'on doit regrouper tous les éléments ayant un prix inférieur à p1), et
- l'opération qui sera appliquée sur chaque partition (ici $Proj_{hn1}$ permet d'extraire le nom des hotels qui ont été regroupés).

La jointure crée alors l'association finale et la projection permet d'éliminer les informations non nécessaires.

La transformation du Map imbriqué en opération de regroupement a été obtenue par l'application d'une des équivalences de [38]. Il existe de nombreuses transformations similaires adaptées aux différents cas de figure. Ces équivalences sont indispensables à la bonne évaluation des requêtes imbriquées. Ceci est illustré ici de plusieurs façons. Tout d'abord, la méthode *avg_price* n'est évaluée que le nombre de fois nécessaire. Ensuite, notons qu'un tri de la collection intermédiaire **Exp1** sur **p1** permet d'effectuer regroupement et jointure en un peu plus que deux passes [13], évitant ainsi les nombreuses itérations requises par l'expression précédente.

Enfin, notons que l'opérateur *Group* qui, comme nous l'avons déjà expliqué est indispensable au traitement des requêtes imbriquées, ne peut être défini sur le modèle relationnel (celui-ci ne permettant pas les structures imbriquées). Les techniques développées dans le relationnel et dont nous sommes très largement inspirées, reposent donc (i) soit sur des réécritures informellement définies des requêtes SQL, (ii) soit sur un opérateur moins général combinant regroupement et application de fonctions agrégats sur certains attributs.

13. Le regroupement se fait par union de la partition précédente et de sa valeur, la jointure par fusion.

Chapitre 3 : Documents structurés

De plus en plus d'applications requièrent l'intégration de données provenant de sources hétérogènes. Ainsi, le serveur Intranet d'un constructeur automobile intégrera des schémas de CAO, des documents techniques ou publicitaires et des données factuelles sur les véhicules. Une librairie numérique pourra contenir des documents, des images mais aussi de l'information sur des librairies non virtuelles. Pour fournir une vision raisonnable de l'application aux utilisateurs, il est nécessaire de créer des liens entre ces différentes entités. Par exemple, entre une documentation technique et son pendant publicitaire ou entre une bibliothèque et les articles qu'on peut y trouver. La manière la plus fréquente de réaliser cette intégration, consiste à utiliser un système centralisateur dont la fonction est de donner une vue globale de l'information, vue sur laquelle l'application sera construite. Les données sont alors le plus souvent matérialisées dans le système centralisateur. Certaines restent dans leurs systèmes sources (par exemple, les documents restent dans des fichiers externes, le système les connaît, sait comment y accéder, mais ne les duplique pas).

Les documents constituent la grande majorité des données manipulées par les applications intégrées et leur importance ne fait que croître. Il est donc indispensable de bien comprendre leurs spécificités et d'offrir des outils d'intégration adaptées à celles-ci. Ce chapitre résume nos efforts dans ce sens et propose des techniques reposant sur l'utilisation d'un SGBDOO comme système centralisateur. Nous verrons que ces techniques vont plus loin que le seul problème de l'intégration. Notamment, le langage que nous proposons ouvre des perspectives particulièrement intéressantes, permettant une interrogation beaucoup plus fine que celle habituellement réservée aux documents et qui se résume souvent à une simple recherche par mot-clé.

La première étape pour intégrer des documents dans une base consiste à les modéliser. Nous nous intéressons ici essentiellement aux documents SGML mais les techniques sont applicables à tous documents structurés par une grammaire connue (par exemple, Latex ou BibTex). Afin de *voir* les documents comme des entités logiques de la base, il est nécessaire d'analyser le fichier qui les contient et de spécifier une transformation adaptée à leur grammaire. Nous proposons une technique déclarative de spécification de ces transformations reposant sur l'annotation de la grammaire par des instructions simples et fondée sur la notion de *schéma structurant*. Nous verrons que cette spécification permet parfois d'interroger ou de modifier le document depuis la base sans le matérialiser. Ce travail nous a amené à comprendre les limitations du modèle ODMG et l'atout que représenterait son extension aux unions de types pour permettre une intégration des documents sans perte d'information et offrant de bonnes performances lors de l'interrogation.

Notre deuxième contribution dans ce domaine porte sur la conception d'un langage adapté à ce nouveau type de données. Ce langage, appelé POQL ou OQL-Doc, est une extension d'OQL permettant (i) l'interrogation des données sans connaître leur structure et (ii) la recherche par mot-clé ou par motif. Cette extension est fondamentale pour l'interrogation des documents qui peuvent être très hétérogènes et dont l'utilisateur ne connaît pas toujours bien la structure. Elle est délicate dans le contexte objet qui repose sur un typage fort des données mais aussi des requêtes. Nous verrons ainsi que l'union de types est essentielle à ce type d'interrogation.

L'extension du langage nous amène naturellement aux questions de performances. Nous proposons deux techniques d'optimisation complémentaires. La première s'applique aux documents non-matérialisés et consiste à "pousser" les requêtes sur le schéma structurant afin de réduire la quantité de données à charger dans le système. La seconde est plus adaptée au cas matérialisé et consiste en une extension de notre algèbre objet. Notamment, nous proposons des équivalences algébriques permettant de réécrire efficacement des requêtes qui peuvent, si elles ne sont pas correctement traitées, nécessiter l'accès à toutes les données d'une base.

Ce chapitre est organisé en trois sections correspondant aux contributions évoquées plus haut. Il constitue un survol de travaux réalisés dans le cadre de différents projets R&D européens (Aquarelle et Wire) et est une traduction courte du papier [2] donnée en annexe. Plus de détails sur les différents points évoqués ici peuvent être trouvés dans [3, 4, 6, 27, 28, 29, 30, 33].

3.1 Modélisation

Dans cette section, nous expliquons pourquoi les unions de type sont utiles à la modélisation de documents et introduisons les schémas structurants. Pour cela, nous utilisons un exemple simple de document SGML dont la grammaire (ou DTD pour *Document Type Definition*) et une instance sont représentées partiellement et respectivement sur les figures 3.1 et 3.2.

```
1. <!DOCTYPE article [
2. <!ELEMENT article   --  (title, author+, section+)>
3. <!ELEMENT title     --  (#CDATA)>
4. <!ELEMENT author    --  (#CDATA)>
5. <!ELEMENT section   --  ((title, body+) | (title, body*,subsec+))>
6. <!ELEMENT subsec    --  (title, body+)>
7. <!ELEMENT body      --  (figure | paragraph)>
.....
]>
```

FIG. 3.1: *La DTD d'un article*

Afin de simplifier la présentation, nous avons limité ici les caractéristiques du format SGML aux simples *Elements* et avons pris quelques libertés avec la grammaire. Les éléments sont similaires aux non-terminaux des grammaires BNF. Chacun permet de décrire une partie plus ou moins fine du document. Ainsi, un élément article est défini comme ayant un titre, des auteurs et des sections, le symbole + (resp. *) dénotant une séquence non vide (resp. potentiellement vide). Notons également l'utilisation du symbole "|" qui permet d'introduire une alternative dans la définition des sections et qui correspond donc à une double définition

```
<article>
<title> From Structured Documents to Novel Query Facilities </title>
<author> V. Christophides </author>
<author> S. Abiteboul </author>
<author> S. Cluet </author>
<author> M. Scholl </author>
<section>
<title> Introduction </title>
....
<body> <paragraph>This paper is organised as follows ....
</paragraph></body></section>
...
</article>
```

FIG. 3.2: *Une document de type article*

d'un non-terminal dans une BNF traditionnelle. Enfin, le symbole *CDATA* dénote du texte non interprété. Comme l'illustre la figure 3.2, des marqueurs indiquent les début (<article>) et fin (</article>) d'un élément dans le document.

Considérons maintenant le problème de la représentation de ce document dans un SGBDOO. Le modèle ODMG est relativement bien adapté à la modélisation des documents. Notamment, en opposition au modèle relationnel, il comprend collections ordonnées, données complexes (comme les images) et imbrications, trois caractéristiques que l'on retrouve dans les documents. De plus, il permet le maintien de références croisées et est ainsi particulièrement bien adaptée à une navigation depuis le *Web*. Il manque cependant à ce modèle (et au relationnel) la notion d'union pour modéliser simplement les alternatives de structure[14]. De fait, il est possible de modéliser celles-ci en se reposant sur l'héritage (par exemple, en définissant une classe *Section* héritant de deux classes représentant l'alternative de structure). Cette solution est, à notre avis, peu satisfaisante. Entre autres raisons, elle peut aboutir à une explosion de la hiérarchie de classes mais, surtout, sa nature statique (toute création de classe est persistante) n'est pas adaptée à l'interrogation qui engendre également des alternatives de type. En effet, comme nous le verrons dans la section suivante, les requêtes sur des documents retournent souvent des résultats hétérogènes. Dans la mesure où les SGBD reposent sur un typage fort, l'union de types vient à point pour typer ces résultats.

L'hétérogénéité des résultats est d'ailleurs un des arguments pour la conception d'une nouvelle famille de modèles sans type, appelés modèles semistructurés[16, 50, 92]. Hélas, les systèmes qui les mettent en œuvre, du fait de leur absence de typage, n'offrent pas encore de performances acceptables sur des gros volumes de données et rendent l'utilisation, par une application, du résultat d'une requête très improbable. L'union de types, même si elle est difficile à gérer, offre une alternative raisonnable. Nous verrons dans le chapitre 5 une autre solution qui nous paraît très prometteuse.

Mais revenons aux documents SGML et à leur modélisation dans un ODMG étendu. La figure 3.3 montre quelques unes des classes qui permettent de modéliser les articles précédemment introduits. De façon assez triviale, les éléments non-atomiques de la DTD sont représentés par des classes. Le schéma proposé représente une parmi les différentes modéli-

14. D'autres particularités du format SGML que nous n'illustrons pas ici bénéficieraient également de l'introduction d'unions de types.

```
Class Article
        title: String,
        authors: list(String),
        sections: list(Section)

Class Section
        union
           s1: tuple (title: String, bodies: list(Body))
           s2: tuple (title: String, bodies: list(Body), subsecs: list(Subsec)
Class Body
        union
           figure: Bitmap
           paragraphe: Paragraphe
....
```

FIG. 3.3: *Classes O_2 décrivant un article*

sations possibles et peut être inféré depuis la DTD. Notez l'utilisation d'une union dans la classe *Section*. Il s'agit d'une union marquée (les attributs de marquage sont *s1* et *s2*) qui autorise un typage statique des requêtes (voir [27, 28] pour plus de détails sur ces différents points).

Voyons maintenant comment les schémas structurants sont utilisés pour générer l'image d'un article dans la base. La figure 3.4 montre (partiellement) celui correspondant à notre exemple. Notez que nous sommes revenus à une spécification plus traditionnelle de la grammaire, plus conforme à celle sur laquelle reposent généralement les analyseurs syntaxiques.

```
/* Type definition */
<article> : Article;
....
<authors> : list(String)
....
```

```
/* Annotated grammar */
<article>    –>  <title> <authors> <sections>
                  $$= Article (title: $1, authors: $2, sections: $3)
<title>      –>  %String
                  $$=$1
.....
<sections>   –>  <section>
                  $$ = list($1)
             –>  <section> <sections>
                  $$ = list($1) concat $2
<section>    –>  <title> <bodies>
                  $$ = Section(title: $1, bodies: $2)
.....
```

FIG. 3.4: *Schéma structurant pour les articles*

Un schéma structurant comprend deux parties. La première spécifie les liens existant entre les non-terminaux de la grammaire et les types de la base, la seconde entre les documents et les objets de la base. Le symbole $$ dans la partie droite d'une règle représente la donnée construite par cette règle, le symbole $i représentant la donnée retournée par son ième composant. Remarquez l'utilisation de requêtes simples, comme la construction d'objets (dans la règle définissant <article>) ou la concaténation de listes (pour <sections>). Nous verrons que cette spécification de la traduction documents-objets, qui a été implantée au-dessus de Yacc [63], est bien adaptée à la mise en place de techniques d'optimisation de requêtes (voir [3, 4, 6] pour plus de détails). D'autres travaux ont été réalisés sur l'intégration de documents dans une base de données [19, 111], mais aucun n'offre la généralité de l'approche proposée ici, ni les techniques d'optimisation qu'elle permet.

3.2 Interrogation

Les langages de requêtes des SGBD proposent une interrogation structurée. Or, les documents sont généralement interrogés par des utilisateurs qui ne connaissent pas ou peu leur structure et qui recherchent des mots-clés ou des motifs dans le texte ou une partie du texte (par exemple, dans le titre). Notons que, s'il est convivial, ce type d'interrogation est peu puissant et ne permet pas, par exemple, de trouver les documents référençant l'auteur d'un document donné. Notre objectif est donc de concevoir un langage mariant ces deux types d'interrogation.

Pour ce faire, nous avons ajouté essentiellement deux fonctionnalités à OQL : (i) une recherche par mot-clé ou motifs et (ii) une interrogation sans connaissance de la structure des données. Si elle nécessite de nouvelles techniques d'optimisation, la première fonctionnalité est facile à obtenir. Il suffit d'étendre le langage par des opérateurs sur les chaînes de caractères un peu plus sophistiquées que ceux existant dans la version standard. La seconde fonctionnalité nécessite une extension plus subtile. Expressément, nous avons introduit deux nouveaux types de données pour comprendre les notions de chemin (**path**) et d'attributs (**att**) et, grâce à des opérateurs appropriés, être capable de les interroger. Le langage ainsi obtenu offre un pouvoir d'expression plus grand que ses prédécesseurs (comme par exemple, [17, 18, 24, 56, 58, 71, 76]), tout en conservant sa simplicité d'origine. Il est comparable à des langages apparus un peu plus tard comme [7, 22, 23].

Nous illustrons ici ces extensions à l'aide de deux exemples.

Exemple 3.1 La première requête recherche les titres (qui peuvent être des titres de document, de section, etc.) du document *my_article* contenant le mot "Web".

select	t
from	my_article @P.title(t)
where	t **contains** "Web"

La requête introduit (i) une variable de chemin (**@P**, le symbole "@" préfixant toujours ce type de variables) et (ii) l'opération **contains**.

Les variables de chemin sont utilisées pour dénoter un chemin quelconque depuis un objet de la base (dans le cas présent *my_article*). Elles sont utilisées pour construire des *expressions de chemin généralisées (ECG)* dont la sémantique diffère des expressions de chemin simples de OQL. Les ECGs retournent des collections de n-uplets avec un attribut par variable. Ainsi, *my_article @P.title(t)* est une ECG retournant une collection de n-uplets avec deux attributs : "@P" dénote un chemin depuis l'objet *my_article* dont l'extrémité est un n-uplet possédant un attribut *title* et t dénote la valeur de cet attribut. L'occurrence de cette ECG dans la clause **from** implique une itération sur la collection résultante.

L'opération **contains** vérifie qu'un mot fait partie d'un texte. □

Comme pour les chemins, il existe des variables permettant d'interroger les attributs. Notons que les variables d'attributs ne sont pas indispensables, dans la mesure où un attribut n'est qu'un cas particulier de chemin. Cependant, elles permettent de formuler certaines requêtes de façon plus concise. Nous n'illustrons pas ce point ici et référons le lecteur à [27, 28].

L'utilisation d'ECG soulève un certain nombre de points intéressants.

- OQL est un langage fonctionnel, les ECG sont donc des opérations comme les autres et leur utilisation n'est pas limitée à la clause **from**. Naturellement, elles peuvent être combinées avec tout opérateur défini sur des collections de n-uplets.

- Comme tout type de la base, les chemins (et les attributs) sont associés à un ensemble de fonctions permettant de les interroger. Par exemple, **length** permet de connaître la longueur d'un chemin, et **[i]** d'extraire un de ses éléments.

- Les bases de données objet constituent des graphes cycliques. Si nous n'imposons pas de contrainte, une variable de chemin est donc potentiellement infinie. Pour résoudre ce problème, deux solutions. La première repose sur une analyse statique du schéma et ne permet pas qu'un chemin passe deux fois par une même classe. Simple à mettre en œuvre, elle a été utilisée dans le prototype qui a été développé et dont une démonstration est proposée sur notre site Web ([65]). La seconde, plus puissante, nécessite une analyse dynamique des objets traversés et consiste à arrêter le chemin à la deuxième occurrence d'un même objet. A l'évidence, cette solution, bien que plus coûteuse, est meilleure du point de vue de l'utilisateur qui a ainsi véritablement accès à tous les objets atteignables depuis un point de départ donné.

- Les données accédées par une variable de chemin sont potentiellement hétérogènes et un typage statique implique donc l'utilisation d'union de types. De fait, dans la majorité des cas, les opérations appliquées sur les données permettent d'affiner le typage. Ainsi, si l'on considère la requête 3.1, l'attribut *title* en bout de chemin et le prédicat **contains** appliqué sur ce titre réduisent la portée de **@P** à des données d'un type inférieur à **tuple(title: String)**.

- Une façon d'accéder à des données dont le type est une union consiste à référencer explicitement les marqueurs. Par exemple, l'expression *x.s2.subsecs* donne accès aux sous-sections de la section représentée par la variable x (voir figure 3.3). Ce type d'interrogation, s'il est fastidieux mais possible sur les données de la base, n'est hélas pas applicable aux données résultant d'une ECG. En effet, les unions étant générées par le système, les marqueurs sont inconnus de l'utilisateur. C'est pour cette raison qu'un accès implicite s'impose. Ainsi, l'expression *x.subsecs* est-elle licite. Notons qu'elle implique l'ajout d'une sélection sur x, qui doit correspondre à une section avec sous-sections. Ceci est expliqué en détail dans [27].

L'exemple suivant montre comment le langage permet retrouver des références croisées.

Exemple 3.2 La requête retrouve dans l'ensemble *Documents*, les couples de documents s'entre-référençant et tels que le titre du premier commence par le mot "Web".

select	d1, d2
from	Documents(d1) @P1,
	Documents(d2) @P2,
where	d1.title **like** "Web*"
	and d1 @P1 = d2
	and d2 @P2 = d1

Notez l'utilisation des variables de chemin **@P1** et **@P2** dans la clause where pour vérifier la référence croisée. Nous aurions pu formuler la même requête en (i) introduisant une variable de données à l'extrémité de chaque ECG de la clause **from** et (ii) comparant ces variables à **d1** et **d2**. □

3.3 Optimisation

Nous évoquons ici très brièvement les techniques d'optimisation que nous avons conçues et, pour certaines, implantées.

Comme nous l'avons vu dans le chapitre 2, un coût important dans l'évaluation d'une requête est celui du parcours des pages contenant les données interrogées. Si celles-ci n'ont pas été matérialisées dans la base, le facteur déterminant devient alors le temps de chargement. Ceci peut paraître anecdotique, considérez l'interrogation d'une encyclopédie pour comprendre son importance. La première technique d'optimisation que nous avons développée concerne ce problème.

L'idée est de pousser la requête de l'utilisateur sur celles utilisées dans le schéma structurant. Le processus est enclenché sur la règle définissant la donnée interrogée (par exemple, la règle <article> si l'on interroge *my_article*). La requête de l'utilisateur est combinée avec celle définissant $$ et des règles de réécriture sont appliquées pour la simplifier et la pousser sur les non-terminaux correspondant aux $is. Les règles de réécriture sont traditionnelle ou spécifiques aux schémas structurants (voir [3, 4, 6]). Le processus continue par appel récursif sur les non-terminaux en partie droite de chaque règle. Ainsi, nous obtenons un nouveau schéma structurant, à la grammaire inchangée, mais dons les annotations sont conçues de façon à ne construire que les données nécessaires. La figure 3.5 montre (de façon très simplifiée) le schéma structurant résultant de l'optimisation d'une requête sélectionnant les sections dont le titre est "Optimisation".

Notez la condition sur la construction des sections. Si elle n'est pas satisfaite, la règle retournera une donnée particulière, appelée $\perp$, dont l'insertion dans une liste est sans effet. Notez également que la technique ne change rien à l'analyse syntaxique de la requête. En d'autres termes, si l'arbre de syntaxe n'est pas mémorisé, le document doit être analysé dans son intégralité.

Bien sûr, les requêtes ne sont pas toujours "poussables" et le "nécessaire" construit est parfois important. Dans [3, 4, 6], nous proposons des classes raisonnables de schémas structurants se prêtant bien à cette manipulation. Nous montrons également comment appliquer une technique similaire pour permettre la mise à jour des documents à partir de la base de données.

```
<article>    -> <title> <authors> <sections>
                $$= $3
<title>      -> %String
                $$=$1
.....
<sections>   -> <section>
                $$ = list($1)
             -> <section> <sections>
                $$ = list($1) concat $2
<section>    -> <title> <bodies>
                $$ = Section(title: $1, bodies: $2)
                     where $1="Optimisation"
.....
```

FIG. 3.5: *Schéma structurant optimisé*

Considérons maintenant l'optimisation dans un contexte matérialisé. La difficulté principale vient, de façon attendue, des ECGs nouvellement introduites dans le langage. Il existe différentes façons d'évaluer ce type de requêtes. La première technique consiste à ne rien faire statiquement et à évaluer les expressions de chemin par un parcours de la base. Ainsi, si l'on considère la requête de l'exemple 3.1, va-t-on parcourir tous les chemins de *my_article* pour ne conserver que ceux satisfaisants les critères de sélection. A l'évidence, cette solution est coûteuse et donc peu satisfaisante. Elle est utilisée par les systèmes semistructurés [77] qui, en l'absence de types, ne peuvent se reposer sur une analyse préalable du schéma de la base. La seconde solution consiste justement à faire une analyse statique des ECG afin de (i) trouver tous les chemins abstraits (c.a.d., sur le schéma plutôt que sur les données) et (ii) transformer la requête en une union de requêtes, une par chemin trouvé. Par exemple, la requête de l'exemple 3.1., est transformée de la façon suivante.

Exemple 3.3

```
select  t
from    my_article.title(t)
where   t contains "Web"
union
select  t
from    my_article.sections{}.title(t)
where   t contains "Web"
union
...
```

Notez l'utilisation de l'opérateur {} pour traverser la collection de sections. □

Cette technique, utilisée dans [18], est plus efficace dans la mesure où elle évite une analyse dynamique des types. Cependant, elle engendre des unions importantes de requêtes qui mettent les optimiseurs à rude épreuve. De plus, la dichotomie de traitement entre ECGs (évaluées partiellement au niveau OQL) et requêtes plus traditionnelles (passées à l'optimiseur) réduit le champ des optimisations.

Dans [29], nous proposons une technique originale basée sur une extension de notre algèbre objet et qui permet des optimisations réellement efficaces des requêtes avec ECG. Cette technique repose sur deux nouveaux opérateurs, *S-inst* et *D-inst*, qui instancient respectivement les ECGs au niveau du schéma et des données. Nous illustrons ceci avec la requête de l'exemple 3.2 dont la traduction algébrique est donnée sur la figure 3.6.

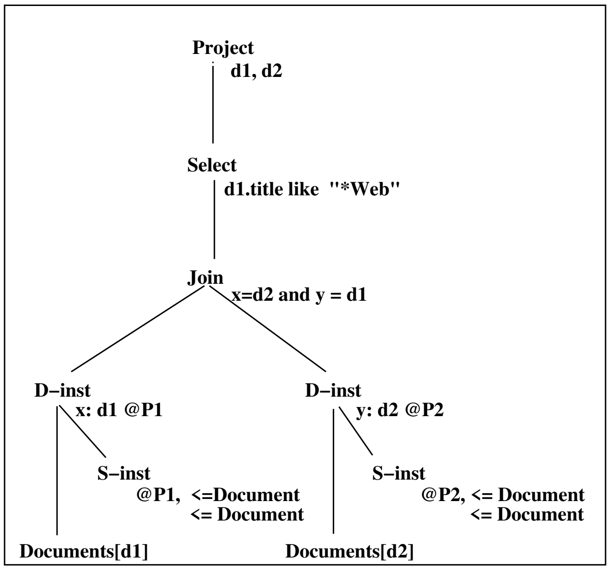

FIG. 3.6: *Traduction algébrique de la requête 3.2*

La partie haute de la figure ne représente pas de difficulté particulière (voir chapitre 2). La partie basse est plus intéressante. A gauche (mais aussi à droite), nous pouvons voir l'utilisation de l'opérateur *S-inst* dont les arguments sont une ECG et ses types de départ et d'arrivée (qui ont été inférés de façon triviale). L'opération retourne ici une collection de n-uplets à un attribut : **@P1** représentant un chemin abstrait d'un document à un autre[15]. L'opération *D-inst* applique alors ces chemins abstraits à chaque document **d1** et mémorise le résultat dans **x**.

L'intérêt de cette approche repose sur un ensemble d'équivalences qui nous permet de déplacer, et parfois de transformer, ces opérations critiques. Ceci est illustré sur la figure 3.7 qui montre une réécriture possible de la requête.

Remarquez que la sélection a été poussée en bas de l'arbre. Ceci présente essentiellement deux avantages. Tout d'abord, l'opérateur *D-inst*, opérateur coûteux s'il en est, est vraisemblablement appliqué sur une collection plus petite. Ensuite, il est maintenant possible d'utiliser un index pour évaluer la sélection sur les documents. Remarquez également que l'opération de jointure a été intégrée dans la seconde opération *D-inst*, qui suit le chemin **@P2** depuis les données x trouvées à l'extrémité du chemin **@P1**. Ceci permet de prendre en compte la première condition de jointure ($x = d2$), la seconde ($y = d1$) est donnée comme paramètre supplémentaire au *D-inst* qui ne considéra que les éléments la satisfaisant.

Cette nouvelle expression est déjà bien meilleure que la première. Une autre amélioration consiste à factoriser la double occurrence de l'opération *S-inst.* Nous ne l'avons pas fait ici afin de pouvoir montrer les possibilités d'optimisation dynamique de l'opérateur *S-inst.* Considérons que la classe *Document* sur laquelle porte l'opération *S-inst* est la racine d'une hiérarchie

15. Notez que cette opération correspond à la phase d'analyse statique de la technique d'optimisation que nous avons évoquée précédemment.

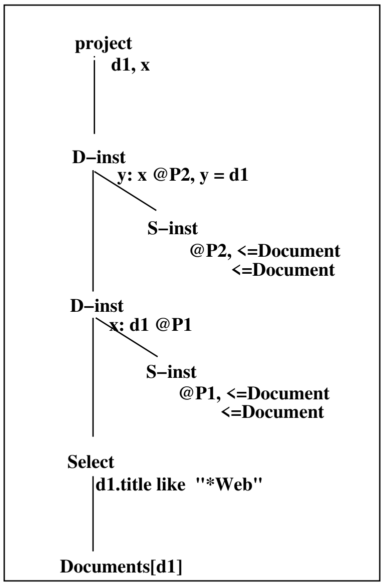

FIG. 3.7: *Optimisation de la requête 3.2*

de classes importante. Suivant cette hypothèse, le coût de l'opération est également important. Considérons maintenant que la sélection ne retourne que des éléments appartenant à la classe *Article*. Dans ce cas, il est possible de simplifier considérablement les deux opérations *S-inst* en restreignant le type de départ de **@P1** et d'arrivée de **@P2** à la seule classe *Article*. Si la deuxième hypothèse n'est pas satisfaite, il est alors toujours possible de mémoriser le résultat du premier *S-inst* pour ne pas avoir à le réévaluer.

Finalement, bien que nous ne l'illustrons pas ici, un opérateur appelé *FTI* (pour Full Text Index) offre une alternative intéressante pour l'évaluation de l'opérateur *D-inst*. Il correspond à une recherche par index plein texte un peu particulière, permettant notamment d'interroger la structure des documents. Cet opérateur est également indispensable à la bonne évaluation des requêtes impliquant les opérateurs nouvellement introduits pour les chaînes de caractères. Plus de détails sur ce dernier point peuvent être trouvés dans [27, 31].

Chapitre 4 : Chargement en masse

Chaque jour, des données en nombre toujours plus important sont transférées puis répliquées dans différents systèmes. Ainsi, certains centres d'analyse du génome humain reposent sur la réplication régulière de gigas octets en provenance d'autres centres. Suivant des fréquences plus faibles, le rafraîchissement des entrepôts de données nécessite la migration de gros volumes d'information [109]. Ces mouvements ne sont généralement pas de simples transferts d'octets mais requièrent une restructuration des données, les systèmes source et cible ne partageant pas toujours le même modèle de représentation. Cette restructuration est la cause du temps très élevé requis par le chargement, qui peut nécessiter des heures, voire des jours. Ceci est particulièrement critique, les systèmes impliqués pouvant être paralysés pendant toute la durée du processus. Il paraît donc essentiel d'optimiser le chargement en masse de données, et ce, quelqu'en soit la fréquence. Notamment, un transfert de technologie (du relationnel à l'objet par exemple) ou la construction ponctuelle d'une librairie numérique suivant une approche matérialisée (voir chapitre précédent) ne peuvent que bénéficier d'un outil de migration efficace.

Jusqu'à présent, seuls les éditeurs de SGBD proposent des logiciels offrant ce type de fonctionnalités (par exemple, [46, 91, 64]). Cependant, les outils sont soit inefficaces, soit de très bas niveau. L'inefficacité est clairement un mauvais point, le deuxième l'est également puisqu'il implique une programmation difficile et non réutilisable. D'un autre côté, la communauté scientifique s'est peu intéressée au problème du chargement proprement dit. Les travaux de recherche portent essentiellement sur les aspects de modélisation et, plus précisément, sur les problèmes de transformation d'un format à l'autre (voir par exemple, [37, 44, 47, 60, 67, 73, 81, 82].

Seule exception, [107, 108] proposent une technique consistant à découper le processus de migration afin d'éviter une saturation de la mémoire vive. De manière similaire, l'approche que nous présentons ici repose sur un chargement incrémental des données. Son originalité tient à sa nature déclarative qui permet une spécification simple du chargement mais surtout la mise en place de techniques d'optimisation appropriées.

Clairement, un bon outil de chargement en masse permet de spécifier de façon simple la transformation des données à migrer et offre de bonnes performances. Trois autres fonctionnalités, moins évidentes, sont également essentielles. Tout d'abord, l'utilisateur doit pouvoir spécifier la façon dont les données seront organisées dans le système cible. Ceci est indispensable pour garantir un déroulement rapide des programmes qui s'effectueront sur les données migrées et évite une réorganisation lourde après chargement. Egalement, il est important que l'utilisateur puisse intervenir dans le découpage du processus de chargement. Comme nous l'avons évoqué précédemment, ce découpage permet d'éviter saturation de mémoire et perte d'information importante en cas de panne. Il est aussi requis pour empêcher une paralysie des systèmes à de moments inopportuns, moments que seul l'utilisateur est capable d'apprécier (par exemple, lorsque certains traitements importants sont en cours d'exécution sur le système source ou cible). Enfin, il est nécessaire de savoir ce qu'il convient d'optimiser et qui peut être une combinaison plus ou moins complexe de quatre facteurs : temps de transfert, gestion d'une mémoire tampon et occupation des ressources du système source ou cible. Encore une fois, seul l'utilisateur connaît la réponse et doit donc pouvoir la spécifier au logiciel de chargement.

Ce chapitre résume les travaux qui ont abouti à la création d'un logiciel offrant ces bonnes propriétés (voir [8]). Fidèles à notre *credo*, nous proposons une approche déclarative du problème reposant sur un modèle de données, un langage déclaratif et des techniques d'optimisation.

- Le modèle de données est encore une fois le modèle ODMG. Dans la mesure où il permet de comprendre la majorité des données structurées, ce modèle est un bon candidat pour une approche de la migration qui se veut générale.

- Le langage doit permettre de filtrer et restructurer des données quelconques du modèle. Il repose donc naturellement sur notre algèbre objet. Il doit également permettre la mise à jour du système cible. Nous avons donc étendu l'algèbre une nouvelle fois pour permettre l'application de fonctions avec effets de bord. Cette extension permet notamment de mémoriser les correspondances existant entre données source et cible, caractéristique indispensable pour (i) gérer les cycles et (ii) permettre un certain type de chargement incrémental (où tous les attributs d'une donnée ne sont pas assignés simultanément).

  L'algèbre n'est clairement pas un langage dédié aux utilisateurs. Cependant, il est difficile de concevoir une interface conviviale, suffisamment abstraite pour satisfaire les utilisateurs de systèmes potentiellement très variés. Aussi, avons nous conçu un langage utilisateur dédié à la migration relationnel-objet.

- Ceci ne réduit pas la portée des techniques d'optimisation que nous avons développées et qui sont effectivement adaptées à des cas très variés de migration (voir [8]). Ces techniques reposent sur de nouvelles équivalences algébriques particulièrement intéressantes puisqu'elles s'appliquent non pas à des expressions simples mais à des séquences d'expressions avec effets de bord.

Le chapitre est un résumé de l'article [9] donné en annexe. Il présente, à travers un exemple simple de migration relationnel-objet, la spécification et l'optimisation d'un processus de chargement. Plus de détails sur d'autres types de chargement et sur le prototype qui a été implanté peuvent être trouvés dans [8].

4.1 Spécification

Suivant notre approche, le processus de migration s'effectue en cinq étapes.

1. L'utilisateur spécifie la transformation source-cible et les propriétés physiques qu'il veut voir réaliser sur le système cible.

2. Cette spécification est donnée à un premier module qui la traduit dans l'algèbre et prépare le système cible (dans le cas relationnel-objet, cette préparation consiste à créer le schéma objet approprié).

3. L'optimiseur entre alors en jeu et, étant donné un modèle de coût paramétrable par l'utilisateur, cherche le meilleur plan pour effectuer un chargement satisfaisant les propriétés physiques spécifiées lors de la première étape.

4. A partir de ce plan, le générateur de code produit un ensemble de programmes paramétrés en temps.

5. L'utilisateur lance les programmes aux moments opportuns et pour des durées choisies. Ceci se fait depuis un environnement d'exécution dont le rôle est de garantir le bon séquençage des programmes et de maintenir l'ensemble des informations nécessaires à l'exécution incrémentale du processus.

Notez que les caractéristiques d'un bon outil de migration, telles que nous les avons décrites précédemment, sont bien prises en compte par cette approche. Nous illustrons dans cette section les étapes 1 et 2 de ce processus. Pour cela, nous utilisons le schéma relationnel illustré par la figure 4.1.

r_emp [emp: integer, lname: string, fname: string, town: string, function: string, salary: real]
r_project [name: string, location: string, boss: string]
r_emp_prj [project: string, employee: integer]

FIG. 4.1: *Le schéma relationnel source*

Le schéma comprend trois relations. Les deux premières, *r_emp* et *r_project*, contiennent de l'information sur les employés et projets d'un institut de recherche. Dans notre exemple, elles seront utilisées pour construire deux classes (*Researcher* et *Project*). Les attributs soulignés sont les clés de chaque relation et servent à identifier les données de façon unique. Cette identification est capitale puisqu'elle permet de maintenir la correspondance entre données source et cible nécessaire notamment à la gestion des cycles. Dans le cas de la migration relationnel-objet, elle repose sur la notion de clé et d'identité d'objet. Tout autre système d'identification est concevable. Finalement, la troisième relation, *r_emp_prj* matérialise l'association entre employés et projets.

Voyons maintenant une traduction ODMG possible de ce schéma. La figure 4.2 montre comment elle est spécifiée dans le langage *RelOO*, langage qui a été implanté dans le système du même nom. Ce logiciel est dédié au cas relationnel-objet et met en œuvre les cinq étapes évoquées ci-dessus. Il est important de comprendre que, si les parties langage et génération de code du système sont spécifiques à un type de migration particulier, il n'en est pas de même de l'optimiseur. Celui-ci repose sur des techniques de réécriture algébrique adaptées à toute migration dont les données sources et cibles sont représentables par des graphes ODMG.

Chaque instruction **create class** spécifie la création d'une classe à partir d'une ou de plusieurs relations. La valeur de ses attributs est définie par des fonctions sur les attributs relationnels (dans l'exemple, l'identité et une concaténation de chaînes de caractères), sa population par la clause **where** et le nom de son extension par la clause **extent**. Deux clauses optionnelles **order by** et **cluster on** permettent additionnellement de spécifier des contraintes

```
create class       Project
from relations     r_project
with attributes    name is name
extent             Projects
where              location = "Rocquencourt"
order by           name
cluster on         staff(Researcher)

create class       Researcher
from relations     r_employee
with attributes    name is lname + fname,
                   salary is salary,
extent             Researchers
where              function = "Researcher"
                   and town = "Rocquencourt"

create link        Project,Researcher
from relations     r_emp_prj, r_employee, r_project
with attributes    staff:set(Researcher) in class Project,
                   projects:set(Project) in class Researcher
where              employee = emp and project = name
```

FIG. 4.2: *Traduction relationnel-objet*

d'organisation physique. Dans l'exemple, seuls les employés chercheurs de Rocquencourt et les projets du même site seront migrés. De plus, les projets seront ordonnés par nom et chaque projet sera suivi sur disque par l'ensemble de ses chercheurs. L'exemple n'illustre pas la possibilité de création d'une hiérarchie de classes qui ne pose pas de problème particulier (voir [8, 9]).

L'instruction **create link** définit une association entre projets et chercheurs matérialisée par les attributs ensemblistes *staff* et *projects*. Cette instruction comprend quatre clauses spécifiant respectivement les (i) classes et (ii) relations impliquées, (iii) les attributs objet créés et (iv) la condition de jointure entre les différentes relations.

A partir de la spécification de l'utilisateur, le système *RelOO* génère un schéma objet et une représentation algébrique. Le schéma est celui de la figure 4.3.

class	Employee name: String, salary : String, projects: set(Project)	**class**	Project name: String, staff: set(researcher)
name	Researchers: set(Researcher)	**name** **cluster**	Projects: set(Project) Project: staff*

FIG. 4.3: *Le schéma objet cible*

La représentation algébrique est donnée sur la figure 4.4. Elle correspond à un chargement par défaut où sont d'abord créés les objets et leurs attributs atomiques. Dans un deuxième temps, les attributs complexes sont évalués. A l'évidence, il existe de nombreuses autres façons de procéder. Le rôle du processus d'optimisation que nous décrivons dans la section suivante est précisément de comparer les différentes alternatives afin de sélectionner la meilleure.

La représentation par défaut consiste, dans le cas présent, en une séquence de quatre expressions algébriques, encore appelées blocs. Les deux premiers permettent la création des classes *Researcher* (**B_res**) et *Project* (**B_proj**), les deux derniers matérialisent l'association entre chercheurs et projets (**B_staff** et **B_projects**). La lecture du chapitre 2 est un pré-requis à la compréhension de ce qui suit.

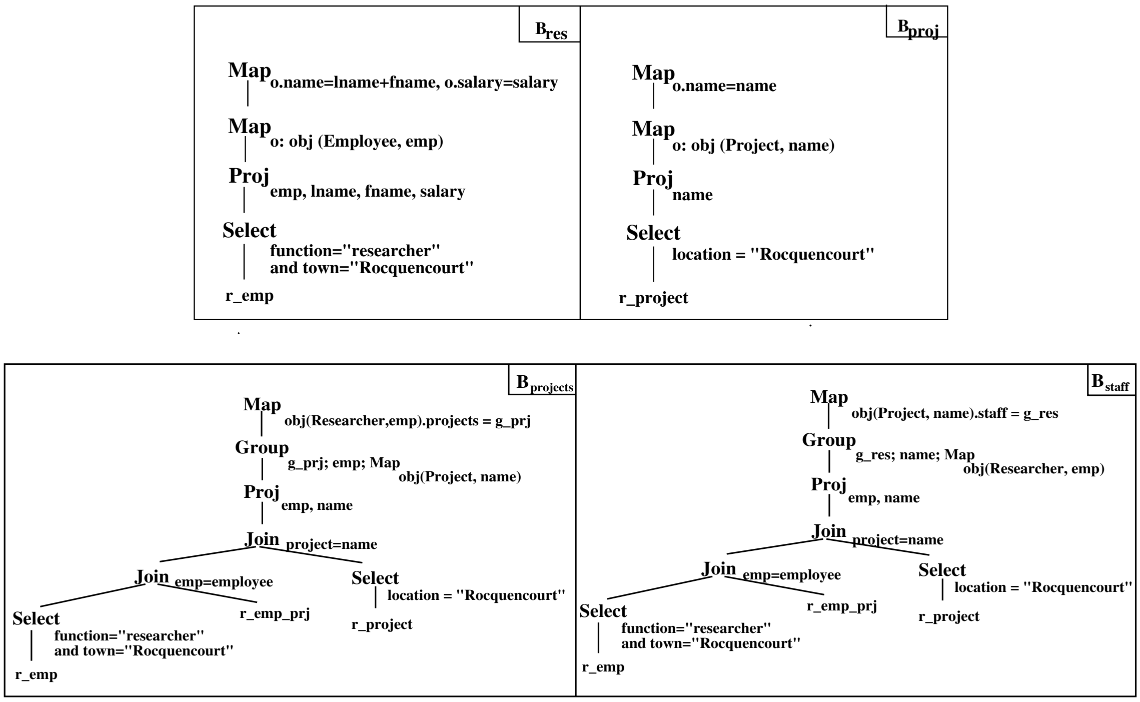

FIG. 4.4: *La représentation algébrique par défaut*

Examinons d'abord les blocs de création des objets. Les deux sont similaires. Ils commencent par une opération de sélection correspondant à la clause **where**. Suit une projection sur les attributs impliqués dans la création. Enfin, deux opérations *Map* sont utilisées pour appliquer les fonctions de mise à jour, **obj** qui crée un objet d'une classe donnée à partir d'une valeur de clé relationnelle et = qui assigne des valeurs aux attributs d'un objet. Ces fonctions sont clairement spécifiques au cas relationnel-objet. De fait, les équivalences supportées par l'algèbre ne reposent pas sur leur sémantique mais sur deux propriétés naturelles dans un contexte de migration: (i) l'idempotence et (ii) l'insensibilité aux valeurs nulles. La première stipule que l'application multiple, sur des paramètres donnés, d'une fonction de mise à jour a le même effet sur la base qu'une application unique. Dans le cas de la fonction **obj**, cette propriété nécessite le maintien d'une table de correspondance entre identifiants source et cible. Lors de chaque application, la table est consultée: si l'objet cible existe déjà, il est simplement retourné. Dans le cas contraire, l'objet est d'abord créé et la table mise à jour. La deuxième propriété garantit qu'une fonction est sans effet si un de ses paramètres est indéfini. Nous verrons que ce cas se présente après certaines réécritures.

Considérons maintenant les blocs de création de l'association *projects/staff*. Là encore, les deux blocs sont similaires. Les opérations de sélection et de jointure permettent de matérialiser l'association, uniquement sur les éléments devant être migrés (les chercheurs et projets de Rocquencourt). L'opération de regroupement permet d'associer à un chercheur (resp. projet) son ensemble d'objets projets (resp. chercheurs). Finalement, le *Map* met à jour les objets de la base de manière appropriée. Notons l'utilisation de la fonction **obj** qui, étant donné l'ordonnancement des blocs, correspond ici à un simple retrait de la table de correspondance (puisque tous les objets ont été créés précédemment).

4.2 Optimisation

Le meilleur programme de migration est celui qui permet de satisfaire les contraintes d'organisation physique spécifiées par l'utilisateur pour un coût d'évaluation minimal. Bien sûr, il est impossible de préciser à un niveau logique (l'algèbre) des contraintes physiques. Cependant, notons que l'ordonnancement des blocs et les opérations de tri et de regroupement de l'algèbre permettent de spécifier la façon dont les données seront données au système cible. L'optimiseur repose ainsi sur un principe simple et, de fait, efficace. Le code permettant le chargement des données suivant une organisation particulière sera plus simple à générer et plus efficace si le niveau logique fournit des données bien ordonnées et regroupées. Ainsi, la première phase de l'optimisation consiste à réorganiser les blocs de façon à satisfaire au mieux les contraintes de l'utilisateur. Nous verrons plus loin comment ceci est possible. Une fois cette première phase effectuée, certaines sous-expressions sont gelées (ne peuvent être modifiées) et un ordre partiel entre les blocs est imposé. La deuxième phase commence alors et va examiner toutes les possibilités de réécriture pour trouver la représentation dont le coût est minimal. Ce coût est calculé suivant différents critères que nous évoquons maintenant.

Trois systèmes participent à un programme de migration : les systèmes source et cible et le logiciel de migration. A l'évidence, les opérations de mise à jour ne peuvent être effectuées que sur le système cible. Par contre, les autres opérations peuvent être évaluées sur tous les sites, en fonction de leurs disponibilités et possibilités. L'équilibrage de charge sur les différents systèmes est donc un des facteurs du processus d'optimisation. Autre facteur important, le temps de transfert. Nous verrons que certaines expressions nécessitent des transferts de données plus faibles que d'autres. Si le réseau connectant les différents systèmes est mauvais, il sera clairement préférable de les privilégier. Dans le cas contraire, un bon compromis devra être trouvé entre temps de travail et de communication. Enfin, chaque opération algébrique a un coût qu'il convient de prendre en compte. Ainsi, la création d'un ensemble d'objets est plus ou moins cher selon qu'elle nécessite un parcours aléatoire ou séquentiel du disque. Le modèle de coût sur lequel repose la deuxième phase du processus d'optimisation englobe ces différents critères. Dans le système *RelOO*, ce modèle est implanté par un module séparé que l'utilisateur peut modifier. Par exemple, en mettant un poids plus fort sur la communication ou en donnant un coût infini à certaines opérations si elles sont affectées à un site particulier.

Voyons maintenant comment réécrire les blocs. Traditionnellement, une équivalence algébrique porte sur une expression sans effet de bord. Clairement, nous ne sommes pas dans un contexte traditionnel. De fait, le flux de données des différentes expressions formant un programme de migration ne nous intéresse que par son influence sur les opérations de mise à jour de la base cible. Nous avons donc conçu une nouvelle classe d'équivalences, appelées DS-équivalences (DS pour Database State), permettant de prendre en compte leurs effets de bords et de raisonner sur plusieurs expressions. Ces équivalences reposent sur les propriétés d'idempotence et d'insensibilité aux valeurs nulles que nous avons évoquées plus haut. Nous ne les donnons pas ici (elles peuvent être trouvées dans [8]) mais illustrons plutôt leur action en montrant un exemple de réécriture du programme algébrique introduit plus haut et illustré par la figure 4.5.

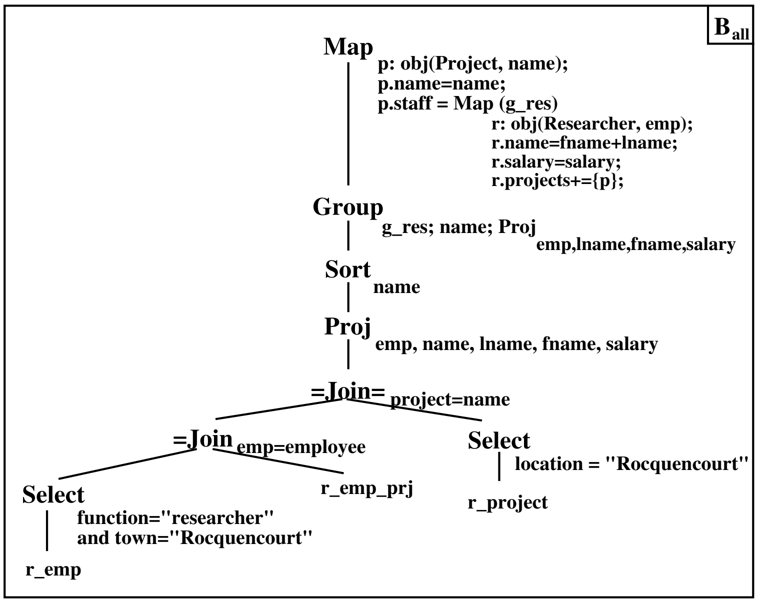

FIG. 4.5: *Une réécriture de la représentation par défaut*

La nouvelle représentation ne comprend qu'un bloc dont l'effet sur la base cible est identique à celui de la représentation par défaut. Dans un premier temps, toutes les informations sont regroupées en une collection unique. Pour cela, nous avons utilisé des jointures externes *=Join* et *=Join=*. La première, ou jointure externe gauche, conserve tous les n-uplets de son paramètre gauche, même s'ils ne satisfont pas le prédicat de jointure. Elle leur associe alors des valeurs nulles. La seconde applique la même stratégie sur les deux paramètres. Ainsi, sommes-nous surs d'avoir conservé dans le flux de données les chercheurs et projets ne figurant pas dans la relation *r_emp_prj*. Les données sont ensuite triées et regroupées sur le nom des projets. Enfin, l'opération *Map* construit tous les objets et leurs attributs atomiques et complexes. Elle travaille pour cela par imbrication. L'attribut *staff* de chaque objet *Project* créé est obtenu par l'application d'un *Map* imbriqué sur la partition *g_res*. Ce dernier construit et met à jour les objets *Researcher* du projet (notez l'utilisation de l'opération d'insertion dans un ensemble $+=$).

Le bloc **B_all** de la figure 4.5 construit l'intégralité de la base. Ceci ne signifie pas que la migration va s'effectuer en une passe. Le générateur de code crée un programme par bloc mais paramètre ce programme sur la durée. Ainsi, chaque lancement du programme correspondant au bloc **B_all** sélectionnera les données pouvant être chargées pendant la durée indiquée. Le point d'arrêt sera mémorisé par l'environnement d'exécution pour savoir où commencer le chargement suivant.

Parmi les différentes représentations possibles, l'optimiseur doit sélectionner la meilleure. Voyons comment les différents critères introduits précédemment permettent de choisir une solution plutôt qu'une autre en comparant les deux programmes algébriques donnés ci-dessus.

– Tout d'abord, remarquons que le bloc **B_all**, en opposition à la séquence de blocs du premier programme, prend en compte les contraintes physiques spécifiés par l'utilisateur. En effet les projets sont bien triés par nom et regroupés avec leurs chercheurs.

– Considérons maintenant les répartitions de charge. Les figures ne l'illustrent pas mais chaque opération est affectée à un système. Le modèle de coût actuel du système *RelOO* a été conçu de façon à privilégier le système relationnel pour toutes les opérations "relationnelles" (formulables en SQL). Ceci permet l'utilisation des index possiblement maintenus par ce système. Tout autre choix est possible. Quoiqu'il en soit, dans le cas présent, le système relationnel a une charge plus importante avec la première représentation qu'avec la seconde puisqu'il devra effectuer quatre requêtes et donc de nombreux parcours de ses relations. Notons tout de même que le résultat des opérations de jointure du bloc **B_projects** peut être mémorisé pour l'évaluation du bloc **B_staff**.

  Du côté objet, les choses sont moins claires et dépendent de la façon dont les données sont stockées sur le disque. Dans le cas présent, nous avons vu que la seconde représentation était meilleure puisqu'elle permettait de prendre en compte les contraintes physiques spécifiées par l'utilisateur [16]. Supposons donc qu'aucune contrainte n'ait été donnée et que le système objet crée un fichier par classe et par valeur complexe. Le programme de la figure 4.4 accédera à ces fichiers de façon séquentielle, celui de la figure 4.5 effectuera des accès aléatoires, et donc plus coûteux, sur le fichier correspondant aux chercheurs et à leurs projets.

  Considérons maintenant le logiciel de migration. Il a la charge du maintien des tables de correspondance et, éventuellement, des opérations intermédiaires (comme les regroupements). En ce qui concerne les tables de correspondance, notons que le bloc **B_all** est intéressant puisqu'il ne nécessite pas le maintien d'une table pour les projets, ces objets étant créés de façon unique et séquentielle. Par contre, même s'il ne contient qu'une opération de regroupement, cette opération est onéreuse puisqu'elle porte sur des volumes de données plus importants que les deux regroupements simples des blocs **B_projects** et **B_staff**.

– La première représentation nécessite un transfert de données moins important que la seconde. Ceci tient aux projections extrêmement réductrices effectuées sur les résultats de jointure et qui sont, nous le rappelons, évaluées du côté relationnel.

– Enfin, en terme d'opérations algébriques pures, nous préférerons le second programme qui nécessite moins d'opérations que le premier.

Les deux représentations du programme de migration données dans cette section forment, en quelque sorte, deux extrêmes: dans un cas, il existe autant de blocs que d'opérations logiques sur la base (création d'objets ou de lien), dans l'autre, un bloc unique les intègre toutes. Les DS-équivalences permettent de considérer toutes les possibilités entre ces deux extrêmes. En les combinant avec les équivalences traditionnelles, l'optimiseur a la possibilité d'examiner un large champ de réécritures qui couvre les multiples solutions qu'un programmeur raisonnable pourrait mettre en œuvre.

16. De fait, l'autre programme sera rejeté par la première phase de l'optimisation puisque, justement, il ne les satisfait pas. Cependant, bien que nous ne l'illustrons pas ici, il existe différents programmes satisfaisant ces contraintes et l'optimiseur devra en sélectionner un.

Chapitre 5 : Données hétérogènes

Si la plupart des applications intégrées s'appuient actuellement sur la matérialisation des informations dans un système centralisateur, la tendance actuelle va vers l'utilisation d'une architecture médiateur-adaptateurs permettant une intégration logique (ou virtuelle) plutôt que physique, évitant d'avoir à gérer la cohérence des données répliquées. Ce type d'architecture repose sur un logiciel pivot, le médiateur, qui constitue l'interface entre une application et les systèmes sources d'informations. Le médiateur propose un modèle unique de représentation des données et des opérations. Chaque système intégré s'y connecte via un logiciel, appelé adaptateur, qui traduit ses données et opérations vers le modèle du médiateur (et vice versa). L'application peut alors être construite directement sur le médiateur ou via un adaptateur. Elle accède aux données par des requêtes sur le médiateur. Ce dernier les décompose en requêtes sur les différentes sources et construit la réponse en intégrant les résultats ainsi obtenus.

Dans le chapitre 3, nous avons proposé une architecture pour l'intégration des documents structurés reposant sur une approche similaire. Le médiateur était un SGBDOO, les adaptateurs étaient des schémas structurants, les applications des programmes O_2. Les techniques d'optimisation reposaient soit sur une matérialisation des documents dans le médiateur, soit sur une réécriture élémentaire de l'adaptateur. Si nous considérions l'utilisation d'index plein-texte pour certaines requêtes, celle-ci reposait sur un codage en dur de l'optimiseur et ne permettait pas une prise en compte dynamique de nouvelles capacités opérationnelles. Ce type d'approche, s'il permet effectivement l'intégration simple et efficace de documents dont la grammaire est figée (comme les documents SGML), n'est pas applicable à des données peu structurées ni à des données arbitraires provenant de systèmes divers, chacun offrant des capacités de calcul différentes. Dans ce chapitre, nous étudions le problème plus général de l'intégration dynamique, virtuelle et efficace de données hétérogènes.

Pour illustrer le problème, nous utilisons un exemple représentatif. L'application est simple mais utilise deux sources de données hétérogènes, à la fois en terme de structures et d'opérations. La première est un SGBDOO, interrogeable par OQL, et contenant des informations sur des artistes et leur œuvre. La seconde contient les biographies des auteurs, représentées par des documents XML dont une partie seulement de la structure est connue. Elle peut être interrogée par des requêtes plein-texte sur l'intégralité d'un document ou sur le contenu d'un attribut particulier appelé *text*. L'application repose sur une vue connectant artistes et biographies.

Comme nous l'avons évoqué, un médiateur propose un modèle de représentation des données et des opérations avec lequel les différents systèmes devront s'interfacer. Ce modèle constitue le cœur du problème. Il doit permettre d'intégrer simplement tout type de données mais également de comprendre les opérations des systèmes sources. Cette dernière caractéristique est essentielle au processus de décomposition des requêtes, lequel permet de ne matérialiser dans le médiateur que les données nécessaires à l'évaluation d'une requête.

Il existe deux familles de médiateurs. La première, dite structurée, repose sur des modèles bases de données, relationnel ou objet (voir [21, 25, 41, 59, 53, 74, 75, 84, 94, 102, 110]). La seconde, dite semistructurée, s'appuie sur des modèles de graphes non typés (voir [50, 92]).

Systèmes structurés : La première étape pour écrire une application intégrant différentes sources au-dessus d'un médiateur structuré consiste à définir son schéma. Les adaptateurs sont ensuite conçus pour l'application et traduisent les données sources conformément au schéma pré-défini. Egalement, ils précisent au médiateur l'ensemble des requêtes qu'ils sont capables d'évaluer [17]. L'avantage principal de cette approche est qu'elle permet l'application de techniques d'optimisation efficaces (celles des SGBDs) et permet une bonne décomposition des requêtes sur les sources. Par contre, elle se prête mal à l'intégration de données et d'opérations peu structurées et nécessite un gros effort de mise en place. Dans le cas de notre application cible, ceci se traduit par la perte vraisemblable d'une partie des biographies (celle dont la structure est arbitraire), et la nécessité de coder deux adaptateurs, logiciels complexes et spécifiques à l'application. De plus, les capacités de recherche plein-texte de la source XML ne pourront pas être intégrées dans le médiateur. Dans le meilleur des cas, ces facilités seront codées dans l'adaptateur et utilisées de façon aveugle par le médiateur qui ne connaît ni leur existence ni leur sémantique.

Systèmes semistructurés : Les médiateurs semistructurés permettent l'utilisation d'adaptateurs génériques, définis pour une technologie plutôt qu'une application. Ainsi, la mise en place d'une application distribuée est simple. Dans le cas présent, le programmeur connectera les deux sources via des adaptateurs fournis par le système (ou programmés, mais réutilisables), qui traduisent des données ODMG et XML quelconques en des graphes. Un langage d'intégration déclaratif permet ensuite de spécifier la façon dont les données doivent être intégrées. Cette simplicité a cependant un prix. Tout d'abord, l'absence de type rend difficile l'écriture d'applications complexes au-dessus de la vue, des langages tels que Java ou C++ requérant des données typées. De plus et toujours par manque de typage, cette famille de médiateurs offre peu de techniques d'optimisation. Si un système comme TSIMMIS [61] résout partiellement ce problème en permettant de pousser certaines requêtes sur les système sources, les optimisations proposées sont limitées et reposent sur un mécanisme qui rompt la nature générique des adaptateurs.

Dans ce chapitre, nous proposons une approche offrant le meilleur des deux mondes. Le système YAT qui a été développé au sein de l'équipe VERSO de l'INRIA repose sur (i) un modèle semistructuré, (ii) un langage d'intégration déclaratif et (iii) un modèle opérationnel avec des techniques d'optimisation appropriées. Cette méthodologie de conception devrait être maintenant familière au lecteur.

- Les données sont représentées en YAT par des graphes arbitraires (en fait, des arbres dont les feuilles peuvent référencer d'autres arbres). L'originalité du modèle vient de sa capacité à représenter des données quelconques mais également (et facultativement)

17. Et éventuellement le coût de ces requêtes.

leur type et/ou leur système de types. La connection entre les différents niveaux de représentation est donnée par un mécanisme d'instanciation. Ainsi, le modèle est à même d'intégrer de manière très flexible données très, peu ou pas structurées, tout en offrant les avantages d'un modèle structuré (typage des programmes, optimisation).

– Le langage déclaratif YATL permet de spécifier la façon dont les données doivent être intégrées et vues par l'application. Cette vue est construite en fonction des spécificités de l'application qui peut requérir une représentation objet (par exemple, une application Java avec accès aux données via OQL), HTML (pour une navigation depuis le Web), relationnelle (pour un accès par formulaires simples, par exemple) ou autre. Bien que ceci ne soit pas illustré ici, le mécanisme d'instanciation a été étendu aux programmes pour permettre leur typage, composition ou combinaison cohérente et la programmation par personnalisation. Par exemple, un programme générant la représentation HTML d'une base ODMG, peut être personnalisé en deux étapes : (i) instanciation sur une classe ODMG donnée (par exemple *Artist*) et (ii) modification par l'utilisateur. Le programme obtenu peut être alors combiné avec le programme initial pour permettre la traduction de toutes données ODMG, avec un traitement particulier pour les artistes. Enfin, en le composant avec un programme donnant la représentation ODMG d'une base relationnelle, on obtient une traduction relationnel-HTML.

– Pour son évaluation YATL repose sur un modèle opérationnel. Celui-ci comprend une algèbre pour données semistructurées qui est une extension de notre algèbre objet. Les atouts de ce modèle tiennent à sa nature extensible et au fait qu'il peut être spécialisé. Ceci lui permet de comprendre de façon générique aussi bien des langages très structurés comme SQL ou OQL que les primitives plein-texte des systèmes de recherche d'information. Ceci offre deux avantages : (i) le médiateur peut être interrogé par différents langages et (ii) les adaptateurs restent génériques tout en permettant une interrogation très fine des systèmes sources. De plus, ce dernier point et les bonnes propriétés de notre algèbre donnent à YAT des possibilités d'optimisation qui vont au-delà de celles des systèmes structurés.

Dans ce chapitre, nous illustrons par l'exemple ces différents points. Le papier donné en annexe [37] présente une version simplifiée du modèle et le langage YATL. Plus de détails sur le modèle et les techniques d'optimisation peuvent être trouvés dans [97].

5.1 Modèle de données

YAT repose sur l'utilisation d'adaptateurs génériques. Notre application exemple nécessite la connection du médiateur avec deux adaptateurs : un pour le système O_2 et un pour le système XML (système très simple et reposant sur une DTD unique que nous avons développé). Une fois cette connection établie, il est possible d'importer dans le médiateur des informations sur les données manipulées par les sources. Cette importation repose sur une traduction simple des données et structures dans le modèle YAT.

Le modèle de données est semistructuré. Il s'inspire d'un premier modèle que nous avions développé [5] et l'enrichit par des constructions de type[18] et un mécanisme d'instanciation. L'instanciation permet de manipuler différents niveaux d'abstraction et de typer données

18. Le modèle propose différents constructeurs que nous ne présentons pas ici (voir [97]).

et programmes[19]. La figure 5.1 illustre quatre différents niveaux de représentation, parmi lesquels trois ont été importés par l'adaptateur du système O_2.

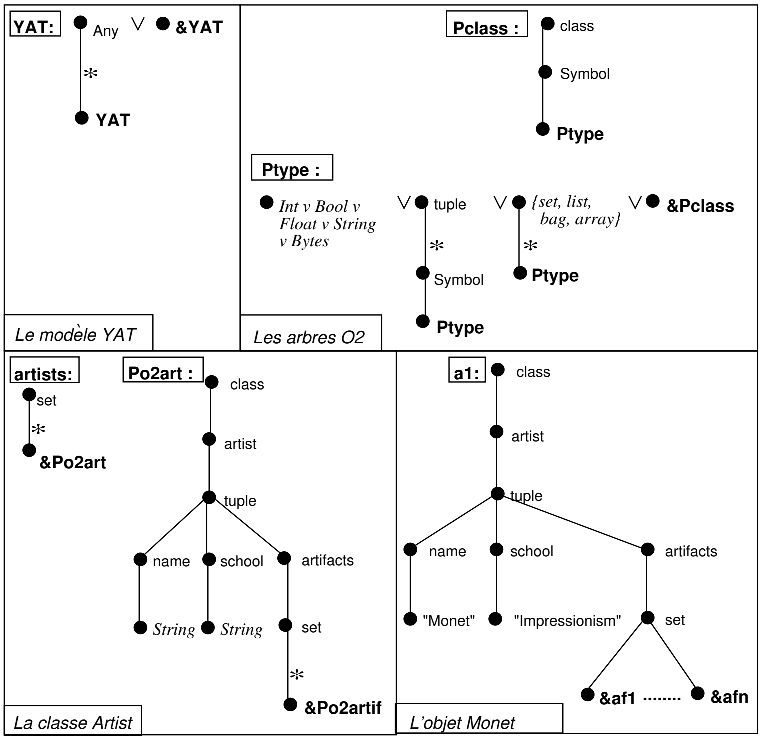

FIG. 5.1: *Du général au spécifique*

L'arbre dont l'identifiant est $\boldsymbol{a1}$, qui est situé en bas et à droite de la figure, représente l'objet O_2 correspondant au peintre "Monet". Les labels des nœuds d'un arbre sont utilisés pour dénoter des valeurs ("Monet"), des attributs (*name*) ou des informations de type (*class*). Le modèle offre également la possibilité de référencer d'autres arbres, permettant ainsi le partage d'information (par exemple, $\boldsymbol{\&af1}$ est une référence à un objet d'une classe *Artifact*).

A gauche de ce premier arbre, se trouve la représentation (i) d'une racine de persistance appelée `artists` et (ii) de la classe *Artist*. Notez la possibilité de représenter des collections par l'utilisation du symbole d'arête "$\star$" qui dénote le fait qu'un nœud *set* peut avoir un nombre arbitraire (éventuellement nul) d'enfants. L'arbre $\boldsymbol{a1}$ représentant l'objet "Monet" est une *instance* de l'arbre plus général $\boldsymbol{Po2art}$: on peut faire correspondre chaque nœud (resp. arête) de $\boldsymbol{a1}$ à un nœud (resp. arête) de $\boldsymbol{Po2art}$ dont le label est égal (par exemple, *class* ou le symbole vide qui marque les arêtes simples) ou plus petit (nous utilisons une notion intuitive de l'ordre : par exemple, "Monet" $\leq$ *String* ou le symbole vide $\leq \star$).

Dans la partie haute de la figure se trouve la description des modèles de données YAT (à gauche) et O_2 (à droite). Notez l'utilisation d'unions ($\vee$) pour matérialiser les alternatives de structures. Ainsi, un arbre $\boldsymbol{YAT}$ est soit un nœud dont le label et le nombre d'enfants sont arbitraires, soit un nœud contenant une référence à un autre arbre $\boldsymbol{YAT}$. Tous les arbres de la figure 5.1 sont instances de $\boldsymbol{YAT}$ et les deux arbres $\boldsymbol{a1}$ et $\boldsymbol{Po2art}$ sont instances de $\boldsymbol{O_2}$.

19. Une formalisation alternative, légèrement plus générale, du modèle en tant que système de types pour les données semistructurées a été récemment proposée dans [16].

De manière similaire à [20], le mécanisme d'instanciation permet ainsi d'englober les notions de typage (***a1*** est de type ***Po2art***) et de subsomption (***Po2art*** subsume ***Pclass***).

Une autre caractéristique importante du modèle YAT est qu'il permet de mélanger information très et peu précise. Cette caractéristique est essentielle pour pouvoir intégrer, par exemple, documents XML, bibliographies BibTex ou sites Web quelconques. Elle est illustrée par la figure 5.2 qui décrit la structure des biographies exportées par notre source XML.

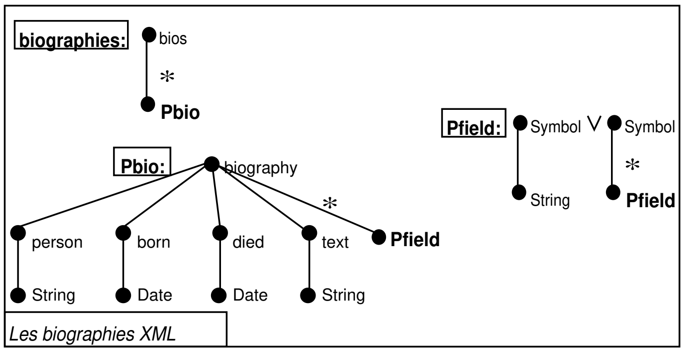

FIG. 5.2: *Les biographies XML*

L'arbre `biographies` est une collection de documents XML. Chaque document (***Pbio***) est décrit par une séquence d'éléments bien définis (*person*, *born*, *died* et *text*) suivie d'un nombre quelconque d'éléments (***Pfield***) dont on sait simplement qu'ils ont un nom (*Symbol*) et qu'ils sont composés soit d'une chaîne de caractères, soit d'éléments imbriqués. Notez que la capacité du modèle à comprendre ce type de description permet d'offrir les avantages d'un modèle structuré (utilisation de la structure) sans en avoir les inconvénients (impossibilité d'importer les éléments additionnels de nos biographies). Dans l'exemple, l'application pourra interroger toute l'information contenue dans les documents mais non spécifiée par sa DTD (qui peuvent être les lieux favoris d'un peintre, les expositions auxquelles il a participé, etc.).

5.2 Spécification de la vue

Une fois l'information structurelle concernant les systèmes sources importée dans le médiateur, l'utilisateur peut spécifier la vue intégrée sur laquelle l'application est construite en utilisant le langage YATL.

YATL est un langage à base de règles. Nous donnons ici un exemple simple d'une règle permettant la création partielle d'une vue objet. Cependant, le langage offre la possibilité de définir des vues conformes à d'autres formats spécifiés par des arbres YAT, formats relationnel, HTML ou autre (voir [98] pour une vue HTML).

Comme pour les vues objet [57, 99, 104], une règle YATL comprend deux parties définissant données et opérations. Considérons tout d'abord la partie correspondant aux données. La règle `odmg_xml_artists` ci-dessous construit une racine de persistance appelée `allartists` qui intègre des informations sur les artistes depuis deux racines importées : `artists` (de la source O_2) et `biographies` (de la source XML). Pour écrire cette règle, le programmeur utilise les descriptions structurelles exportées par les adaptateurs (voir les figures 5.1 et 5.2).

Dans un premier temps, la règle extraît les données nécessaires telles que le nom (N), l'école (S) ou la date de naissance (B) d'un artiste en utilisant un mécanisme de filtrage basé sur l'instanciation. Les filtres sont des arbres YAT quelconques avec variables (les noms de variables commencent par une majuscule). La règle en comprend cinq dans sa partie droite, le premier permet d'extraire les artistes, le second des informations sur ceux-ci, etc. Lorsqu'un arbre concret est instance d'un filtre, il en résulte une liaison des variables. A partir des liaisons obtenues sur les différents filtres et réunies par jointure, la règle construit un nouvel arbre $\boldsymbol{Partist}$ pour chaque nom distinct et les regroupe dans `allartists`. L'utilisation de fonctions de Skolem permet de contrôler la création de nouveaux identifiants ($\boldsymbol{Partist}(N)$) et de créer des références à des arbres définis dans d'autres règles ($\boldsymbol{\&Partifact}(N_A, D_A)$). Ces fonctions maintiennent la correspondance entre données entrantes et leur représentation dans la vue et sont particulièrement utiles à un accès par navigation (voir [98]). Certaines des primitives de YATL peuvent engendrer du non-déterminisme (voir [37]). Pour simplifier la discussion, nous ne considérons que des programmes déterministes. Notamment, nous considérons que les paramètres des fonctions de Skolem sont des clés.

`rule odmg_xml_artists =`

`allartists :`
$root \xrightarrow{\star} \boldsymbol{Partist}(N) :$
$artist\langle \rightarrow name \rightarrow N,$
$\rightarrow born \rightarrow B,$
$\rightarrow died \rightarrow D,$
$\rightarrow school \rightarrow S,$
$\rightarrow artifacts \xrightarrow{\star} \boldsymbol{\&Partifact}(N_A, D_A),$
$\rightarrow description \rightarrow T,$
$\rightarrow biography \xrightarrow{\star} \boldsymbol{PField} \rangle$

$\Longleftarrow$

`artists :`
$set \xrightarrow{\star} \boldsymbol{\&Po2art},$
$\boldsymbol{Po2art} :$
$class \rightarrow artist \rightarrow tuple\langle \rightarrow name \rightarrow N,$
$\rightarrow school \rightarrow S,$
$\rightarrow artifacts \rightarrow set \xrightarrow{\star} \boldsymbol{\&A}\rangle,$
$\boldsymbol{A} :$
$class \rightarrow artifact \rightarrow tuple\langle \rightarrow name \rightarrow N_A,$
$\rightarrow date \rightarrow D_A\rangle,$
`biographies :`
$bios \xrightarrow{\star} \boldsymbol{Pbio}$
$\boldsymbol{Pbio} :$
$biography\langle \rightarrow person \rightarrow P,$
$\rightarrow born \rightarrow B,$
$\rightarrow died \rightarrow D,$
$\rightarrow text \rightarrow T,$
$\xrightarrow{\star} \boldsymbol{PField} \rangle,$
$N = P$

`view operations:`

`description_contains`(X: $String$, $\boldsymbol{Partist}$) $\equiv$ `text_contains`(X: $String$, $\boldsymbol{Pbio}$)

Une vue peut cacher certains arbres (ou types) et, par là même, les opérations définies sur ces arbres (la section suivante montre comment importer de nouvelles opérations dans le médiateur). Pour cette raison, il est parfois nécessaire de redéfinir des opérations sur la vue. Ceci est illustré dans la partie basse de la règle `odmg_xml_artists`. L'opération `text_contains`, qui a été importée depuis la source XML, est définie sur les arbres $\boldsymbol{Pbio}$. Elle permet de savoir si l'élément `text` d'une biographie contient un mot donné. Elle est redéfinie, sous le nom `description_contains`, sur les arbres $\boldsymbol{Partist}$.

5.3 Optimisation

Une fois la vue spécifiée, l'utilisateur peut l'interroger. Cette interrogation se fait directement sur le médiateur en utilisant le langage YATL [20] ou, via un adaptateur, dans un autre langage. La seule contrainte concernant le langage utilisé est qu'il soit traduisible dans le modèle opérationnel de YAT. Nous verrons que c'est le cas de langages tels que SQL ou OQL. Une extension des filtres lui permettrait également de comprendre des langages plus puissants comme, par exemple, POQL [28], Lorel [7] ou StruQL [49].

Les requêtes sont ensuite traduites dans le modèle opérationnel, combinées avec la définition de la vue et optimisées. L'optimisation consiste en des réécritures de la requête qui peuvent être "locales" (similaires à celles présentées dans le chapitre 2) ou distribuées (qui sont des applications au contexte semistructuré de celles utilisées traditionnellement dans les systèmes distribués [90]). Ces dernières correspondent à une décomposition de la requête en sous-requêtes qui seront évaluées par les différentes sources. L'optimisation distribuée est essentielle : elle permet (i) d'éviter la matérialisation de toutes les données accédées dans le médiateur [21], (ii) de diminuer la charge du réseau, (iii) d'exploiter les potentielles structures d'accès rapide (index) des sources et (iv) de paralléliser l'évaluation sur les différents systèmes impliqués. Bien évidemment, elle nécessite que le médiateur connaisse les capacités opérationnelles des sources intégrées. Nous présentons, dans un premier temps, le modèle opérationnel et la façon dont cette connaissance est acquise. Puis, nous illustrons par deux exemples les possibilités d'optimisation offertes.

Modèle opérationnel et importation des capacités des sources

Le modèle opérationnel de YAT a été conçu de façon à maximiser la connaissance que le médiateur a des sources intégrées. Un autre objectif important était la possibilité d'importer ces informations de façon générique. Pour offrir ces deux caractéristiques, nous avons conçu un modèle extensible et spécialisable. Notre expérience avec OQL nous a appris qu'un langage fonctionnel pouvait être simplement étendu. La première caractéristique est donc obtenue par une approche similaire : le modèle YAT est fonctionnel. Dans un autre contexte, le mécanisme d'instanciation de YAT s'est montré particulièrement utile pour intégrer des informations sur les types. En utilisant ce mécanisme sur les paramètres des opérations du modèle, nous obtenons la deuxième caractéristique. Un opérateur peut être spécialisé par instanciation de ses paramètres.

La figure 5.3 montre en partie le modèle opérationnel de YAT après que les adaptateurs génériques de nos deux sources aient spécifié leurs capacités. Les opérateurs (représentés par des rectangles) sont organisés en fonction de leurs catégories (par exemple, `algebra` *vs* `function`) ou de leur signature (par exemple, `Predicate`). A chaque opérateur est associé un descripteur contenant sa signature et l'ensemble des systèmes pouvant l'évaluer (*suppliers*). Egalement, l'opérateur `Bind` est spécialisé (les arêtes épaisses dénotent la spécialisation).

Examinons d'abord la partie gauche de la figure. Elle montre quelques opérations de base (comme `=`) évaluables par les systèmes O_2 et YAT. D'autres opérations sont spécifiques à

20. YATL a été principalement conçu pour spécifier intégration et conversion. Pour faciliter son utilisation dans un contexte de requêtes, nous avons ajouté des raccourcis syntaxiques permettant notamment d'utiliser des filtres plus simples, ne décrivant qu'une partie de la structure des données filtrées.

21. Notons que la matérialisation implique que le médiateur possède son propre système de stockage sur disque. Nous n'avons pas encore considéré ce problème.

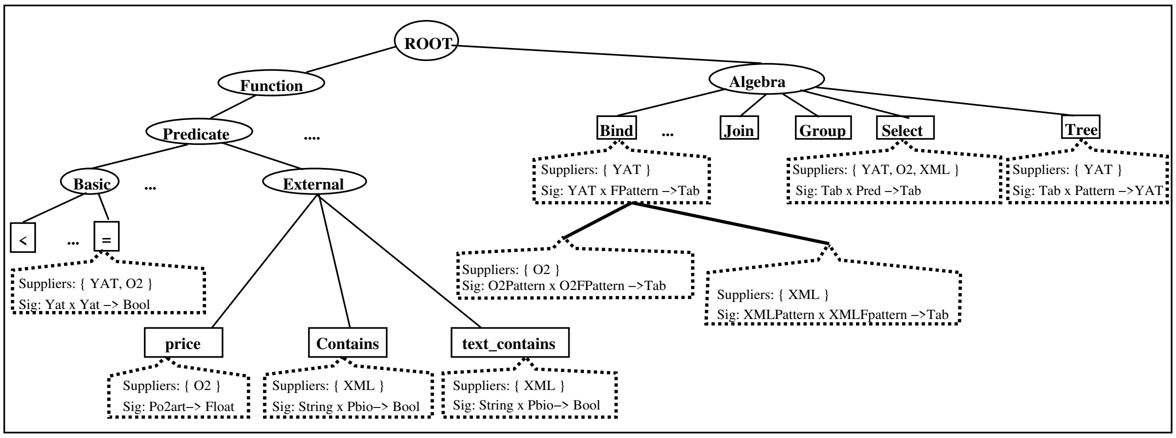

FIG. 5.3: *Le modèle opérationnel après importation des capacités externes*

un système particulier : `text_contains` et `contains` ont été ajoutées au modèle de base par l'adaptateur de notre système XML, `price` par celui d'O_2. Les deux premières correspondent à une recherche plein-texte sur un document (dont le type YAT est ***Pbio***) ou l'élément `text` d'un document. La dernière est une méthode de la classe *Artifact* (***Po2art***).

Regardons maintenant la partie droite de la figure et considérons les opérateurs algébriques avant spécialisation. L'algèbre est celle présentée dans le chapitre 2 enrichie par deux opérateurs *Bind* et *Tree*. Nous la présentons très succinctement (voir [97] pour plus de détails).

L'opérateur *Bind* permet de s'extraire des structures arbitraires de YAT. Nous l'utilisons, comme l'opérateur *Map* dans le contexte objet, pour construire des structures (similaires aux collections de n-uplets et appelées ici *Tab*) sur lesquelles il est possible de définir des opérateurs ayant de bonnes propriétés (ceux de l'algèbre objet). Pour cela, nous utilisons les capacités de filtrage de YAT pour extraire l'information des arbres et les lier à des variables. A chacune de ces variables, correspondra un attribut dans la collection résultante. Ainsi, les paramètres de cet opérateur sont un arbre et un filtre YAT (un arbre avec variables dénoté ***Fpattern***). Le résultat est un arbre à la structure particulière, appelée `Tab`. La figure 5.4 montre l'application d'un *Bind* à la racine de persistance *artists* (voir la figure 5.1 de la section précédente). L'opération extrait le nom, l'école et les œuvres de chaque artiste (le symbole \$ préfixe les variables, le symbole $\star$ permet d'extraire un sous-arbre entier plutôt que sa racine)[22]. Le résultat est un arbre (`Tab`) dont la structure n'est pas sans rappeler celle d'une collection de n-uplets.

L'opération *Tree* est le complément du *Bind* et permet de construire des arbres quelconques à partir d'un *Tab*. Ses paramètres sont donc un *Tab* et un arbre spécifiant la construction (*Pattern* qui ne sera pas détaillé ici). Cet opérateur est très puissant, permettant notamment d'effectuer regroupement et tri. La figure 5.5 montre l'application d'un *Tree* sur le résultat du *Bind* précédent. Les données sont regroupées par école. Pour chaque école, un arbre est construit contenant autant de branches que d'artistes lui appartenant.

La sémantique des autres opérateurs de notre algèbre ne diffère pas de celle présentée dans le chapitre 2, la seule différence est qu'ils sont définis sur des *Tab*s plutôt que sur les collections de n-uplets du modèle objet. Dans la mesure où ces opérateurs ne voient que le

22. Notez que nous utilisons ici un raccourci syntaxique permettant de ne spécifier qu'une partie de la structure de la donnée entrante.

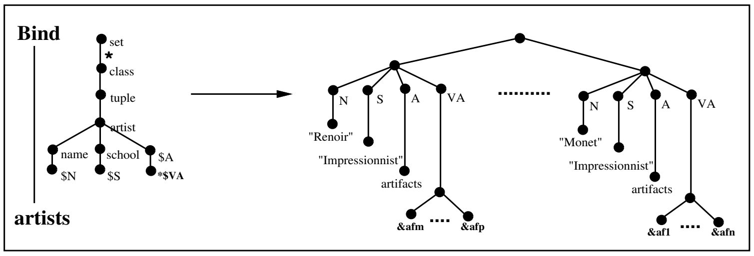

FIG. 5.4: *Une opération Bind*

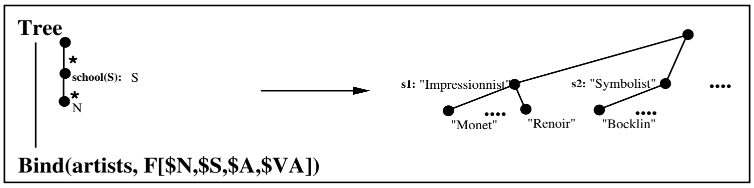

FIG. 5.5: *Une opération Tree*

niveau supérieur de la structure à laquelle ils s'appliquent, ceci ne soulève aucun problème particulier.

Voyons maintenant les spécialisations réalisées par nos adaptateurs. L'opérateur *Bind* a été spécialisé pour O_2 et XML. Cette spécialisation ne change pas sa sémantique, elle réduit simplement sa portée. Ainsi, notez que les paramètres du *Bind* d'O_2 sont un arbre décrivant une structure O_2 (***O2pattern***, qui regroupe les arbres du modèle O_2 de la figure 5.1) et un filtre ***O2Fpattern***, chacun respectivement instance de ***YAT*** et ***Fpattern***. Nous ne donnons pas ici la description des arbres ***O2Fpattern*** (voir [97]). Ils correspondent aux capacités de filtrage d'OQL qui sont plus ou moins celles de YAT sans la possibilité de lier une variable à une information de type. Ainsi, l'opération de la figure 5.4 pourrait être poussée sur la source O_2. L'opérateur *Bind* a des capacités de filtrage encore plus réduites dans la source XML. Les filtres ***XMLFpattern*** spécifient que seuls les documents dans leur intégralité peuvent être liés à une variable.

Notez que l'opération de sélection peut être évaluée par les deux systèmes intégrés sans que cela ait donné lieu à une spécialisation. Celle-ci est rendue inutile par le fait qu'une opération ne peut être poussée que si ses paramètres l'ont été. Par exemple, une sélection ne sera poussée sur O_2 que si (i) son prédicat est poussable et (ii) un *Bind* a été poussé précédemment, ce qui garantit un bon typage de l'opération.

Il est important de comprendre que le pouvoir d'expression des langages de requêtes des deux sources intégrées est englobé dans le modèle opérationnel partiellement illustré par la figure 5.3. A notre connaissance, seul le modèle YAT offre cette possibilité de comprendre finement et de façon générique les capacités opérationnelles de sources hétérogènes.

Optimisation

Nous donnons ici deux exemples de requêtes YATL qui illustrent bien les possibilités d'optimisation offertes par la bonne connaissance que YAT a des capacités de nos deux sources. L'optimiseur de requêtes est le seul module du système YAT encore en développement. Nous présentons donc un travail conceptuel qui n'a pas encore été validé.

Exemple 5.1 La requête extraît le nom des artistes impressionnistes qui ont participé à l'exposition de 1874. Elle est formulée en YATL de la façon suivante :

$$\boldsymbol{names} \xrightarrow{\star} \boldsymbol{N} \quad \Longleftarrow \quad \begin{array}{l} \texttt{allartists :} \\ \quad \boldsymbol{root} \xrightarrow{\star} \boldsymbol{PA} : \\ \qquad \boldsymbol{artist}\langle \rightarrow \boldsymbol{name} \rightarrow \boldsymbol{N}, \\ \qquad\qquad\qquad \rightarrow \boldsymbol{school} \rightarrow \boldsymbol{S}\rangle, \\ \boldsymbol{S} = \text{“Impressionnist”}, \\ \texttt{description\_contains}(\text{“exhibition of 1874”}, \boldsymbol{PA}) \end{array}$$

La requête extraît l'identifiant ($\boldsymbol{PA}$), le nom ($\boldsymbol{N}$) et l'école ($\boldsymbol{S}$) des artistes de `allartists`. Elle sélectionne ensuite ceux dont l'école est "Impressionnist" et dont la description contient "exhibition of 1874". Enfin, elle construit l'arbre résultant avec une branche par nom sélectionné. □

La requête est traduite et combinée avec l'expression algébrique correspondant à la définition de la racine `allartists` dans la vue. Le résultat de cette première étape est illustré sur la figure 5.6, où la frontière entre vue (partie basse) et requête (partie haute) est caractérisé par une séquence d'opérations $\boldsymbol{Bind} - \boldsymbol{Tree}$. La lecture du chapitre 2 est un pré-requis à la compréhension de ce qui suit.

La vue est matérialisée par (i) une succession d'opérations *Bind* (qui extraient les informations des sources et correspondent aux filtres utilisés pour spécifier la vue), (ii) des opérations *D-join* et *Join* qui associent les variables et (iii) un *Tree* qui construit la racine `allartists`. La requête est, quant à elle, une simple séquence *Bind-Select-Tree*.

Voyons maintenant une version optimisée possible de cette requête illustrée par la figure 5.7. Nous n'expliquons pas en détail quelles équivalences nous ont permis d'aboutir à ce résultat (voir [97]). Notons juste la plus importante qui repose sur une relation d'instanciation entre les filtres des deux opérations frontières et permet de les réécrire en une simple projection avec renommage (voir la figure 5.6). Les autres sont traditionnelles (réordonnancement et élimination de jointure, sélection poussée, etc.) ou spécifiques à YAT (simplification ou division de *Bind*, traduction de l'opérateur `description_contains` de la vue en `text_contains` sur la source XML)[23].

L'expression optimisée peut être évaluée de la façon suivante. Tout d'abord, YAT demande à la source XML les biographies dont l'élément $\boldsymbol{text}$ contient la chaîne de caractères "exhibition of 1874". Ceci correspond aux opérations *Bind* et *Select* dans la partie basse et gauche de la figure. Notez que le *Bind* a été simplifié (en fait, coupé en deux) pour être poussé sur le système XML. Pour évaluer cette requête, le système utilisera l'index plein-texte dont l'existence a été spécifiée au médiateur. Ensuite, un deuxième *Bind*, évalué cette fois dans le médiateur, permet d'extraire le nom des artistes des biographies sélectionnées. Pour chacun de ces noms, la source O_2 est appelée (notez l'opérateur *D-join* et la variable liée $\boldsymbol{N}$) et retourne les informations sur l'artiste si il appartient à l'école impressionniste. Ceci correspond à un passage d'information entre sources qui peut être particulièrement intéressant si peu de

23. Notons que la réécriture repose sur le fait que le résultat de la requête est ensembliste et sur une propriété sémantique des données O_2 : tout artiste a au moins une œuvre associée.

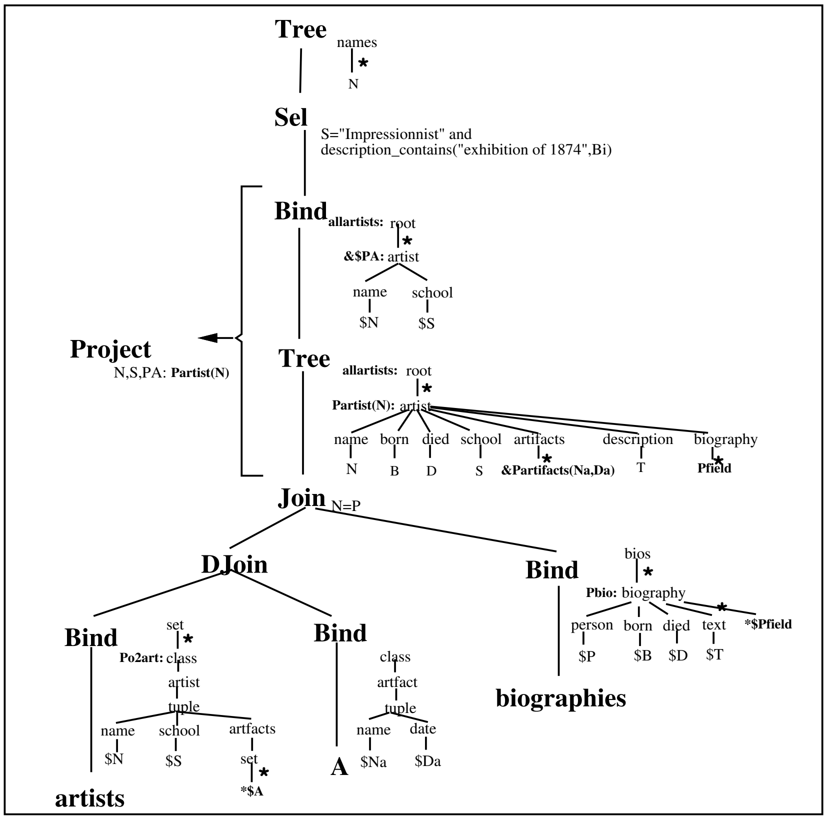

FIG. 5.6: *Traduction algébrique de la requête 5.1*

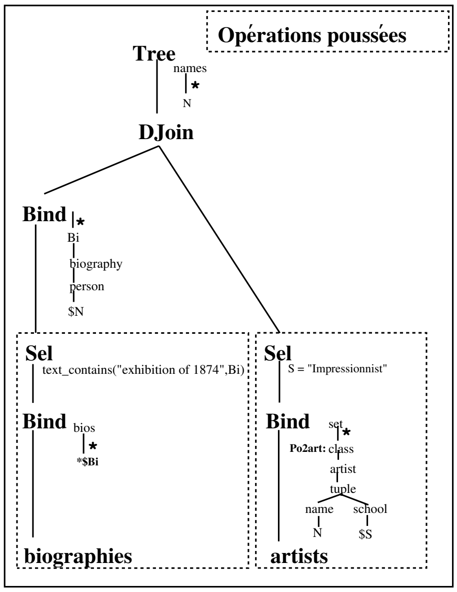

FIG. 5.7: *Optimisation de la requête 5.1*

biographies sont sélectionnées. Il aurait été également possible de travailler par jointure, sans passage d'information.

La requête optimisée est clairement meilleure que la précédente. Elle évite la matérialisation d'une information qui n'est pas nécessaire à la requête et comprend moins d'opérations.

Considérons maintenant un deuxième exemple qui montre comment (i) interroger la partie des biographies dont la structure est peu précise et (ii) utiliser les capacités plein-texte de la source XML pour évaluer des requêtes structurées.

Exemple 5.2 La requête extraît le nom des artistes dont le lieu favori est Giverny.

$$\begin{array}{lcl} & & \texttt{allartists :} \\ \boldsymbol{names} \xrightarrow{\star} \boldsymbol{N} & \Longleftarrow & \boldsymbol{root} \xrightarrow{\star} \boldsymbol{artist}\langle \rightarrow \boldsymbol{name} \rightarrow \boldsymbol{N}, \\ & & \qquad\qquad\qquad \rightarrow \boldsymbol{biography} \rightarrow \boldsymbol{favorite\_place} \rightarrow \boldsymbol{P}, \\ & & \boldsymbol{P} = \text{“Giverny”} \end{array}$$

□

Sa traduction algébrique est donnée par la figure 5.8.

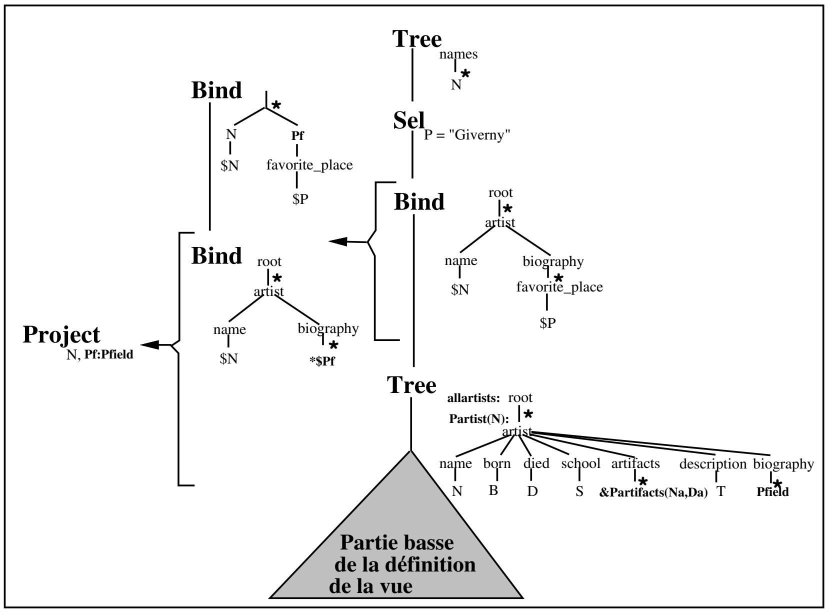

FIG. 5.8: *Traduction algébrique de la requête 5.2*

Pour réécrire cette requête, il est d'abord nécessaire de diviser l'opérateur *Bind* à la frontière vue-requêtes. Ceci est dû au fait que son filtre original est plus profond que celui du *Tree*. Une fois cette réécriture effectuée, l'équivalence utilisée sur l'exemple précédent peut être appliquée et les opérations frontières réécrites en une projection. D'autres réécritures (voir [97]) nous permettent d'aboutir à la requête optimisée illustrée en partie gauche de la figure 5.9.

L'expression optimisée est certainement plus efficace que la première puisqu'elle correspond à une simple sélection. Cependant, notez que les capacités de filtrage de la source XML

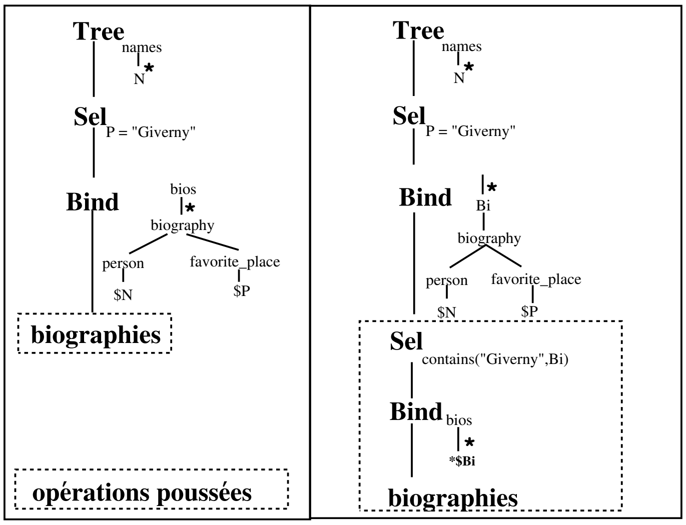

FIG. 5.9: *Deux réécritures de la requête 5.2*

ne sont pas utilisées. Ceci est dû au fait que le prédicat de sélection porte sur un élément non indexé par la source. Pourtant, celle-ci pourrait effectuer une pré-selection en ne retournant que les documents contenant une occurrence de “Giverny” à un endroit quelconque du texte. Pour effectuer cette réécriture, l'optimiseur a besoin de connaître la relation existant entre sélections structurée et plein-texte. Dans YAT, cette relation peut être spécifiée par l'adaptateur de la façon suivante ([**X**] indique la présence de la variable $\boldsymbol{X}$ dans un arbre).

`biographies :` $\boldsymbol{bios} \xrightarrow{\star} \boldsymbol{Pbio}$: $\boldsymbol{Pattern}[\boldsymbol{X}]$ X= Y:String	$\equiv$	`biographies :` $\boldsymbol{bios} \xrightarrow{\star} \boldsymbol{Pbio}$: $\boldsymbol{FPatternt}[\boldsymbol{X}]$ X=Y `contains`(Y, $\boldsymbol{Pbio}$)

En utilisant une représentation interne de cette règle, l'optimiseur est alors capable de réécrire la requête comme illustrée sur la partie droite de la figure 5.9.

A notre connaissance, seul YAT permet, par une analyse statique des informations fournies par les adaptateurs, les optimisations présentées dans cette section. De façon générale, sa capacité à utiliser le plein pouvoir d'expression de sources réellement hétérogènes offre à YAT des opportunités d'optimisation plus importantes que celles des autres médiateurs.

Conclusion

Les travaux présentés dans cette thèse ont en commun une thématique et une méthodologie. Nous nous sommes attachés à comprendre certains des problèmes importants soulevés par l'ouverture des systèmes informatiques vers le réseau. Nous avons préconisé l'utilisation systématique d'une approche déclarative, approche dont les avantages sont qu'elle simplifie grandement la vie des programmeurs et permet de réagir efficacement, de façon dynamique et transparente aux différents changements pouvant intervenir dans la représentation physique des données manipulées. Pour cela, nous nous sommes reposés sur une algèbre que nous avons étendu de différentes façons. Cette algèbre a ainsi montré qu'elle était particulièrement bien adaptée aux nombreux problèmes liés à la manipulation des données, qu'elles soient locales à un système, intégrées ou matérialisées depuis d'autres systèmes.

Un autre point commun à ces travaux, est le plaisir qu'a eu l'auteur à les réaliser en collaboration avec des personnes dont elle a apprécié l'intelligence, le charme, la gentillesse et le sens de l'humour.

XML apparaît maintenant comme un standard incontournable pour l'échange des données sur le Web. De nombreux projets cherchent à concevoir un langage d'interrogation, des techniques de stockage et d'optimisation appropriés. Nos recherches sur OQL et les données documentaires ou semistructurées devraient se révéler utiles dans ce contexte et nous espérons bien contribuer à cette nouvelle problématique. Pour ce faire, nous pensons suivre deux directions : (i) concevoir un langage d'interrogation et (ii) finaliser le système YAT en lui offrant un optimiseur et le support d'une mémoire secondaire appropriée. L'optimiseur devra intégrer les différentes techniques que nous avons présentées dans cette thèse et qui incluent matérialisation efficace, optimisations locale et distribuée sur des requêtes structurées ou semistructurées. De nouvelles techniques devront également être développées pour permettre une bonne évaluation des requêtes, même en l'absence d'information fine sur les types. Maintenant que le nécessaire mais, doit on l'avouer, ennuyeux travail de rédaction de ce document est achevé, nous comptons nous atteler à cette tâche.

Annexe: Algèbre objet

L'algèbre comprend des opérateurs définis sur les listes, ensembles et multi-ensembles. La définition des opérateurs, d'une type de collection à l'autre, change peu. Les équivalences s'appliquent toutes aux opérations ensemblistes. Les multi-ensembles en permettent moins, et les listes encore moins. Nous donnons ici les définitions ensemblistes. Pour des raisons de concision, nous ne distinguons pas une expression algébrique de son évaluation : si elle figure à droite d'une définition, il s'agit de son évaluation. Nous utilisons les notations suivantes :

- e, e_i sont des expressions algébriques,
- p est un prédicat,
- f est une fonction quelconque,
- a et a_i sont des attributs, $\vec{a}$ désigne une liste d'attributs. Si t est un n-uplet, $t.a$ désigne la valeur de l'attribut a de t et $t.\vec{a}$ dénote la valeur de la sous-partie du n-uplet de t contenant les attributs de $\vec{a}$. Les opérations de jointure et de *Map* reposent sur l'unicité de nom des attributs, unicité garantie par l'optimiseur.

Les opérateurs sont les suivants:

Map : L'opérateur Map existe sous trois formes syntaxiques :

1. $e[a] = \{[a : x] | x \in e\}$
2. $Map_f(e) = \{f(t) | t \in e\}$
3. $Map_{a:f}(e) = \{t \circ [a : f(t)] | t \in e\}$

Projection :

$Proj_{\vec{a}}(e) = \{t.\vec{a} | t \in e\}$

La projection est un cas particulier de *Map*.

Jointure :

$e_1 \; Join_p \; e_2 = \{t_1 \circ t_2 \mid t_1 \in e_1, t_2 \in e_2, p(t_1, t_2)\}$ où $\circ$ représente la concaténation de n–uplets.

Sélection :

$Select_p(e) = \{t | t \in e, p(t)\}$

Group :

$$Group_{g;\vec{a}\theta;f}(e) = \{t_1.\vec{a} \circ [g : G] | t_1 \in e, G = \{f(t_2) | t_2 \in e, t_2.\vec{a}\theta t_1.\vec{a}\}\}.$$

Sort :

$$\boldsymbol{Sort_{\vec{a}}(e) = \langle t_1, t_2, ..., t_n \rangle} \text{ tel que}$$
$$\boldsymbol{(\forall i \in [1, n], t_i \in e)}$$
$$\boldsymbol{\wedge(\forall t \in e, \exists i \in [1, n], t_i = t)}$$
$$\boldsymbol{\wedge(\forall i \in [1, n-1]\ t_i.\vec{a} \leq t_{i+1}.\vec{a})}$$

Table des figures

Bibliographie

[1] S. Abiteboul and C. Beeri. The Power of Languages for the Manipulation of Complex Values. *The VLDB Journal*, 4(4), October 1995.

[2] S. Abiteboul, S. Cluet, V. Christophides, T. Milo, G. Moerkotte, and J. Siméon. Querying documents in object databases. *International Journal on Digital Libraries*, 1(1), April 1997.

[3] S. Abiteboul, S. Cluet, and T. Milo. Querying and updating the file. In *Proceedings of International Conference on Very Large Databases (VLDB)*, Dublin, Ireland, August 1993.

[4] S. Abiteboul, S. Cluet, and T. Milo. A database interface for file updates. In *Proceedings of ACM SIGMOD Conference on Management of Data*, San Jose, California, May 1995.

[5] S. Abiteboul, S. Cluet, and T. Milo. Correspondence and Translation for Heterogeneous Data. In *Proceedings of International Conference on Database Theory (ICDT)*, Delphi, Greece, January 1997.

[6] S. Abiteboul, S. Cluet, and T. Milo. A logical view of structured files. *The VLDB Journal*, 7(2), May 1998.

[7] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1), April 1997.

[8] S. Amer-Yahia. *Du Chargement en Masse dans une Base de Données (en Général) et de la Migration Relationnel-Objet (en particulier).* Thèse de doctorat, Université Paris XI, January 1999.

[9] S. Amer-Yahia, S. Cluet, and C. Delobel. Bulk Loading Techniques for Object Databases and an Application to Relational Data. In *Proceedings of International Conference on Very Large Databases (VLDB)*, New York, August 1998.

[10] ANSI. *http://www.objs.com/x3h7/sql3.htm*.

[11] Apertus. *http://www.apertus.com/*.

[12] The o_2 database system.
http://www.ardentsoftware.fr/.

[13] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented system manifesto. In *Proc. DOOD, Kyoto, Japan*, December 1989.

[14] F. Bancilhon, S. Cluet, and C. Delobel. Query languages for object-oriented database systems: the O2 proposal. In Morgan Kaufmann, editor, *Proceedings of International Workshop on Database Programming Languages*, Salishan Lodge, Oregon, June 1989.

[15] C. Beeri and Y. Kornatzky. Algebraic optimization of object-oriented query languages. In *Proceedings of International Conference on Database Theory (ICDT)*, Paris, France, 1990.

[16] C. Beeri and T. Milo. Schemas for Integration and Translation of Structured and Semi-Structured Data. In *Proceedings of International Conference on Database Theory (ICDT)*, LNCS, Jerusalem, Israel, January 1999.

[17] AIS Berger-Levrault. SGML/Search, Description du Langage. Internal Document, 34 Avenue du Roule 92200 Neuilly sur Seine, 1993.

[18] E. Bertino, F. Rabitti, and S. Gibbs. Query Processing in a Multimedia Document System. *ACM TOIS*, 6(1), January 1988.

[19] G. Blake, M. Consens, P. Kilpeläinen, P. Larson, T. Snider, and F. Tompa. Text / relational database management systems: Harmonizing SQL and SGML. In W. Litwin and T. Risch, editors, *Applications of Databases: First International Conference*, pages 267–280. Vadstena, Sweden, 1994.

[20] P. Buneman, S. B. Davidson, M. Fernandez, and D. Suciu. Adding Structure to Unstructured Data. In *Proceedings of International Conference on Database Theory (ICDT)*, Delphi, Greece, January 1997.

[21] P. Buneman, S. B. Davidson, K. Hart, C. Overton, and L. Wong. A data transformation system for biological data sources. In *Proceedings of International Conference on Very Large Databases (VLDB)*, Zurich, Switzerland, September 1995.

[22] P. Buneman, S. B. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of ACM SIGMOD Conference on Management of Data*, Montreal, Canada, June 1996.

[23] P. Buneman, S. B. Davidson, and D. Suciu. Programming constructs for unstructured data. In *Proceedings of International Workshop on Database Programming Languages*, Gubbio, Italy, 1995.

[24] F. J. Burkowski. The Use of Retrieval Filters to Localize Information in a Hierachically Tagged Text-Dominated Database. In *RIAO'91*, Barcelona, Spain, April 1991.

[25] M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. F. Cody, R. Fagin, M. Flickner, A. W. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J. H. Williams, and E. L. Wimmers. Towards heterogeneous multimedia information systems: The garlic approach. In *Research Issues in Data Engineering*, Los Alamitos, California, March 1995.

[26] R. G. Cattell. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, 1997.

[27] V. Christophides. *Document Structurés et Bases de Données Objet.* Thèse de doctorat, Conservatoire National des Arts et Métiers, October 1996.

[28] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *Proceedings of ACM SIGMOD Conference on Management of Data*, Minneapolis, Minnesota, May 1994.

[29] V. Christophides, S. Cluet, and G. Moerkotte. Evaluating queries with generalized path expressions. In *Proceedings of ACM SIGMOD Conference on Management of Data*, Montreal, Canada, June 1996.

[30] V. Christophides, S. Cluet, G. Moerkotte, and J. Siméon. Un cocktail de bases de données et d'index plein-texte. In *Congrès INFORSID*, Bordeaux, France, June 1996. Journées O_2 Technology.

[31] V. Christophides, S. Cluet, G. Moerkotte, and J. Siméon. Optimizing Generalized Path Expressions Using Full Text Indexes. In *Proceedings of 13es Journées Bases de Données Avancées*, Grenoble, France, September 1997.

[32] S. Cluet. *Langages et Optimisation de Requêtes pour SGBDOO*. Thèse de doctorat, Université Paris XI, June 1991.

[33] S. Cluet. Modeling and querying semi-structured data. In *International Summer School, SCIE-97*, volume 1299 of *Lecture Notes in Artificial Intelligence*. Springer Verlag, 1997.

[34] S. Cluet. Designing OQL: Allowing Objects to be Queried. *Information Systems*, 23(5), 1998.

[35] S. Cluet and C. Delobel. A General Framework for the Optimization of Object-Oriented Queries. In *Proceedings of ACM SIGMOD Conference on Management of Data*, San Diego, California, May 1992.

[36] S. Cluet, C. Delobel, C. Lécluse, and P. Richard. An Algebra Based Query Language for an Object-Oriented Database System. In *Proceedings International Conference on Deductive and Object-Oriented Databases (DOOD)*, Lecture Notes in Computer Science. Springer-Verlag, Kyoto, Japan, December 1989.

[37] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your Mediators Need Data Conversion! In *Proceedings of ACM SIGMOD Conference on Management of Data*, Seattle, Washington, June 1998.

[38] S. Cluet and G. Moerkotte. Nested queries in object bases. In *Proceedings of International Workshop on Database Programming Languages*, New York City, USA, August 1993.

[39] S. Cluet and G. Moerkotte. Efficient Evaluation of Aggregates on Bulk Types. In *Proceedings of International Workshop on Database Programming Languages*, LNCS, Gubbio, Italy, September 1995. Springer-Verlag.

[40] S. Cluet and G. Moerkotte. On the Complexity of Generating Optimal Left-Deep Processing Trees with Cross-Products. In *Proceedings of International Conference on Database Theory (ICDT)*, Pragua, Rcheque Republic, January 1995.

[41] Quelques systèmes d'intégration commerciaux.
http://www.software.ibm.com/data/datajoiner/,
http://www.sybase.com/products/entcon/,
http://www.oracle.com/products/servers/rdb/html/fs_dbi.html,
http://www.unisql.com/product_info/unisqlm.html,
http://www.cincom.com/totalframework/index.html,
http://www.enterworks.com/products.html,
http://www.cerfnet.com/~margie/dii/,
http://www.junglee.com/,
http://www.crossaccess.com/product.htm,
http://www.ibi.com/.

[42] Computer Associate. *http://www.cai.com/*.

[43] S. Daniels, G. Graefe, T. Keller, D. Maier, D. Schmidt, and B. Vance. Query optimization in revelation, an overview. *Data Engineering Bulletin*, 14(2), 1991.

[44] S. Davidson and A.S.Kosky. WOL: A Language for Database Transformations and Constraints. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, pages 55–65, UK, April 1997.

[45] G. Lohman et al. Optimization of nested queries in a distributed relational database. In *Proceedings of International Conference on Very Large Databases (VLDB)*, 1984.

[46] F. Exertier. ROBIN: Generating Object-Oriented and WEB Interfaces for Relational Databases. Technical report, Bull, January 1997. White paper.

[47] C. Fahrner and G. Vossen. Transforming Relational Database Schemas into Object-Oriented Schemas acoording to ODMG-93. In *Proceedings International Conference on Deductive and Object-Oriented Databases (DOOD)*, Singapore, December 1995.

[48] L. Fegaras and D. Maier. Towards an effective calculus for object query languages. In *Proceedings of ACM SIGMOD Conference on Management of Data*, San Jose, California, May 1995.

[49] M. Fernandez, D. Florescu, A. Y. Levy, and S. Suciu. A query-language and processor for a web site management system. In *Workshop on Management of Semistructured Data*, Tucson, Arizona, May 1997. In conjunction with PODS/SIGMOD.

[50] M. Fernandez, D. Florescu, A. Y. Levy, and S. Suciu. Warehousing and incremental evaluation for web site management. In *Proceedings of 14es Journées Bases de Données Avancées*, Hammamet, Tunisie, October 1998.

[51] Daniela Florescu. *Search Spaces for Object-Oriented Query Optimization.* Thèse de doctorat, Université Paris VI, 1996.

[52] R. Ganski and H. Wong. Optimization of nested SQL queries revisited. In *Proceedings of ACM SIGMOD Conference on Management of Data*, 1987.

[53] G. Gardarin, S. Gannouni, B. Finance, P. Fankhauser, W. Klas, D. Pastre, R. Legoff, and A. Ramfos. IRO-DB : A Distributed System Federating Object and Relational Databases. In *Object Oriented Multibase Systems : A Solution for Advanced Applications*. Prentice Hall, 1995.

[54] G. Gardarin, F. Machuca, and P. Pucheral. OFL: A functional execution model for object query languages. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 59–70, San Jose, CA, 1995.

[55] Gemstone. `http://www.gemstone.com`.

[56] G. H. Gonnet and F. W. Tompa. Mind Your Grammar: a New Approach to Modeling Text. In *VLDB*, Brighton, 1987.

[57] G. Guerrini, E. Bertino, B. Catania, and J. Garcia-Molina. A formal view of object-oriented database systems. *Theory and Practice of Object Sytems*, 3(3), 1997.

[58] M. Gyssens, J.Paredaens, and D. V. Gucht. A Graph-Oriented Object Database Model. In *Proceedings of ACM Symposium on Principles of Database Systems*, Nashville Tennesee, April 1990. ACM.

[59] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proceedings of International Conference on Very Large Databases (VLDB)*, Athens, Greece, August 1997.

[60] Jean-Luc Hainaut. Database Reverse Engineering - Problems, Techniques and CASE tools. In *Conference on Advanced Inf. Syst. Engineering*, Jvaskla, Finland, June 1995. Tutorial.

[61] J. Hammer, H. Garcia-Molina, S. Nestorov, R. Yerneni, M. Breuning, and V. Vassalos. Template-based wrappers in the tsimmis system. In *Workshop on Management of Semistructured Data In conjunction with PODS/SIGMOD*, Tucson, Arizona, May 1997.

[62] J. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of ACM SIGMOD Conference on Management of Data*, 1993.

[63] Laurent Herr. O2yacc. Technical report, Projet Verso, INRIA, 1994.

[64] Persistence Software INC. Bridging the Gap between Relational Data and Object Oriented Development. Technical report, Persistence Software INC., September 1993.

[65] INRIA. `http://cosmos.inria.fr:8080/o2_list-bases-inria.html/`.

[66] ISO. Information processing – text and office systems – standard generalized markup language (sgml), 1986. ISO 8879:1986(E), ISO/IEC 10744.

[67] J. Jahnke, W. Schäfer, and A. Züendorf. A Design Environment for the Migration of Relational to Object-Oriented Database Systems. In *ICSM*. IEEE Computer Society, 1996.

[68] JavaSoft. `http://www.javasoft.com/`.

[69] P. Jenq, D. Woelk, W. Kim, and W. Lee. Query processing in distributed ORION. In *Proceedings of the International Conference on Extending Database Technology*, Venice, Italia, 1990.

[70] A. Kemper and G. Moerkotte. Query optimization in object bases: Exploiting relational techniques. In G. Vossen J. C. Freytag and D. Maier, editors, *Proceedings of Dagstuhl Workshop on Query Processing in Object-Oriented, Complex-Object and Nested Relation Databases*. Morgan Kaufmann, 1994.

[71] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proceedings of ACM SIGMOD Conference on Management of Data*, San Diego, California, June 1992.

[72] W. Kim. On optimizing an SQL-like nested query. *ACM Transactions on Database Systems*, 7(3), Sep 82.

[73] Franck Lebastard. *DRIVER: Une couche Objet Virtuelle Persistante pour le Raisonnement sur les Bases de Données Relationnelles.* PhD thesis, Institut National des Sciences Appliquées, Lyon, France, March 1993.

[74] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of International Conference on Very Large Databases (VLDB)*, Bombay, India, September 1996.

[75] L. Liu, C. Pu, and Y. Lee. An adaptive approach to query mediation across heterogeneous information sources. In *Proceedings of International Conference on on Cooperative Information Systems (CoopIS)*, Brussels, Belgium, June 1996.

[76] I. A. Macleod. SEQUEL as a Language for Document Retrieval. *American Society for Information Science*, 30(9), September 1979.

[77] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3), September 1997.

[78] J. Melton. *ISO-ANSI SQL2.* International Standard Organization and American National Standards Institute, 1988.

[79] J. Melton and A. Simon. *Understanding the New SQL: A complete Guide.* Morgan Kaufmann, 1993.

[80] D. Michie. Memo Functions and Machine Learning. *Nature*, (218), 1968. pp 19-22.

[81] T. Milo and S. Zohar. Using Schema Matching to Simplify Heterogeneous Data Translation. In *Proceedings of International Conference on Very Large Databases (VLDB)*, New York, August 1998.

[82] R. Missaoui, J.M. Gagnon, and R. Godin. Mapping an Extended Entity-Relationship Schema into a Schema of Complex Objects. In *Object Oriented and Entity Relationship*, Australia, December 1995.

[83] M. Muralikrishna. Optimization and dataflow algorithms for nested tree queries. In *Proceedings of International Conference on Very Large Databases (VLDB)*, 1989.

[84] H. Naacke, G. Gardarin, and A. Tomasic. Leveraging mediator cost models with heterogeneous data sources. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, Orlando, Florida, February 1998.

[85] O2Technology. *O2DBaccess User Manual Release 4.6*, September 1996.

[86] Objectivity. `http://www.objectivity.com/`.

[87] Objectstore. `http://www.odi.com`.

[88] Ontos. `http://www.ontos.com/`.

[89] Oracle. *Oracle 7 Server Utilities*, 1996. Release 7.3.

[90] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1991.

[91] Hewlett Packard. OpenODB: Facilitating Change. Technical report, Hewlett Packard, September 1991.

[92] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, Taipei, Taiwan, March 1995.

[93] Poet. `http://www.poet.com/`.

[94] M. T. Roth and P. M. Schwarz. Don't scrap it, wrap it! a wrapper architecture for legacy data sources. In *Proceedings of International Conference on Very Large Databases (VLDB)*, Athens, Greece, August 1997.

[95] D. Rozen and D. Shasha. Rationale and design of bulk. In Morgan Kaufmann, editor, *Proceedings of International Workshop on Database Programming Languages*, Naphlion, Greece, June 1991.

[96] G. M. Shaw and S. B. Zdonik. An Object-Oriented Query Algebra. In Morgan Kaufmann, editor, *Proceedings of International Workshop on Database Programming Languages*, Gleneden Beach, Oregon, June 1989.

[97] J. Siméon. *Intégration de sources de données hétérogènes (Ou comment marier simplicité et efficacité)*. PhD thesis, Université de Paris XI, January 1999.

[98] J. Siméon and S. Cluet. Using YAT to Build a Web Server. In *International Workshop on the Web and Databases (WebDB'98)*, Valencia, Spain, March 1998.

[99] C. Souza dos Santos, S. Abiteboul, and C. Delobel. Virtual schemas and bases. In *Proceedings of the International Conference on Extending Database Technology*, Cambridge, March 1994.

[100] H. J. Steenhagen, P. M. G. Apers, and H. M. Blanken. Optimization of nested queries in a complex object model. In *Proceedings of the International Conference on Extending Database Technology*, 1994.

[101] D. Straube and T. Özsu. Queries and query processing in object-oriented database systems. *ACM Trans. on Information Systems*, 8(4), 1990.

[102] A. Tomasic, L. Raschid, and P. Valduriez. Scaling heterogeneous databases and the design of disco. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, Hong Kong, May 1996.

[103] J. D. Ullman. *Database and Knowledge base systems.* Computer Science Press, 1988.

[104] J. Van den Bussche and E. Waller. Methods and views. In *Proceedings of 14es Journées Bases de Données Avancées (BDA'98)*, October 1998.

[105] S. L. Vandenberg and D. DeWitt. Algebraic support for complex objects with arrays, identity, and inheritance. In *Proceedings of ACM SIGMOD Conference on Management of Data*, 1991.

[106] Versant. `http://www.versant.com/`.

[107] J. L. Wiener and J. F. Naughton. OODB Bulk Loading Revisited: The Partitioned-List Approach. In *Proceedings of International Conference on Very Large Databases (VLDB)*, Zurich, Switzerland, 1995.

[108] J.L. Wiener. *Algorithms for Loading Object Databases.* PhD thesis, University of Wisconsin - Madison, 1995.

[109] W.J.Labio, J.L.Wiener, and H.Garcia-Molina. Algorithms for resuming interrupted warehouse loads. Technical report, Stanford University, USA, February 1998.

[110] L. L. Yan, M. T. Özsu, and L. Liu. Accessing heterogeneous data through homogenization and integration mediators. In *Proceedings of International Conference on on Cooperative Information Systems (CoopIS)*, Charleston, South Carolina, June 1997.

[111] T. Yan and J. Annevelink. Integrating a structured-text retrieval system with an object-oriented database system. In *Proceedings of International Conference on Very Large Databases (VLDB)*, Santiago, Chile, September 1994.

[112] L. Yu and S. L. Osborn. An evaluation framework for algebraic object-oriented query models. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, 1991.