N^o d'ordre :

THÈSE

présentée à

Université Paris XI

pour obtenir le grade de

DOCTEUR EN SCIENCES

Spécialité

INFORMATIQUE

par

Sihem AMER-YAHIA

Sujet de la thèse :

Du Chargement en Masse dans une Base de Données (en Général) et de la Migration Relationnel-Objet (en Particulier)

Soutenue le 06 Janvier 1999 devant le jury composé de :

M.	Claude DELOBEL	Directeur
Mme.	Geneviève JOMIER	Rapporteurs
M.	C. MOHAN	
Mme.	Sophie CLUET	Examinateurs
Mme.	Marie-Christine ROUSSET	
M.	Serge ABITEBOUL	
M.	Laurent VIEILLE	

À mes parents qui m'ont tellement manqué ...

Aux femmes de mon pays ...

Remerciements

Je tiens tout particulièrement à remercier Sophie Cluet pour m'avoir fait partager son enthousiasme pour la recherche. J'ai beaucoup appris en travaillant avec elle. Elle a lu et relu mon manuscrit en entier. Elle a cru en moi et n'a pas cessé de m'encourager. Je ne pourrais la remercier assez pour tout ce qu'elle m'a apporté. Je suis très contente qu'elle fasse partie de mon jury.

J'aimerais remercier Claude Delobel d'avoir accepté de me suivre durant ma thèse et pour les nombreuses et longues discussions que nous avons eues ensemble. Il me fait l'honneur de faire partie de mon jury.

J'aimerais également remercier Serge Abiteboul pour m'avoir acceptée dans son équipe et m'avoir soutenue dans mes investigations après-thèse. Mon jury aurait été orphelin sans lui.

Je remercie Geneviève Jomier et C. Mohan pour avoir accepté d'être mes rapporteurs. Je remercie C. Mohan pour avoir suivi très attentivement mon exposé et ma démonstration.

Je remercie Marie-Christine Rousset. Elle me fait l'honneur de faire partie de mon jury.

Je remercie Bjorn Bergstein et Laurent Vieille de Next Step Century Media pour avoir discuté avec moi en début de thèse et apporté un avis industriel sur le sujet. Laurent Vieille a accepté de faire partie de mon jury. J'en suis très touchée.

Je remercie Michel Scholl pour sa gentillesse et son amitié. Il a pris le temps de lire mon introduction.

Je remercie Peter Bunemann de l'Université de Philadelphie, Guy Ferran et Jean-Claude Mamou d'Ardent Software, Dave Maier de l'Université d'Oregon, Tova Milo de l'Université de TelAviv et Alberto Mendelson de l'Université de Toronto pour toutes les critiques et suggestions qu'ils m'ont apportées lors de nos discussions.

J'aimerais aussi remercier Ségolène Galabert avec laquelle j'ai eu le plaisir d'optimiser.

Je remercie l'équipe du LRI, en particulier Emmanuel Waller et Nicolas Spyratos qui m'ont invité à exposer mon travail à l'Université d'Orsay.

J'ai une pensée particulière pour Danny pour son aide et son soutien de tous les jours. Elle a lu ma thèse. Elle a été là à chaque fois que j'avais besoin d'elle.

Je veux également remercier toute l'équipe Verso : Bernd, Irini, Stéphane, Laurent pour avoir lu et relu mon introduction et pas que ça, Luc, Pierreangelo, Anne-Marie et Fanny. Ainsi que les visiteurs Verso : Leonid Libkin de Bell-Labs, Rodney Topor de l'Université de Griffith qui a lu mon introduction et Victor Vianu qui me fait l'honneur d'assister à ma thèse. Toute l'équipe va me manquer.

Je remercie Patrick Valduriez et Éric Simon de l'équipe Rodin. Ils m'ont aidé dans mes investigations après-thèse. Je tiens à remercier Mokrane Amzal pour ses commentaires et Khaled Yagoub pour son aide à mes débuts avec java.

Je remercie mes parents et ma sœur Amel d'être là. Sans eux je n'en serai pas là.

Je pense très fort à Mouni, Lotfi et Sekoura.

Je ne trouve pas de mots pour remercier Jérôme.

Table des matières

Introduction

La démocratisation de l'informatique accroît considérablement le nombre d'utilisateurs accédant aux mêmes données et favorise leur distribution. Les systèmes conçus au départ pour de simples applications de gestion, doivent aujourd'hui s'adapter rapidement à cette évolution. Pour répondre à des besoins d'efficacité, certains d'entre eux dupliquent les données sur des systèmes équivalents pour avoir des temps de réponse plus intéressants. Dans ce cas, les données sont soit importées telles quelles dans le système cible (construction de sites internet «miroirs») soit restructurées avant de les importer (entrepôts de données, vues). D'autres systèmes migrent les données sources vers un système cible offrant une technologie plus avancée (migration relationnel-objet, migration de fichiers contenant des données du génome humain dans une base de données pour bénéficier de fonctionnalités telles que la concurrence d'accès, l'organisation et le partage efficace des données, leur interrogation à travers un langage de requête déclaratif).

Les «aspirateurs de sites internet» tels que HTTrack [RP] permettent de construire et de maintenir une image d'un ou plusieurs sites existants en transférant les données qui les constituent vers un autre afin d'être exploitées localement. Les entrepôts de données [HMWJZ98] et plus généralement, la matérialisation de vues, nécessitent aussi de migrer régulièrement des quantités de données qui peuvent être très importantes pour construire et maintenir l'entrepôt. Par exemple, Sagent Technology construit des entrepôts de plus de 100 gigas octets de données et l'entrepôt de Walmart contient 24 téras octets de données [WJHM98]. Une des préoccupations du séquençage du génome humain [Pea91] est le chargement du contenu de fichiers existants dans d'autres fichiers ou dans une base de données pour que les données soient interrogées et exploitées par une communauté d'utilisateurs plus large. La quantité des données manipulées devenant de plus en plus importante, leur duplication et leur migration deviennent de plus en plus complexes. Nous parlerons de chargement en masse.

Jusqu'à présent, le chargement de données d'un système à l'autre était réalisé de deux façons : par l'écriture de programmes en utilisant les interfaces des systèmes source et cible ou par l'utilisation d'un outil de chargement existant [Ora, Fa87, Aa93, INC93, Ont96]. L'écriture de programmes de chargement est, d'une part un travail long et fastidieux et, d'autre part difficile sinon impossible à réutiliser du fait que les programmes sont adaptés à des types de données source et cible particuliers. En offrant un langage déclaratif pour l'expression du chargement, l'utilisateur pourra se contenter de spécifier les transformations source-cible et le système générera pour lui les programmes correspondants. Se limiter aux outils existants n'est pas non plus une solution : ils se comportent comme une boîte noire, ne laissant souvent aucune latitude à leurs utilisateurs (dans le sens où ils génèrent des programmes de chargement par défaut) et n'implantant pas de techniques d'optimisation (certains systèmes comme SQL*Loader d'Oracle [Ora96], permettent de faire du chargement parallèle mais c'est à l'utilisateur de déterminer les programmes de chargement à paralléliser). Notons également que l'interruption d'un chargement nécessite de le reprendre à zéro si aucun point de reprise n'est défini. Ce processus étant souvent long (du fait du volume important des données), les risques d'interruption se voient multipliés. Un autre point clé est alors la décomposition du chargement en parties distinctes pouvant être exécutées à des moments différents et pour une durée limitée.

Actuellement, aucun outil de chargement ne permet de concilier déclarativité pour libérer l'utilisateur des détails d'implantation et mettre en œuvre des techniques d'optimisation de façon transparente, efficacité pour optimiser l'écriture des programmes, leur évaluation et la reprise sur panne et flexibilité pour donner plus de contrôle à l'utilisateur tant du point de vue de l'organisation de la base cible que du processus de chargement lui-même.

La présente thèse porte sur la conception et la validation de techniques d'optimisation adaptées au chargement en masse dans une base relationnelle ou à objets. Elle définit un cadre général pour s'abstraire des particularités des systèmes sous-jacents. Elle propose une approche qui s'inspire de l'utilisation des langages de requête et de leur optimisation. Elle se base sur de nouvelles méthodes d'optimisation pour assurer un chargement en masse efficace. Le principe est de permettre à l'utilisateur l'expression déclarative du chargement et la spécification de contraintes d'organisation physique des données de manière flexible. La réalisation de ces contraintes lors du peuplement de la base cible évite sa réorganisation ultérieure en vue d'optimiser les applications. Notre approche définit aussi le moyen de partitionner «intelligemment» le chargement en transactions optimisées et adaptées aux spécifications de l'utilisateur.
Notre travail s'articule en deux parties. La première s'attache à l'étude des problèmes de chargement de données en masse en général et la définition de techniques d'optimisation appropriées [AYCD98]. La deuxième porte sur la validation de nos idées par le développement d'un prototype de migration de données relationnelles dans un système à objets [AY97].

Du chargement de données en masse

Dans cette partie, nous proposons un langage de haut niveau adapté au chargement de différents types de données et des techniques d'optimisation.
Les données sources peuvent provenir de fichiers ayant une grammaire simple, d'une base de données relationnelle ou encore d'une base de données à objets. Un chargement consiste à exporter les données sources, les restructurer et enfin les importer dans le système cible. Pour être à même de réaliser ces trois étapes sur différents types de données, nous avons choisi un modèle objet appelé IQL [AK89] (Identity Query Language) et une algèbre adaptée à ce dernier [CM94] pour l'expression du chargement.

Le modèle IQL permet d'exprimer les modèles de type relationnel, objet ainsi que les fichiers texte ayant une grammaire simple. Nous pouvons alors représenter des objets, valeurs, classes, relations, schémas et bases. Nous avons simplement étendu le modèle avec la notion de liste ordonnée.
L'algèbre peut être vue comme un sur-ensemble de l'algèbre relationnelle. Elle contient des

opérateurs qui peuvent être compris par un système à objets. Les programmes de chargement sont exprimés sous forme d'expressions algébriques. La recherche des expressions les plus efficaces du point de vue de l'exécution utilise des règles de réécriture. Nous retrouvons les règles bien connues dans l'algèbre relationnelle comme pousser des sélections et réordonner des jointures. Toutefois, le cadre de peuplement d'une base de données étant plus général, nous avons défini une nouvelle classe d'équivalences algébriques qui tiennent compte des modifications sur la base cible (effets de bord). Les nouvelles équivalences reposent sur deux propriétés principales des fonctions d'effet de bord : idempotence et insensibilité à la valeur nulle. Ainsi, nos techniques d'optimisation peuvent être utilisées pour optimiser tout programme ou ensemble de programmes dont l'objectif est de peupler une base (relationnelle ou à objets) à condition de vérifier ces deux propriétés.

De la migration relationnel-objet

Dans la première partie, nous avons développé des techniques algébriques qui permettent d'optimiser les programmes de chargement, de les découper en transactions et d'assurer une organisation des données qui optimise l'accès des applications à la base cible. Afin de valider ces idées, nous avons développé un prototype de migration de données d'une base relationnelle vers une base à objets. Ce prototype propose un langage de transformation (création d'un schéma et d'une base à objets à partir d'un schéma et d'une base relationnels) [AYCD98]. Le langage est conçu au dessus de l'algèbre et permet d'exprimer la migration de manière déclarative et conviviale. Les primitives du langage sont traduites dans l'algèbre pour optimiser les programmes de chargement. Notre prototype implante une stratégie de recherche qui applique des réécritures algébriques. La recherche de la meilleure solution utilise un modèle de coût pour évaluer chaque ensemble de programmes. Une fois les programmes optimaux générés, ils peuvent être exécutés dans un environnement complet de migration pour peupler la base à objets cible.

Le langage d'expression des transformations relationnel-objet que nous proposons est déclaratif, simple et convivial. Il offre les possibilités de restructuration d'un langage de vues [AB91, SLT91, Ber91, Run92, DdST95] et permet également d'exprimer des contraintes physiques sur la base cible. C'est ce qui fait son originalité puisque l'utilisateur peut ainsi spécifier, de manière déclarative, l'organisation des données sur disque. Nous montrons comment les primitives du langage sont traduites dans l'algèbre pour générer une représentation algébrique correspondant à une évaluation possible du chargement.

La phase d'optimisation prend en entrée une représentation algébrique pour en générer une représentation optimisée. Nous avons développé une stratégie de recherche qui utilise un modèle de coût pour le choix de la meilleure solution, i.e. celle qui réalise les contraintes physiques et offre le compromis le plus satisfaisant entre les paramètres du modèle de coût (temps et espace). L'intérêt de notre stratégie est qu'elle est générique et peut être facilement adaptée au chargement d'autres types de données. La spécificité des systèmes sous-jacents est codée dans le modèle de coût. Celui-ci estime le coût d'un chargement en fonction des temps de traitement sur les systèmes source et cible, du temps de communication entre ces systèmes et du volume de stockage que nécessitent les traitements intermédiaires. Ainsi, l'approche que nous proposons permet non seulement d'utiliser les techniques d'optimisation des systèmes distribués mais aussi d'exploiter le fait que l'on peuple une base de données.

Une fois le chargement optimisé, le système génère les programmes correspondants. Un programme est une transaction paramétrée par le temps ou le nombre d'objets à créer. Il est capable d'interroger la base relationnelle source, de restructurer le résultat et de peupler tout ou partie de la base à objets cible.
Les programmes générés peuvent ensuite être lancés dans un environnement de contrôle qui gère leur enchaînement. Ils peuvent être exécutés à des moments choisis par l'utilisateur et pour une période donnée, ce qui se traduit par un gain de temps appréciable en cas de reprise à la suite d'une défaillance. Le système offre ainsi le moyen d'effectuer une migration incrémentale.
Nous avons effectué des tests sur notre prototype qui nous ont permis de valider l'approche adoptée et de tester les performances des programmes générés sur le chargement de données dont le volume est important. Ces tests ont également servi à valider et «raffiner» le modèle de coût.

Organisation de la thèse

Nous commençons, dans le chapitre 1, par présenter un état de l'art portant sur les deux aspects du chargement : l'expression des transformations source-cible et le peuplement effectif de la base de données cible. Nous situons nos contributions dans chacun de ces deux domaines.
Dans la première partie de cette thèse, nous suggérons des solutions générales à l'optimisation du chargement en masse. Nous décrivons le modèle de données que nous avons choisi (chapitre 2), le langage algébrique d'expression des transformations source-cible (chapitre 3) et enfin les techniques d'optimisation développées (chapitre 4).
Dans la deuxième partie, nous présentons le langage *RelOO* d'expression des transformations et des contraintes physiques (chapitre 5). Les techniques d'optimisation adaptées au cas relationnel-objet sont données dans le chapitre 6 à travers une discussion des paramètres qui sont pris en considération dans le modèle de coût, pour appliquer les équivalences algébriques. Ce chapitre se termine par plusieurs exemples qui illustrent quelques possibilités de réécriture. Le chapitre 7 détaille l'architecture de notre prototype. Il montre comment procède la phase d'optimisation (stratégie de recherche et modèle de coût). La phase de génération de code et le fonctionnement de l'environnement de contrôle du chargement terminent ce chapitre. Quelques résultats expérimentaux sont présentés dans le chapitre 8. Nous terminons par rappeler les contributions apportées par notre travail et les différentes perspectives envisagées.

Chapitre 1

État de l'art et motivations

Le nombre d'applications requérant, de façon occasionnelle ou répétitive, le chargement en masse de données va croissant : entrepôts de données, entreprises virtuelles, gestion de documents, ... Or, il existe peu d'outils dédiés à cette tâche. À notre connaissance, aucun SGBD à objets commercial n'offre des fonctions de chargement adaptées. Des techniques de chargement ont été développées et mises en œuvre pour certains systèmes relationnels mais, de ces logiciels, aucun n'a la convivialité et n'offre les performances que l'on pourrait attendre d'une industrie dont la spécialité est le traitement efficace de grands volumes de données. Il est temps d'appréhender le chargement de données en masse d'un point de vue bases de données et d'offrir enfin aux utilisateurs indépendance logique/physique et efficacité.

Le chargement en masse soulève deux problèmes : (i) la spécification des transformations source-cible, et (ii) le peuplement de la base de données cible le plus rapidement possible. Nous étudions l'état de l'art suivant ces deux volets. Dans le premier volet, nous verrons que des travaux aussi variés que les vues, l'intégration de données hétérogènes, la conversion de formats et la réingénierie nous ont inspiré dans notre choix d'un modèle de données assez général pour représenter aussi bien les données sources et cibles que les correspondances et transformations entre ces données. Dans le deuxième volet, nous verrons que, contrairement à l'expression des transformations source-cible, il n'existe quasiment pas de travaux portant sur l'optimisation du chargement en masse.

1.1 Spécification des transformations

Nous pouvons distinguer deux types d'approches qui traitent de l'expression de transformations source-cible : des solutions spécifiques à des types de données ou à des systèmes particuliers et des solutions plus générales dans lesquelles des modèles permettant de représenter tout type de données sont proposés.

1.1.1 Approches spécifiques

Cette catégorie regroupe les travaux qui ont concerné la rétro-conception de données et d'applications existantes et certains systèmes de chargement qui existent dans l'industrie.

La communauté scientifique a principalement mis l'accent sur les problèmes de réingénierie et a fourni de nombreuses solutions à la génération automatique d'un schéma cible à partir d'un schéma source. Plusieurs travaux ont porté sur la rétro-conception d'une base de données relationnelle pour en extraire un schéma conceptuel Entité-Relation étendu [Pet96, CBS94, JK89]. D'autres travaux tels que [Hai95a] se sont intéressés à l'extraction de structures relationnelles à partir de programmes COBOL, ou d'applications SQL [VA95]. Enfin, des exemples comme [Leb93, FV95, KFL95, Hai95b, MGG95, MR96, JSZ96, RH96, RH97, DA97, Fon97, BGD97, IUT98], montrent l'intérêt porté sur l'inférence d'un schéma orienté-objet à partir de données provenant d'une base relationnelle. Dans ce cas, lorsque le schéma relationnel n'est pas connu à l'avance, une phase de réingénierie extrait la structure des données sources. La conversion source-cible se fait ensuite en utilisant un ensemble de règles de correspondance prédéfinies.

Toutes les approches de réingénierie existantes ont un point commun, elles définissent un ensemble minimal de transformations source-cible qui concernent un type de données particulier. L'inconvénient majeur de ces approches est qu'elles ne sont pas flexibles. L'utilisateur n'intervient dans le processus de transformation que pour apporter une information sémantique qui aidera le système à appliquer une tranformation ou une autre. À aucun moment, il ne choisit les transformations à appliquer et ne peut pas en créer de nouvelles.

La phase d'extraction des informations du schéma source peut être très longue. Dans le cas particulier du relationnel-objet, Premerlani et Baha [PB94], illustrent la difficulté de capturer toute l'information source nécessaire pour générer «un schéma objet fidèle au schéma relationnel». Cette difficulté est accrue par le fait que, dans la réalité, le schéma relationnel ne respecte pas toujours les contraintes de normalité des relations qui le constituent. Dans la suite, nous expliquons comment se passe en général une rétro-conception de données relationnelles :

1. La première étape consiste à récupérer les méta-données relationnelles soit en analysant le code source de création du schéma relationnel soit en exécutant des requêtes SQL pour l'interrogation de ce schéma. Cette étape répertorie les relations, attributs, clés primaires et étrangères du schéma relationnel.

2. La deuxième étape est la description sémantique du schéma relationnel. Elle consiste à extraire des informations supplémentaires des entités relationnelles obtenues dans l'étape précédente. Des représentations fondées sur des extensions du modèle Entité-Relation, par exemple, le *Schema Intension Graph* (SIG) défini dans [MIR93], l'approche *Partial Universal Relation* (PUR) [FP] ou encore le modèle sémantique structurel dans le système Penguin [Kel94], sont utilisés pour capturer un schéma relationnel avec quelques nouvelles dépendances fonctionnelles et d'inclusion ainsi que des propriétés d'héritage.

3. La troisième étape est la phase d'enrichissement et de validation du schéma qui résulte de la phase précédente. L'objectif principal de cette phase est d'assurer que le nouveau schéma ne réduit pas la «capacité d'information» du schéma initial. Elle se distingue par l'utilisation d'heuristiques comme dans [CBS94, RH97] ou en analysant les données de la base [JK89, MR96] pour extraire les dépendances d'inclusion. L'utilisateur peut valider et compléter certaines propriétés obtenues.

4. La dernière étape est l'application des transformations relationnel-cible, la cible pouvant être un schéma relationnel étendu ou un schéma objet. Cette étape décrit non seulement les transformations entre les entités du modèle relationnel source et le modèle cible mais induit aussi une transformation sur les données. Elle peut être entièrement automatisée. Les transformations appliquées correspondent à une base de règles. Un exemple complet peut être trouvé dans [FV95] dans lequel les schémas objet générés sont des schémas compatibles ODMG-93 [ABD$^+$94].

Dans l'industrie, certains systèmes sont utilisés comme systèmes pivots entre un langage de programmation objet et une base de données relationnelle. Ceci nécessite de définir des correspondances entre les concepts manipulés dans les deux mondes. L'OIS (Object Integration Server) d'Ontos par exemple, donne la possibilité au développeur d'exécuter des transformations relationnel-objet interactivement en sélectionnant une transformation possible parmi celles proposées [Ont96]. Pegasus [Aa93] aussi fournit un mécanisme de traduction d'un schéma relationnel vers un schéma de base de données Iris [Fa87]. L'inconvénient majeur de l'approche adoptée est que le schéma objet cible est vu comme un schéma relationnel ce qui limite considérablement les capacités de modélisation. Plus récemment, des produits comme Persistence [INC93] offrant une interface homogène d'accès aux données relationnelles à partir de programmes C++, sont apparus sur le marché.

Pour résumer. Les travaux présentés ci-dessus ont deux inconvénients. Le premier est que la puissance d'expression est limitée par l'ensemble des transformations et heuristiques proposées. Le souci majeur de ces approches est de *trouver le schéma cible qui reflète le plus fidèlement possible la structure des données sources*. Aucune latitude n'est laissée à l'utilisateur. Le deuxième inconvénient est que ces approches traitent de la transformation de schéma et négligent leur effet sur les données sous-jacentes. Il peut en effet exister plusieurs interprétations d'une transformation de schéma au niveau des données. Lorsque le chargement des données est considéré, il est codé en dur et nécessitera d'être complètement redéveloppé si l'utilisateur veut organiser le chargement autrement. De plus, quasiment aucune technique d'optimisation n'est utilisée. Nous verrons ce point plus en détail dans le paragraphe 1.2.

1.1.2 Approches génériques

Les approches génériques préconisent l'utilisation d'un modèle de données unique pour capturer les données sources et cibles. Ces modèles sont essentiellement utilisés pour l'intégration de données provenant de sources hétérogènes, pour la conversion de données et pour l'expression de vues au dessus de données sources. Il existe deux tendances : la définition d'un modèle général pour capturer le maximum de propriétés et la définition d'un modèle minimaliste qui ne pose aucune contrainte sur les types de données représentées.

Dans la première catégorie, nous classons des systèmes fondés sur un modèle orienté-objet comme GARLIC [CHS$^+$95] qui utilise un modèle d'intégration de données multimédia, MORPHASE [Kos96, DK97] qui se base sur un modèle permettant d'exprimer des transformations dans un langage de règles, les systèmes de vues objets comme [AB91, SLT91, Ber91, Run92, DdST95] qui définissent des langages déclaratifs pour l'expression de transformations source-cible. Nous verrons que nous utilisons un modèle de données similaire. Notre approche se distingue par le fait que nous considérons la matérialisation de données en masse en opposition à des données virtuelles ou partiellement matérialisées, ce qui nécessite de développer des techniques appropriées. Il existe aussi des modèles relationnels comme par exemple DISCO [TRV96] et Jung-Lee[Jun] pour l'intégration de données distribuées et hétérogènes. Ces modèles sont bien adaptés à la représentation de données relationnelles et de données provenant de fichiers ASCII mais sont insuffisants lorsque les données sont plus complexes. Le modèle relationnel alourdit le traitement de ce type de données (nécessité de jointures fréquentes par exemple).

La deuxième tendance concerne la définition d'un modèle semistructuré dont la caractéristique principale est qu'il est minimal. Ce sont souvent des modèles d'arbres/graphes qui sont utilisés comme dans TSIMMIS [PGMW95], STRUDEL [FFK+98], deux systèmes d'intégration de données hétérogènes, YAT [CDSS98] et [MZ98] pour la conversion de données hétérogènes. L'inconvénient de ces approches est qu'elles véhiculent peu de sémantique ce qui rend le raisonnement sur les données difficile.

1.1.3 Notre approche pour la représentation des données

Notre modèle se veut assez général pour représenter les données sources et cibles que nous considérons et assez spécifique pour pouvoir raisonner sur ces données et envisager un cadre d'optimisation du chargement. Pour ce faire, nous avons choisi d'utiliser IQL, un modèle de données orienté-objet, défini dans [AK89]. IQL est un modèle riche qui permet de représenter des données de type relationnel, objet, etc. Afin d'exprimer les transformations source-cible, nous utilisons une algèbre objet définie dans [CM94] à laquelle nous rajoutons des fonctions de création et de mise à jour de données dans une base. Enfin, nous proposons un langage utilisateur d'expression des transformations dans le cas précis du relationnel-objet qui constitue une application de notre travail. Il est intéressant de noter que le modèle et l'algèbre que nous utilisons nous permettent de définir un cadre général de chargement de données en masse dans une base de données. Par conséquent, les approches spécifiques discutées ci-dessus peuvent être facilement réalisées dans notre contexte.

1.2 Chargement de données

Le chargement en masse peut prendre des heures voire des journées. Du fait de leur volume, les données ne tiennent pas en mémoire centrale. La migration ne peut donc être effectuée dans une seule transaction. De plus, si aucune technique de découpage de ce processus n'est prévue, il suffit d'une interruption pour que toutes les données déjà migrées soient perdues. Rajoutons à cela que les systèmes source et cible restent entièrement bloqués durant le chargement.

Contrairement à l'expression des transformations source-cible, il existe très peu de travaux de recherche sur le chargement de données en masse [Wie95, Fon97, IUT98] et seuls [WN95, Wie95, WN96] proposent d'optimiser ce processus.

Le chargement en masse a intéressé essentiellement les industriels. Plusieurs systèmes commerciaux, qu'ils soient relationnels (ex. Oracle V8 [Ora]) ou à objets (ex. ObjectStore DBConnect [Obj], O2DBAccess [O2T96], GemConnect [Gem], ROBIN [Exe97], OpenODB [Pac91]), fournissent des facilités de chargement de données. Néanmoins, ces outils sont souvent propriétaires et de bas niveau et quasiment aucune technique d'optimisation n'est étudiée. Ils concernent un système source et cible particuliers et ne peuvent être généralisés. Par exemple, SQL*Loader d'Oracle [Ora96] offre la possibilité de reprendre un chargement interrompu après une panne. L'inconvénient est que c'est à la charge de l'utilisateur. Le système met à sa disposition quelques indicateurs à regarder pour savoir à quel moment reprendre le chargement.
SQL*Loader est un outil de chargement de données provenant de fichiers dans une base de données Oracle. Il donne la possibilité à l'utilisateur de définir un chargement en définissant des *fichiers de contrôle* (fic.CTL). Dans chacun, l'utilisateur donne le nom d'un ou plusieurs fichiers de données sources et les noms des relations cibles ainsi que la description de leurs attributs. Il peut également spécifier des informations supplémentaires telles que les conditions que les données sources doivent satisfaire, les index sur lesquels elles sont triées, les index à créer et le nombre maximal d'enregistrements non chargés au delà duquel le chargement doit s'arrêter. Le système génère alors des commandes d'insertion des données sources dans la base cible. Le système garde trace du déroulement du chargement dans un fichier fic.LOG généré pour chaque fichier de contrôle défini. Celui-ci contient le nombre de n-uplets créés dans chaque relation, l'espace utilisé pendant le chargement, les raisons pour lesquelles certains n-uplets ne sont pas chargés (non conformité au type défini par exemple), etc. Deux autres fichiers fic.DSC qui répertorie tous les n-uplets qui ne satisfont pas les conditions du chargement et fic.BAD contenant les enregistrements non chargés à cause d'une erreur sont générés. Si le chargement est interrompu (soit parce que le nombre maximal de n-uplets que peut contenir une table est atteint ou à cause d'une défaillance), c'est l'utilisateur qui doit le reprendre au bon endroit en examinant le fichier fic.LOG. Il doit alors écrire un nouveau fichier de contrôle qui utilise les informations contenues dans fic.LOG. L'utilisateur doit lui-même supprimer les éventuels index créés auparavant et redemander leur création dans le nouveau fichier de contrôle. Par exemple, si le fichier fic.LOG contient l'information selon laquelle 345 enregistrements ont été lus, l'utilisateur peut écrire :
sqlldr userid=scott/tiger control=fic.ctl skip=345
SQL*Loader offre un autre type de chargement : le chargement direct dans lequel les données sont directement créées dans la base (sans générer les commandes d'insertion). Dans ce cas, le système n'est pas en concurrence avec les autres applications et les performances du chargement sont améliorées. C'est la seule optimisation proposée.

En recherche, citons comme exemple le travail de [Fon97] qui introduit une méthode de chargement naïf de données relationnelles dans un systèmes à objets. Cette méthode est fondée sur l'utilisation de fichiers intermédiaires. Le chargement s'effectue en deux étapes. La première consiste à décharger chaque n-uplet de relation dans un fichier accompagné de commandes d'insertion dans la base à objets. La seconde étape consiste à lire chaque fichier généré pour peupler la base. Aucune technique d'optimisation n'est proposée.

L'un des rares travaux de recherche autour du thème du chargement en masse qui propose des techniques d'optimisation est celui présenté dans [WN95, Wie95, WN96] où plusieurs algorithmes de chargement incrémental de données provenant de fichiers ASCII vers une base de données à objets sont proposés. Le souci principal de cette approche est de minimiser le temps de chargement en réduisant le nombre d'entrées/sorties nécessaires à la création et à la mise à jour des objets de la base. Elle repose principalement sur l'utilisation de tables qui maintiennent la correspondance entre les identifiants de données sources et cibles. Ces tables sont nécessaires pour effectuer un chargement incrémental et pouvoir gérer les cycles d'objets. Des identifiants logiques qui assurent qu'un objet n'est effectivement créé que lorsque toutes les données le concernant sont disponibles, sont utilisés.
Un algorithme de chargement naïf a d'abord été réalisé. Les performances de cet algorithme se dégradent lorsque les données à migrer ou les tables de correspondance ne tiennent plus en mémoire centrale. D'autres algorithmes utilisant des tables d'inverses pour le chargement des objets impliqués dans des associations ont été ensuite développés. Afin de retrouver toutes les données qui constituent la valeur d'un objet, le système fait des jointures entre ces tables. Celles-ci sont triées sur l'identifiant logique de l'objet à modifier pour minimiser le coût de sa mise à jour.
Le dernier algorithme proposé offre en outre la possibilité de partitionner les tables de correspondance utilisées de manière à ce que chaque partition tienne en mémoire centrale et minimise ainsi les entrées/sorties. Enfin, tous ces algorithmes ont été testés sur le système Shore [DCa94] et avec des volumes de données croissants. Les auteurs sont arrivés à la conclusion que lorsque moins d'1% des données tiennent en mémoire centrale, de bonnes performances peuvent être réalisées avec leurs algorithmes.
Cette approche propose aussi l'introduction de points de reprise périodiques afin d'éviter de perdre les données chargées à la suite d'une interruption du processus. Un point de reprise définit les informations qui doivent être stockées pour caractériser un instant du chargement. Par exemple, entre deux lectures d'objets dans les fichiers sources, un point de reprise enregistre les tables de correspondance et la position du curseur dans le fichier source. De la sorte, lorsqu'une interruption intervient entre deux points de reprise, toutes les entrées des tables de correspondance créées après le premier point sont annulées et le chargement peut reprendre à ce point. Les auteurs ne proposent pas de compromis entre points de reprise fréquents (pour ne perdre que très peu de ce qui a été fait) et rares (pour éviter de perdre du temps à stocker les informations qui seront utilisées après une interruption).

1.2.1 Notre approche pour le chargement de données

L'intérêt de l'approche présentée dans [Wie95] réside dans (i) l'optimisation du temps de chargement en minimisant le nombre d'entrées/sorties et dans (ii) l'introduction de points de reprise pour gérer les interruptions. Néanmoins, elle s'intéresse au cas particulier du chargement de fichiers ASCII dans une base de données à objets et nécessiterait de réécrire un autre algorithme de chargement pour d'autres types de données. En outre, toute nouvelle optimisation engendre l'écriture d'un nouvel algorithme.
Notre approche se veut générique et déclarative. Les programmes de chargement sont exprimés dans une algèbre qui capture aussi bien des données de type relationnel, objet ou

des données provenant de fichiers ASCII de grammaire simple. Nous pouvons alors peupler une base relationnelle ou à objets à partie d'une autre base ou de fichiers. L'optimisation du chargement consiste à travailler dans le cadre algébrique défini, ce qui permet de se libérer des spécificités des systèmes sous-jacents. Les techniques d'optimisation développées utilisent des équivalences algébriques qui tiennent compte des effets de bord sur la base cible.
Nous les avons validées par le développement d'un prototype pour le cas relationnel-objet. Pour cela, nous avons d'abord conçu un langage déclaratif et de haut niveau qui, en plus de permettre à l'utilisateur l'expression des transformations relationnel-objet, lui donne la possibilité de spécifier des contraintes de regroupement et d'ordonnancement des objets de la base cible sur disque. Ces spécifications sont alors traduites en des programmes de chargement écrits dans l'algèbre. Ces programmes sont ensuite optimisés pour en générer d'autres qui non seulement optimisent le chargement (nombre d'entrées/sorties, temps de communication entre les systèmes, volume de stockage, etc) mais satisfont aussi les contraintes physiques exprimées sur la base cible. Tout cela est réalisé par notre stratégie de recherche qui a l'avantage d'être générique et utilise un modèle de coût spécifique pour le choix des programmes de chargement optimaux.
La phase de génération de code suit pour produire plusieurs transactions contenant le code nécessaire à l'exécution de ces programmes. Cette décomposition permet (i) d'éviter de monopoliser les systèmes sous-jacents pour une longue période, (ii) de perdre le minimum de données dans le cas d'une interruption et (iii) de libérer la mémoire régulièrement. Si de nouvelles optimisations sont envisagées, il suffira de les coder dans la stratégie de recherche, la phase de génération de code reste inchangée.
Le prototype *RelOO*, que nous avons développé, est construit au dessus du SGBD O_2 [BDK92] du côté objet et utilise l'interface standard ODBC [Mic94] pour interroger n'importe quelle base relationnelle.

Première partie

Du Chargement en Masse en Général

Un chargement est défini comme une combinaison de trois fonctions : la fonction d'exportation des données du système source, la fonction de restructuration de ces données et enfin la fonction d'importation dans un système cible. Chacune de ces fonctions peut être entièrement ou en partie exécutée à différents endroits : sur le système source, sur le système cible mais aussi sur un système intermédiaire. L'équilibrage de charge est un paramètre important qui dépend de la disponibilité des systèmes. La communication entre ces trois composants doit aussi être prise en considération. La fonction de restructuration peut être plus ou moins complexe en fonction des capacités des systèmes sous-jacents et aussi de l'application de chargement. Elle peut être exprimée dans un langage de transformation qui met en correspondance les entités sources et cibles ou être définie par défaut. Dans ce dernier cas, elle apparaît comme une boîte noire contenant un ensemble de règles. L'utilisateur intervient pour éventuellement apporter des informations supplémentaires sur les données sources mais n'agit pas sur les transformations.

Une fois les données sources exportées et restructurées, elles sont importées dans le système cible. Cette opération nécessite de conserver la correspondance entre entités sources et cibles pour (i) effectuer un chargement incrémental (mise à jour incrémentale sur la base cible), (ii) établir des points de reprise dans le cas d'une interruption du chargement et enfin, (iii) optimiser l'accès aux entités cibles. La mise en correspondance source–cible repose sur l'utilisation de tables dont l'accès et la taille nécessitent d'être optimisés du fait de la masse importante des données.

Nous commençons par présenter le modèle de données sur lequel repose notre approche. Nous décrivons ensuite un langage formel pour exprimer les transformations source-cible. Nous finissons en présentant les techniques algébriques développées pour réaliser un chargement «intelligent». Notre modèle de données ainsi que le langage de transformation ne couvrent pas tous les cas de chargement. Ils permettent néanmoins de traiter des cas aussi généraux que le chargement de données provenant de fichiers ASCII simples, de bases de données relationnelles et de bases de données à objets et ce vers une base relationnelle ou à objets. Tout au long de cette première partie, nous utiliserons les exemples suivants :

Exemples de fichiers

Fichier `persons` qui contient un ensemble d'enregistrements dont la grammaire est :
record : "person":*string*,"town":*string*. Chaque enregistrement concerne une personne et répertorie son nom et la ville où elle habite.
Fichier `hobbies` dont la grammaire des enregistrements est :
record : "name":*string*,"hobby":*string*. Ce fichier associe à chaque personne un loisir.

Exemples de classes

`Person` [personName:*string*,hobs:{*string*},lives:`City`]. Cette classe contient des objets où chaque objet est une personne donnée par son nom, l'ensemble de ses loisirs et une référence à sa ville.
`City` [cityName:*string*]. Cette classe répertorie des noms de villes.
Nous définissons également une collection d'objets appelées `somePersons` qui contient quelques

personnes créées dans la classe `Person`.

Exemples de relations
`allHobbies`[hobName:*string*]. Cette relation contient des loisirs.
`allPersons`[pName:*string*]. Cette relation contient des noms de personnes.
`perHobs`[pers:*string*,hob:*string*]. Cette relation matérialise une association où une personne est liée à plusieurs loisirs et un loisir intéresse plusieurs personnes.

Chapitre 2

Modèle de données

Le modèle de données que nous présentons est le modèle objet IQL [AK89]. Il nous paraît très satisfaisant puisqu'il couvre les modèles relationnel, objet et permet également de représenter les grammaires simples de nombreux fichiers ASCII. Nous l'avons simplement étendu avec la notion de liste ordonnée.

2.1 Définitions

Le modèle IQL manipule des objets et des valeurs. À partir de l'ensemble de valeurs atomiques D et de l'ensemble des identifiants d'objets O, nous définissons l'ensemble de valeurs $Val(O)$ en utilisant les constructeurs d'ensemble, de liste et de n–uplet.

Soient les ensembles infinis dénombrables suivants :

- $\mathcal{R} = \{r_1, r_2, \ldots\}$ qui contient des noms de relations (ex. `allHobbies`), de fichiers (ex. `persons`) ou de racines de persistance objet (ex. `somePersons`).
- $\mathcal{C} = \{c_1, c_2, \ldots\}$ qui contient des noms de classes telles que définies dans le modèle objet (ex. `City`).
- $\mathcal{A} = \{a_1, a_2, \ldots\}$ qui contient des noms d'attributs de relations ou d'attributs de classes (ex. "person", "cityName", "hob").
- $O = \{o_1, o_2, \ldots\}$ qui contient les identifiants d'objets créés dans les classes de l'ensemble C.
- $D = \{d_1, d_2, \ldots\}$ qui est un ensemble de valeurs atomiques (ex. $\{5.2, 'Paris', 0, \ldots\}$).

Définition 2.1.1 Valeurs. L'*ensemble des valeurs* notées $Val(O)$ est le plus petit ensemble contenant $D \cup O$ et tel que, si $v_1, \ldots, v_k$ $(k \geq 0)$ sont des valeurs et $a_1, \ldots, a_j$ $(j \geq 0)$ sont des noms d'attributs distincts alors $\{v_1, \ldots, v_k\}$ est une valeur *ensemble*, $\langle v_1, \ldots, v_k \rangle$ est une valeur *liste* et $[a_1 : v_1, \ldots, a_k : v_k]$ est une valeur *n–uplet*. Une valeur relationnelle est un cas particulier $[a_1 : v_1, \ldots, a_k : v_k]$ où $v_1, \ldots, v_k$ $(k \geq 0)$ sont des valeurs appartenant à

l'ensemble D. Le symbole spécial *nil* dénotant l'identifiant indéfini est une valeur sur O. De la même manière, le symbole *null* dénote la valeur indéfinie dans D.
L'ensemble des enregistrements d'un fichier ASCII simple est vu comme une liste de valeurs de type n–uplet. L'enregistrement [name:John, hobby:skying] est de type n–uplet. □

Définition 2.1.2 Assignation de valeurs et assignation d'oids. Soit R un ensemble fini de racines et C un ensemble fini de classes.

Une assignation de valeurs pour R est une fonction ρ qui associe à chaque racine dans R un ensemble fini de valeurs. Ainsi, ρ`(hobbies)` contient tous les enregistrements du fichier *hobbies*.

Une assignation d'oids pour C est une fonction π qui associe à chaque classe c dans C un ensemble fini d'identifiants d'objets distincts dans O. $\pi(c)$ est aussi appelée *population* de c. π est *disjoint* si : pour tout c, c' de C, $c \neq c'$ alors $\pi(c) \cap \pi(c') = \emptyset$. □

Définition 2.1.3 Objet. Un *objet* est une paire (o, v) où o est un oid (identifiant) dans O et v une valeur dans $Val(O)$.
Dans IQL, la notion d'identité d'objet est fondamentale. Par rapport aux modèles par valeur, où l'identité est définie en fonction de la valeur (ex. la clé d'un n–uplet dans le modèle relationnel), l'identité dans ce modèle est indépendante de la valeur des objets. Ainsi un objet possède un identifiant et un état qui peut changer sans pour autant que son identifiant soit modifié. □

Définition 2.1.4 Expressions de type. L'ensemble des *expressions de types* définies sur C, $types(C)$, est donné par la syntaxe suivante où τ est une expression de type, c est un élément de C et les a_k $(k \geq 0)$ sont des attributs :

$$\tau : \, : = D|c|[a_1 : \tau_1, \ldots, a_k : \tau_k]|\{\tau\}|\langle\tau\rangle$$

La structure d'une valeur est décrite par son type. Les types sont définis à partir des noms de classes et de l'ensemble des valeurs atomiques D en utilisant des constructeurs ([] pour le type n–uplet, {} pour le type ensemble). Nous avons étendu la définition de types initiale dans IQL en rajoutant la notion de liste d'éléments ordonnés représentée par le constructeur $\langle\rangle$. Notons que les éléments de l'ensemble C sont des types. Un nom de classe c est un type et ne doit pas être confondu avec *l'extension de la classe* $\pi(c)$ qui représente l'ensemble des objets qui lui appartiennent. □

Définition 2.1.5 Interprétation de type. Pour une assignation d'oids π, à chaque expression de type τ est associé un ensemble de valeurs et une *interprétation de type* $[\![\tau]\!]_\pi$ définie comme suit :

- $[\![D]\!]_\pi = D, [\![c]\!]_\pi = \pi(c)$ (pour tout $c \in C$),
- $[\![[a_1 : \tau_1, \ldots, a_k : \tau_k]]\!]_\pi = \{[a_1 : v_1, \ldots, a_k : v_k] | v_i \in [\![\tau_i]\!]_\pi, i \in [1, k]\}$,
- $[\![\{\tau\}]\!]_\pi = \{\{v_1, \ldots, v_j\} | j \geq 0$, et $v_i \in [\![\tau]\!]_\pi, i \in [1, j]\}$,
- $[\![\langle\tau\rangle]\!]_\pi = \{\langle v_1, \ldots, v_j\rangle | j \geq 0$, et $v_i \in [\![\tau]\!]_\pi, i \in [1, j]\}$,

□

2.2 Notions de schéma et d'instance

Afin de représenter les données sources et cibles, nous utilisons les notions de schéma et d'instance tels que définis dans IQL. Un schéma est constitué d'un ensemble de racines (fichiers, relations ou racines de persistance) et d'un ensemble de classes. Une classe décrit des objets ayant la même structure. L'instance d'un schéma, appelée base, regroupe les objets et les valeurs qui respectent les contraintes de type de ce schéma. Par exemple, dans le cas de fichiers ASCII, le schéma nous donne l'ensemble des noms de fichiers et le type de données qu'ils contiennent et une base, instance de ce schéma, représente les fichiers et leur contenu.

Définition 2.2.1 Schéma. *Un schéma* est défini par la donnée de $(R, C, T, \prec)$, où R est un ensemble fini de racines, C est un ensemble fini de noms de classes, T est une fonction de $R \cup C$ dans $types(C)$ qui associe à chaque racine dans R, son type et à chque classe dans C, son type (et non le nom de la classe). $\prec$ est une relation d'héritage d'ordre partiel entre les classes de C. □

Définition 2.2.2 Héritage et sous-typage. La relation d'héritage entre classes induit la relation de *sous–typage*, notée $\leq$, qui est la plus petite relation d'ordre partiel sur $types(C)$ telle que :

- si $c \prec c'$ alors $c \leq c'$
- $\forall \tau_i, s_i \in types(C)$ tels que $\tau_i \leq s_i$ et $i \in [1, n]$, $[a_1 : \tau_1, \ldots, a_n : \tau_n, a_{n+1} : \tau_{n+1}, \ldots, a_{n+p} : \tau_{n+p}] \leq [a_1 : s_1, \ldots, a_n : s_n]$
- si $\tau_1 \leq \tau_2$ alors $\{\tau_1\} \leq \{\tau_2\}$ et $\langle\tau_1\rangle \leq \langle\tau_2\rangle$

□

Il est commode de distinguer deux grandes familles de schémas : les schémas à valeurs et les schémas à objets.
Un schéma à valeurs est défini par le 4-uplet $(R, \emptyset, T, \_)$. C'est le cas d'un schéma contenant des fichiers avec R représentant l'ensemble des noms de fichiers et T la fonction de typage de chaque r dans R. L'ensemble des classes C est vide et la relation d'héritage n'est pas considérée. Plus concrètement, l'ensemble des racines sources R peut contenir le fichier `hobbies` dont le type est ⟨[name:string, hobby:string]⟩. Un schéma relationnel sera représenté par un schéma à valeurs.
ar contre, un schéma à objets est défini par le 4-uplet $(R, C, T, \prec)$ où R est l'ensemble des racines de persistance, C est l'ensemble des noms de classes par exemple `City`.

Définition 2.2.3 Schéma objet bien formé. Les schémas de type objet que nous considérons sont bien formés : $\forall c, c' \in C, c' \prec c \Rightarrow T(c') \leq T(c)$. □

Définition 2.2.4 Base. *Une instance* I d'un schéma S est un 3-uplet (ρ, π, ν) défini comme suit :

- $\rho : R \longrightarrow Val(O)$ est une assignation de valeurs pour chaque racine $r \in R$.

- $\pi : C \longrightarrow 2^O$ est une fonction de population pour chaque classe $c \in C$. L'ensemble O est maintenant limité aux objets de la base $O = \bigcup\{\pi(c) | c \in C\}$.

- $\nu : O \longrightarrow Val(O)$ qui associe à chaque objet sa valeur.

Les contraintes suivantes doivent toujours être vérifiées :

1. $\rho(r) \in [\![T(r)]\!]_\pi$, pour toute racine $r \in R$,

2. $\forall o \in \pi(c), \nu(o) \in [\![T(c)]\!]_\pi$, pour toute classe $c \in C$,

3. ν est totale sur $\pi(c)$, pour tout $c \in C$ avec $T(c) = \{\tau\}$ ou $T(c) = \langle\tau\rangle$,

□

Schéma source et schéma cible. Nous pouvons maintenant représenter un schéma source S_s et son instance source I_s ainsi qu'un schéma cible S_t et son instance I_t. Nous verrons plus loin comment exprimer I_t à partir de I_s. Prenons l'exemple donné précédemment sur le chargement d'un ensemble de fichiers ASCII dans une base à objets :

- S_s est donné par $(R, \emptyset, T, \_)$ où R est l'ensemble des fichiers sources (dans ce cas, nous considérons le fichier `hobbies`).

- I_s est donnée par $(\rho_s, \_, \_)$. L'ensemble des classes dans S_s étant vide, π_t et ν_t ne sont pas définies. ρ_s(`hobbies`) est la liste des enregistrements du fichier `hobbies` donnés dans le tableau suivant :

ρ_s(`hobbies`)

name	hobby
John	skying
Maria	reading
Georges	dancing
Maria	swimming
John	surfing
Helena	playing tennis
Helena	watching TV
Georges	swimming
John	reading
Mary	playing piano

- S_t est donné par $(R, C, T, \_)$ où R contient la racine de persistance `somePersons` associée à la classe `Person`, C contient la classe `Person` où T(`Person`) est [person:*string*,hobs:{*string*},lives:`City`]. La relation d'héritage n'est pas considérée.

– I_t est donnée par (ρ_t, π_t, ν_t). À la classe `Person` est associée sa population π_t(`Person`) $= \{o_1, o_2, o_3, o_4, o_5\}$ définie à partir de l'instance source I_s. $\nu(o)$ attribue sa valeur à chaque objet o créé dans π_t(`Person`).
Par exemple, $\nu_t(o_2)$ = [personName:Helena,hobbs:{playing tennis, watching TV},lives:*nil*].
La racine de persistance `somePersons` peut contenir quelques personnes créées dans `Person`. ρ_t(`somePersons`) $\subseteq$ π_t(`Person`). Les objets contenus dans π_t(`Person`) ne sont pas ordonnés puisqu'il s'agit d'un ensemble et non d'une liste. Néanmoins, nous utiliserons la représentation tabulaire ci-dessous pour les listes aussi.

π_t(`Person`)

oid	person	hobs	lives
o_1	Georges	{dancing,swimming }	nil
o_2	Helena	{playing tennis,watching TV}	nil
o_3	John	{skying,surfing,reading}	nil
o_4	Maria	{reading,swimming}	nil
o_5	Mary	playing piano	nil

Quelques remarques :

1. De la même façon que ci-dessus, nous pouvons considérer un schéma source avec, cette fois-ci, les fichiers `persons` et `hobbies` et comme schéma cible, l'exemple relationnel contenant les relations `allHobbies, allPersons` et `perHobs`.

2. D'un point de vue général, un schéma à objets peut avoir zéro, une ou plusieurs racines de persistance. Chaque racine est associée à une ou plusieurs classes. Plusieurs racines peuvent être associées à la même classe. Dans notre cas, nous nous limiterons à associer à une classe c au plus une racine de persistance dont le type sera $\{c\}$ ou $\langle c \rangle$. Si r est une une racine de persistance associée à la classe c alors :
$\forall o \in \rho(r), o \in \pi(c)$.

3. Nous n'avons pas parlé de la notion de clé telle que définie dans le modèle relationnel. Nous verrons que cette notion n'est pas indispensable grâce à l'utilisation de fonctions de peuplement particulières. De ce fait, nous ne posons aucune condition sur les schémas relationnels considérés (dépendance fonctionnelle et d'inclusion. ...).

Après avoir présenté le modèle IQL et la possibilité de représenter des schémas et des données, nous donnons, dans le chapitre suivant, un langage formel pour l'expression des transformations source-cible. Ce langage permet de spécifier des opérations sur toute donnée dont le type est inclus dans $types(C)$.

Chapitre 3

Langage formel pour les transformations

Partant du modèle, nous avons choisi un langage déclaratif de transformations source-cible qui nous permet d'optimiser le chargement : l'algèbre de [CM94]. Cette algèbre nous donne la possibilité de restructurer librement toutes les données du modèle IQL.

Les algèbres sont communément utilisées pour optimiser des requêtes. Comme nous l'avons dit auparavant, un chargement de données est constitué de trois fonctions principales : exportation des données sources, restructuration et importation dans le système cible. Les deux premières phases s'expriment aisément dans l'algèbre, la dernière nécessite de travailler par effets de bord sur la base cible. Ces effets de bord ne sont pas arbitraires et utilisent le résultat des requêtes évaluées. De ce fait, l'algèbre sert de support au calcul temporaire. Par conséquent, l'optimisation d'un système de chargement repose essentiellement sur la réécriture de requêtes.

Le chargement de données contenues dans une instance I_s d'un schéma S_s, est exprimé par la donnée :

– d'un schéma cible S_t,

– et d'un programme permettant de passer de l'instance I_s de S_s à une instance I_t de S_t. Nous exprimerons ce programme par une séquence d'*expressions algébriques* qui mettent à jour les fonctions π_t, ρ_t et ν_t.

Reprenons notre exemple de chargement d'un ensemble de fichiers dans une base de données à objets. Le schéma source S_s contient les fichiers `persons` et `hobbies`. Supposons que l'on définisse le schéma cible $S_t(R, C, T, \prec)$ avec R contenant les racines de persistance des classes `Person` et `City` et C contenant les classes `Person` et `City`. Par la donnée de π_t et ν_t (ρ_t n'est pas définie dans ce cas puisque nous ne considérons pas de racine de persistance), nous définissons une instance I_t. Dans la suite, nous verrons comment π_t, ρ_t et ν_t peuvent être exprimées en utilisant les opérateurs de l'algèbre. Nous donnons d'abord la définition formelle des opérateurs ainsi que les fonctions qui réalisent les effets de bord. Dans le chapitre 4, nous verrons que de nouvelles équivalences algébriques qui tiennent compte des effets de bord sur la base cible permettent de raisonner sur les spécifications de π, ρ et ν.

3.1 Algèbre

L'algèbre que nous utilisons est celle de [CM94]. Elle est bien adaptée au problème de chargement que nous traitons puisqu'elle supporte les notions de n–uplet, ensemble et liste ordonnée. Elle est constituée d'opérateurs définis, pour la plupart, sur des collections de n–uplets dont les attributs peuvent être de type quelconque (atomique ou complexe). Ci-dessous, nous donnons la définition des opérateurs pour la manipulation d'ensembles. Ils sont définis de manière similaire pour les listes.

3.1.1 Opérations de base

Afin de mieux comprendre chacune des opérations algébriques que nous allons présenter, nous en donnerons des exemples en utilisant les fichiers `hobbies` et `persons` dont les contenu est donné ci-dessous :

$\rho($`hobbies`$)$

name	hobby
John	skying
Maria	reading
Georges	dancing
Maria	swimming
John	surfing
Helena	playing tennis
Helena	watching TV
Georges	swimming
John	reading
Mary	playing piano

$\rho($`persons`$)$

person	town
John	Paris
Maria	New York
Georges	New York
Helena	Paris
Marc	Paris
Mary	Vienna

Dans la suite, les notations e et e_i désignent des expressions algébriques composées d'opérateurs quelconques. Si e figure à droite d'une définition, il s'agira de son évaluation. Si elle est à gauche, il s'agira de sa définition. p est un prédicat. c est une classe. r est une racine. a et a_i désignent des attributs de classe ou de racine. La notation $\vec{a}$ désigne une liste d'attributs. Si t est une valeur de type n–uplet, $t.a$ désignera la valeur de l'attribut a de t et $t.\vec{a}$

dénotera la valeur de la sous–partie du n–uplet de t contenant les attributs de $\vec{a}$. Dans un schéma de racines ou de classes, chaque nom d'attribut est unique. Si tel n'est pas le cas, un attribut sera préfixé par le nom de la racine ou de la classe à laquelle il appartient.

Map : Le rôle de l'opérateur Map est fondamental dans l'algèbre. Comme nous le verrons, les autres opérateurs sont définis sur des ensembles de n-uplets. Ceci permet d'attribuer certaines propriétés à l'algèbre. L'opérateur permet alors de transformer des valeurs arbitraires en collections de n-uplets et vice versa. De cette façon, les autres opérateurs de l'algèbre peuvent être appliqués. Il permet d'appliquer une fonction aux éléments d'une collection. En particulier, il prend en entrée une collection quelconque (ensemble ou liste d'éléments ayant un type arbitraire) et peut construire une collection de n-uplets qui pourra ensuite être manipulée par les autres opérateurs de l'algèbre. De façon analogue, il permet de casser les n-uplets d'une collection pour en extraire un ensemble de chaînes de caractères ou d'objets par exemple. Également important, il permet d'appliquer des fonctions arbitraires à la collection qu'on lui donne entrée. En particulier, nous verrons que cet opérateur est utilisé pour appliquer des fonctions de création et de mise à jour des entités cibles. Il existe sous trois différences syntaxiques :

1. $e[a] = \{[a : x] | x \in e\}$ qui construit un n-uplet de toute donnée en entrée. Par exemple, cette forme sera utilisée pour transformer une collection d'objets en une collection de n-uplets dans le cas d'un chargement objet-cible.
2. $Map_f(e) = \{f(t) | t \in e\}$ où f est une fonction dont les variables sont des attributs de e. Par exemple,
   Map_{person}(`persons`)$=$ {John, Maria, Georges, Helena, Marc, Mary} extrait les valeurs de l'attribut "person" du fichier `persons`.
3. $Map_{\circ[a:f]}(e) = \{t \circ [a : f(t)] | t \in e\}$ qui permet de conserver dans l'attribut a le résultat de l'évaluation de f sur chaque n–uplet t de e ($\circ$ est la concaténation de n-uplets). Nous utiliserons $Map_{a:f}(e)$ pour dénoter $Map_{\circ[a:f]}(e)$. Par exemple, $Map_{country:Country(town)}$(`persons`) où *Country* est une fonction qui associe son pays à une ville. Cette expression retrourne l'ensemble des n-uplets suivants :

   {[person:John, town:Paris, country:France], [person:Maria, town:New York, country:USA], [person:Georges, town:New York, country:USA], [person:Helena, town:Paris, country:France], [person:Marc, town:Paris, country:France], [person:Mary, town:Vienna, country:Austria]}

L'utilisation de l'attribut a pour récupérer le résultat de l'évaluation de $Map_{a:f}(e)$ permet non seulement de conserver l'effet de f sur chaque n-uplet de e mais aussi de conserver dans le flux algébrique résultant, le résultat de l'évaluation de e sur chacun de ses n-uplets.
Lorsque deux opérations Map sont appliquées de manière successive, nous les regrouperons en une seule. Ainsi, $Map_{a:f_1}(e)$ suivie de $Map_{f_2}(e)$ s'écrit $Map_{f_1;f_2}(e)$. L'attribut a peut être supprimé puisque e n'est pas perdu.

Jointure : $e_1 \bowtie_p e_2 = \{t_1 \circ t_2 \mid t_1 \in e_1, t_2 \in e_2, p(t_1, t_2)\}$ où $\circ$ représente la concaténation de n–uplets. Cette définition implique que les attributs de t_1 et t_2 soient disjoints.

Supposons que l'on veuille mettre dans un même résultat toutes les informations dont nous disposons sur les personnes répertoriées dans les deux fichiers. Ceci s'exprime par la jointure (`hobbies` $\bowtie_{name=person}$ `persons`) dont le résultat est donné ci-dessous.

`hobbies` $\bowtie$ `persons`

name	hobby	person	town
John	skying	John	Paris
Maria	reading	Maria	New York
Georges	dancing	Georges	New York
Maria	swimming	Maria	New York
John	surfing	John	Paris
Helena	playing tennis	Helena	Paris
Helena	watching TV	Helena	Paris
Georges	swimming	Georges	New York
John	reading	John	Paris
Mary	playing piano	Mary	Vienna

Jointure externe : (en anglais Left–outer–join) $e_1 \sqsupset\!\bowtie_p e_2 =$

$$\{t_1 \circ t_2 \mid t_1 \in e_1, t_2 \in e_2 \mid p(t_1, t_2)\}$$
$$\cup$$
$$\{t_1 \circ t_3 \mid t_1 \in e_1, \neg\exists t_2 \in e_2, p(t_1, t_2), atts(t_3) = atts(e_2),$$
$$\forall a \in atts(e_2), t_3.a = NULL\}.$$

Cette définition implique que tous les attributs de t_1 et t_2 sont disjoints. Elle agit comme une jointure pour le côté droit et laisse passer dans le résultat, tous les n–uplets de e_1 (côté gauche) en leur concaténant la valeur indéfinie *null* lorsqu'il n'existe aucun n–uplet de e_2 qui satisfait la condition de jointure.

Cette opération est utilisée pour des raisons d'optimisation. Il existe d'autres variantes telles que la jointure externe à droite (en anglais right–outer–join) ($e_1 \bowtie\!\sqsubset_p e_2$) et la jointure externe double (en anglais outer–join) ($e_1 \sqsupset\!\bowtie\!\sqsubset_p e_2$).

Sélection : $\sigma_p(e) = \{t | t \in e, p(t)\}$ qui sélectionne les n–uplets d'une expression algébrique e qui satisfont un prédicat p dont les variables sont des attributs de t.
Ainsi, $\sigma_{HOBBY='reading'}$(`hobbies`) a comme résultat :

$\sigma_{HOBBY='reading'}$(`hobbies`)

name	hobby
Maria	reading
John	reading

Projection : $\pi_{\vec{a}}(e) = \{t.\vec{a} \mid t \in e\}$ où $t.\vec{a}$ retourne la sous-partie du n–uplet t constituée des attributs $\vec{a}$. La projection sur un attribut a_i avec $a_i \notin \vec{a}$ n'est pas définie. Comme nous travaillons sur des ensembles de valeurs distinctes, les redondances sont éliminées. Ainsi, π_{name}(`hobbies`) donne :

π_{name}(`hobbies`)

name
John
Maria
Georges
Helena
Mary

Notons que l'opération de projection est un cas particulier d'application du *Map* : $\pi_{\vec{a}}(e) = Map_{[a_1:\tau_1,\ldots,a_n:\tau_n]}(e)$ avec $\vec{a} = \{a_1, \ldots, a_n\}$.

Group : $Group_{g;\vec{a_1};f}(e) = \{t_1.\vec{a_1} \circ [g : G] | t_1 \in e, G = \{f(t_2) | t_2 \in e, t_2.\vec{a_1} = t_1.\vec{a_1}\}\}$. Cette opération regroupe dans l'attribut g, l'ensemble des résultats de l'évaluation de f sur les n–uplets de e qui ont la même valeur pour $\vec{a_1}$.
Par exemple, $Group_{g;\vec{a_1};\pi_{\vec{a_2}}}(e)$ regroupe dans g, toutes les valeurs de $\vec{a_2}$ qui ont une-même valeur pour $\vec{a_1}$.
Pour des raisons de simplification, nous utiliserons
$Group_{g;\vec{a_1};\vec{a_2}}(e)$ pour exprimer $Group_{g;\vec{a_1};\pi_{\vec{a_2}}}(e)$. Ainsi, $Group_{g;name;hobby}$(`hobbies`) regroupe, par nom, l'ensemble des loisirs d'une personne.

$Group_{g;name;hobby}$(`hobbies`)

name	g
Georges	{dancing,swimming}
Helena	{playing tennis,watching TV}
John	{skying,surfing,reading}
Maria	{reading,swimming}
Mary	playing piano

Sort : $Sort_{\vec{a}}(e) = \langle t_1, t_2, ..., t_n \rangle$ tel que

$$\begin{aligned}&(\forall i \in [1, n], t_i \in e)\\ &\wedge(\forall t \in e, \exists i \in [1, n], t_i = t)\\ &\wedge(\forall i \in [1, n-1]\ t_i.\vec{a} \leq t_{i+1}.\vec{a})\end{aligned}$$

Cette opération trie le résultat de e par ordre ascendant sur les valeurs de $\vec{a}$. Elle produit une liste. Le tri est non déterministe ce qui signifie que lorsque deux entrées ont la même valeur pour le critère de tri, le choix de leur position dans la liste résultante est aléatoire. Par exemple : $Sort_{name}$(`hobbies`) a comme résultat la liste des n-uplets du fichier `hobbies`, ordonnée par nom de personnes. Dans le résultat, l'enregistrement [name:Maria,hobby:reading] peut aussi bien se trouver avant ou après l'enregistrement [name:Maria,hobby:swimming].

$Sort_{name}$(`hobbies`)

name	hobby
Georges	dancing
Georges	swimming
Helena	playing tennis
Helena	watching TV
John	reading
John	surfing
John	skying
Maria	reading
Maria	swimming
Mary	playing piano

3.1.2 Fonctions d'effet de bord

Afin d'exprimer les effets de bord sur la base cible, nous avons défini deux familles de fonctions qui sont appliquées à travers l'opérateur Map de l'algèbre : les *fonctions de création* qui créent des entités cibles à partir d'entités sources et les *fonctions d'assignation* qui mettent à jour les entités cibles créées. Les opérations de base, définies dans le paragraphe précédent, permettent d'importer et de restructurer les données de la base source. Les fonctions d'effet de bord réalisent l'exportation des données dans le système cible.

Fonctions de création

Ce sont des fonctions de Skolem dont les paramètres sont des données sources et qui retournent des identifiants cibles. Une fonction de Skolem associe à chaque donnée une donnée unique. Pour garantir l'unicité des identifiants pour un ensemble de paramètres donnés, elles utilisent des tables de correspondance. Celles-ci sont toujours accédées depuis les paramètres sources. Leur utilisation est indispensable pour (i) gérer le fait qu'une entité peut être référencée par plusieurs autres (dans ce cas, la table de correspondance permet de s'assurer que l'entité en question est créée une seule fois), pour (ii) retrouver à tout moment l'entité cible qui correspond à la source et enfin pour (ii) mettre à jour les données cibles de manière incrémentale.

Les paramètres d'une fonction de création sont appelés *clés de transformation*. Ainsi, dans le cas du chargement de fichiers dans une base de données à objets et pour un couple (fichier r,classe c), la clé de transformation source peut être la valeur d'un ou de plusieurs champs des enregistrements du fichier r. Si nous considérons le champ "name" du fichier `hobbies` comme clé de transformation pour la création des objets dans la classe `Hobbies`, il y aura autant d'objets créés dans cette classe que de noms de personnes différents dans le fichier `hobbies`. La table de correspondance de `(hobbies,Hobbies)` contiendra la paire (name,identifiant cible).

Un ou plusieurs attributs peuvent être utilisés comme clés de transformation. Dans le cas de la non unicité des valeurs de ces attributs, la fonction de Skolem garantit qu'une seule entité source est créée pour ces valeurs (de manière non déterministe). Cette approche a l'avantage d'être flexible en ne posant aucune condition sur les données sources ou cibles.

Nous verrons dans le chapitre 4 que nous utilisons trois types de fonctions de création pour le cas du chargement relationnel-objet. Dans l'immédiat, nous introduisons la fonction $condnew(r, c, \vec{a})$.

Si r est un fichier et c une classe, $condnew(r, c, \vec{a})$ associe à chaque valeur distincte de $\vec{a}$ (clé de transformation formée de champs de r), un identifiant d'objet dans $\pi(c)$. À chaque invocation, cette fonction teste si l'objet correspondant à la clé de transformation donnée en paramètre a été créé et le crée sinon. Elle met à jour la table de correspondance définie pour la paire (r, c) si nécessaire (si la paire (clé,objet) n'y est pas déjà).
Si nous prenons le cas du fichier `persons`, nous pouvons peupler la classe `City` en considérant le champ "town" comme clé de transformation.
Ainsi $Map_{condnew(\texttt{persons},\texttt{City},town)}$ `(persons)` créera autant d'objets dans la classe `City` qu'il y a de noms de villes différents dans le fichier `persons`.

De manière similaire, nous définissons une fonction de Skolem $condnew(c, r, \vec{a})$ qui associe une valeur d'un attribut ou d'un champ de r à chaque valeur distincte des attributs $\vec{a}$ de c et met à jour la table de correspondance (classe source c, racine cible r). Par exemple, si nous supposons que la classe `Hobbies` a une racine de persistance appelée `theHobbies` et que la relation `allPersons` est peuplée à partir de cette racine, ceci s'écrira :
$Map_{condnew(\texttt{Hobbies},\texttt{allPersons},personName)}$ `(theHobbies)` qui créera autant de personnes dans `allPersons` qu'il y a de valeurs distinctes de l'attribut "personName" dans la racine `theHobbies`.

Par définition, si la fonction *condnew* a comme clé de transformation une valeur indéfinie, son résultat est aussi une valeur indéfinie :
$condnew(r, c, null) = nil$ et
$condnew(c, r, nil) = null$
condnew n'a alors aucun effet sur la base cible ni sur la table de correspondance. Cette propriété est appelée *insensibilité à la valeur nulle*. Les fonctions de création vérifient une autre propriété que nous appelons *idempotence*. Celle-ci signifie que, appliquée plus d'une fois avec les mêmes paramètres, la fonction *condnew* assure que l'entité cible correspondante n'est créée qu'une seule fois. Nous verrons dans la suite que ces propriétés nous permettent de supporter certaines équivalences algébriques.

Fonctions de mise à jour

Nous avons défini une fonction de mise à jour qui nous permet d'exprimer l'évaluation des attributs formant la valeur d'une entité. Cette fonction est définie par les opérateurs «=» (assignation de valeur) et «+=» (insertion d'un élément dans un ensemble ou une liste). La partie gauche de l'assignation est un attribut d'une entité cible. Sa partie droite est un appel de fonctions quelconques (identité, fonction arithmétique, fonction de chaînes de caractères, fonction de conversion de type ou encore une combinaison de fonctions de création ou de mise à jour d'entités). Ainsi, si la classe `Person` est peuplée à partir du fichier `persons`, les

attributs "personName" et "lives" de chaque objet o de cette classe seront mis à jour de la façon suivante :
o.person=name
o.lives= *condnew*(`persons`,`Person`,town). Nous verrons plus loin des exemples plus élaborés.

De la même façon que la fonction *condnew*, l'assignation va être appliquée à travers l'opérateur Map. Si une valeur est assignée plusieurs fois à un attribut de l'entité cible donnée en paramètre, elle ne doit être assignée qu'une seule fois *(idempotence)*. Si ses paramètres ont la valeur *null*, la partie droite va être effectuée mais aucune valeur n'est assignée *(insensibilité à la valeur nulle)*.
Ainsi, en écrivant o.lives = *condnew*(`persons`,`Person`,town), et si $o = nil$ alors la partie droite de l'égalité va être évaluée (un objet sera créé dans la classe `City`) mais l'assignation ne sera pas effectuée.

3.2 Expression des effets de bord

Comme les données sources proviennent de relations, fichiers ou racines de persistance objet, une requête sera exprimée sur des racines de R. C'est sur le résultat de cette requête que sont ensuite évaluées les fonctions de création et de mise à jour des entités cibles. Comme nous l'avons dit plus haut, les effets de bord permettent de peupler l'instance cible I_t. En d'autres termes, ils définissent $\rho_t(r)$ pour chaque racine r, $\pi_t(c)$ pour chaque classe c et $\nu_t(o)$ pour chaque objet o dans $\pi_t(c)$. Un chargement revient donc à écrire l'expression permettant d'évaluer chacune de ces fonctions. Si la base cible est relationnelle, il suffira d'exprimer ρ_t pour chacune des relations cibles. Si elle est de type objet, il s'agira d'exprimer π_t et ν_t (pour créer les objets et évaluer leurs attributs) et aussi ρ_t pour peupler les racines de persistance. Nous donnons l'expression générale de chacune de ces fonctions. Pour des raisons de simplicité, nous n'écrirons que les clés de transformation comme paramètres de la fonction *condnew*. Ainsi, $condnew(source, cible, \vec{a})$ est écrit $condnew(\vec{a})$.

3.2.1 Peuplement de racines

Une racine r peut être peuplée à partir d'une ou de plusieurs racines ou classes sources. Par conséquent, $\rho_t(r)$ est exprimée comme suit :

$$\rho_t(r) = \{\cup \mathcal{Q}_i(\rho_s, \pi_s, \nu_s), i \in [1, n]\}$$

où $\mathcal{Q}$ est une requête sur I_s exprimée en fonction d'opérations algébriques. La racine r est peuplée par l'union des résultats de requêtes sur toutes les racines ou classes sources de r.
Soit le fichier `persons`. Supposons que l'on veuille créer une base relationnelle qui contiendrait une seule relation `allPersons` dans laquelle seront répertoriées les personnes du fichier `persons` qui habitent Paris.

$\rho_t(\texttt{allPersons}) = \{\mathcal{Q}(\rho_s, \pi_s, \nu_s)\}$ avec
$\mathcal{Q}(\rho_s, \pi_s, \nu_s) = Map_{condnew(p.person);condnew(p.person).pName=p.person}(\sigma_{p.town='Paris'}(\texttt{persons}[p]))$

avec $\texttt{persons}[p]$ qui construit des n-uplets.

3.2.2 Peuplement de classes et assignation d'attributs

Une classe c peut être peuplée à partir d'une ou de plusieurs racines ou classes sources. Le peuplement d'une classe s'exprime en deux phases : (i) création des objets de la classe dans $\pi(c)$ et (ii) évaluation de la valeur de ces objets (de leurs attributs) $\nu(o), \forall o \in \pi(c)$.

$\pi_t(c) = \{\cup \mathcal{Q}_i(\rho_s, \pi_s, \nu_s)), i = [1, n]\}$

ν_t s'exprime par un ensemble de fonctions d'assignation pour chaque attribut de c.

Reprenons notre exemple de peuplement de la classe `Person`, l'instance cible I_t est entièrement définie par la donnée de :

$\pi(\texttt{Person}) = Map_{condnew(name)}(\texttt{hobbies})$
$Map_{o.personName=name;o.hobs=g}(Group_{g;name;hobby}(\texttt{hobbies}))$ calcule la valeur de chaque objet o de π`(Person)`. Nous pouvons alors écrire :
$\nu_t(o) = [personName : t.name, hobs : t.g]$ avec $t \in (Group_{g;name;hobby}(\texttt{hobbies}))$.

Ainsi un objet est créé pour chaque personne dans le fichier `hobbies`. Les attributs "personName" et "hobs" sont évalués pour chacun de ces objets. L'évaluation de l'attribut "hobs" nécessite l'utilisation de l'opérateur de regroupement *Group*.

Chapitre 4

Optimisation

Dans ce chapitre, nous allons illustrer les possibilités d'optimisation supportées par l'algèbre. Une application de ces optimisations dans le cas du chargement relationnel-objet est donnée dans la partie II de cette thèse.

De la même manière que dans l'optimisation de requêtes, l'optimisation d'un système de chargement revient à trouver le scénario d'exécution optimal. Chaque scénario correspond à un ensemble de programmes de chargement qui, lorsqu'ils sont tous exécutés, peuplent la base cible. L'optimalité est définie par rapport à une fonction d'évaluation du coût de chaque scénario (le *modèle de coût*). Afin de choisir une solution plutôt qu'une autre, l'optimiseur utilise une *stratégie de recherche.* C'est un algorithme selon lequel des réécritures algébriques sont appliquées successivement pour examiner l'ensemble des solutions potentielles. Ces solutions forment *l'espace de recherche* (ensemble des possibilités). La stratégie de recherche utilisée peut être exhaustive et examiner tous les plans possibles, ce qui peut conduire à un temps d'optimisation prohibitif. C'est pour cela que certains systèmes utilisent des *heuristiques.* La qualité des solutions trouvées dépend alors de la qualité des heuristiques choisies.

Dans le domaine de l'optimisation de requêtes, la recherche de la meilleure solution est rendue difficile par la nécessité d'un compromis entre le temps que prend la phase d'optimisation et la qualité des plans d'exécution générés (ce qui correspond au coût d'exécution de la requête) [LV91]. Dans notre cas, le temps que prend la phase d'optimisation n'est pas inclus dans le temps de chargement. Notre objectif est donc de développer une stratégie de recherche qui examine toutes les solutions possibles sans se soucier du temps qu'elle prend à le faire. Néanmoins, d'autres critères tels que celui qui consiste à éviter les solutions redondantes (*complexité en nombre d'instructions*) et celui qui consiste à optimiser la taille de l'espace de recherche (*complexité en mémoire utilisée*) sont pris en considération.

4.1 État de l'art

Il existe plusieurs types de stratégies de recherche utilisées dans le monde relationnel et objet. La stratégie *énumérative* considère quasiment toutes les solutions possibles et tente de réduire l'espace de recherche en appliquant des heuristiques. Plusieurs types d'algorithmes peuvent implanter une stratégie énumérative : largeur-d'abord (qui génère toutes les solutions possibles à partir de la solution la plus ancienne), meilleur-d'abord (qui développe la meilleure solution en premier) et profondeur-d'abord (qui génère toutes les solutions possibles à partir de la solution la plus récente). L'algorithme *branch-and-bound* donné dans [CP82] est aussi une variante de la stratégie énumérative. Les stratégies énumératives peuvent générer la meilleure solution mais risquent de causer une explosion combinatoire lorsque les requêtes à optimiser sont complexes [IK84]. L'optimiseur de System R [Sa79] par exemple, restreint l'espace au traitement d'arbres binaires et utilise *la programmation dynamique* pour trouver la solution optimale. La programmation dynamique est la stratégie de recherche utilisée dans les systèmes industriels. Elle examine toutes les possibilités de manière constructive. Elle construit chaque solution étape par étape.

Dans le but d'explorer des espaces de recherche plus importants, des stratégies dites *aléatoires* (*recuit simulé* [IW87], *améliorations itératives* [SG88], *optimisation à deux phases* [IK90]) ont été proposées. Elles sont généralement fondées sur une technique transformationnelle (par opposition aux techniques constructives). Alors que les stratégies énumératives considèrent tout l'espace de recherche, les stratégies aléatoires s'intéressent à trouver un *optimum local* autour de quelques points de départ. Elles sont effectuées en deux étapes. Primo, une ou plusieurs solutions sont obtenues en appliquant une recherche en profondeur-d'abord et en utilisant éventuellement des heuristiques. Secundo, ces solutions sont améliorées en appliquant des transformations jusqu'à trouver les meilleures. Ces stratégies nécessitent la définition de plusieurs paramètres tels que le nombre de transformations à appliquer, le critère selon lequel une solution est optimale, etc. Les transformations appliquées peuvent être spécifiées à travers un ensemble de règles comme dans [Fre87].
Le recuit simulé examine une partie de l'espace de recherche et tolère aussi bien les transformations qui baissent le coût de la solution courante que celles qui l'augmentent (avec une certaine probabilité appelée *température*). Cette probabilité diminue tout au long de l'algorithme. Lorsqu'elle devient très proche ou égale à zéro, l'algorithme de recherche s'arrête et la solution courante est délivrée.
La stratégie d'améliorations itératives effectue un grand nombre d'*optimisations locales*. Chacune de ces optimisations part d'une solution choisie aléatoirement qui est tranformée en une solution moins coûteuse jusqu'à l'obtention d'un optimum local. L'algorithme retourne l'optimum local le moins coûteux. Le critère d'arrêt utilisé pour cet algorithme est le temps.
La stratégie d'optimisation à deux phases est une combinaison des deux dernières [IK90]. Elle commence d'abord par appliquer une stratégie d'améliorations itératives pendant une période de temps relativement courte (quelques optimisations locales sont effectuées). En résultat, le meilleur minimum local trouvé est fourni. En deuxième phase, l'algorithme de recuit simulé est appliqué pour chercher les solutions générées à partir du minimum trouvé.

Selon [Ioa97], jusqu'à dix jointures, la programmation dynamique est la stratégie la plus intéressante parce qu'elle est plus rapide et qu'elle garantit une solution optimale. Pour des requêtes plus larges, malgré la nature probabiliste des algorithmes aléatoires, ils sont plus efficaces. Parmi ces algorithmes, la stratégie d'améliorations itératives trouve une solution raisonnable très rapidement. Le recuit simulé peut trouver une solution plus optimale en plus de temps. Enfin, l'optimisation à deux phases est capable de trouver des solutions qui sont aussi intéressantes que celles du recuit simulé en moins de temps.

Il existe plusieurs autres stratégies de recherche telles que l'algorithme A^* [YL89] utilisé en intelligence artificielle considéré comme une extension de la programmation dynamique [YL89] et les algorithmes génétiques [Gol89] qui généralisent les stratégies aléatoires et sont fondés sur des mécanismes d'échange appelés *cross over* et de *mutation*. La stratégie génétique réduit le temps de recherche de la solution optimale mais ne garantit pas la qualité de la solution trouvée. Des aperçus plus complets de tous ces algorithmes sont donnés dans [JK84, MCS84, Ioa97].

4.2 Objectifs

Tel que nous l'avons vu dans l'état de l'art (chapitre 1), il n'existe quasiment pas de techniques d'optimisation pour un système de chargement. Jusque là, le chargement de données d'un système à l'autre se faisait soit en écrivant des programmes ou en utilisant des «outils de connectivité» qui existent dans l'industrie. Prenons l'exemple de chargement de données, provenant de fichiers simples, dans une base de données à objets. Supposons que l'on veuille peupler la classe `Person` à partir des fichiers `hobbies` et `persons`. La classe `Person` est définie par les attributs "personName" de type *string*, "hobs" de type $\{string\}$ et "lives" de type `City`. Ci-après, les données des fichiers `hobbies` et `persons` :

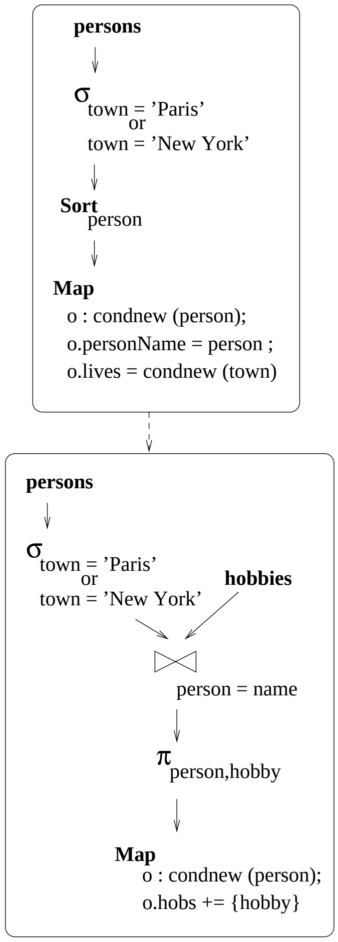
ρ_s(`hobbies`)

name	hobby
John	skying
Maria	reading
Georges	dancing
Maria	swimming
John	surfing
Helena	playing tennis
Helena	watching TV
Georges	swimming
John	reading
Mary	playing piano

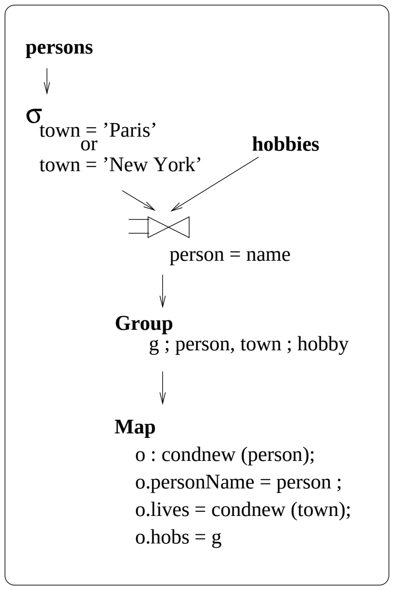
ρ_s(`persons`)

person	town
John	Paris
Maria	New York
Georges	New York
Helena	Paris
Marc	Paris
Mary	Vienna

Supposons que l'on veuille que la classe `Person` soit créée à partir de ces données et qu'elle contienne les informations de nom, loisirs et ville des personnes qui habitent «Paris» ou «New York». Cette classe peut alors être peuplée de différentes manières. Nous donnons deux exemples d'algorithmes qui permettent de le faire.



ALGORITHME I



ALGORITHME II

L'algorithme **I** est constitué de deux parties indépendantes. La première crée autant d'objets dans la classe `Person` qu'il y a de noms de personnes différents dans le fichier `persons` en sélectionnant bien auparavant les personnes habitant «Paris» ou «New York». À chaque objet créé, elle assigne la valeur de son attribut "personName" et celle de l'attribut "lives". La deuxième partie assigne la valeur de l'attribut "hobs" à chaque personne de manière incrémentale avec l'opérateur «+=». Une jointure est d'abord effectuée entre les fichiers `hobbies` et `persons` pour sélectionner dans `hobbies` les personnes qui nous intéressent. Puis, les identifiants de ces personnes sont extraits en utilisant la fonction *condnew* pour leur assigner la valeur de leur attribut "hobs". Cette fonction garantit que les objets ne sont pas créés deux fois. Après avoir entièrement exécuté ces deux parties, tous les objets de la classe `Person` sont créés et leurs attributs évalués.

L'algorithme **II** est constitué d'une seule partie à l'issue de laquelle tous les objets de la classe `Person` sont créés et leurs attributs évalués. Une jointure externe (⟕) est d'abord effectuée entre les fichiers `hobbies` et `persons`. À la différence d'une jointure classique, cette opération sélectionne dans le fichier `persons`, les personnes qui apparaissent dans `hobbies` mais laissent aussi passer la totalité des données sélectionnées dans le fichier `persons` en concaténant la valeur *null* aux personnes qui n'apparaissent pas dans `hobbies`. Ceci assure par la suite que tous les objets qui doivent être créés le sont. L'opération de regroupement

Group regroupe les loisirs de chaque personne ensemble de sorte que la mise à jour de l'attribut "hobs" des personnes se fasse avec l'opérateur «=». Les autres attributs sont mis à jour de la même façon que dans l'algorithme **I**.

Les algorithmes **I** et **II** ont le même effet sur la base de données cible (celui de créer tous les objets de la classe `Person` et de calculer leurs attributs). Ils sont donc «équivalents» par rapport à leur effet sur la base. Quelque soit le chargement auquel on s'intéresse, il existe toujours plusieurs algorithmes différents, chacun répondant à une logique particulière. Il existe aussi plusieurs variantes d'un même algorithme. Reprenons, par exemple, l'algorithme **II** et considérons que les fichiers sources sont plus importants (du point de vue du nombre de données qu'ils contiennent) que ceux de l'exemple (ce qui est sûrement le cas lorsque l'on fait du chargement en masse), nous pouvons par exemple, choisir de travailler sur des partitions de chacun des fichiers. Ce qui donnera autant de sous-parties du même algorithme qu'il y a de partitions. De plus, en considèrant que l'architecture du système de chargement s'articule autour d'un système de fichiers source, d'un composant intermédiaire et d'un système à objets cible, l'exécution d'un même algorithme peut se faire de manière «répartie» sur chacun de ces composants selon leur disponibilité (équilibrage de charge). Ci-après, nous dressons un tableau comparatif des algorithmes **I** et **II** selon certains critères qui permettent d'apprécier le coût global de chaque algorithme («+» signifie mieux, «-» signifie moins bien).

Critères	Algorithme **I**	Algorithme **II**
temps de communication	+	-
temps de traitement source	-	+
temps de traitement cible	-	+
volume de stockage	+	-
reprise après panne	+	-
organisation physique		loisirs regroupés par personne

Supposons que le chargement se fasse d'abord en sélectionnant les données qui nous intéressent dans le système source, en les important ensuite dans un composant intermédiaire chargé de faire quelques restructurations (essentiellement en utilisant l'opération *Group*) et enfin en créant les entités correspondantes dans le système à objets cible. L'algorithme **I** est alors plus intéressant du point de vue du *temps de communication* entre les systèmes source et intermédiaire puisque la quantité de données importées 'a la fois est relativement plus petite que dans l'algorithme **II**. De plus, dans la deuxième partie de l'algorithme **I**, une projection est introduite pour réduire le résultat de la jointure et n'importer que ce qui est nécessaire au calcul de l'attribut "hobs" ce qui diminue considérablement le temps de communication.

Le *temps de traitement source* est par contre plus intéressant dans l'algorithme **II** du fait que les fichiers sources sont utilisés une seule fois (nombre d'itérations sur le même fichier réduit et une seule jointure globale).

Le *temps de traitement cible* se reflète principalement dans le nombre d'entrées/sorties effectuées pour peupler la base cible. Il dépend du nombre de fois qu'on accède au même objet dans des programmes différents et de l'ordre dans lequel les objets sont accédés). Il est plus intéressant dans l'algorithme **II** puisque les objets créés sont accédés une seule fois pour les

créer et les mettre à jour, ce qui n'est pas le cas dans l'algorithme **I** (deux fois la même opération *condnew*). De plus, l'algorithme **I** crée les objets de la classe `Person` dans l'ordre ascendant de leur attribut "person" (opération *Sort* appliquée auparavant) et les met à jour dans la deuxième partie dans un autre ordre ce qui implique des accès aléatoires à la mémoire secondaire et peut être très coûteux en nombre d'entrées/sorties.
Pour ce qui est du *volume de stockage* que nécessitent les traitements intermédiaires, on peut penser que l'algorithme **I** est plus intéressant puisque le volume de données importées à la fois est moins important que dans l'algorithme **II**. Néanmoins, si nous considérons que le volume de stockage concerne non seulement la mémoire nécessaire aux traitements intermédiaires mais aussi la nécessité de conserver les tables de correspondance (ce qui en réalité est plus important en taille), l'algorithme **II** est plus intéressant. Ceci est dû au fait que dans cet algorithme, il n'est pas indispensable de conserver la table `(persons,Person)` puisque les objets de la classe `Person` ne sont accédés qu'une seule fois (création et mise à jour). Néanmoins, pour pouvoir peupler la classe `Persons` de manière incrémentale, cette table reste indispensable.
Si nous considérons la *reprise après panne*, l'algorithme **I** étant «naturellement» décomposé en deux parties indépendantes qui peuvent être assimilées à deux transactions différentes, les risques de perdre une partie du travail fourni pour cause d'interruption du chargement sont aussi amoindris.
Dans l'algorithme **II**, les loisirs d'une personnes sont regroupés sur disque avec cette personne (puisqu'ils lui sont assignés à sa création). L'algorithme **I** engendre une organisation différente des données sur disque. Dans certaines applications, l'organisation physique des données influe considérablement sur le temps d'exécution des programmes qui les manipulent et constitue un critère pertinent d'optimisation. Il est donc indispensable, pour éviter une réorganisation longue et coûteuse de la base générée, de le prendre en considération dans le choix de l'algorithme de chargement à utiliser.

À partir du tableau précédent, il apparaît que «trouver un algorithme optimal» revient à trouver un *compromis* entre deux aspects du chargement : *efficacité* et *flexibilité*.
Par efficacité, on entend les critères de temps et d'espace. En d'autres termes, il s'agit du temps de communication entre les systèmes utilisés pour le chargement, du temps de traitement sur le système source (nombre d'itérations sur la même source dans des programmes différents) et enfin, du temps de traitement sur la base cible (ex. nombre d'entrées/sorties, ordre dans lequel les objets sont créés versus ordre dans lequel ils sont accédés).
Par flexibilité, on entend d'une part, la possibilité laissée à l'utilisateur de spécifier des contraintes sur le peuplement de la base cible (contraintes de regroupement sur disque par exemple) et d'autre part, afin de faciliter la reprise sur panne, la décomposition du processus de chargement en plusieurs transactions.

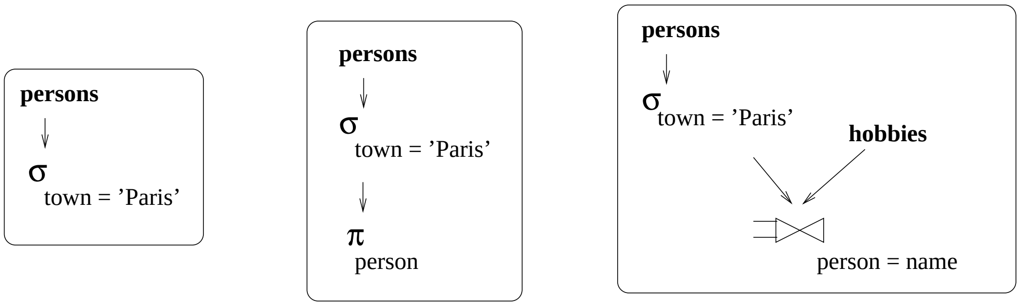
Étant donnés plusieurs algorithmes de chargement «équivalents», le système doit être capable de choisir «le meilleur». Dans la suite les programmes qui constituent un même processus de chargement seront appelés «blocs».

4.3 Paramètre DS et nouvelles équivalences

Nous avons vu qu'un algorithme de chargement peut être exprimé en utilisant les opérations de l'algèbre. Plusieurs algorithmes qui procèdent différemment les uns des autres peuvent avoir le même effet sur la base cible. Les expressions algébriques qui les représentent seront alors considérées non pas par rapport au flux de données qu'elles véhiculent (équivalence standard) mais par rapport à leur effet de bord. Également, il est important d'être capable de raisonner sur différents programmes faisant partie du même processus de chargement. Ces programmes sont rattachés les uns aux autres par l'effet que chacun d'eux a sur la base cible. Nous appelons ce lien «paramètre DS» pour Database State (ou état de la base).

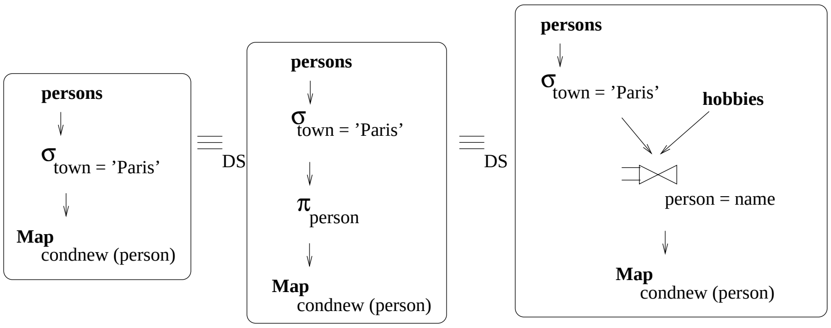
DS nous permet de définir une nouvelle classe de réécritures algébriques qui reposent sur les effets de bord. Ces nouvelles équivalences sont moins restrictives que les réécritures standards. En effet, deux expressions algébriques qui ne véhiculent pas le même flux de données, peuvent être DS-équivalentes (peuvent avoir le même effet sur la base). Si e_1 et e_2 sont deux expressions algébriques, $e_1 \equiv e_2 \Rightarrow e_1 \equiv_{DS} e_2$. L'implication n'est bien sûr pas vérifiée dans l'autre sens. Dans les figures, les flèches en pointillé représentent le DS et les autres le flux algébrique.

Soient les blocs suivants qui ne véhiculent pas le même flux de données.



I II III

Par contre, les blocs suivants sont équivalents :



I' II' III'

Regardons de plus près le résultat algébrique de chacun de ces blocs.

Bloc I

person	town
John	Paris
Helena	Paris
Marc	Paris

Bloc II

person
John
Helena
Marc

Bloc III

person	town	name	hobby
John	Paris	John	skying
John	Paris	John	surfing
John	Paris	John	reading
Helena	Paris	Helena	playing tennis
Helena	Paris	Helena	watching TV
Marc	Paris	NULL	NULL

Ces résultats ne sont effectivement pas équivalents. Néanmoins, en appliquant l'opérateur $Map_{condnew(person)}$ sur chacun de ces résultats, le même effet de bord est réalisé sur la base cible. Ceci est dû aux propriétés d'idempotence et d'insensibilité à la valeur nulle de la fonction *condnew* :
$condnew($`hobbies,Person,`$null) = nil$
et
$condnew($`hobbies,Person,`$val)$ exécutée plus d'une fois avec la même valeur val en paramètre, est équivalente à ne l'exécuter qu'une seule fois.

Pour pouvoir passer d'un algorithme de chargement à l'autre, nous avons défini des équivalences qui respectent l'état de la base. Il existe deux types d'équivalences DS : des *équivalences inter-blocs* qui concernent plusieurs blocs à la fois et des *équivalences intra-bloc* qui travaillent à l'intérieur d'un unique bloc. Dans ce paragraphe, nous donnons ces réécritures à travers des exemples. Les preuves de chacune d'elles sont détaillées en annexe B. Dans la suite, les entrées des jointures ont des noms d'attributs disjoints. Aussi, lorsqu'un nouvel attribut est créé dans le flux algébrique d'une expression, en appliquant une opération $Map_{a:f}$, cet attribut n'est pas utilisé pas les opérations qui suivent celle-ci. Les modifications apportées par une équivalence seront encadrées en pointillé.

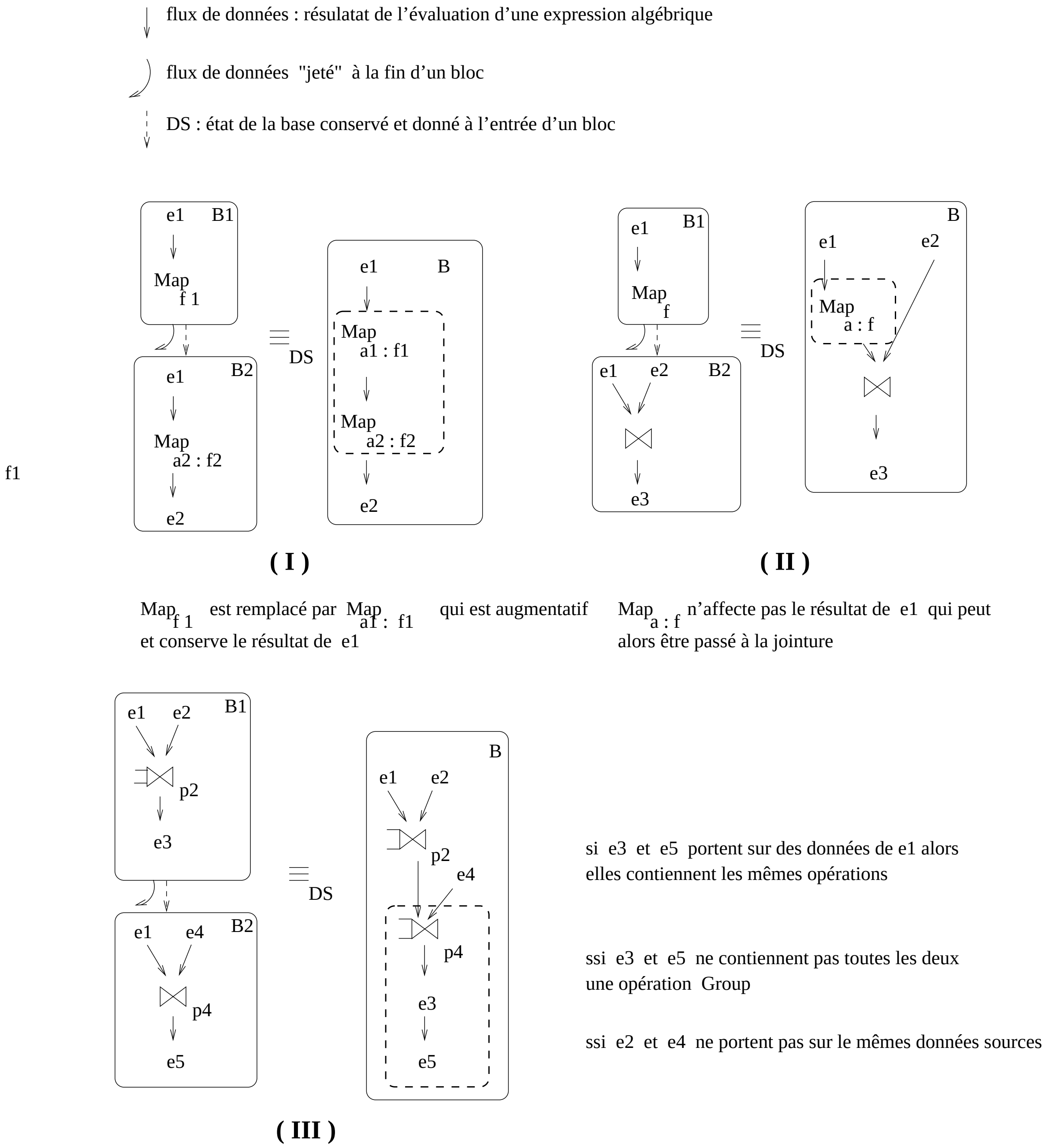
4.3.1 Équivalences inter-blocs

Une équivalence inter-blocs prend deux blocs en entrée et renvoie en sortie (i) ces mêmes blocs réordonnés, (ii) parallélisés ou (iii) un bloc unique dont l'exécution est équivalente à celle des deux blocs en entrée.

Réordonnancement de blocs. Grâce à la propriété d'idempotence, l'ordre d'exécution des blocs formant le même processus de chargement n'est pas significatif. Si nous reprenons l'algorithme **I** du paragraphe 4.2, les deux blocs qui le constituent peuvent être exécutés dans n'importe quel ordre. Soit on crée d'abord les objets de la classe `Person` avec leurs attributs "personName" et "lives" et on calcule ensuite l'attribut "hobs" de chaque personne soit on crée les personnes avec leur attribut "hobs" et on calcule, dans un deuxième temps, les attributs "personName" et "lives" de chaque personne. Ceci est rendu possible par le fait que la fonction *condnew* invoquée plusieurs fois avec la même valeur de l'attribut "person" ne crée l'objet correspondant qu'une seule fois. On verra, dans la deuxième partie de cette thèse, que si on s'intéresse au regroupement des objets créés sur disque ou à vouloir minimiser le nombre d'entrées/sorties dans la base à objets, il existe un ordre optimal des blocs.

Parallélisation de blocs. Deux blocs B_1 et B_2 sont parallélisables si et seulement s'ils ne contiennent pas de fonction de création des mêmes entités cibles. Par conséquent, si B_1 et B_2 ont en commun un couple $(source, cible)$, leurs prédicats de sélection sur les entités sources doivent être disjoints. Ceci garantit que les opérations qui suivent la sélection dans chacun des deux blocs ne portent pas sur les mêmes entités cibles.
Dans la parallélisation de blocs, toute la question est d'éviter d'accéder en écriture à une table de correspondance pour mettre à jour le même objet. Il faut donc garantir que deux blocs à paralléliser ne contiennent pas une opération de création avec les mêmes valeurs de paramètres (puisque c'est la seule qui est susceptible d'accéder à la même table de correspondance et à la même entité dans la base cible). Avec cette réécriture, si les systèmes sous-jacents offrent des possibilités de parallélisation, le temps global du chargement peut être optimisé.

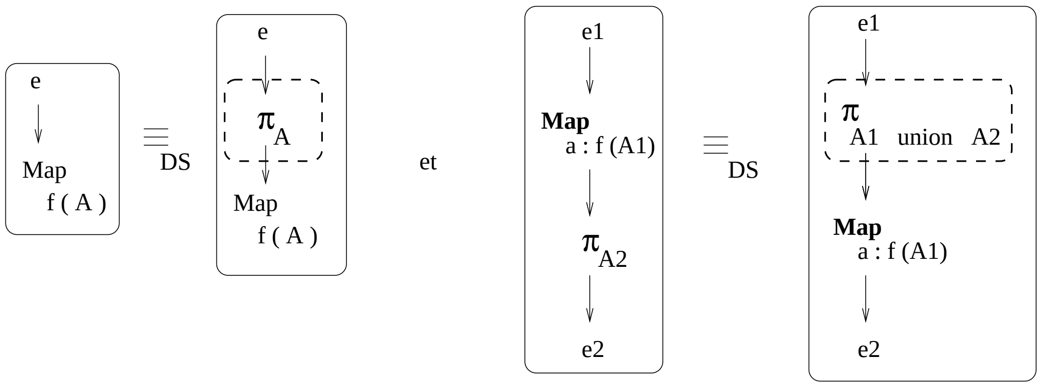
Regroupement de blocs. Afin de réécrire deux blocs B_1 et B_2 en un seul bloc B, nous utilisons plusieurs équivalences. Si B_1 et B_2 contiennent la même expression algébrique e_1 à laquelle des opérations Map sont appliquées, ces blocs peuvent être réécrits en un seul. Ceci est par exemple le cas de regroupement d'un bloc de création de classe avec un bloc qui calcule la valeur d'un attribut de cette classe (cas **(I)**). Si une expression algébrique e_1 du bloc B_1 apparaît dans une jointure d'un bloc B_2, B_1 et B_2 peuvent être réécrits en un bloc B (cas **(II)**). La dernière réécriture **(III)** est utilisée pour calculer les attributs complexes d'une même classe dans un seul bloc. Nous verrons des exemples de ces réécritures dans le cas relationnel-objet.



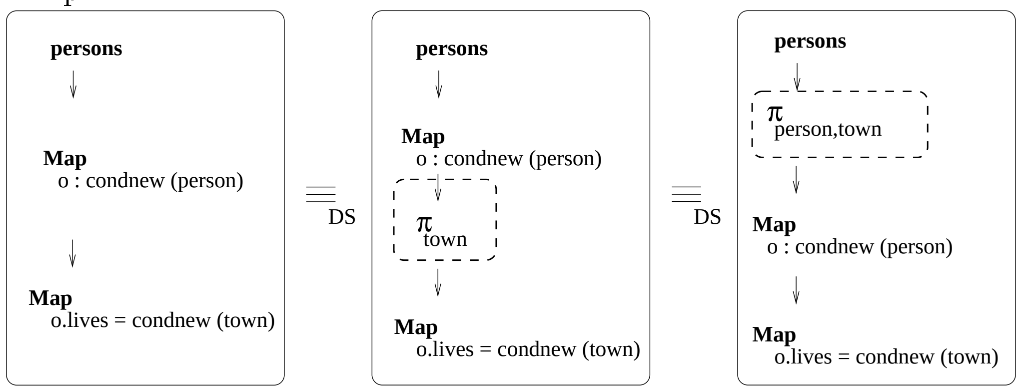
4.3.2 Équivalences intra-bloc

Une équivalence intra-bloc prend un bloc en entrée et délivre un ou plusieurs blocs en sortie. Elle est souvent utilisée avant ou après une équivalence inter-blocs. Dans cette catégorie, nous classons les équivalences : (i) insérer et pousser une projection, (ii) inverser une opération Map et une jointure, (iii) inverser une opération Map et un regroupement $Group$, (iv) combiner plusieurs Map, (v) remplacer un $Group$ par une assignation d'éléments dans un ensemble (+=), (vi) introduire une opération d'ordre $Sort$ et (vii) fragmenter un bloc en plusieurs. Dans la suite, A_i désigne un ensemble d'attributs (c'est en fait une autre écriture de $\vec{a}$ qui est plus facile à obtenir de manière graphique).

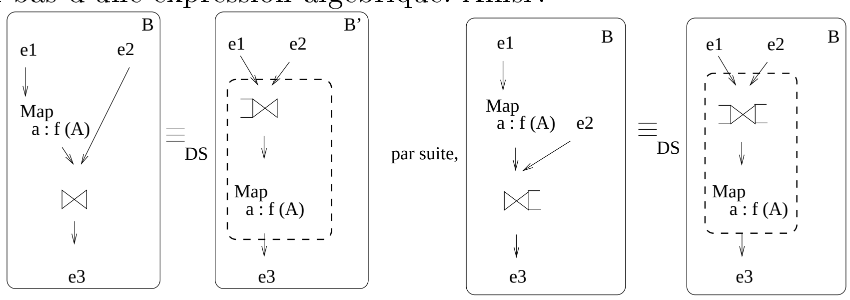
Insérer et pousser une projection. Cette équivalence est utilisée pour réduire le temps de communication entre les systèmes du chargement. Il est possible d'insérer une projection dans un bloc pour ne garder que l'ensemble minimal des données manipulées dans ce bloc. Cette opération est d'abord insérée avant la dernière opération *Map* du bloc. Elle est ensuite poussée vers le haut en récupérant dans la projection les attributs de l'opération qui la précède. Ceci s'écrit :



Par exemple :

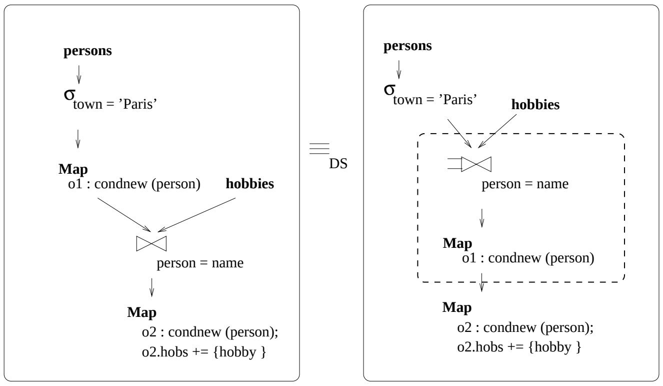


Inverser un *Map* et une jointure. Cette équivalence est utilisée pour pousser une opération *Map* en bas d'une expression algébrique. Ainsi :

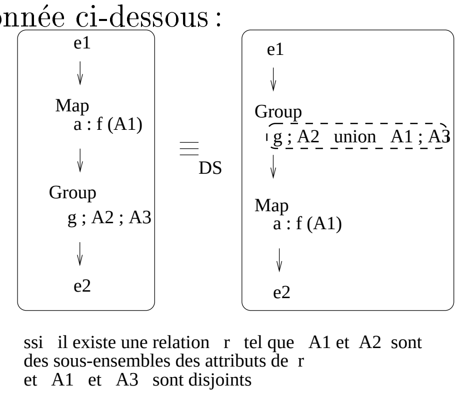


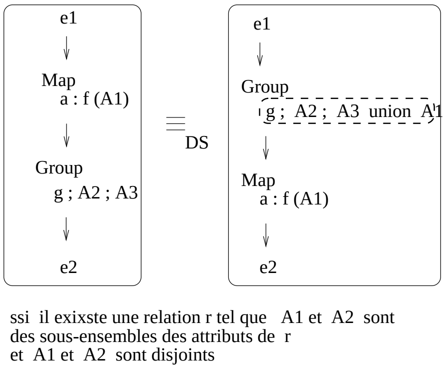
ssi les opérations de e3 portant sur le résultat de l'évaluation de e1 est $\text{Map}_{a\,:\,f\,(A)}$

Par exemple :

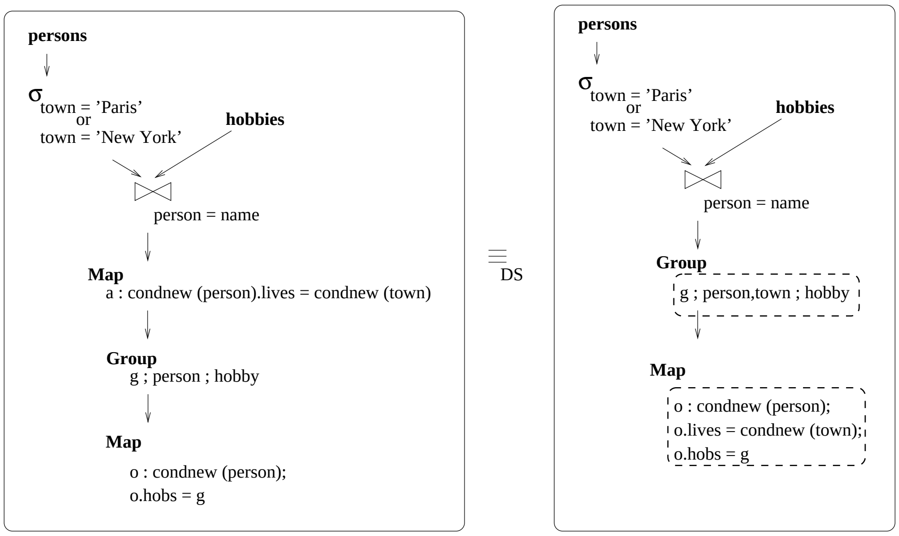


Inverser un *Map* **et un** *Group***.** Cette équivalence permet de pousser une opération *Map* en bas d'un bloc. Elle est donnée ci-dessous :



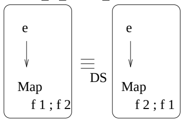


Par exemple :



Combiner plusieurs Map**.** On peut combiner des opérations Map dans un bloc de deux façons différentes :

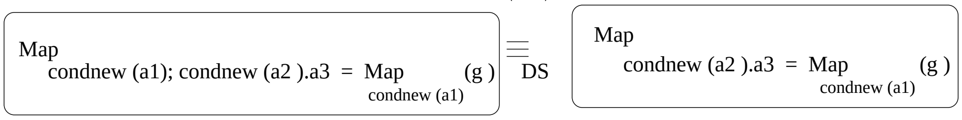
1. Soit en permutant les fonctions appliquées à travers cet opérateur :

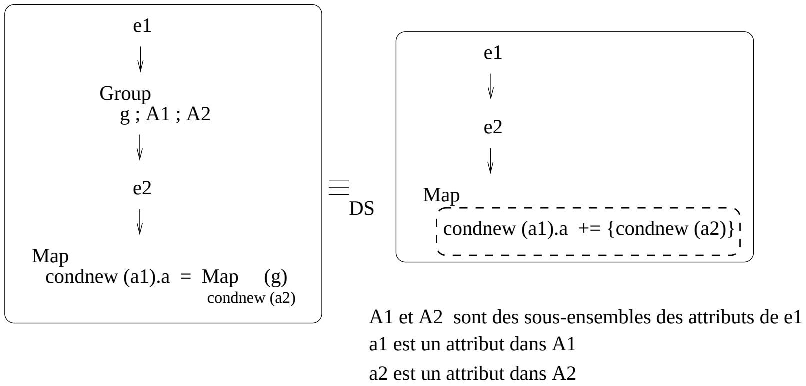
   ssi f1 ne crée pas des objets qui sont mis à jour dans f2

   Par exemple :
   $Map_{o.lives=condnew(town);o.hobbies=g}(e) \equiv_{DS} Map_{o.hobbies=g;o.lives=condnew(town)}(e)$
   par contre,
   $Map_{o:condnew(person);o.personName=person}(e)$ n'est pas équivalent à
   $Map_{o.personName=person;o:condnew(person)}(e)$ alors que
   $Map_{condnew(person);condnew(person).personName=person}(e)$ est équivalent à
   $Map_{condnew(person).personName=person;condnew(person)}(e)$

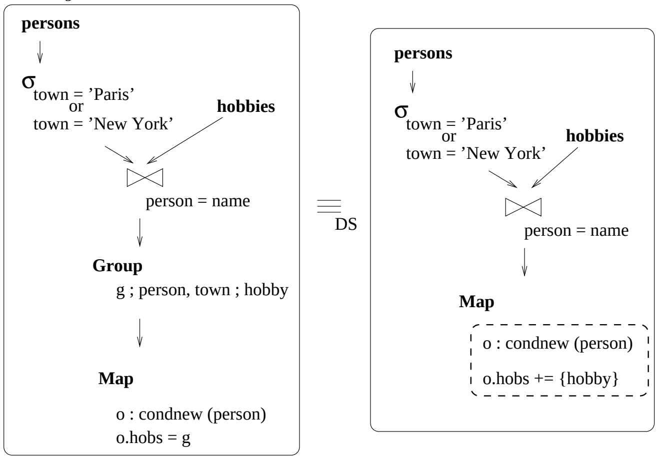
2. Soit en imbriquant des fonctions dans d'autres. Dans ce cas, l'opération $condnew(a_1)$ sera exécutée à travers le calcul de l'attribut a_3 même si l'objet $condnew(a_2)$ a la valeur nil (insensibilité à la valeur nulle de la fonction d'assignation). C'est ce qui nous autorise à supprimer la fonction $condnew(a_1)$.

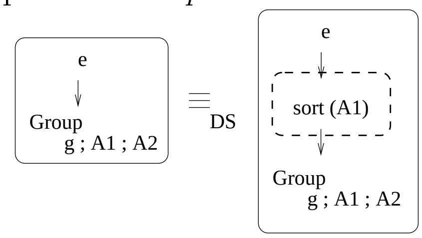
Remplacer un $Group$ **par un +=.** Elle est utilisée pour éliminer l'opérateur $Group$ dans un bloc. Elle peut être intéressante lorsqu'une mise à jour incrémentale d'une collection d'éléments est plus intéressante qu'un regroupement suivi de l'assignation d'une collection.



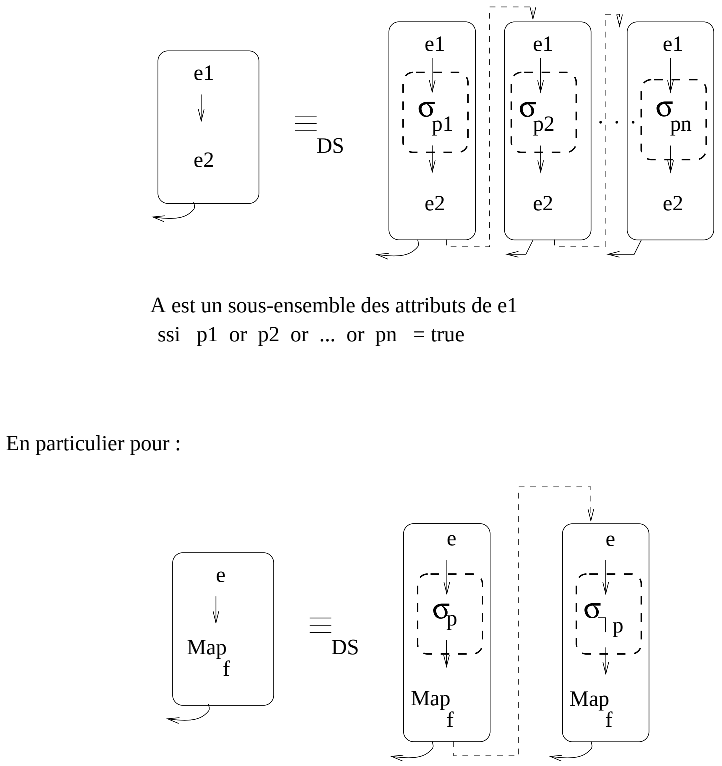
Elle est donnée de manière plus explicite sur l'exemple ci-dessous où l'opération $Group_{g;A1;A2}$ et l'assignation $Map_{o.hobs=g}$ sont remplacées par l'assignation $Map_o.hobs+=\{hobby\}$.



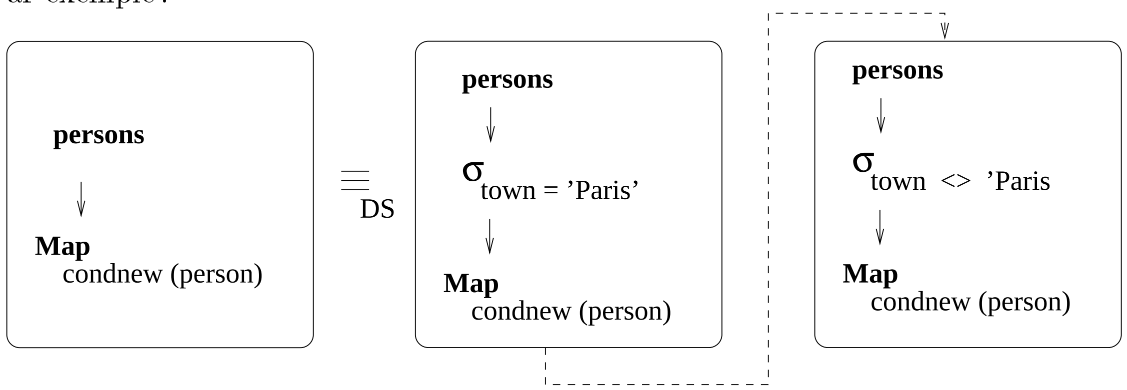
Introduire un ***Sort*****.** Elle est utilisée pour ordonner les éléments créés pour une entité cible. Nous verrons aussi que, pour des raisons d'efficacité, l'opération *Sort* sera insérée avant une opération de regroupement *Group*.



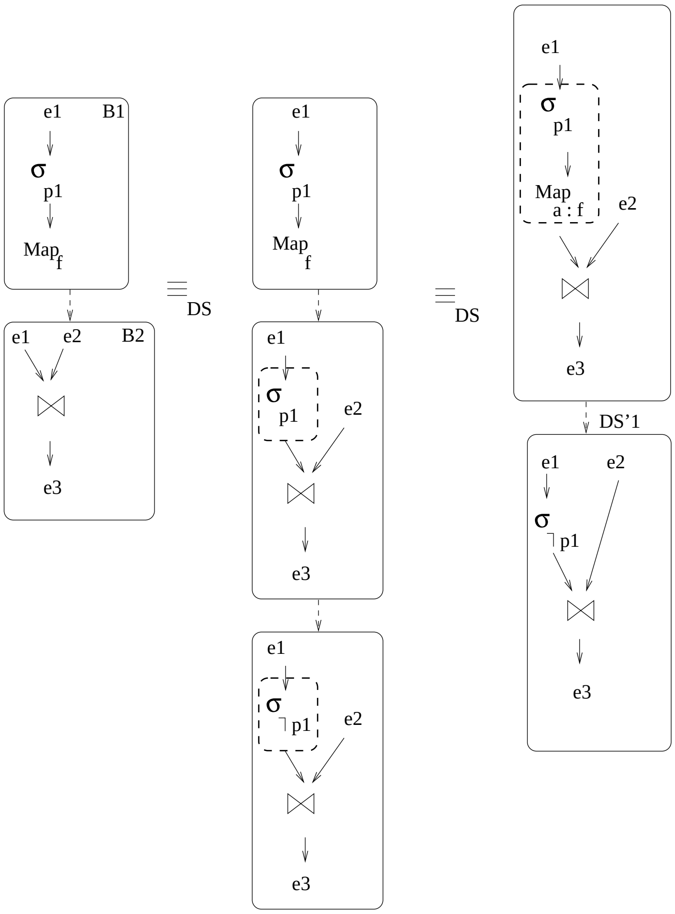
Fragmenter un bloc. Elle est utilisée pour traiter les éléments d'une même entité cible différemment.



Par exemple :



Cette équivalence est souvent utilisée afin de pouvoir appliquer une équivalence intra-blocs. C'est par exemple le cas lorsque des contraintes physiques doivent être réalisées sur une partie des objets d'une classe. Dans ce cas, il faut d'abord d'appliquer une équivalence intra-bloc qui fragmente un bloc en deux selon un critère de sélection et effectuer ensuite une équivalence inter-blocs. Ceci est donné ci-dessous :


e1
B1
σ
p1
Map
f
e1
e2
B2
e3
≡
DS
e1
σ
p1
Map
f
e1
σ
p1
e2
e3
e1
σ
┐p1
e2
e3
≡
DS
e1
σ
p1
Map
a : f
e2
e3
DS'1
e1
e2
σ
┐p1
e3

Deuxième partie

De la Migration Relationnel-Objet en Particulier

Afin de valider nos idées, nous avons choisi de les appliquer au cas relationnel-objet. L'algèbre définie précédemment nous permet d'exprimer de manière déclarative des transformations source-cible. Cependant, elle n'est pas adaptée à des utilisateurs de langages tels que SQL ou OQL. Nous avons donc conçu un langage plus convivial et qui se traduit facilement dans l'algèbre. Avec ce langage, l'utilisateur a toute la flexibilité des langages de vues pour créer classes et objets à partir de relations.
Notons également que le choix d'un langage autre que l'algèbre est motivé aussi par la volonté de donner à l'utilisateur un moyen simple d'exprimer, en plus des transformations relationnel-objet, des contraintes d'organisation physique des objets à leur création. Ceci fera l'objet du paragraphe 5.2.

Le chapitre 5 décrit le langage *RelOO* à travers un exemple simple de transformation relationnel-objet. Sa syntaxe complète est donnée en annexe A.
Le chapitre 6 porte sur l'algébraïsation du langage *RelOO* à des fins d'optimisation.
Le chapitre 7 décrit le prototype *RelOO* ainsi que la stratégie de recherche et le modèle de coût utilisés pour optimiser le chargement relationnel-objet.
Enfin, le chapitre 8 présente quelques résultats de tests de validation et de performance.

Chapitre 5

Le langage *RelOO*

5.1 Transformations Logiques

L'objectif de notre travail étant d'étudier les techniques d'optimisation du chargement en masse, nous avons défini un ensemble de transformations simples permettant néanmoins de couvrir la majorité des situations pratiques. Notre approche ne se soucie pas des propriétés sémantiques du schéma relationnel où de la façon de les obtenir. Elle n'est pas fondée sur une transformation automatique par défaut mais laisse à l'utilisateur la flexibilité de choisir les transformations qu'il désire réaliser. Elle vient alors compléter les approches de [Leb93, FV95, KFL95, Hai95b, MGG95, MR96, JSZ96, RH96, RH97, DA97, Fon97, BGD97, IUT98].

Nous avons sélectionné un certain nombre de primitives sur lesquelles le langage *RelOO* est conçu. Dans la suite, nous présentons chaque primitive en donnant son expression dans le langage et son effet de bord sur la base cible. Les transformations considérées sont :

- La création de classes à partir d'une ou de plusieurs relations.
- La création d'attributs atomiques ou complexes.
- La création de liens d'héritage entre classes.
- La création d'associations ODMG [CBB+97] entre classes (attribut complexe et son inverse).

Jusque là, nous avons vu un programme de chargement comme étant défini par la donnée d'un schéma cible S_t et d'un ensemble de transformations qui créent une instance cible I_t à partir d'une instance source I_s avec la condition que I_t est bien une instance de S_t. Pour des raisons de simplicité, le langage *RelOO* repose sur une spécification schéma-données. Le schéma cible est spécifié dans les primitives de transformation. À partir d'un programme unique, le système va générer schéma cible et programme de chargement des données. Nous allons voir comment le schéma est généré. La façon dont les programmes le sont sera expliquée dans le chapitre 7.

5.1.1 Exemple illustratif

Soit le schéma relationnel d'un centre de recherche qui gère des employés (chercheurs et administratifs confondus) répertoriés dans la relation *rEmployee* et des projets dans la relation *rProject*. La relation *rEmpPrj* représente une association de type n,m entre les employés et les projets : un projet concerne différents employés et un employé travaille sur plusieurs projets.

rEmployee[emp:integer,lname:string,fname:string,town:string,function:string,salary:real]
rProject[name:string,topic:string,address:string,boss:integer]
rEmpPrj[project:string,employee:integer]

Dans chaque relation, les attributs soulignés représentent sa clé. Du fait de l'utilisation de la fonction de Skolem *condnew* et de la possibilité de spécifier, dans le langage *RelOO*, n'importe quels attributs des relations sources comme clés de transformation, la notion de clé relationnelle n'est pas indispensable. Toutefois, si aucun autre attribut n'est donné en paramètre de la fonction de Skolem, celle-ci sera utilisée par défaut. si la clé d'une relation n'est pas donnée, l'ensemble de ses attributs seront considérés comme clé. La correspondance entre les paramètres de la fonction de Skolem et les objets créés permet de savoir pour une classe et une relation données, si l'objet correspondant à une valeur de clé de transformation a été créé dans la base à objets. Elle permet aussi de traiter les références cycliques entre objets. Enfin, elle permet de savoir à tout moment, ce qui a été migré et ce qui reste à migrer (chargement incrémental) et aussi de maintenir la cohérence entre les bases relationnelle et à objets dans le cas de mises à jour sur la base relationnelle.

5.1.2 Création de classes

Ci-dessous, nous donnons un programme qui contient des créations de classes en *RelOO* :

```
create class      Employee
from relations    rEmployee
with attributes   name:string is lname+fname,
                  salary:real is salary
extent            Employees
where             function ≠ 'Researcher'
                  and town = 'Rocquencourt'
end

create class      Researcher
from relations    rEmployee
inherits          Employee
extent            Researchers
where             function = 'Researcher'
                  and town = 'Rocquencourt'
end

create class      Project
from relations    rProject
with attributes   name:string is name,
extent            Projects
where             address = 'Rocquencourt'
end

create class      City
from relations    rEmployee[town]
with attributes   name:string is town
end
```

Une classe est créée à partir d'une ou de plusieurs relations. Les attributs de la classe sont dérivés d'attributs de ses relations sources en utilisant des fonctions simples (identité, concaténation de chaînes de caractères, fonctions arithmétiques,...). En fait, toute la puissance d'un langage comme OQL pourrait être utilisée mais ceci limiterait sérieusement les possibilités d'optimisation. Dans l'exemple précédent, la relation *rEmployee* a été partitionnée pour construire deux classes (`Employee` et `Researcher`) définie ci-dessus. La relation *rProject* est utilisée pour créer la classe `Project`. Seuls, les employés et projets situés à 'Rocquencourt' sont considérés. Ceci est spécifié dans le filtre de la clause `where`. La classe `Researcher` hérite de la définition des attributs de la classe `Employee` puisqu'ils ne sont pas redéfinis. L'héritage est seulement structurel. Les populations des deux classes sont spécifiées de manière disjointe. La clause `extent` sert à attribuer à la classe, un nom d'extension (ou racine de persistance) qui contiendra tous les objets créés dans cette classe lors de la phase de peuplement. Si un nom n'est pas attribué à l'extension, le système en attribue un par défaut. L'utilisateur peut également spécifier qu'il ne veut pas d'extension grâce à l'option `none` de la clause `extent` auquel cas, la classe sera peuplée (éventuellement à travers d'autres classes) mais, son extension ne sera pas maintenue. L'extension d'une classe est stricte. Elle ne contient pas les extensions de ses sous-classes. Par exemple, le nom `Employees` contiendra tous les employés habitant 'Rocquencourt' et qui ne sont pas des chercheurs tel que spécifié dans le filtre de la classe `Employee`.

La syntaxe générale de création d'une classe est donnée ci-dessous. Notons que lorsqu'une classe est créée à partir de plusieurs relations, son filtre ainsi que les fonctions de calcul de ses attributs sont spécifiés pour chacune des relations sources.

```
create class     c
inherits         c'
from relations   r_1[k_{1,1}, ...], ..., r_n[k_{n,1}, ...]
with attributes  a_1 : τ_1 is  f_{a_{1,1}} in relation r_1,
                               ...
                               f_{a_{1,n}} in relation r_n;
                 a_m : τ_m is f_{a_m,1} in relation r_1,
                               ...
                               f_{a_{m,n}} in relation r_n;
extent           e_c
where            p_1 in relations r_1,
                 ...
                 p_n in relations r_n
end
```

c est le nom de la classe à créer, c' la classe dont va hériter c, les $a_i, i \in [1, m]$ sont les attributs de la classe c, les r_j les relations sources de c, les p_j sont les prédicats qui définissent un filtre sur chaque relation source r_j et qui sont utilisés pour peupler la classe c. Pour tout j, $f_{a_{i,j}}$ sont des fonctions d'attributs relationnels permettant de calculer la valeur de l'attribut a_i. Il y a autant de fonctions pour un attribut a_i que de relations sources r_j pour la classe

c. Si d'autres attributs que ceux formant la clé de la relation r sont donnés comme clés de transformation (les $k_{i,j}$ dans la clause `from relations`), ils seront utilisés dans la fonction *condnew*. Ceci est le cas pour la définition de la classe `City` à partir de la relation *rEmployee* avec comme paramètre l'attribut *town* de cette relation.

Nous voyons bien que la définition ci-dessus donne aussi bien la structure de la classe c (son type) que son peuplement. De ce fait, le schéma cible est créé de façon incrémentale. Au départ, S_t est totalement vide. Chaque instruction *RelOO* l'enrichit. Ainsi, la transformation `create class` appliquée sur un schéma source $S_s = (\mathcal{R}_s, \emptyset, T_s, \_)$ enrichit le schéma cible $S_t = (\mathcal{R}_t, \mathcal{C}_t, T_t, \prec)$ de la façon suivante :

- $\mathcal{C}_t = \mathcal{C}_t \cup \{c\}$.
- $T(c) = T(c') \circ [a_i : \tau_i], \forall i \in [1, m]$.
- $c \prec c'$.

Cette transformation implique que les conditions suivantes soient vérifiées :

- $c \notin \mathcal{C}_t, c' \in \mathcal{C}_t$.
- Tous les attributs qui interviennent dans les fonction $f_{a_{i,j}}$ et dans la définition des prédicats p_j sont des attributs des relations r_j.
- Toutes les relations sources de la classe c doivent aussi définir sa super-classe c'.

Nous avons présenté jusque là la primitive `create class` qui permet de créer une classe à partir d'une ou de plusieurs relations, de créer plusieurs classes à partir d'une même relation, de créer des liens d'héritage entre classes et enfin de créer des attributs atomiques dans une classe. La primitive `create link` que nous présentons ci-dessous permet de spécifier des attributs complexes et des associations ODMG.

5.1.3 Création d'attributs complexes

Ci-dessous, nous donnons l'autre partie de notre programme de transformation qui contient des créations d'attributs complexes :

```
create link       Project,Researcher
from relations    rEmpPrj, rEmployee, rProject
with attributes   staff:set(Researcher)
                  in class Project,
                  projects:set(Project)
                  in class Researcher
where             employee=emp and project=name
end
```

```
create link       Project,Employee
from relations    rEmployee, rProject
with attributes   manages:Project
                  in class Employee
where             boss=emp
end
```

Un attribut complexe est un attribut ayant un type collection (liste ou ensemble d'éléments) ou une référence à un objet. Les collections peuvent elles-mêmes contenir des objets ou des structures plus complexes telles que données dans les expressions de type du modèle IQL. Si un attribut référençant une classe est défini avec son inverse, ils constituent à eux deux une association ODMG. De la même façon que les associations du modèle Entité-Relation, celles de l'ODMG sont de trois types : (1,1) lorsqu'elle est matérialisée par deux attributs dans deux classes chacun référençant un objet de l'autre classe, (1,n) lorsqu'elle est matérialisée par un attribut dans chacune des classes, l'un référençant un objet de l'une et l'autre une collection d'objets, et enfin (n,m) lorsque les deux attributs référencent chacun une collection d'objets de l'autre classe.

Les attributs complexes sont créés par la primitive `create link` du langage *RelOO*. Dans l'exemple, un lien matérialisé par `manages` et une association matérialisée par un attribut `staff` et son inverse `projects` sont définis. Le lien `manages` associe à chaque employé le projet qu'il gère. Il est créé dans la classe `Employee` et référence la classe `Project`. Les objets de la classe `Researcher` étant des employés, ils héritent ce lien. L'association quant à elle est définie entre les projets et les chercheurs et associe à chaque projet l'ensemble des chercheurs qui y travaillent et vice versa. Afin de peupler cette association, la relation *rEmpPrj* est utilisée. Nous verrons que cette relation sera d'abord filtrée pour ne garder que les employées qui sont chercheurs et qui habitent 'Rocquencourt' ainsi que les projets qui se trouvent à 'Rocquencourt'. Ces sélections sont possibles grâce aux filtres des classes `Researcher` et `Project` et à l'utilisation de jointures. Dans cet exemple, tous les attributs complexes sont de type ensemble. L'utilisateur peut bien sûr construire des attributs complexes de n'importe quel type (ex. list(real), set(integer), ...) en donnant les conditions appropriées dans la clause `where` de la commande `create link`. La syntaxe générale de cette commande est :

`create link`	c_1, c_2
`from relations`	$r_{1,1}, r_{2,1}, r_1;$
	$\ldots$
	$r_{1,n}, r_{2,n}, r_n$
`with attributes`	$a_1 : \tau_1$ `in class` c_1,
	$a_2 : \tau_2$ `in class` c_2
`where`	p_1 `in relation` $r_{1,1}, r_{2,1}, r_1;$
	$\ldots$
	p_n `in relation` $r_{1,n}, r_{2,n}, r_n;$
`end`	

où $\forall i \in [1, n]$, les relations $r_{1,i}$(resp. $r_{2,i}$) sont les relations sources de la classe c_1(resp. c_2). La relation r_i n'est pas toujours indispensable et dépend des types de liens à créer (cas de la relation *rEmpPrj*). a_1 est l'attribut créé dans la classe c_1, a_2 est l'attribut créé dans la classe c_2 et les p_i sont les prédicats de jointure des $r_{1,i}, r_{2,i}, r_i$. Il s'agira d'un attribut sans inverse lorsque seulement un des deux attributs a_1 ou a_2 est défini, d'une association lorsque les deux attributs sont donnés dans un même `create link`. Le schéma cible S_t résultant est donné par :

- $T(c_1) = T(c_1) \circ [a_1 : \tau_1], T(c_2) = T(c_2) \circ [a_2 : c_1]$ où $\circ$ représente la concaténation de n–uplets.

sous la condition que les attributs qui apparaissent dans la définition des prédicats p_i sont des attributs des relations $r_{1,i}, r_{2,i}, r_i$.

5.2 Organisation physique

Une fois la base de données à objets spécifiée logiquement, l'utilisateur peut vouloir donner la façon dont les objets doivent être organisés sur disque pour améliorer l'accès aux données. Afin d'éviter une réorganisation ultérieure de la base, le système *RelOO* lui donne la possibilité d'organiser ses données sur disque avant même de peupler la base i.e. au moment de définir les transformations relationnel-objet. Ces spécifications constituent des contraintes qui seront prises en considération lors de la phase d'optimisation du chargement. Ces contraintes sont de deux types : ordonnancement et regroupement d'objets sur disque.

5.2.1 Ordonnancement d'objets sur disque

Le modèle ODMG [CBB+97] (et la plupart des systèmes à objets) manipulent les listes et les tableaux qui sont utilisés pour accéder aux objets dans un ordre logique prédéfini. Une liste (ou un tableau) d'objets est le plus souvent représentée sur disque comme une suite d'identifiants. Si l'ordre de ces objets sur disque est différent de l'ordre logique de la liste, l'accès à la mémoire secondaire peut être aléatoire et donc coûteux du fait de nombreux défauts de page. Afin d'éviter cela, le langage *RelOO* offre une primitive permettant de spécifier un ordre de stockage que les objets d'une classe doivent respecter. Aucune hypothèse n'est faite sur le SGBD objet sous-jacent. L'idée principale est qu'il lui sera plus aisé de respecter un ordre de création spécifié par l'utilisateur si les objets sont créés en respectant cet ordre. Ceci est le cas pour le système O_2.

5.2.2 Regroupement d'objets sur disque

Les techniques de regroupement d'objets sur disque ont été largement étudiées et sont fournies par la majorité des SGBD objet existants ([BDK92, TN92]). L'objectif du regroupement est de minimiser les entrées sorties en réduisant le nombre de pages disque que le système doit lire pour accéder à des objets en naviguant dans la base. Le langage *RelOO* offre des primitives permettant de spécifier un placement d'objets et un mécanisme de réécritures qui prend en considération ces spécifications. Là encore, aucune hypothèse n'est faite quant à la politique de placement du système à objets sous-jacent. L'idée principale est que le SGBD objet a de fortes chances de respecter une instruction de regroupement si tous les objets à regrouper sont créés ensemble dans la même transaction. Ceci est vrai pour le système O_2 [VMa] mais aussi pour ObjectStore [Obj] par exemple.

Soient les deux exemples de contraintes utilisateur ci-dessous. La première spécifie un regroupement entre chaque projet de type 'Network' de la classe `Project` et l'ensemble constitué par les chercheurs travaillant dans ce projet et ce à travers son attribut `staff`. Des regroupements plus profonds peuvent être donnés à travers des attributs complexes. Par exemple, la deuxième contrainte spécifie un regroupement entre chaque projet de la classe

`Project` et les chercheurs de ce projet. Cette fois-ci, ce ne sont pas seulement les identifiants des chercheurs qui seront regroupés avec leur projet mais aussi les enregistrements physiques qui leurs sont associés. De plus, cette deuxième contrainte définit un ordre descendant sur les noms de projets selon lequel les objets de la classe `Project` doivent être créés. L'ordre est par défaut ascendant.

`class` Project `if` topic='Network' `cluster on` staff	`class` Project `order by` name `desc` `cluster on` staff*

Un regroupement est défini pour une classe c et regroupe des objets de c avec des objets d'autres classes qui sont atteignables par des liens d'aggrégation (i.e. des attributs complexes référençant des objets). Il peut aussi servir à regrouper ensemble les objets d'une même classe. L'utilisateur peut aussi spécifier une contrainte de regroupement partiel des objets d'une classe avec d'autres objets grâce à la clause `if`, telle que spécifiée ci-dessus.

La syntaxe générale de contraintes données par l'utilisateur est la suivante :

`class`	$c[r_1, \ldots, r_n]$
`order by`	$\vec{a_1}$ `in relation` r_1
	$\ldots$
	$\vec{a_n}$ `in relation` r_n
`if`	p_1 `in relation` r_1
	$\ldots$
	p_n `in relation` r_n
`cluster on`	$a_1^*(a_{1,1}(\ldots), \ldots a_{1,m}(\ldots)),$
	$\ldots$
	$a_m^*(a_{m,1}(\ldots), \ldots)$

où c est la classe de regroupement, les $\vec{a_i}$ sont des attributs utilisés pour l'ordre et sont donnés pour chaque relation source considérée, p_i est le prédicat de regroupement pour la paire (r_i, c). $\forall i \in [1, n]$. $\forall j$, les a_j sont les attributs complexes de c et les $a_{j,k}, \forall k$ sont les attributs des classes atteignables par c à travers chaque a_j. Le regroupement peut être arbitrairement profond.

Le partage d'objets rend impossible de garantir les regroupements. Par exemple, sachant qu'un chercheur appartient à plusieurs projets, il ne sera réellement regroupé sur disque qu'avec un seul projet. Les instructions de regroupement sont seulement des «indications» à respecter autant que possible. L'ordre dans lequel elles sont données est donc important. et doit être respecté.

Jusque là, nous avons vu que le langage *RelOO* offre des primitives simples permettant de spécifier et de combiner des transformations relationnel-objet. Il permet également d'exprimer des contraintes physiques à réaliser sur la base de données à objets. Dans le chapitre suivant, nous expliquerons comment chaque primitive du langage *RelOO* est traduite dans l'algèbre afin de faire de l'optimisation.

Chapitre 6

Optimisation

Dans ce chapitre, nous allons d'abord voir comment les transformations exprimées dans le langage *RelOO* sont traduites dans l'algèbre. Nous nous intéresserons ensuite à illustrer les techniques d'optimisation développées au chapitre 4 à travers des réécritures algébriques appliquées à notre exemple de chargement relationnel-objet.

L'originalité de la technique d'optimisation que nous proposons réside dans le fait qu'elle intègre des réécritures standards [Ull88, CM94] avec de nouvelles réécritures qui prennent en considération les effets de bord sur la base cible. Ceci nous permet de réaliser les contraintes physiques exprimées par l'utilisateur et d'optimiser le chargement. Elle propose aussi de fragmenter le chargement en plusieurs parties pouvant être exécutées indépendamment les unes des autres afin (i) d'éviter de bloquer les systèmes sous-jacents pendant une longue période, (ii) de perdre le minimum de données si une interruption survient pendant le chargement et enfin (iii) de libérer régulièrement la mémoire utilisée.

Dans la dernière partie de ce chapitre, nous verrons que les facteurs qui rentrent en considération durant la phase de réécriture sont parfois contradictoires et nécessitent de trouver un compromis satisfaisant.

6.1 Traduction algébrique des primitives *RelOO*

Le langage *RelOO*, ainsi que nous l'avons présenté dans le chapitre précédent, est composé de deux types de primitives :

- *des primitives de création de classes* dans lesquelles sont exprimées la création d'une classe, d'attributs dans cette classe, chaque attribut ayant une requête de calcul par relation source, de liens d'héritage avec d'autres classes et du filtre permettant le peuplement de la classe créée à partir de ses relations sources,

- *des primitives de création de liens entre classes* dans lesquelles sont exprimées les créations d'attributs complexes et les créations d'associations en spécifiant les deux classes pour lesquelles les attributs sont créés, les relations entre lesquelles les jointures sont effectuées pour calculer les valeurs des attributs complexes ainsi que les prédicats de jointure pour chaque triplet (ou paire) de relations sources.

L'algèbre sert de support de base à notre phase d'optimisation. Chaque primitive du langage *RelOO* est traduite en une séquence d'opérations algébriques.
Une primitive de création de classe génère autant d'expressions algébriques qu'il y a de relations sources et de filtres associés à cette classe. Si c est une classe définie à partir d'un ensemble de relations $\{r_1, \ldots, r_n\}$ avec les prédicats associés $\{p_1, \ldots, p_n\}$, il y aura autant d'expressions algébriques qu'il y a de triplets (c, r_i, p_i) avec $i \in [1, n]$.
De la même façon, pour les primitives de création d'attributs complexes entre classes, il y aura autant d'expressions algébriques qu'il y a de 7-uplets $(c_i, r_i, c_j, r_j, r_{i,j}, a_{i,j}, p_{i,j})$ différents où $a_{i,j}$ est un attribut de la classe c_i qui référence la classe c_j et r_i (resp. r_j) est la relation source de c_i (resp. c_j). $r_{i,j}$ est une relation tierce (de type n,m) qui peut intervenir dans l'évaluation de l'attribut $a_{i,j}$. $p_{i,j}$ est le prédicat de jointure entre les relations impliquées.

Si un lien a est défini entre une classe c_1 et c_2, alors les sous-classes de c_1 héritent de ce lien. Ceci signifie que plusieurs blocs de calcul de la valeur de cet attribut seront générés (un pour c_1 et un pour chacune de ses sous-classes). Si c_2 a des sous-classes et que l'attribut a contient une collection d'objets, sa valeur pour chaque objet de la classe c_1 doit comprendre les objets de la classe c_2 et ceux de ses sous-classes. Par conséquent, des blocs seront générés pour les sous-classes. L'opération d'assignation «+=» permet alors de calculer la valeur de cet attribut de manière incrémentale.

Chaque expression algébrique est appelée *bloc* (c'est le terme utilisé dans la partie I de cette thèse pour désigner un programme du processus de chargement). L'ensemble des blocs générés définissent *la représentation algébrique par défaut* du processus de chargement. Nous considérons la partie de l'exemple donné dans le chapitre précédent qui porte sur la création des classes `Researcher` et `Project` et de l'association `staff/projects`. La figure 6.1 contient deux blocs : $\mathbf{B}_{\text{Res}}$ et $\mathbf{B}_{\text{Prj}}$. $\mathbf{B}_{\text{Res}}$ (resp. $\mathbf{B}_{\text{Prj}}$) peuple la classe `Researcher` (resp. `Project`). Dans la suite, une expression algébrique est schématisée par un arbre dont les nœuds sont constitués de relations sources et d'opérateurs algébriques.

Le bloc $\mathbf{B}_{\text{Res}}$ sélectionne d'abord l'ensemble des employés chercheurs qui habitent 'Rocquencourt'. Cette opération correspond au filtre de la classe `Researcher` exprimé dans la clause `where` de sa définition. Ensuite, une opération *Map* est appliquée pour créer un objet par n-uplet sélectionné[1]. Cette opération mémorise le nouvel objet (attribut o). Elle utilise la fonction de Skolem *condnew* avec comme paramètre l'attribut clé *emp* de la relation *rEmployee* qui est pris par défaut puisqu'aucun autre attribut n'est donné dans la définition de la classe `Researcher`. Comme nous l'avons vu dans le chapitre précédent, cette fonction repose sur l'utilisation d'une table de correspondance créée pour la paire (*rEmployee*,`Researcher`). Cette table associe les identifiants des objets créés dans `Researcher` aux valeurs de l'attribut *emp*, paramètre de la fonction de Skolem. Nous verrons qu'elle est toujours indexée sur les paramètres de la fonction de Skolem. Les tables de correspondance que nous utilisons deviennent importantes en taille et demandent des techniques de stockage et d'accès

1. Pour alléger les schémas, nous avons simplifié les expressions algébriques. Par exemple, $condnew(r, c, emp)$ est remplacée par $condnew(emp)$

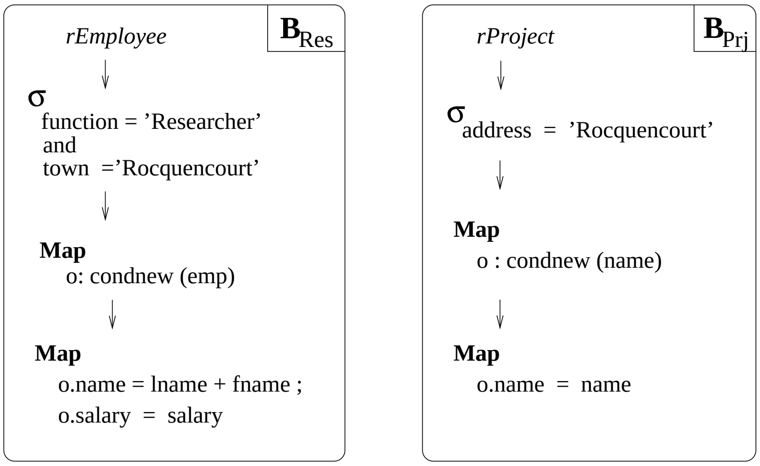


FIG. 6.1 – *Création des Chercheurs et des Projets*

comme dans [WN95]. Nous verrons que certaines de nos réécritures nous permettent d'éliminer quelques-unes d'entre elles ce qui représente un gain de temps et d'espace considérable. Enfin, une opération Map suit pour mettre à jour les valeurs des attributs `name` et `salary` des objets de la classe `Researcher`.

Considérons à présent les blocs $\mathbf{B}_{\text{staff}}$ et $\mathbf{B}_{\text{projects}}$ qui calculent l'association `staff/projects` (figure 6.2). Un bloc est créé pour chaque attribut complexe. Le bloc $\mathbf{B}_{\text{staff}}$ contient deux jointures qui permettent de filtrer les n-uplets de la relation *rEmpPrj*. Ceci évite de créer des associations entre des objets qui ne devraient pas appartenir à la base cible (par exemple, les employés chercheurs qui n'habitent par 'Rocquencourt' ou qui ne travaillent pas dans des projets situés à 'Rocquencourt'). Les chercheurs sont ensuite regroupés par projet grâce à l'opérateur *Group*. Celui-ci a trois paramètres : le nom de l'attribut qui contient les chercheurs regroupés (g), le nom de l'attribut de regroupement ($name$) et l'attribut sur lequel chaque groupe résultant est projeté (emp). Un ensemble g de chercheurs est constitué pour chaque valeur distincte de l'attribut $name$. L'opération Map qui suit assigne à l'attribut `staff` de chaque objet de la classe `Project` ($condnew(name)$) sa valeur (un ensemble d'objets de la classe `Researcher`). Cette valeur est calculée en appliquant la fonction $condnew(emp)$ à chaque élément de l'ensemble g (opération Map imbriquée).

De la même façon, le bloc $\mathbf{B}_{\text{projects}}$ évalue l'ensemble des projets d'un chercheur ($condnew(emp).projects$). Les attributs `staff` et `projects` étant inverse l'un de l'autre, ils ont exactement la même partie relationnelle. La seule différence réside dans l'opération *Group* qui, dans un cas regroupe les chercheurs par projet et dans l'autre, les projets par chercheur. Nous verrons qu'il sera envisagé de factoriser des parties de blocs communes. Pour des raisons de simplicité, aucun lien d'héritage n'est spécifié entre les classes de l'association. Si tel est le cas, les attributs complexes d'une classe sont hérités par toutes ses sous-classes. Nous en verrons des exemples dans le chapitre 8.

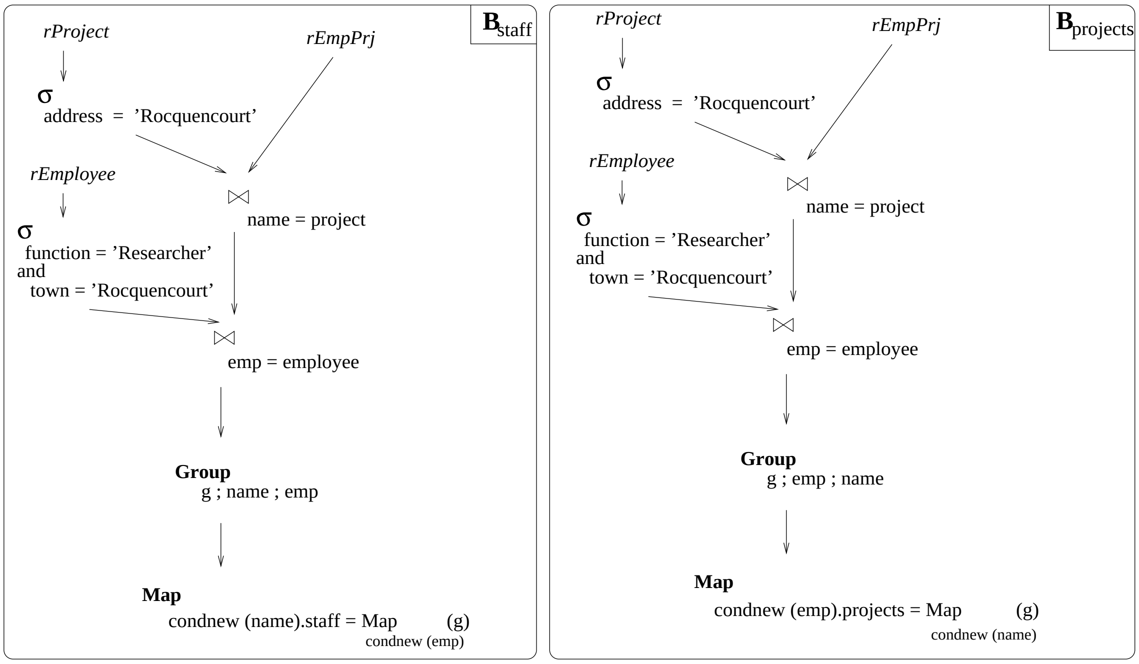


FIG. 6.2 – *L'association staff/projects*

6.2 Paramètre DS

La représentation algébrique par défaut correspond à une évaluation «naïve» du processus de chargement [2]. Elle est constituée d'un ensemble de blocs dont certains créent des objets et d'autres leur assignent des valeurs. Les blocs générés peuvent être exécutés dans n'importe quel ordre. Ceci est rendu possible grâce à la fonction de Skolem de création d'objets que nous avons défini. Cette fonction permet de garder trace des objets créés pour garantir que pour une paire (r, c) où r est une relation source de la classe c, seulement un objet est créé pour chaque valeur distincte des clés de transformation. Néanmoins, afin de pouvoir appliquer nos réécritures, nous choisissons un ordre des blocs par défaut. Cet ordre reflète l'ordre dans lequel les primitives du langage *RelOO* sont données et sera modifié lors de la phase d'optimisation (nous verrons que certains ordres sont plus intéressants que d'autres).

En rajoutant l'ordre, la représentation algébrique par défaut devient une liste ordonnée de blocs distincts. En l'état, il est impossible de détecter et de réaliser des réécritures concernant plus d'un bloc à la fois (par exemple, une réécriture qui remplacerait $\mathbf{B}_{\mathrm{Prj}}$ et $\mathbf{B}_{\mathrm{staff}}$ par un seul bloc afin de créer les projets et calculer leur valeur pour l'attribut `staff` en même temps). Notre solution est de fournir des liens sémantiques explicites entre les blocs. Le flux de données d'un bloc peut être insignifiant pour un autre (ex. nous n'avons pas besoin du flux de $\mathbf{B}_{\mathrm{Prj}}$ pour exécuter $\mathbf{B}_{\mathrm{Res}}$) et ce qui nous intéresse à la fin d'un bloc est son impact (ou effet de bord) sur la base cible. Par conséquent, le lien entre les blocs doit représenter l'état logique de la base de données cible. C'est le paramètre DS (Database State) défini

2. Nous verrons plus loin qu'un algorithme naïf peut être approprié dans certains cas.

dans le chapitre 4. La figure 6.3 montre l'évolution du paramètre DS entre les opérations d'un même bloc et entre les blocs.

Si nous considérons à nouveau la création des chercheurs, des projets et de l'association entre eux. Nous pouvons d'abord commencer par les créer dans l'ordre (les chercheurs, les projets ensuite l'association). Nous pouvons aussi exécuter les blocs des attributs `staff` et `projects` qui non seulement créeront les chercheurs et les projets mais calculeront aussi les valeurs de l'association entre eux. Les blocs $\mathbf{B}_{\mathrm{Res}}$ et $\mathbf{B}_{\mathrm{Prj}}$ exécutés plus tard vont créer le reste des chercheurs et projets (ceux qui ne sont pas liés les uns aux autres) et calculer les valeurs des attributs atomiques de tous les chercheurs. Le paragraphe 6.4 contiendra des exemples concrets d'application de ces réécritures. Nous verrons qu'en appliquant des réécritures successives, nous pouvons déduire un unique bloc dont l'exécution est DS-équivalente à l'exécution de tous les blocs de la représentation par défaut.

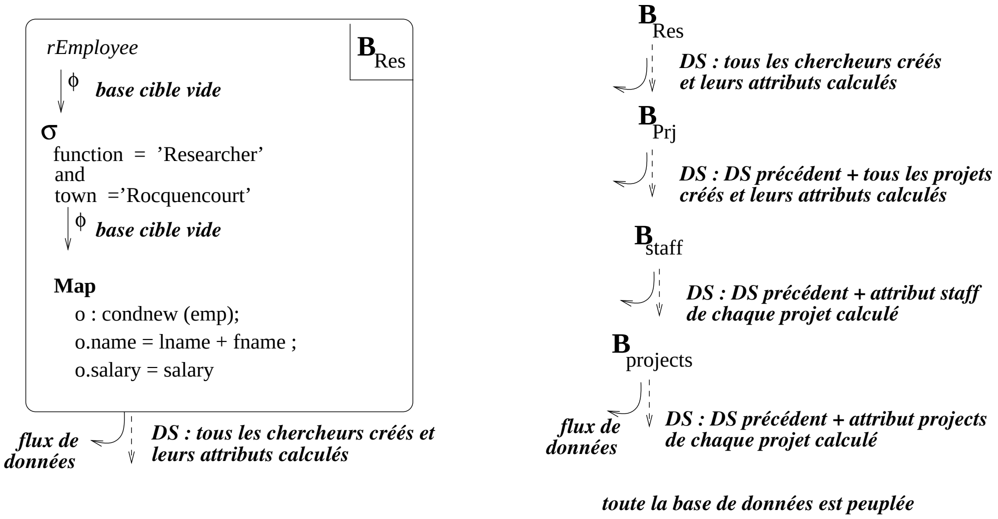


FIG. 6.3 – *Paramètre DS*

Le paramètre DS représente l'évolution de l'état de la base cible. DS n'est rien d'autre que l'ensemble des objets créés dans la base à un instant donné et les valeurs de leurs attributs. La partie gauche de la figure 6.3 montre l'évolution de l'état de la base pour le bloc $\mathbf{B}_{\mathrm{Res}}$. À noter que la seule opération qui modifie le paramètre DS est l'opération Map (puisque c'est elle qui effectue les effets de bord).

À la fin d'un bloc, son résultat algébrique est «jeté» alors que DS est passé en entrée du bloc suivant. Ceci est illustré dans la partie droite de la figure 6.3 qui montre le DS obtenu après chaque bloc de la représentation algébrique par défaut. Dans cette figure, nous avons choisi d'ordonner les blocs dans l'ordre d'apparition des primitives qui les ont générés.

6.3 Facteurs de la réécriture

Malgré le fait que l'objectif principal de la réécriture soit d'optimiser le temps global du chargement, certains facteurs qui influent sur l'équilibrage de charge, les temps de communication et de traitement ainsi que le volume de stockage doivent être pris en considération. Nous montrons comment ces facteurs vont guider la phase de réécriture. Nous verrons que la représentation algébrique par défaut peut être intéressante dans certains cas.

6.3.1 Où effectuer les traitements?

Le chargement de données consiste en trois phases. (i) Une requête est évaluée sur le système source et le résultat est récupéré dans le composant intermédiaire, (ii) un traitement local peut éventuellement suivre et enfin (iii) une transaction est exécutée dans le système cible pour peupler la base de données. Ceci est illustré sur la figure 6.4. Il est possible d'imaginer que le composant intermédiaire est partie intégrante du système source ou cible. Du point de vue de l'optimisation, la différence se ressentirait dans les coûts de communication.

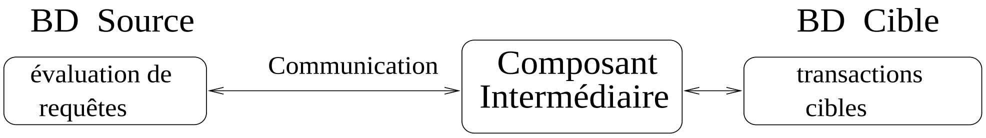


FIG. 6.4 – *Les trois phases du chargement*

Étant donnée cette architecture, nous pouvons imaginer plusieurs façons de décomposer le processus de chargement. Une décomposition naturelle des blocs $\mathbf{B}_{\text{staff}}$ et $\mathbf{B}_{\text{projects}}$ serait de considérer que les opérations *Map* sont effectuées dans des transactions du côté cible, que les opérations de *Group* sont exécutées localement (dans le composant intermédiaire) et que toutes les autres opérations sont effectuées sur le site source. Une autre décomposition serait de choisir d'exécuter toutes les opérations, excepté *Map*, dans le composant intermédiaire (en supposant que les algorithmes de sélection, projection et jointure soient développés localement et que les relations sources soient chargées dans ce composant). Enfin, un autre choix serait de tout effectuer sur le site cible après y avoir importé les relations sources. Il existe, bien sûr, plusieurs autres alternatives. Le choix d'une solution plutôt qu'une autre dépend de différents facteurs : la disponibilité des systèmes source et cible, le coût de chaque opération sur les différents systèmes, la capacité du composant intermédiaire à stocker et à gérer de grands volumes de données et enfin le coût et la fiabilité de la communication entre les systèmes sous-jacents.

La décision de favoriser une solution plutôt qu'une autre a des conséquences directes sur le processus de réécriture (la phase d'optimisation) et a une influence importante sur le coût global du chargement. Par exemple, dans le cas du chargement relationnel-objet, si nous décidons de pousser autant d'opérations que possible sur le site relationnel, la réécriture devra séparer les opérations non relationnelles (un regroupement qui contient des ensembles et les opérations *Map*) des opérations relationnelles. Si nous décidons par contre d'effectuer tous les traitements en dehors du système relationnel et du système à objets, le coût global du chargement pourrait être augmenté à partir du moment où les techniques d'optimisation relationnelles comme par exemple les index pour les jointures, ne seront pas utilisées. Néanmoins, ce choix peut être intéressant puisqu'il minimise les coûts de communication ainsi que la charge des traitements sur le système source.

6.3.2 Quand effectuer les traitements ?

Les systèmes source, intermédiaire et cible ne sont pas toujours disponibles. Il est donc impératif de décomposer le processus de chargement en parties pouvant être exécutées quand c'est possible et pour une durée déterminée. Pour décomposer un programme, nous pouvons charger toutes les données sources dans le composant intermédiaire, y effectuer tous les traitements et ensuite lancer une séquence de transactions jusqu'à ce que toutes les données soient traitées. Cette approche ne bénéficie pas des techniques d'optimisation relationnelles et peut avoir un coût prohibitif. L'autre approche serait de générer des requêtes qui chargent seulement la partie des données nécessaires à une transaction (décomposition d'une requête en plusieurs requêtes plus petites). L'avantage de cette solution est que, contrairement à la précédente, elle est flexible en termes d'équilibrage de charge. Les traitements peuvent être évalués à n'importe quel endroit. Bien sûr, elle a aussi un coût qui doit être pris en considération par le processus d'optimisation : n petites jointures peuvent être plus coûteuses qu'une seule. C'est cette solution que nous avons adopté. Nous avons choisi de générer plusieurs programmes de chargement, chacun étant paramétré par le temps (ou le nombre d'entités à traiter). L'utilisateur peut alors invoquer un programme en donnant une valeur de temps ou de nombre fixe.

6.3.3 Organisation physique des données

Tel que nous l'avons expliqué dans le chapitre précédent, nous voulons être capables, lorsque la base de données cible est de type objet, de spécifier des contraintes de regroupement et d'ordonnancement afin de stocker les objets sur disque d'une manière intéressante pour les traitements futurs. Considérons la commande de regroupement suivante selon laquelle les chercheurs doivent être stockés par projet :

```
class Project
cluster on staff
```

Une façon de réaliser ce regroupement serait que le système objet connaisse le nombre de chercheurs par projet avant de créer chaque projet afin d'être capable de réserver un espace approprié. Mieux encore, avoir toutes les informations sur les projets et les chercheurs qui leur sont liés au même moment, devrait permettre de regrouper les objets sur disque.

Regardons de plus près la représentation par défaut. Elle crée les chercheurs d'abord, les projets ensuite, avant d'évaluer l'association `staff/projects`. Le système n'est évidemment pas en position de garantir la contrainte de regroupement ci-dessus sans réorganiser le stockage des objets. L'expression par défaut doit donc être réécrite pour regrouper les trois blocs $\mathbf{B}_{\mathrm{Res}}$, $\mathbf{B}_{\mathrm{Prj}}$ et $\mathbf{B}_{\mathrm{staff}}$ en un seul.

Un autre exemple de regroupement pourrait être de décomposer l'ensemble des projets en deux parties pour permettre des regroupements différents des objets d'une même classe. La commande *RelOO* suivante spécifie que seuls les chercheurs travaillant sur des projets de type 'Network' doivent être regroupés avec leur projet :

```
class Project
if topic='Network'
cluster on staff
```

Par conséquent, les projets qui ne satisfont pas le prédicat topic='Network' ne doivent pas être regroupés avec leurs chercheurs. Enfin, l'ordre spécifié dans :

```
class Project
order by name
```

implique l'addition d'une opération *Sort* dans le bloc $\mathbf{B}_{\mathrm{Prj}}$.

Nous verrons plus loin, à travers des exemples, comment les équivalences inter-blocs, la décomposition des objets d'une classe et l'ordonnancement des objets sur disque peuvent être réalisés (figures 6.5, 6.6 et 6.7). Pour le moment, nous nous contentons de commenter l'influence de ces opérations sur le coût global du chargement et par conséquent sur la phase de réécriture. Considérons encore une fois la possibilité de réécrire les trois blocs $\mathbf{B}_{\mathrm{Res}}$, $\mathbf{B}_{\mathrm{Prj}}$ et $\mathbf{B}_{\mathrm{staff}}$ en un seul et de ce fait, créer les projets, les chercheurs et l'association `staff/projects` ensemble. Bien sûr, excepté les créations et mises à jour d'objets, tout travail de restructuration pourrait se faire dans le composant intermédiaire. Une autre alternative serait d'effectuer la jointure entre les trois relations sur le système relationnel et de fournir au composant intermédiaire les informations nécessaires à l'évaluation des attributs des classes `Project` et `Researcher`. Notons que, dans ce cas, le coût de traitement va être réduit (moins d'itérations sur les relations sources puisque seul un bloc subsiste). Pourtant, il est à noter aussi que (i) le temps de communication global va être augmenté (une grande jointure est plus coûteuse à communiquer que la somme de deux sélections et d'une petite jointure), (ii) le coût de l'opération *Group* sera plus important (puisque effectué sur un ensemble plus important) ainsi que (iii) le coût de stockage, et enfin (iv) la flexibilité de la décomposition en plusieurs transactions est amoindrie puisque tout est effectué dans un bloc unique.

6.3.4 Optimisation

Jusque là, nous avons discuté des paramètres de la réécriture et avons étudié leur influence sur le processus de chargement. Considérons maintenant les techniques d'optimisation.

Le coût d'un chargement est évalué en termes de (i) communications, (ii) volume de stockage, et (iii) traitement source, intermédiaire et objet. Tous ces coûts doivent bien sûr être minimisés.

Le moyen le plus simple et le plus efficace de réduire le temps de communication est d'effectuer toutes les sélections et les projections sur le site source et seulement ces opérations (pas les jointures puisque ce sont elles qui coûtent le plus en termes de communication). Ensuite, le résultat est chargé dans le composant intermédiaire qui peut effectuer toutes les opérations algébriques restantes. En termes de réécritures, cela implique seulement de garder les sélections et les projections dans la partie supérieure des expressions algébriques (ne pas les mélanger avec les autres opérations). Pourtant, ceci peut affecter sérieusement le temps de traitement puisque les index du système relationnel, s'il y en a, ne peuvent pas être utilisés pour effectuer les jointures. De plus, ceci demande un stockage local potentiellement important. La stratégie de recherche et le modèle de coût, que nous présenterons dans le chapitre 7, tiennent compte de cela. Par exemple, conformément aux exigences de l'application, les coûts de communication peuvent avoir un poids plus ou moins important dans le modèle

de coût.
Le volume de stockage doit être pris en compte par le processus de réécriture soit comme un paramètre du modèle de coût (comme c'est le cas dans le système *RelOO*) ou comme faisant partie de la stratégie de recherche. Le système stocke aussi bien les tables de correspondance source-cible que les résultats temporaires. Si nous voulons maintenir la cohérence entre les données sources et cibles dans le cas de mises à jour, toutes les tables doivent être conservées. Si ce n'est pas le cas, il est possible de supprimer certaines tables. Nous verrons cela à travers des exemples (ex. 6.4.1). Une autre possibilité qui permettrait de réduire le volume de stockage local serait d'éviter certains traitements (comme le regroupement). Nous verrons que c'est ce qui a été adopté dans le prototype *RelOO*.

Certaines opérations dans les blocs effectuent des mises à jour sur la base cible et doivent donc être évalués sur celle-ci. D'autres peuvent être effectuées soit par le système cible soit dans le composant intermédiaire (les regroupements qui construisent des ensembles). Enfin, certaines d'entre elles peuvent être effectuées sur les trois composants (ex. les sélections). Dans ce qui suit, nous donnons les principales techniques d'optimisation qui s'appliquent à ces différents types d'opérations.

- Le seul moyen d'optimiser les opérations de mise à jour d'une base (opération Map) est de prendre en considération les entrées/sorties. Nous verrons que certaines expressions algébriques effectuent des accès aléatoires aux données (ex. la représentation par défaut de l'exemple relationnel-objet) alors que d'autres le font de manière séquentielle.

- Le nombre de tables de correspondance peut être réduit. Nous verrons que ceci peut se faire en appliquant des équivalences inter-blocs.

- Si aucune contrainte de regroupement sur disque n'est demandée, les opérations $Group$ peuvent être supprimées et remplacées par une utilisation itérative de l'opération d'insertion d'un élément dans un ensemble «+=». Ceci est bien sûr plus coûteux en termes d'entrées/sorties. Mais nous verrons que nous pouvons utiliser l'ordre des objets sur disque et dans les tables de correspondances pour minimiser ce coût (voir exemple 6.4.1).

- L'algèbre permet l'application de règles de réécriture traditionnelles [Ull88, CM94] mais aussi de nouvelles règles fondées sur des équivalences DS. En utilisant les premières, nous pouvons réécrire toutes les opérations algébriques. Nous pouvons aussi appliquer des optimisations sémantiques pour réduire le temps de traitement. Par exemple, si nous savons que tous les chercheurs de 'Rocquencourt' appartiennent à des projets de 'Rocquencourt', nous pouvons éliminer la jointure avec la relation *rProject* des blocs $\mathbf{B}_{\text{staff}}$ et $\mathbf{B}_{\text{projects}}$ de la représentation par défaut. En utilisant des équivalences DS, nous pouvons par exemple, diminuer le nombre d'itérations sur les entités sources et aussi le nombres d'entrées/sorties. En outre, si le système cible supporte le parallélisme, les équivalences DS peuvent aussi être utilisées pour prendre en considération la parallélisation de blocs.

6.3.5 Pour conclure...

Plusieurs facteurs doivent être pris en considération durant le processus de réécriture. Certains d'entre eux sont contradictoires. Un chargement peut avoir un coût global élevé et peut être cependant «intéressant» parce qu'il minimise le temps de communication ou le volume de stockage ou favorise des regroupements intéressants, etc.

Reprenons par exemple la représentation par défaut de l'exemple précédent. Ses avantages principaux sont qu'elle réduit les coûts de communication et facilite la décomposition en transactions à partir du moment où le chargement est composé de plusieurs blocs. De plus, en mémorisant localement les résultats de jointures apparaissant dans différents blocs (factorisation), les temps de communication et de traitement relationnel peuvent se voir réduits. Pourtant, (i) le temps de traitement objet n'est pas optimisé à cause des accès fréquents aux tables de correspondances, (ii) cette représentation ne garantit pas les regroupements sur disque donnés plus haut en exemple (regroupement des chercheurs par projet).

6.4 Exemples de réécritures

Ce paragraphe contient trois exemples qui illustrent quelques représentations algébriques équivalentes (qui ont le même effet sur la base cible). Ces exemples partent de la représentation par défaut du cas relationnel-objet. Nous ne donnons pas les différentes réécritures qui nous ont permis de les obtenir, l'objectif étant de discuter les avantages et inconvénients de chaque représentation.

6.4.1 Regroupement de blocs

La première réécriture (voir figure 6.5) réécrit les quatre blocs $\mathbf{B}_{\mathrm{Res}}$, $\mathbf{B}_{\mathrm{Prj}}$, $\mathbf{B}_{\mathrm{staff}}$ et $\mathbf{B}_{\mathrm{projects}}$ dans un bloc unique $\mathbf{B}_{\mathrm{all}}$ qui crée tous les objets et les associations. Les opérations de jointure ont été transformées en jointures externes. Celles-ci garantissent la présence des chercheurs et projets qui n'apparaissent pas dans la relation *rEmpPrj* mais qui doivent quand même être créés. Si nous avions une information supplémentaire selon laquelle tous les chercheurs qui habitent 'Rocquencourt' et les projets situés à 'Rocquencourt' sont liés, les jointures externes pourraient être remplacées par des jointures (équivalence sémantique). Nous avons également introduit une projection pour minimiser le temps de communication. L'opération *Group* de l'attribut `projects` a été supprimée et remplacée par une insertion d'éléments dans un ensemble «+=». Les chercheurs sont créés et mis à jour dans une opération *Map* imbriquée. Nous avons groupé toutes les opérations *Map* en une seule. Enfin, les projets sont ordonnés selon leur nom.

Le Bloc $\mathbf{B}_{\mathrm{all}}$ a cinq caractéristiques : (1) la table de correspondance (*rProject*,`Project`) peut être supprimée sachant que tous les projets sont créés dans ce bloc et qu'il n'y a pas de cycle. La table (*rEmployee*,`Researcher`) doit néanmoins être conservée. Un chercheur peut travailler dans plusieurs projets et peut par conséquent être consulté à plusieurs reprises $condnew(emp)$. (2) Le nombre de jointures et d'itérations sur les relations sources est réduit dans ce bloc. (3) Nous ne posons aucune hypothèse sur la localisation des traitements : ainsi, les jointures peuvent être évaluées sur le site relationnel, intermédiaire ou du côté objet.

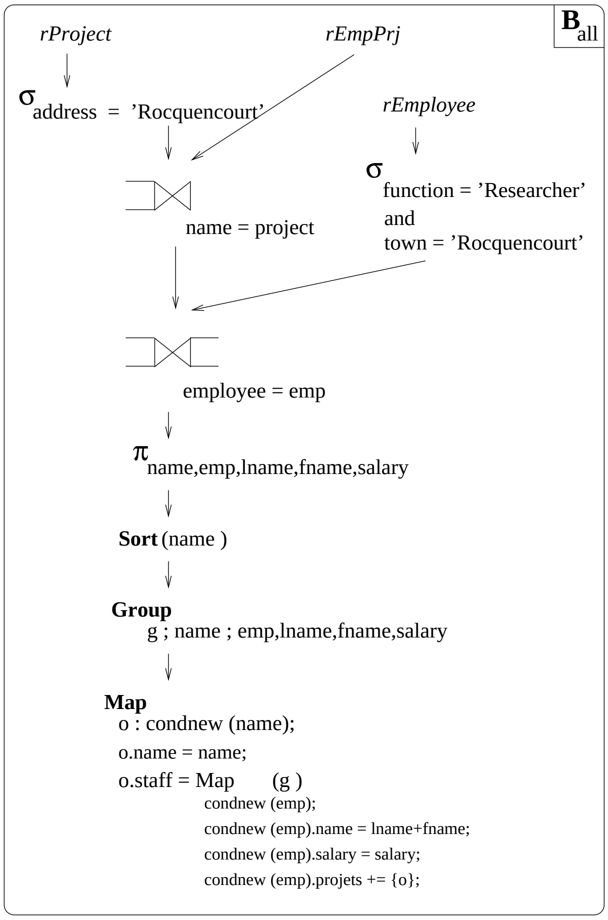


FIG. 6.5 – *Regroupement de blocs*

(4) Le temps de communication est relativement réduit par l'introduction de la projection. (5) En supposant que le chargement est incrémental, l'intégrité référentielle de l'association `staff/projects` est maintenue (à tout moment, un projet référencé par un chercheur, le référence lui-même). Enfin, (6) en regroupant la création des chercheurs par projet, le nouveau bloc $\mathbf{B}_{\text{all}}$ réalise la contrainte physique suivante :

`class` Project `order by` name `cluster on` staff

Les aspects négatifs de cette réécriture sont : (1) l'opération *Group* est effectuée sur un ensemble plus large que dans la représentation algébrique par défaut. Ceci implique une mémoire plus importante et un coût de traitement du regroupement plus important. (2) Si les jointures sont effectuées sur le site relationnel, le coût de communication global sera plus important que celui de la représentation par défaut (ce qui rend la projection indispensable). Enfin, (3) le découpage du processus de chargement en plusieurs parties est plus difficile.

6.4.2 Décomposition et regroupement

Le deuxième exemple de réécriture est donné figure 6.6. Le bloc $\mathbf{B}_{\mathrm{Res}}$ a été réécrit avec une sous-partie des blocs $\mathbf{B}_{\mathrm{Prj}}$ et $\mathbf{B}_{\mathrm{staff}}$ reliés aux projets de type 'Network'. Par conséquent, à la fin du bloc $\mathbf{B}_{\mathrm{P/R/staff}}$, tous les chercheurs sont créés ainsi que tous les projets de type 'Network' et leur attribut `staff` est évalué. La partie restante de $\mathbf{B}_{\mathrm{Prj}}$ et $\mathbf{B}_{\mathrm{staff}}$ suit et l'expression complète se termine avec le bloc de la représentation par défaut $\mathbf{B}_{\mathrm{projects}}$. Le regroupement ci-dessous est réalisé :

```
class Project
if topic='Network'
cluster on staff
```

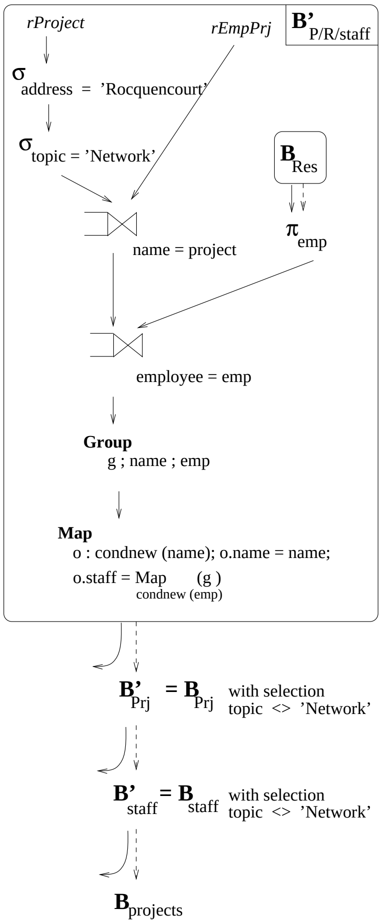


FIG. 6.6 – *Décomposition et regroupement*

Nous pouvons réécrire les trois derniers blocs de n'importe quelle façon sans pour autant affecter le regroupement demandé. Le bloc $\mathbf{B}_{\mathrm{P/R/staff}}$ contient une opération *Group* plus petite que celle de l'exemple précédent (seulement les paramètres de la fonction de Skolem sont regroupés). Le coût de communication est donc réduit. Néanmoins, cette réécriture force l'évaluation d'une jointure à l'extérieur du système relationnel ($\sqsupset\!\!\bowtie_{employee=emp}$) puisqu'elle est précédée par le bloc $\mathbf{B}_{\mathrm{Res}}$. Pour la même raison, le coût de stockage du côté objet peut être important (le résultat de $\mathbf{B}_{\mathrm{Res}}$ n'est pas jeté).

6.4.3 Manipulations de l'opérateur *Group*

Notre troisième exemple (voir figure 6.7) montre comment pousser une opération *Group* sur le site relationnel en utilisant une fonction agrégat *count*. La représentation algébrique consiste en trois blocs: le bloc $\mathbf{B}_{\mathrm{Prj}}$ qui crée les objets de la classe `Project`, $\mathbf{B'}_{\mathrm{Res}}$ qui crée des chercheurs et $\mathbf{B}_{\mathrm{staff/projects}}$ qui matérialise l'association entre les chercheurs et les projets.

Dans le bloc $\mathbf{B'}_{\mathrm{Res}}$, une opération *Sort* est introduite pour ordonner les chercheurs selon leur numéro (*emp*). Cette opérateur sert aussi à simplifier l'opération *Group* qui suit. Cette dernière associe à chaque chercheur le nombre de projets dans lesquels il travaille. Finalement, les chercheurs sont créés à travers une opération *Map*. Même si nous ne disposons pas de toutes les informations concernant les projets dans lesquels travaille un chercheur, nous connaissons la taille de leur attribut `projects`. Par conséquent, étant donnée la spécification de regroupement suivante, nous pouvons organiser les objets de manière appropriée:

```
class Researcher
order by desc emp
cluster on projects
```

Analysons maintenant les aspects positifs et négatifs de cette expression. (i) Elle garantit le regroupement ci-dessus et (ii) pousse une opération *Group* sur le site relationnel (à partir du moment où la fonction *count* est appliquée et qu'il ne s'agit pas d'une construction d'ensemble). En supposant que le système relationnel est plus efficace que les autres composants, cette réécriture est intéressante.

Malheureusement, cette nouvelle expression induit un traitement plus important sur le site objet ainsi que la nécessité de conserver deux tables de correspondance (pour le bloc $\mathbf{B}_{\mathrm{staff/projects}}$). Notons néanmoins que les chercheurs (qui constituent la table la plus grande), ont été ordonnés sur les paramètres de la fonction de Skolem utilisée pour leur création (*emp*). Nous pouvons donc supposer un accès séquentiel à la table des chercheurs ainsi qu'au disque pour éviter des entrées/sorties superflues.

Avec ces trois exemples, nous avons donné l'intuition d'un modèle de coût qui prendrait en compte certains paramètres afin de satisfaire le compromis escompté. Le chapitre suivant concerne le prototype *RelOO*. Il contient une présentation des différents modules et détaille la stratégie de recherche et le modèle de coût utilisés durant la phase d'optimisation pour le choix de la solution de chargement optimale.

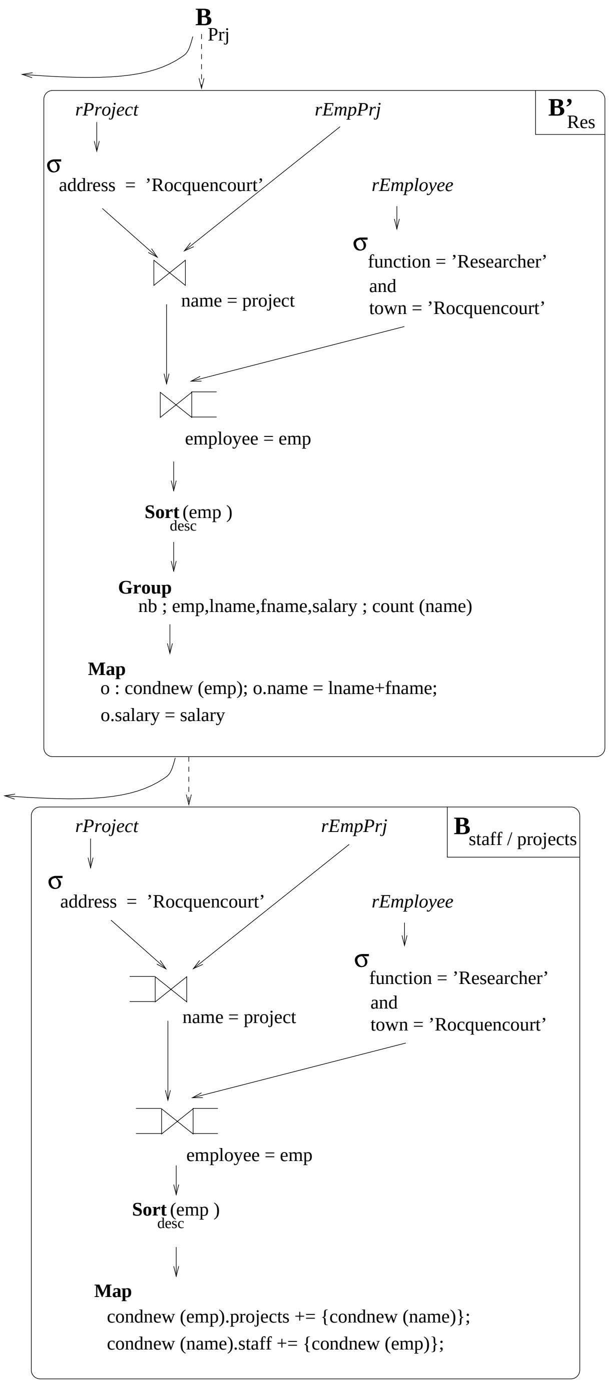


FIG. 6.7 – *Introduction d'un regroupement relationnel*

Chapitre 7

Le prototype *RelOO*

Dans ce chapitre, nous donnons l'architecture du prototype *RelOO* et détaillons chacun de ses modules. Nous présentons la stratégie de recherche et le modèle de coût développés. Nous décrivons ensuite la phase de génération de code. Nous terminons par l'environnement de contrôle du chargement dans lequel sont exécutés les programmes générés à l'issue de la phase d'optimisation.

7.1 Architecture

Le prototype *RelOO* est écrit en Java. Il est construit au dessus du SGBD O_2 [BDK92] du côté objet et utilise l'interface standard ODBC [Mic94] pour interroger n'importe quelle base de données relationnelle. La figure 7.1 présente une vue générale des modules qui le constituent. Elle est découpée verticalement en quatre parties. La partie gauche (resp. droite) contient le système relationnel (resp. le système objet). Le deuxième rectangle représente l'interaction de l'utilisateur avec le système : l'utilisateur donne un fichier de spécification et a ensuite la charge de lancer les programmes de chargement générés. Enfin, les modules du prototype *RelOO* sont illustrés dans le troisième rectangle. Ils sont au nombre de quatre : (i) *le module de traitement préliminaire* qui se charge de la génération du schéma objet et de la représentation algébrique par défaut, (ii) *le module d'optimisation du chargement*, (iii) *le module de génération de programmes* et enfin, (iv) *le module de contrôle de l'exécution des programmes générés*. Nous décrivons succinctement chacun de ces modules. Plus de détails sur les trois derniers seront donnés plus loin.

Traitement préliminaire. Les transformations relationnel-objet et les contraintes d'organisation physique sont exprimées dans un fichier de spécification(1). La compilation de ce fichier peuple un *méta schéma* qui contient toutes les informations de mise en correspondance des entités sources (relations et attributs) et des entités cibles (classes, liens d'héritage, attributs, racines de persistance et associations) ainsi que les propriétés physiques à réaliser sur la base cible. Le méta schéma est en fait un package java que nous avons appelé *mapping*.

Ses principales classes sont données dans le tableau ci-dessous.

Class (*.java)	Description
`AssocDescriptor`	décrit une association entre deux classes constituée d'un attribut et de son inverse
`ClassDescriptor`	décrit une classe, ses clés de transformation, sa relation source, ses attributs, sa clé de transformation, sa racine de persistance, sa super-classe, ses sous-classes et ses contraintes physiques
`ClusterDescriptor`	contient la description d'un regroupement (sa classe racine, les conditions du `if` par relation source et les attributs de regroupement)
`Constraints`	contient une contrainte d'ordre et/ou de regroupement
`NameDescriptor`	décrit la racine de persistance d'une classe (son nom et les requêtes de cette classe sur les données sources (une requête par relation))
`ObjectAttributeDescriptor`	décrit un attribut de classe (son nom, son type, la classe qu'il référence et ses requêtes d'évaluation par relation source)
`Pred`	décrit une requête de classe ou d'attribut
`RelOOSchema`	décrit un schéma objet (son schéma relationnel source, sa base, ses classes, ses racines de persistance, ses associations et les contraintes physiques qu'il contient)
`RelSchema`	décrit un schéma relationnel (son nom et ses relations)
`RelationDescriptor`	décrit une relation (son nom, sa clé primaire et ses attributs)
`RelationalAttributeDescriptor`	décrit un attribut relationnel (son nom et son type)

Les requêtes de peuplement des classes du schéma objet ainsi que celles permettant le calcul des valeurs de chacun des attributs de ces classes sont aussi répertoriées. C'est l'ensemble des classes du package *mapping* qui est ensuite utilisé pour générer le fichier de définition du schéma objet et des contraintes de regroupement des objets sur disque(2) dans le langage du système O_2 (O_2C) ainsi que la représentation algébrique du chargement par défaut(3).

Le fichier généré pour l'exemple du chapitre 5 est donné ci-dessous. Notons que les associations entre classes sont matérialisées par des liens de référence. Le schéma objet comprend également (i) des méthodes de mise à jour des attributs des classes et (ii) les commandes de regroupement des objets sur disque. Dans le cas présent, la commande générée correspond

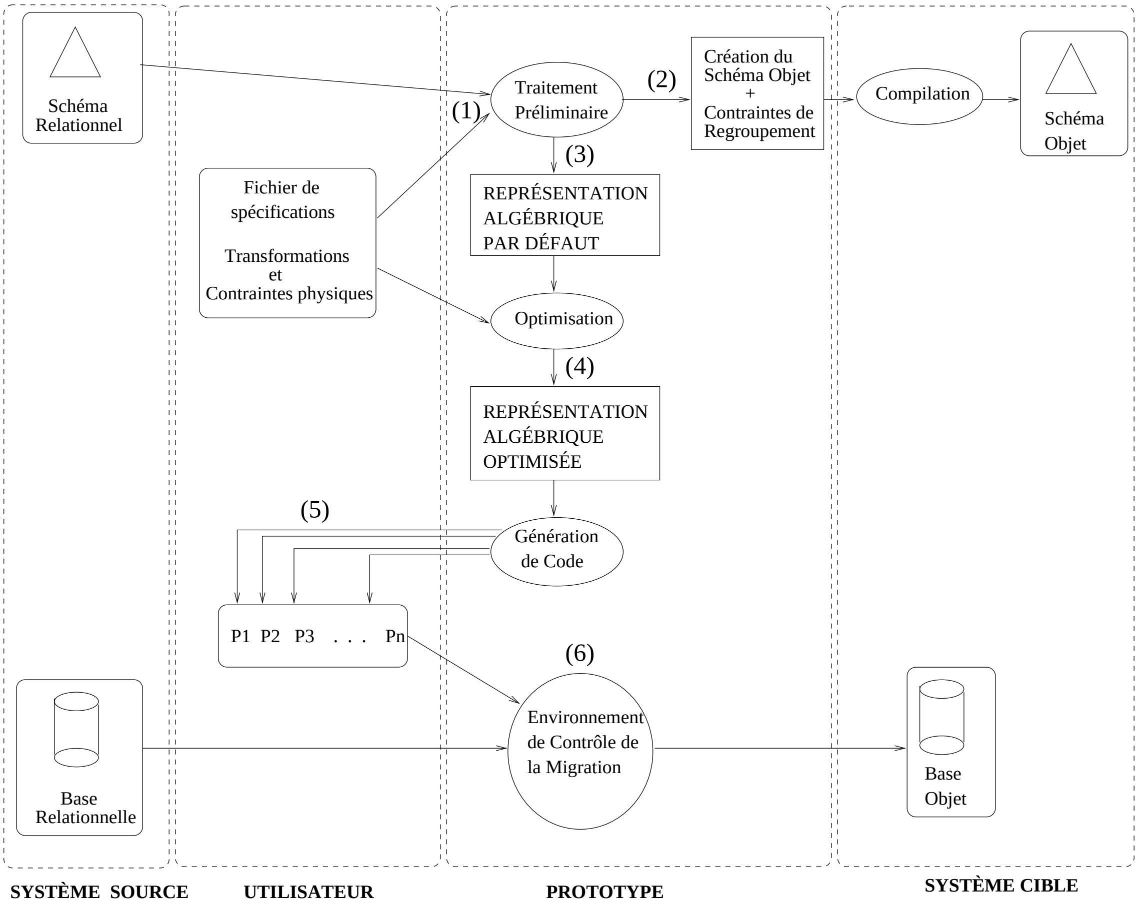


FIG. 7.1 – *Architecture du prototype RelOO*

au regroupement des chercheurs par projet.

```
class Project
cluster on staff
```

```
class         Employee type tuple(        class Project type tuple(
              name:string,                        name:string,
              salary:real,                        staff:set(Researcher))
              manages:Project)

class         Researcher                  name Employees:set(Employee);
inherits      Employee                    name Researchers:set(Researcher);
type tuple(   projects:set(Project))      name Projects:set(Project);


class         Project
cluster on    staff
```

Optimisation du chargement. La phase d'optimisation prend en entrée la représentation algébrique du chargement par défaut et retourne en sortie une représentation algébrique optimisée(4). Cette dernière est ensuite utilisée pour générer un ensemble de programmes paramétrés qui seront fournis à l'utilisateur pour peupler la base à objets.

L'optimiseur a comme objectif principal de minimiser le temps global du chargement tout en respectant les contraintes de l'utilisateur. Il est fondé sur une stratégie de recherche et un modèle de coût qu'il utilise pour trouver le meilleur compromis entre les temps de communication, de traitement relationnel et objet ainsi que les volumes de stockage utilisés. L'optimiseur travaille en deux étapes. En appliquant certaines équivalences, il réalise, en priorité, les contraintes spécifiées par l'utilisateur. Ensuite, tout en conservant les propriétés physiques, il applique les réécritures algébriques présentées au chapitre 4.

La phase d'optimisation repose sur un package que nous avons baptisé *optimizing*. Les classes principales de ce package sont données ci-dessous. Un bloc est une instance de la classe `Bloc.java` qui est décrite par une succession d'instances de la classe `AlgOpt.java`. Une instance de `AlgRep.java` est associée à un schéma objet et contient une représentation algébrique constituée d'une liste de blocs.

Class (*.java)	Description
`AlgOpt`	représente une opération algébrique (contient l'opération qui précède et suit celle-ci)
`AlgRep`	décrit une représentation algébrique (son schéma objet et la liste de ses blocs)
`Bloc`	décrit un bloc
`CostModel`	décrit les paramètres du modèle de coût
`Map`	hérite de `AlgOpt` décrit un *Map* (liste des fonctions qu'il applique)
`Func`	décrit une fonction appliquée à travers l'opérateur *Map*
`Obj`	hérite de `Func`, décrit une fonction de création d'objet
`Assign`	hérite de `Func`, décrit une opération d'assignation de valeur à un attribut
`Project`	hérite de `AlgOpt` décrit une projection
`Group`	hérite de `AlgOpt` et de `Project` décrit les paramètres d'un *Group*
`Sort`	hérite de `AlgOpt` et de `Project` décrit un tri
`Select`	hérite de `AlgOpt` décrit une sélection
`Join`	hérite de `AlgOpt` et de `Select` décrit une jointure contient aussi la deuxième entrée d'une jointure

Génération de code. Une fois le chargement optimisé, la phase de génération de code(5) construit les programmes. Chaque programme correspond à un bloc de l'expression algébrique optimisée. Il est constitué d'une partie SQL pour l'interrogation des données relationnelles et d'une partie en O_2C pour la manipulation des données dans le système O_2. Ces programmes sont des transactions paramétrées par le temps ou le nombre d'objets à créer à la fois.

Contrôle d'exécution. Le peuplement de la base à objets est réalisé de manière incrémentale dans un environnement de contrôle du chargement(6). Cet environnement gère l'enchaînement des différents programmes générés précédemment. Les programmes étant paramétrés, ils peuvent être lancés à plusieurs reprises et à des moments différents. Ceci implique de conserver toutes les informations concernant ce qui a été chargé et ce qui reste à charger à tout moment. Ces informations seront utilisées à la suite d'une interruption (volontaire ou involontaire) du chargement. D'autre part, l'environnement de contrôle les utilise pour reconstruire dynamiquement les requêtes SQL de façon à n'importer que la partie des données qui dépend du paramètre du programme lancé. La phase de peuplement sera terminée lorsque le dernier programme aura été entièrement effectué.

7.2 Stratégie de recherche et modèle de coût

Nous avons vu, dans le chapitre précédent, les deux aspects du chargement : *efficacité* et *flexibilité*. L'efficacité est exprimée en termes de temps et espace nécessaires au chargement. Le facteur temps englobe les temps de communication, traitement relationnel et traitement objet. Le facteur espace représente le volume de stockage nécessaire aux traitements intermédiaires ainsi que celui nécessité par le stockage des tables de correspondance (qui est potentiellement plus important). La flexibilité est en fait reflétée dans les possibilités accordées à l'utilisateur en termes de choix de l'organisation des données sur disque et de la possibilité de lancer des programmes de chargement paramétrés. Partant de la représentation du chargement par défaut, la stratégie de recherche applique une série d'équivalences afin de trouver la représentation algébrique optimale. Le choix de la solution optimale repose sur un modèle de coût. Notre stratégie de recherche travaille en deux étapes :

1. *Une étape linéaire* qui applique des réécritures en vue de réaliser les contraintes physiques exprimées par l'utilisateur.

2. *Une étape énumérative* qui explore toutes les possibilités de réécriture en respectant le résultat de la première étape.

L'étape linéaire prend en entrée une représentation algébrique par défaut et génère en sortie une représentation satisfaisant les contraintes de regroupement et d'ordonnancement des objets sur disque. C'est une ligne droite qui ne nécessite à aucun moment de choisir une représentation plutôt qu'une autre.

L'étape énumérative quant à elle, repose sur un algorithme qui applique des réécritures algébriques. Cet algorithme explore la totalité de l'espace de recherche. Nous avons commencé par développer un algorithme de recherche en largeur d'abord. Son inconvénient majeur est qu'il peut consommer un espace mémoire beaucoup trop important et causer ainsi une explosion combinatoire (voir chapitre 8). Nous avons alors développé un algorithme de recherche en profondeur d'abord qui limite la complexité en espace mais dont la complexité en nombre d'instructions (temps d'exécution) est plus importante que l'algorithme en largeur.

Étant donné que le temps que prend la phase d'optimisation importe peu et que l'intérêt principal de notre approche est de s'assurer que les programmes générés seront bien optimaux, nous pouvons opter pour l'algorithme de recherche en profondeur d'abord. Cependant, nous tenterons, dans le chapitre 8, de déterminer la taille du schéma objet en dessous de laquelle il reste intéressant d'utiliser l'algorithme en largeur d'abord.

Que ce soit à l'étape linéaire ou énumérative, le passage d'une représentation algébrique à une autre se fait en appliquant une *fusion*. Nous commençons d'abord par expliquer ce que signifie une fusion et quelles sont les contraintes de son application. Nous parlerons ensuite des étapes linéaire et énumérative. Nous finirons par le modèle de coût et quelques exemples de calcul du coût.

7.2.1 Fusions et contraintes d'application

Une fusion est une réécriture algébrique qui transforme deux blocs en un seul. Elle est réalisé par une combinaison d'équivalences intra-bloc et d'une équivalence inter-blocs. Elle a deux sémantiques distinctes : en profondeur et en largeur.

Fusion en profondeur. Celle-ci augmente un bloc de calcul d'un attribut complexe a d'une classe c_1 soit en calculant dans ce bloc la valeur d'un attribut atomique de tout ou partie des objets de cette classe soit, si l'attribut a référence une autre classe c_2, en calculant un attribut atomique ou complexe de tout ou partie des objets de c_2. Ceci est illustré par la figure 7.2.

La fusion en profondeur peut être appliquée à n'importe quelle profondeur du graphe du schéma objet (par exemple en calculant un attribut atomique des objets d'une classe c_3 référencée par c_2). L'exemple de la figure 7.3 est le cas simple de calcul de l'attribut atomique `name` de tous les objets de la classe `Researcher` à travers l'attribut `staff` de la classe `Project`. Notons que l'opération $\sqsupset\!\bowtie$ assure que tous les objets de la classe `Researcher` sont créés et leurs attributs évalués (même ceux qui ne sont pas membres d'un projet). Les parties encadrées en pointillé montrent ce qui change dans les blocs de départ.

Une équivalence inter-blocs est d'abord appliquée. Ensuite, des équivalences intra-bloc sont éventuellement appliquées pour pousser l'opérateur Map le plus bas possible avant de le combiner avec le Map déjà présent. Bien entendu, l'opérateur Map n'est pas obligatoirement poussé, cette réécriture dépend du modèle de coût.

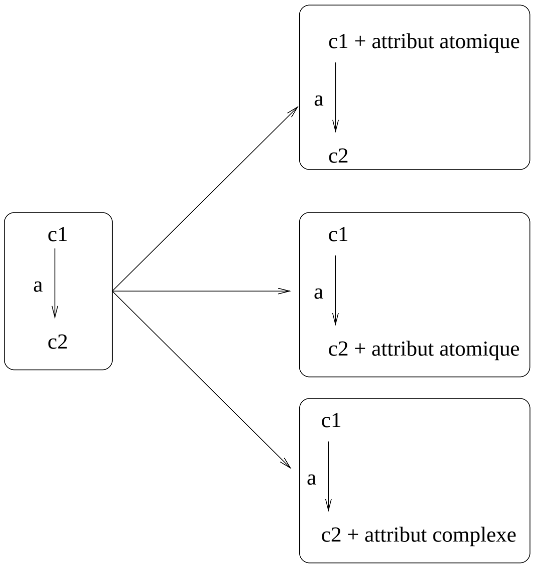


FIG. 7.2 – *Fusion en profondeur*

Fusion en largeur. Elle permet de calculer les attributs complexes de tout ou partie des objets d'une classe dans un bloc unique. En particulier, elle est appliquée pour calculer deux attributs complexes a_1 et a_2 d'une classe c dans le bloc de création des objets de cette classe. Elle est illustrée par la figure 7.4.

L'exemple de la figure 7.5 suivant montre comment fusionner les blocs de calcul de deux attributs complexes de la classe `Researcher` : `projects` (ensemble des projets dans lesquels un chercheur travaille) et `ownes` (voiture possédée par un chercheur) dans un même bloc. Le bloc **I** crée tous les objets de la classe `Researcher` et calcule la valeur de leurs attributs atomiques et de leur attribut complexe `projects`. Le bloc **II** calcule la valeur de leur attribut `ownes` (on suppose que la relation *car* contient un attribut *car* et un attribut *owner*). Dans le cas présent, il n'y a pas de conflit d'opérations *Group*. Si chacun de ces attributs était calculé en utilisant chacun une opération *Group* différente, une d'elle nécessiterait d'abord d'être supprimée avant la fusion par une réécriture intra-bloc (remplacée par l'opérateur «+=»). Dans le cas de l'exemple, la projection $\pi_{emp,car}$ dans le bloc **II** est d'abord éliminée. L'opération $⟖$ assure que même si un chercheur n'a pas de voiture, son attribut `projects` est calculé.

Pour créer les objets d'une classe avec tous ses attributs complexes dans un même bloc, il suffit d'appliquer une fusion en profondeur entre le bloc de création des objets de la classe et celui d'un de ses attributs complexes et d'appliquer ensuite une succession de fusions en largeur entre le bloc résultant et les blocs de calcul des attributs complexes de la classe.

Contraintes sur l'application des fusions. Nous avons défini des *contraintes locales* qui gèrent les possibilités de réécriture d'un bloc, et des *contraintes globales* pour gérer la cohérence des blocs qui forment une même représentation algébrique. Nous avons identifié

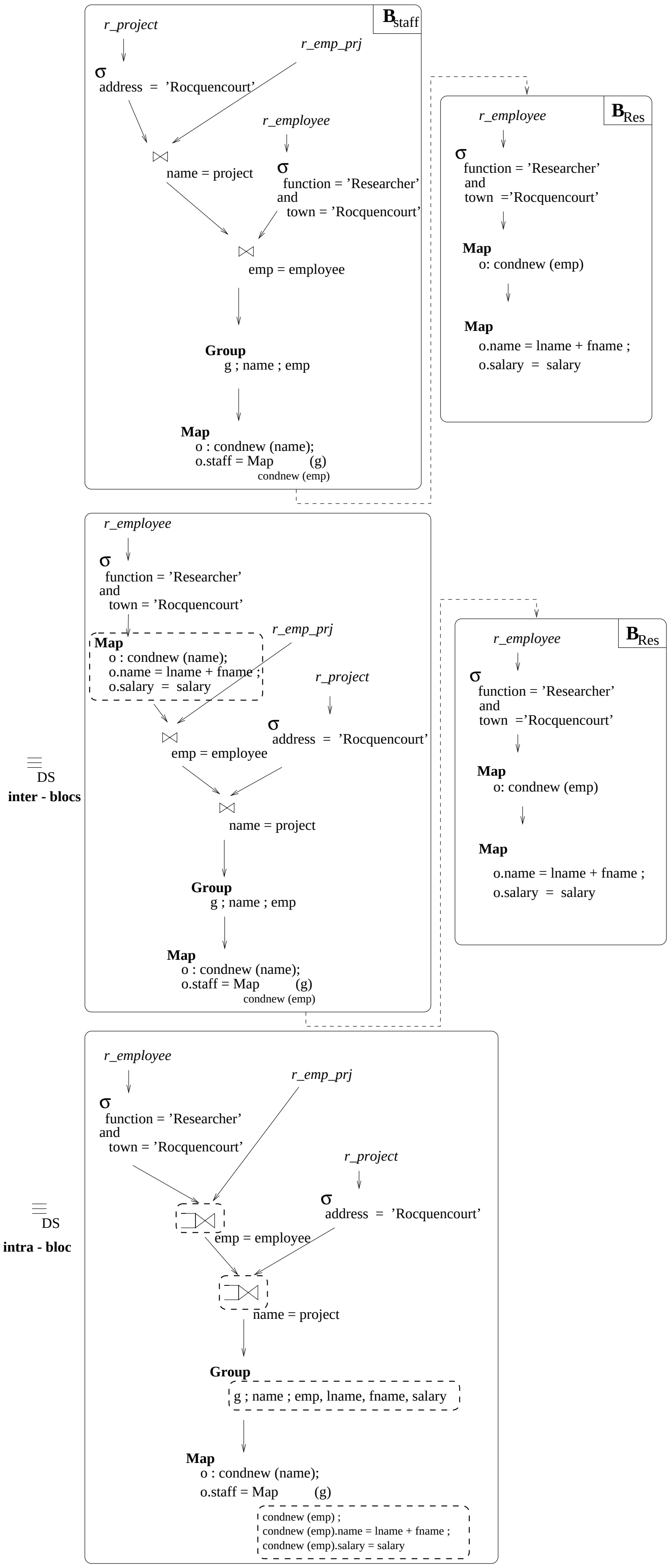


FIG. 7.3 – *Exemple de fusion en profondeur*

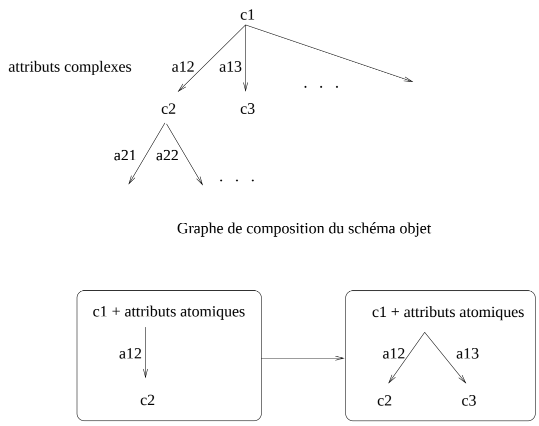


FIG. 7.4 – *Fusion en largeur*

deux contraintes locales :

1. Dans un bloc, un même attribut doit être calculé une seule fois pour éviter de boucler. En effet, si un attribut a est du même type que l'objet qui le contient (a est un attribut de c qui référence c), alors il est possible théoriquement de lui fusionner le bloc de calcul du même attribut (de calculer l'attribut a de l'attribut a du premier objet). Cependant, cela implique la possibilité de fusionner indéfiniment un même bloc de calcul d'attribut dans un autre. Pour éviter de se trouver dans cette situation, nous avons posé une contrainte sur les blocs qui évite de boucler : un même attribut ne peut être calculé qu'une seule fois dans un bloc.

2. Si la valeur d'un attribut a de type collection est calculée dans un bloc $\mathbf{B}_a$ et que cet attribut représente un lien entre une classe c_1 et c_2 et que c_2 est une super-classe alors il faut calculer la valeur de a en considérant tous les objets de ses sous-classes. Cette valeur est calculée de manière incrémentale en utilisant l'opérateur «+=».

Nous avons identifié deux contraintes globales :

1. Les blocs d'une représentation algébrique sont ordonnés. Cet ordre influe sur les réécritures et sur le coût de la représentation. En réalité, nous avons trois fonctions qui implantent la création d'objets. La première consiste à créer l'objet et l'insérer dans la table de correspondance si elle existe (fonction *new*). La deuxième, utilisée jusque là, teste d'abord si l'objet existe dans la table si oui, elle le récupère, si non, elle le crée et l'insère dans la table (fonction *condnew*). La dernière consiste à récupérer l'objet dans la table sachant qu'il a été préalablement créé (fonction *getnew*). Chacune de ces fonctions a un coût différent. Pour savoir laquelle appliquer dans un bloc, il est important de savoir, entre autres, si tout ou partie des objets d'une classe ont été créés avant le bloc courant. Pour cela, il faut regarder tous les blocs qui précèdent celui-ci.

   Lorsque plusieurs blocs créent des objets d'une même classe, le premier appelle la fonction *new*, qui crée les objets et leur assigne les valeurs atomiques, les autres blocs doivent appeler la fonction *condnew*. Si l'on est sûr que tous les objets d'une classe ont été créés dans les blocs précédents, il faut utiliser la fonction *getnew*.

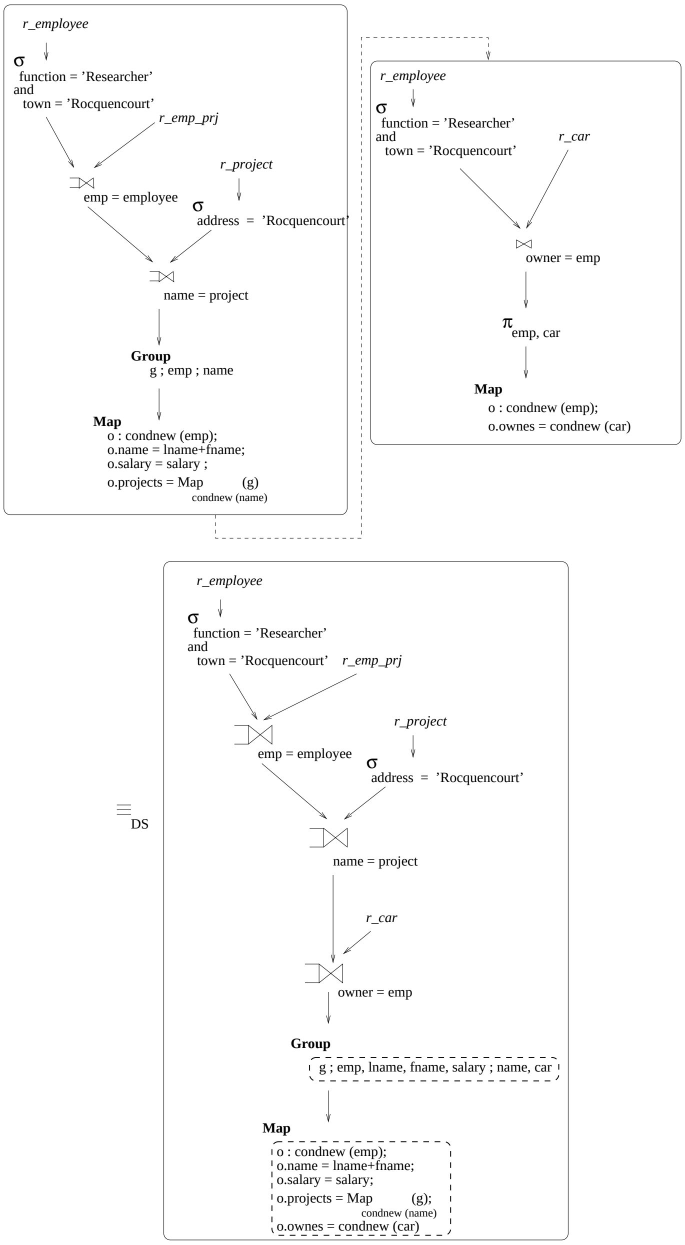


FIG. 7.5 – *Exemple de fusion en largeur*

2. Soit un bloc $\mathbf{B}_a$ qui calcule la valeur d'un attribut agrégat a d'une classe c_1 qui fait référence à une classe c_2. Si un ordre est défini par l'utilisateur sur les objets de la classe c_2, celle-ci ne doit pas être peuplée à travers la classe c_1. Cela signifie que le bloc qui crée réellement les objets de c_2 dans le bon ordre doit précéder $\mathbf{B}_a$.

7.2.2 Étape linéaire

Cette étape prend en entrée la représentation algébrique par défaut, réécrit la sous-liste de blocs concernée par les contraintes d'ordre et de regroupement et place les blocs résultants en tête de la représentation algébrique en sortie. Les blocs dans la sous-liste réécrite sont ordonnés par rapport à l'ordre d'apparition des contraintes physiques dans le fichier de spécification de l'utilisateur. Les contraintes d'ordre et de regroupement peuvent être combinées dans une même spécification.

Contrainte d'ordre. Si une contrainte d'ordre est donnée sur tout ou partie des objets d'une classe, un tri (opérateur *Sort*) est introduit dans le bloc de création des objets de cette classe. Ce bloc ne doit en aucun cas être décomposé en blocs qui casseraient l'ordre. Dans le cas où une clause `if` est spécifiée, le bloc de création de la classe est d'abord décomposé en deux blocs : celui réalisant la condition du `if` et celui réalisant le complément. Ce n'est que dans le premier bloc que le tri demandé est inséré. En plaçant celui-ci en tête de la représentation algébrique, on s'assure que les objets seront créés dans le bon ordre, à travers ce bloc sachant qu'il doit contenir des opérations *new* (de création effective). Si un ordre est demandé sur tous les objets d'une classe, toute invocation d'un objet de cette classe dans un autre bloc que celui qui crée les objets dans l'ordre est faite avec l'opérateur *getnew*.

Contrainte de regroupement. Soit la contrainte de regroupement suivante :

`class`	c
`if`	p
`cluster on`	$a_1(a_{11}^*), a_2$

Cette contrainte signifie que l'utilisateur veut obtenir un programme qui créerait les objets de la classe c qui satisfont le prédicat p et calculerait leurs attributs atomiques et leur attributs complexes a_1 et a_2 et que pour tout objet de la classe référencée par a_1, l'attribut complexe a_{11} soit évalué. De plus, sachant que a_{11} référence des objets, leur attributs atomiques doivent aussi être calculés (du fait de l'*).
Le bloc de peuplement de la classe c est d'abord décomposé en deux blocs : celui qui réalise la contrainte de la clause `if` et son complément. Le premier est alors fusionné avec les blocs de calcul des attributs a_1 et a_2. Une fusion en profondeur est ensuite appliquée afin de calculer l'attribut a_{11} et ses valeurs atomiques à travers l'attribut a_1.
Dans le cas d'un regroupement, le bloc de création de la classe racine du regroupement est fusionné avec tous les blocs d'attributs apparaissant dans la clause `cluster on`. Pour chaque attribut suivi d'une étoile, le bloc résultant de la première fusion est aussi fusionné avec le bloc de création de la classe référencée par l'attribut (pour le calcul des attributs atomiques de cette classe). Le résultat est ensuite placé en tête de la représentation algébrique à laquelle il appartient en respectant l'ordre de priorité des contraintes telles que spécifiées par l'utilisateur.

7.2.3 Étape énumérative

Cette étape a été réalisée en collaboration avec Ségolène Galabert, stagiaire de DEA.
La représentation algébrique qui réalise les contraintes physiques est donnée à l'étape énumérative qui produit la représentation optimisée. L'espace de recherche peut être vu comme un arbre dont chaque nœud est une représentation algébrique et chaque branche une fusion.

Notre stratégie explore toutes les possibilités de fusion de blocs. Par opposition aux stratégies de recherche dynamiques qui travaillent par construction, notre stratégie opère par transformation. De ce fait, chaque nœud de l'arbre est une représentation algébrique à part entière (voir figure 7.6).

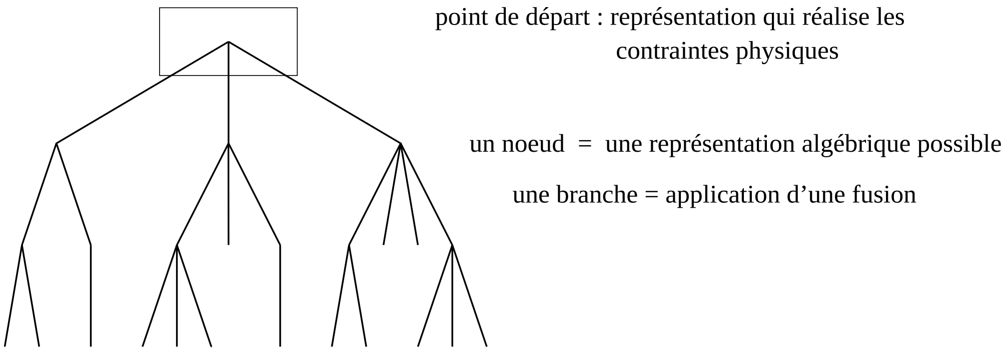
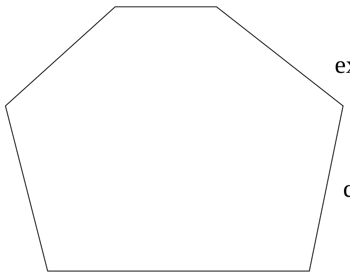


FIG. 7.6 – *Arbre de génération des solutions*

Redondances. À partir de de la racine de l'arbre, plusieurs chemins différents peuvent mener à la même représentation algébrique. Partant du bloc qui calcule l'attribut `staff` de chaque projet, on peut appliquer (i) une fusion en profondeur qui calculerait, dans ce bloc, l'attribut `name` de chaque projet et (ii) une autre fusion en profondeur qui calculerait les attributs atomiques `name` et `salary` de chaque chercheur créé à travers un projet. L'ordre dans lequel ces fusions sont appliquées est arbitraire, ce qui résulte en deux chemins différents menant au même résultat.
Deux blocs sont identiques s'ils effectuent les mêmes opérations dans le même ordre. Deux représentations algébriques identiques ont le même coût et le même nombre de blocs, qui apparaissent dans le même ordre. Deux représentations ne peuvent être identiques que si elles ont été obtenues en appliquant le même nombre de fusions. C'est pour cela que pour diminuer le risque de solutions redondantes, toute fusion est faite avec un bloc qui n'a jamais été modifié (bloc de création ou bloc d'assignation d'attribut complexe).

Terminaison de l'algorithme. L'algorithme s'arrête lorsqu'il n'y a plus de représentation à exploiter. Le nombre de blocs d'une représentation est fini, donc celui des blocs qui peuvent supporter une fusion aussi. Par conséquent, il existe une profondeur maximale de l'arbre des solutions (le passage d'un niveau à l'autre se faisant par une fusion). La forme générale de l'espace de recherche est illustrée ci-dessous.



Dans la suite, nous donnons le déroulement des algorithmes de recherche en largeur et en profondeur.

7.2.4 Algorithme en largeur d'abord

Cet algorithme utilise les variables et fonctions suivantes :

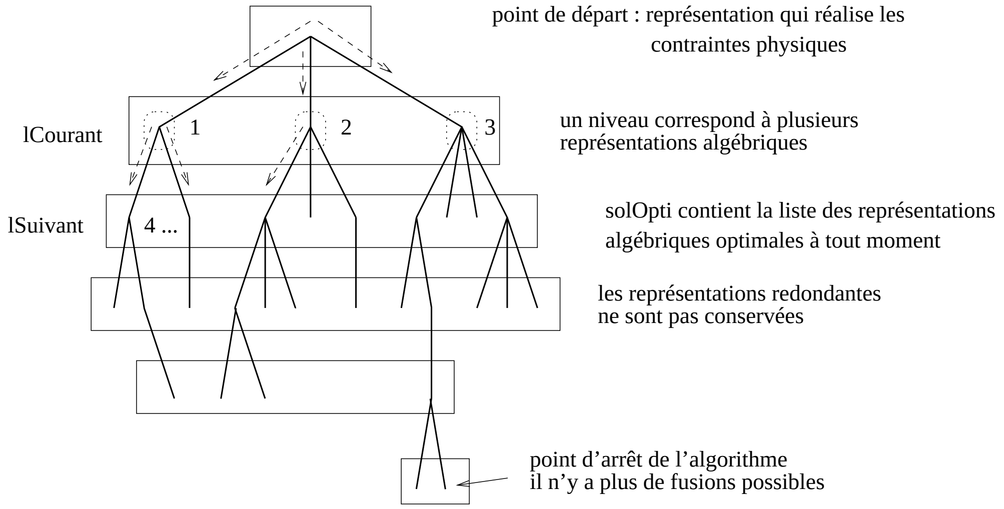
- `rep` et `nouvelleRep` sont des représentations algébriques du chargement.
- `repContraintes` est la représentation obtenue après prise en compte des contraintes physiques. C'est le point de départ de l'algorithme de recherche.
- `solOpti` contient la liste des solutions optimales et le coût de ces solutions (plusieurs solutions peuvent avoir le coût optimal). Une solution est une représentation algébrique constituée d'une succession de blocs. Au départ, `solOpti` contient `repContraintes`.
- `lCourant` contient les solutions courantes à examiner.
  Au départ, elle contient `repContraintes`.
- `lSuivant` contient les solutions générées par les solutions courantes.
  Au départ, elle est vide.

```
function merge(rep,b,b') {
  copier rep dans nouvelleRep
  fusionner b dans nouvelleRep avec b'
  Si nouvelleRep n'appartient pas à lSuivant
    lSuivant += [nouvelleRep];
    testInsert(nouvelleRep,solOpti)
}

function testInsert(nouvelleRep,solOpti) {
  Si coût(nouvelleRep) < coût(solOpti)
       SolOpti = [nouvelleRep]
  Sinon
    si coût(nouvelleRep) = coût(solOpti)
         solOpti += [nouvelleRep]
}
```

```
Tant que lCourant n'est pas nulle
  Pour chaque représentation rep dans lCourant
    Pour chaque bloc b de rep
       Pour chaque classe c dont on crée un ou plusieurs objets dans b
                   et dont les valeurs atomiques ne sont pas assignées
           récupérer bc, le bloc de peuplement de c
           merge(rep,b,bc)
       FinPour
       Pour chaque bloc batt de rep qui calcule un attribut a
          Si des objets de la classe référencée par a sont créés dans b
                        et que l'attribut a n'apparaît pas déjà dans b
             récupérer batt, le bloc de calcul de a
             merge(rep,b,batt)
       FinPour
    FinPour
  FinPour
  lCourant = lSuivant; lSuivant = []
FinTant que
```

L'algorithme travaille en deux étapes. Pour chaque représentation courante, il regarde les fusions en profondeur et en largeur qu'il peut appliquer à chacun de ses blocs. Il génère alors dans `lSuivant`, tous les résultats possibles après une première fusion (fonction `merge()`) et teste à chaque fois, si les représentations générées ont un coût minimal (fonction `testInsert()`). C'est ce qui fait que la stratégie de recherche est indépendante du modèle de coût. Si l'on reprend le schéma de l'espace de recherche donné sur la figure 7.6, l'algorithme de recherche en largeur peut être illustré par la figure qui suit. Les numéros montrent l'ordre dans lequel les représentations sont générées.



À tout moment, on ne garde que deux niveaux de l'arbre de recherche (`lCourant`, `lSuivant`) ainsi que les solutions optimales (`solOpti`). Deux représentations algébriques sont identiques lorsqu'elles sont issues du même nombre de fusions. Pour éviter les redondances et diminuer la complexité en nombre d'instructions de l'algorithme, il suffit de tester qu'une représentation n'existe pas dans `lSuivant` avant de l'y insérer. Cependant, la complexité en espace croit avec le nombre de représentations générées par niveau. L'algorithme en profondeur d'abord permet d'y remédier.

7.2.5 Algorithme en profondeur d'abord

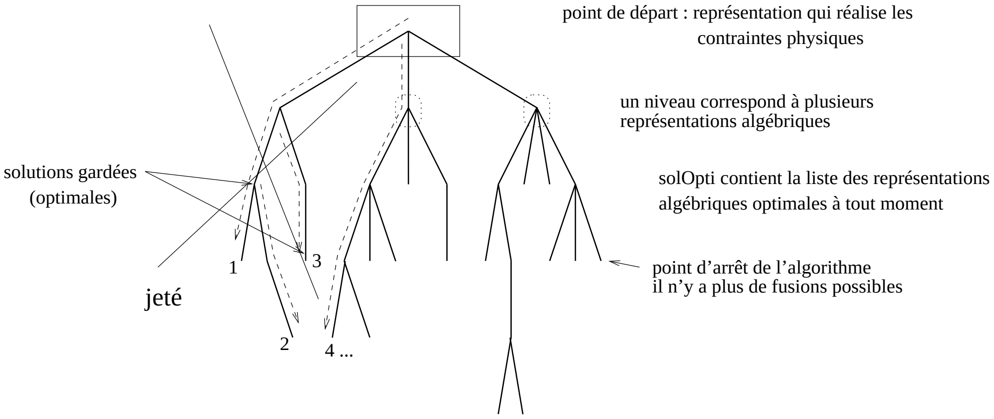
Comme précédemment `rep`, `nouvelleRep` et `repContraintes` sont des représentations algébriques du chargement, `repContraintes` étant celle obtenue après la réalisation des contraintes physiques. `solOpti` est la liste des représentations de coût minimal.

```
// initialisations
rep = repContraintes
solOpti = [rep]
solOpti = optimize(rep, solOpti)
```

Avec comme fonction récursive :

```
function optimize(rep, solOpti) {
  Pour chaque bloc b de rep
    Pour chaque classe c dont on crée un ou plusieurs objets dans b
                et dont les valeurs atomiques ne sont pas assignées
        récupérer bc le bloc de peuplement de c
        merge(rep,b,bc)
        solOpti = optimize(nouvelleRep, solOpti)
    FinPour
    Pour chaque bloc batt de rep qui calcule un attribut a
       Si des objets de la classe référencée par a sont créés dans b
                          et que l'attribut a n'apparaît pas dans b
         récupérer batt, le bloc de calcul de a
         merge(rep,b,batt)
         solOpti = optimize(nouvelleRep, solOpti)
    FinPour
  FinPour
```

Cette fonction fait en sorte que c'est la représentation la plus récemment générée qui est développée à chaque fois. L'algorithme, donné dans la figure ci-dessous, commence alors par aller de la racine de l'arbre jusqu'à la feuille 1. Il développe ensuite la feuille 2 ... etc.



À tout moment, seuls sont conservés les nœuds qui mènent de la racine de l'arbre de recherche au nœud où l'on se trouve ainsi que les solutions optimales obtenues à ce point. La complexité en espace peut alors être considérablement réduite. L'inconvénient de cet algorithme est que l'on n'est plus capable de reconnaître une solution déjà rencontrée puisqu'on ne dispose pas, à la fois, de toutes les solutions obtenues avec le même nombre de fusions (même niveau de l'arbre). Ceci implique que le nombre de représentations redondantes augmente. La complexité en nombre d'instructions de cet algorithme est alors plus importante que celle de l'algorithme en largeur d'abord.

7.2.6 Modèle de coût

Les paramètres du modèle de coût, à savoir, l'équilibrage de charge, le temps de traitement relationnel et objet, le temps de communication, le volume de stockage et le nombre de blocs constituant une représentation algébrique, sont traduits en termes de taille des relations et des tables de correspondance manipulées, de nombres d'itérations sur les relations sources, d'accès séquentiels vs aléatoires aux objets, etc.

Quasiment toutes nos opérations algébriques travaillent sur des ensembles de n-uplets. Chacune prend un nombre n de n-uplets en entrée et en retourne m. Leur coût dépend bien évidemment de n. Le coût d'un bloc devrait être tout simplement la somme des coûts de chacune des opérations qui le constituent. Seulement, nous verrons que grâce à la factorisation d'une même expression relationnelle entre différents blocs, le coût des opérations relationnelles dans un bloc peut être nul mais son coût de stockage se voit augmenté. Nous donnons d'abord, le coût de chaque opération et ensuite, la formule de coût pour un bloc et enfin pour une représentation algébrique i.e. une succession de blocs.

Dans notre cas de chargement relationnel-objet, nous avons choisi de pousser sur le site relationnel, toutes les opérations qui peuvent y être effectuées et enfin, d'exécuter les opérations de création et de mise à jour d'objets sur le site objet. Le site relationnel est le mieux placé pour effectuer les opérations relationnelles et les opérations de création et de mise à jour d'objets ne peuvent être exécutées ailleurs que sur le site objet. Afin de conserver une stratégie de recherche indépendante des systèmes sous-jacents, ce choix est codé dans le modèle de coût. Il est exprimé par le fait que toute opération relationnelle dans un bloc précédée d'une opération non relationnelle a un coût infini. Ceci garantit que toute expression algébrique ne vérifiant pas cette condition sera très coûteuse et ne sera pas marquée comme optimale par la stratégie de recherche. De ce fait, l'équilibrage de charge n'est pas codé en dur dans la stratégie de recherche et peut être modifié aisément en modifiant les coûts attribués aux opérations algébriques.

Coût des opérations algébriques

Tri : $n \rightarrow n$

coût $= 0$ si factorisation

$= \infty$ si précédé d'un Map ou d'un $Group$

$= n\ logn$ sinon

$n\ logn$ est le coût attribué à un tri. Il dépend du nombre d'éléments entrants.

Group : $n \rightarrow n{*}taux\ de\ partage$

coût $= \infty$ si non précédé d'un tri

$=$ constant et faible (c_G) sinon

Le coût d'un regroupement garantit qu'il n'est intéressant que s'il est précédé d'un tri effectué sur le site relationnel. Si ceci est le cas, son coût devient négligeable. Il correspond au coût de construction de l'ensemble des éléments regroupés. Le *taux de partage* est une constante qui réduit le nombre de n-uplets résultants. C'est une valeur déterminée par le nombre moyen de fois qu'un élément apparaît dans les n-uplets de départ.
Nous avons attribué à c_G la valeur nulle puisqu'en réalité, nous ne construisons pas d'ensemble contenant la partition du regroupement. Nous verrons à la génération du code comment nous traitons un regroupement. Pour la même raison, le *taux de partage* est égal à 1. De ce fait, si deux blocs $\mathbf{B}_1$ et $\mathbf{B}_2$ contiennent les mêmes opérations dans le même ordre mais que $\mathbf{B}_1$ contient une commande de regroupement et que $\mathbf{B}_2$ une mise à jour incrémentale dans un ensemble «+=» qui lui est équivalente, la différence de coût entre ces deux blocs se reflétera dans le coût de calcul de la valeur de l'attribut concerné par le regroupement (coût de l'opération d'assignation).

Sélection : $n \rightarrow n{*}facteur\ de\ sélectivité$

où le *facteur de sélectivité* est un quotient qui réduit la taille des éléments en entrée.

coût $= 0$ si factorisation

$= \infty$ si précédé d'un Map

$= n * c_s$ sinon

Le *facteur de sélectivité* est un quotient qui réduit la taille des éléments en entrée en fonction de la satisfaction du prédicat de sélection. Nous avons attribué à chaque prédicat de sélection un facteur donné. La constante c_s représente le coût engendré par la lecture de chaque n-uplet et le test de satisfaction du prédicat de sélection. Pour estimer le coût d'une sélection, c_s est multiplié par le nombre de n-uplets entrants. En considérant que le coût d'une sélection unitaire effectuée sur le site relationnel est toujours le même, le coût de sélectionner sur n n-uplets ne va dépendre que de n. Nous avons alors fixé c_s à 1.

Projection : $n \rightarrow n{*}$*facteur de projection*
coût $= 0$ si factorisation
$= \infty$ si précédé d'un Map
$= c_p$

Le *facteur de projection* est une constante qui réduit le nombre de n-uplets en entrée. Comme la projection élimine les redondances dans le résultat, le nombre d'éléments entrants se voit réduit. Nous verrons dans le calcul du temps de communication d'un bloc que nous utilisons la taille de chaque attribut conservé par la projection et le nombre de n-uplets. Pour les mêmes raisons que la sélection, nous avons fixé la constante c_p à 1.

Jointure : $n + m \rightarrow n * m{*}$*facteur de jointure*
Jointure externe : $n + m \rightarrow n * m$
où n et m sont les tailles des relations qui interviennent dans la jointure
coût $= 0$ si factorisation
$= \infty$ si précédé d'un Map
$= n * m * c_j$ sinon

De la même façon que pour la sélection et la projection, nous fixons le coût unitaire c_j à 1. Ce qui différencie la jointure et la jointure externe c'est le *facteur de jointure*. En réalité, la taille du résultat de la jointure externe est inférieure à $n * m$ puisqu'il y a quand même des n-uplets qui ne se retrouvent pas dans le résultat (ce n'est pas un produit cartésien) mais nous omettons ce détail et considérons que c'est le cas. Le nombre d'éléments sortants est donc $n * m$. Ceci n'est pas le cas pour la jointure puisque les n-uplets ne vérifiant pas le prédicat de jointure ne se trouvent pas dans le résultat. Par conséquent, à chaque prédicat de jointure, nous associons une constante qui représente son *facteur de jointure*.

$\mathbf{Map}_h(e) : n \rightarrow n$
$\mathbf{Map}_{a:h}(e) : n \rightarrow n$
coût $= n{*}\text{coût}(h)$

où n est la taille du résultat de l'évaluation de e et h est une fonction de création ou de mise à jour dont nous donnons le coût dans la suite. À noter que $Map_{a:h}(e)$ augmente la taille de chaque n-uplet entrant de l'attribut a qui contient le résultat de l'évaluation de h sur le n-uplet.
Nous allons voir que le coût de h n'est a priori pas précis. Notre objectif premier est d'avoir une estimation qui tient compte des différences entre les fonctions appliquées à travers l'opérateur Map. La phase de validation du modèle de coût (voir chapitre 8) servira à le rendre plus précis. C'est pour cela que nous avons fait en sorte que sa modification soit aisée.

Nous calculons maintenant le coût des fonctions *getnew*, *new*, *condnew* et de l'assignation de valeur.

getnew : $getnew(r, c, \vec{a})$ est utilisée pour récupérer de la table de correspondance de (r, c) l'objet correspondant à un enregistrement donné. Cette fonction suppose que l'objet a été préalablement créé. Elle évite de tester son existence. Son coût est minimal puisqu'il correspond à un simple accès à la table de correspondance qui est déjà indexée. Nous fixons son coût à 1.

coût = coût accès à table corresp. = 1

new : $new(r, c, \vec{a})$ est utilisée lorsque l'on sait que l'objet qui correspond à la valeur de la clé de transformation donnée en paramètre n'a pas été créé auparavant. Elle crée un identifiant d'objet o dans $\pi(c)$ pour chaque valeur distincte de $\vec{a}$ dans r. Elle insère o dans la racine de persistance de c si elle existe. Le couple $(\vec{a}, o)$ est ensuite inséré dans la table de correspondance définie pour la paire (r, c) (si elle existe).

coût = coût accès à table corresp.(si table existe)+
coût création obj+
coût m.a.j. racine de persist.(si racine existe)+
coût m.a.j. table corresp.(si table existe)

Dans le cas où la table de correspondance n'existe pas, le coût est réduit au coût de création de l'objet et au coût de mise à jour de la racine de persistance dans laquelle il est inséré (si bien sûr cette racine existe). Nous évaluons à 1 la formule : coût de création d'un objet + coût m.a.j. racine de persist.(si racine exite) + coût m.a.j. table corresp.(si table existe). La fonction *new* est évaluée à 3.

condnew : $condnew(r, c, \vec{a})$ qui teste si la paire $(\vec{a}, o)$ existe déjà dans la table de correspondance de (r, c) et crée l'objet s'il n'y est pas. Elle est utilisée dans le cas le plus défavorable où l'on n'a aucune connaissance de l'existence de l'objet correspondant à la valeur de $\vec{a}$ en paramètre. Dans le cas de l'utilisation de cette fonction, on sait que la table de correspondance existe.

coût = coût accès à la table corresp.+
prob∗(coût création obj+
coût m.a.j. table corresp.+
coût m.a.j. racine de persist.(si racine exite))

prob est la probabilité qu'un objet existe ou non dans la base. Selon le cas, la fonction *condnew* correspondra soit à la fonction *getnew* soit à la fonction *new*. La fonction *condnew* est potentiellement plus chère que la fonction *new* puisqu'elle teste toujours l'existence de l'objet dans la table de correspondance (parcous de la table). Nous la fixons à 4.

assignation: Si h est une fonction d'assignation, elle s'écrit : $o.a = f$ ou $o.a{+}{=}f$. Dans les deux cas, o est l'objet à mettre à jour, a est l'attribut auquel est assignée le résultat de l'évaluation de f éventuellement de manière incrémentale «+=».

coût = coût m.a.j. obj+coût de f

Ci-dessous, la fonction récursive `costAssign()` qui calcule le coût de h.

```
function costAssign()
  Si a est une référence à un ou plusieurs objets
      Si l'opérateur utilisé est <<=>>
        retourner (coût m.a.j.obj+coût de f*taux de partage)
      Sinon
        retourner (coût m.a.j.obj+coût de f)
  retourner coût m.a.j.obj
```

Nous supposons que le coût de mise à jour d'un objet est égal à 1. Si a est une référence à un objet, le coût de l'assignation correspond au coût de mise à jour de l'objet o et au coût de la fonction f (qui crée et met à jour l'objet référencé par a). Si a est une référence à plusieurs objets, le coût de f est multiplié par le *taux de partage* qui représente le nombre moyen d'objets contenus dans la partition de regroupement. Dans tous les autres cas, le coût de l'assignation est égal au coût de mise à jour de l'objet o. Par exemple, si a est de type atomique, le coût de f est négligé. À noter que dans le cas de l'utilisation de l'opérateur «+=», le coût de mise à jour de o est comptabilisé à chaque itération dans la partition de regroupement ce qui n'est pas le cas lorsque c'est l'opérateur «=» qui est utilisé.

Formule de coût

La formule de coût d'un bloc s'exprime en fonction des paramètres : (i) coût de traitement relationnel, (ii) coût de communication, (iii) coût de stockage et (iv) coût de traitement objet.
Le coût d'une représentation algébrique est la somme de chacun des coûts des blocs qui la constituent. Chaque somme est pondérée du poids attribué par l'utilisateur à chaque coût. La formule de coût d'une représentation algébrique ne selimite pas à cela. Elle comprend aussi le volume de stockage des tables de correspondance créées et du nombre de blocs de cette représentation.

Le coût de traitement relationnel est calculé en fonction de la taille des relations manipulées et du coût des opérations de sélection, projection, jointure et tri qui constituent la partie relationnelle d'un bloc.
Le coût de communication s'exprime en fonction de la taille des données à communiquer (en termes de nombre de n-uplets, de nombre d'attributs par n-uplet et de la taille en octets de chaque attribut) et du coût unitaire de communication (coût pour un octet).

Le coût de stockage s'exprime en fonction de la taille des données communiquées et de la taille des tables de correspondance qui est prise en considération dans le calcul du coût global, celui d'une représentation algébrique. En effet, une table de correspondance ne dépend pas d'un bloc mais de la représentation algébrique qui le contient. Nos tables sont stockées dans le SGBD O_2 [BDK92] (notre système objet cible) comme des collections de paires (clé de transformation, identifiant d'objet). Elles sont indexées sur les valeurs de la clé de transformation. Pour une paire (relation, classe), ce n'est que dans le cas où seule la fonction *new* est utilisée dans tous les blocs d'une représentation algébrique que la table de correspondance n'est pas créée pour cette paire .
Le coût de traitement objet correspond au coût de l'opération *Map* dans ce bloc. Afin de contrôler le chargement incrémental des données d'un bloc, nous introduisons un tri qui sera utilisé dans l'évaluation de la requête SQL d'un bloc. Ce tri sert également à minimiser le nombre d'entrées/sorties dans un bloc (son coût de traitement objet). En effet, si les objets sont consultés dans l'ordre dans lequel ils ont été créés, les accès sur disque sont séquentiels. Bien sûr, ceci n'est pas vérifié lorsqu'un objet est créé ou mis à jour à travers un autre objet.

Si l'on dénote P_{Rel} le poids attribué par l'utilisateur au traitement relationnel, P_{Com} celui attribué à la communication, P_{Sto} celui attribué au stockage et P_{Obj} celui attribué au traitement objet et sachant que pour une représentation algébrique **AlgRep**, la somme des coûts relationnels de ses blocs est C_{Rel}, celle des coûts de communication est C_{Com} et celle des coûts de traitement objet est C_{Obj}, la formule de calcul du coût de **AlgRep** est la suivante :
$C_{\mathbf{AlgRep}} = (P_{Rel} * C_{Rel} + P_{Com} * C_{Com} + P_{Obj} * C_{Obj} + P_{Sto} * C_{Sto})/$ nombre de blocs
Avec C_{Sto}, le coût de stockage des tables de correspondance.

Les deux algorithmes implantés retournent un ensemble de solutions équivalentes en termes de coût. Afin de réduire cet ensemble, on privilégie dans le modèle de coût la ou les solutions ayant le nombre maximal de blocs. Comme un programme est généré pour chaque bloc et que nous tenons à décomposer le processus de chargement, plus il y a de programmes, plus c'est intéressant. Afin de réaliser cela, le coût d'une représentation algébrique est divisé par son nombre de blocs. Ceci n'empêche pas d'obtenir plusieurs solutions optimales (celles ayant le même coût et le même nombre de blocs). Dans ce cas, le choix d'une solution est non déterministe.
La factorisation est prise en compte dans le coût des opérations relationnelles de chaque bloc. Si $\mathbf{B}_1$ précède $\mathbf{B}_2$ et que les deux blocs ont la même partie relationnelle, une copie du résultat de l'évaluation de la partie relationnelle de $\mathbf{B}_1$ est stockée dans un fichier intermédiaire. Son contenu sera utilisé pour $\mathbf{B}_2$. Bien sûr, le volume de stockage se voit augmenté. Ceci est pris en considération dans le calcul du coût de stockage global. La factorisation est intéressante lorsque les blocs concernés calculent l'un un attribut complexe et l'autre son inverse. Si la partie relationnelle de ces blocs n'a pas été modifiée par la phase de réécriture, ils sont factorisables. Notons que, comme l'exécution d'un bloc est paramétrée par le facteur temps ou nombre, le résultat de l'évaluation de sa partie relationnelle est récupéré de manière incrémentale et la copie de ce résultat dans le fichier de factorisation correspondant se fait aussi de manière incrémentale. Dans certains cas, la factorisation nécessite un coût de traitement supplémentaire. Ceci est le cas lorsque les deux attributs concernés par la factorisation (un attribut et son inverse) sont des collections d'objets. En effet, dans ce cas, chacun d'entre

eux nécessite un regroupement (donc un tri) différent. Il serait intéressant de savoir s'il n'est pas mieux dans ce cas précis d'éviter une factorisation.

Soient les relations suivantes :

Relation [nombre de n-uplets]	Attributs [taille en octets]
r_1[2000]	att_1[5] att_2[3] att_3[20]
r_2[390]	att_4[20] att_5[35] att_6[2] att_7[5]

Supposons les opérations suivantes :

Opération	Facteur
$join_p$	1/2
σ_{p_1}	1/10
σ_{p_2}	1/5
π_{att_1,att_2}	1/2

Supposons les blocs suivants :

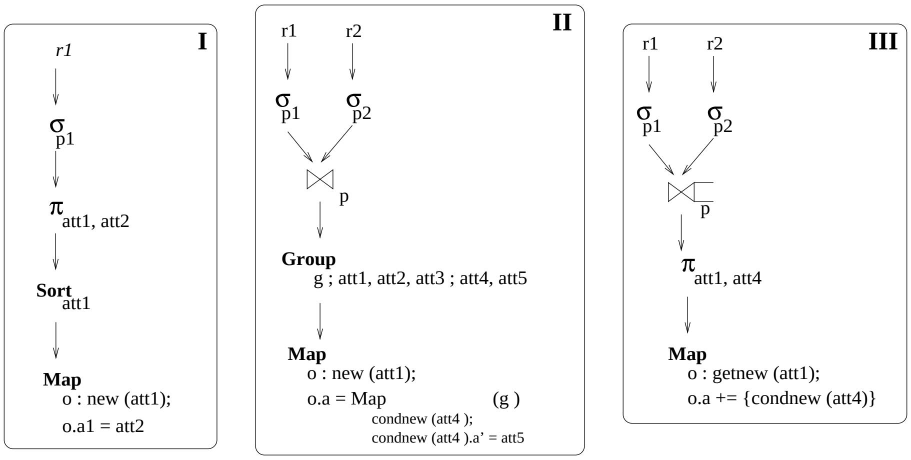


FIG. 7.7 – *Exemples*

Calculons maintenant les coûts des blocs **I**, **II** et **III**.

Opération	entrée→sortie	Coût
σ_{p_1}	2000→2000/10=200	2000
π_{att_1,att_2}	200→200/2=100	200
$Sort_{att_1}$	100→100	100*log 100=460.51
$o : new(att_1)$		3
$o.a = att_2$		1
$Map_{...}$	100→100	100*(3+1)=400

Le coût de traitement relationnel du bloc **I** est estimé à 2000 + 200 + 460.51, ce qui fait 2660.51. Son coût de traitement objet est égal à 400. Son coût de communication est égal au résultat de l'opération $Sort$ (nombre de n-uplets égal à 100) multiplié par la largeur de chaque n-uplet. Comme la projection conserve les attributs att_1 et att_2, le coût de communication est égal au coût de communication d'un octet (nous le fixons à 1) multiplié par (100*(5+3)). Il est donc estimé à 800.

Regardons maintenant le coût du bloc **II** :

Opération	entrée→sortie	Coût
σ_{p_1}	2000→2000/10=200	2000
σ_{p_2}	390→390/5=78	390
$\bowtie_p$	200*78→200*78/2=7800	200*78=15600
$Group_{g;att1,att2,att3;att4,att5}$	7800→7800	0
$o : new(att_1)$		3
$condnew(att_4)$		4
$condnew(att_4).a' = att_5$		1
$o.a = Map_{...}(g)$		1+$partage$*(4+1)=26
$Map_{...}$	7800→7800	7800*(3+26)=226200

Notons que la constante $partage$ est le facteur de partage du regroupement (taille moyenne de la partition de regroupement). Dans ce cas, on lui donne la valeur 5. Le coût de traitement relationnel du bloc **II** est estimé à 2000 + 390 + 15600 = 17990. Son coût de traitement objet est égal à 226200. Son coût de communication est égal au résultat de l'opération $\bowtie$ multiplié par 1, ce qui donne 7800. C'est le même coût qui est attribué au stockage local. Notons que dans le cas du bloc **II**, il est impératif de conserver la table de correspondance de la classe pour laquelle l'opération $condnew(att_4)$ est utilisée. Ceci est pris en compte dans le calcul du coût global de la représentation algébrique qui contient ce bloc.
Enfin, intéressons nous au coût du bloc **III** :

Opération	entrée→sortie	Coût
σ_{p_1}	2000→2000/10=200	2000
σ_{p_2}	390→390/5=78	390
$⟗_p$	200*78→200*78	200*78=15600
π_{att_1,att_4}	15600→15600/2=7800	15600
$o : getnew(att_1)$		1
$condnew(att_4)$		4
$o.a$+=$condnew(att_4)$		1+4=5
$Map_{...}$	7800→7800	7800*(1+5)=46800

Le coût de traitement relationnel du bloc **III** est estimé à 2000 + 390 + 15600 + 15600 = 33590. Son coût de traitement objet est égal à 46800. Le coût de communication est égal à la taille en octets du résultat de la projection (15600*(5+20)) soit 390000.

7.3 Génération de code

Une fois la phase d'optimisation terminée, le système procède à la génération d'un programme par bloc. Ce programme respecte la structure du bloc telle que donnée dans la figure 7.8.

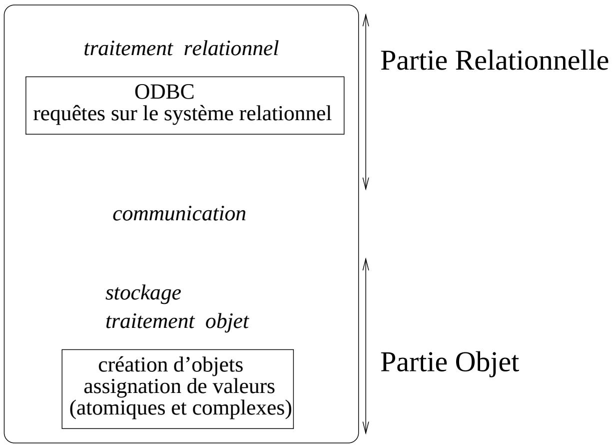


FIG. 7.8 – *Structure d'un Bloc*

Un programme de chargement est constitué d'*une partie relationnelle* qui contient une requête SQL générée dynamiquement à chaque invocation du programme et d'*une partie objet* qui contient les commandes de création et de mise à jour d'objets.

La partie relationnelle utilise des commandes ODBC pour invoquer des requêtes sur les relations sources. La partie objet est écrite dans le langage de programmation du système O_2 (O_2C) pour créer et de mettre à jour des objets. Nous utilisons la version 5.0.2.P.6 du système O_2. À noter que c'est à la phase de génération de code que le système décide s'il faut ou non générer une table de correspondance pour une paire (relation, classe). Si ce couple n'apparaît qu'avec des opérations «new», la table correspondante n'est pas créée sinon elle est stockée dans la base de données à objets. Les tables de correspondance utilisées sont détruites à la fin du chargement.

Un des principes de notre approche est de découper le chargement. Ceci nécessite de maintenir certaines structures de stockage similaires à celles utilisées pour stocker les informations d'un point de reprise. Nous les verrons plus en détail au paragraphe 7.4. Chaque programme est paramétré de deux façons : (i) avec un paramètre de temps qui signifie : exécuter ce programme pour un temps donné ou (ii) avec un paramètre de nombre d'objets à créer. Lorsque le temps est écoulé (ou le nombre d'objets à créer est atteint), le système stocke la position du dernier n-uplet utilisé dans chacune des relations impliquées dans un bloc. Lorsque l'utilisateur reprend le processus, le système génère la requête SQL correspondante de manière dynamique pour reprendre le chargement à partir du dernier n-uplet traité. L'utilisateur peut aussi exprimer le fait qu'il veut exécuter un programme entièrement sans contrainte de temps ni de nombre. Un programme sera terminé lorsque toutes les données relationnelles qu'il filtre ont été traitées. Sauf dans le cas d'une factorisation, le flux de données d'un programme est jeté. Les blocs étant ordonnés, un programme de chargement ne peut être invoqué que si tous les programmes correspondant aux blocs qui le précédent ont été exécutés entièrement (i.e pour tous les n-uplets filtrés).

Comme il n'existe pas de pilote ODBC (driver) pour oracle sur plate-forme Sun (utilisées pour nos tests), nous avons opté pour l'utilisation d'un autre SGBD relationnel appelé MiniSql. Ce système offre sa propre bibliothèque de fonctions d'accès à la base relationnelle. Ces fonctions travaillent de la même manière que les fonctions de l'interface ODBC.

Afin de mieux comprendre les programmes générés, nous donnons succinctement le code de leur génération. Ce code est en fait une méthode `genMigCode()` définie dans la classe `AlgRep.java`. Son pseudo-code est le suivant :

```
{ genCommonNames();
  genCors();
  genBlocNames();
  genFuns();
  genMigStart();
  genMigFinish();
  genMigAll();
  Pour chaque bloc b de cette représentation
     b.genObjFuncs();
     b.genRelPart();
     b.genObjPart();
     b.genMigBloc();
  FinPour
}
```

où,
`genCommonNames()` génère les structures contenant les informations relatives à une session relationnelle ainsi que le n-uplet courant.
`genCors()` génère les tables de correspondance.
`genBlocNames()` génère les structures contenant les informations relatives à un bloc (indicateur de fin de bloc, dernier n-uplet traité dans le bloc pour une classe, temps de traitement relationnel, traitement objet, communication et volume de stockage calculé à chaque exécution du programme relatif à ce bloc).
`genFuns()` génère les fonctions `inCor()` pour une paire (relation, classe) qui vérifie si la valeur de la clé de transformation courante a déjà été insérée dans la table de correspondance relative à cette paire et retourne l'objet correspondant si elle y est et `fieldPos()` utilisée pour récupérer la position d'un attribut relationnel dans le n-uplet courant.
`genMigStart()` et `genMigFinish()` qui génèrent les fonctions `migStart()` et `migFinish()` utilisées pour délimiter des appels de programmes de chargement dans l'environnement de contrôle. `migStart()` initialise la connection avec le système relationnel.
`genMigAll()` qui charge tout ce qui n'a pas été chargé. Elle peut être invoquée pour finir

le chargement. En l'occurence, elle peut être la seule fonction utilisée pour le chargement. Dans ce cas, chaque programme généré est une unique transaction.

D'autres méthodes sont définies sur les blocs d'une représentation algébrique. C'est le cas de `genObjFuncs()` qui génère les fonctions de création et d'assignation qui seront invoquées dans la partie objet d'un bloc. Ces fonctions sont principalement `createObj()` pour chaque fonction *new*, *getnew* et *condnew* apparaissant dans ce bloc, `atomicAssign()` pour chaque assignation atomique dans le bloc et `complexAssign()` pour chaque assignation complexe dans le bloc. `genRelPart()` est une autre méthode dans la classe `Bloc.java`. Elle génère une requête SQL paramétrée pour ce bloc. Quant à `genObjPart()`, elle combine dans une transaction, les fonctions de création et d'assignation générée pour un bloc. Enfin, `genMigBloc()` génère un programme `migBloc()` pour un bloc qui sera invoqué dans l'environnement de contrôle du chargement.

Partant de la représentation algébrique par défaut de l'exemple ci-dessous, le système produit trois programmes principaux : `migBloc0()` qui traite les conférences, `migBloc1()` qui traite les chercheurs et `migBloc2()` qui calcule le lien `participants`.

```
create class     Conf                              create class     Researcher
from relations   rConf                             from relations   rEmployee
with attributes  cName:string is confName,         with attributes  name:string is lname,
                 date:string is confDate,          where            function = 'Researcher'
end                                                end

create link      Researcher,Conf
from relations   rResConf,rConf,rEmployee;
with attributes  participants:set(Researcher) in class Conf
where            cEmp=emp && rConf=conf;
end
```

Lorsqu'un programme de chargement est invoqué avec le paramètre temps, le système utilise une fonction qui calcule le nombre approximatif d'objets qui doivent être créés dans ce temps. La fonction `migBloc0()` est alors invoquée pour un nombre d'objets `n`.

```
function migBloc0(n:integer) {
  classQuery=relPart0();
  res=msqlQuery(sock,classQuery);
  if (res!=-1)  objPart0(n);
}
```

avec les fonctions : `relPart0()` qui retourne une requête relationnelle à évaluer en fonction du dernier n-uplet traité et `objPart0()` qui s'occupe de créer les chercheurs :

```
function relPart0:string {
  classQuery="select * from rEmployee where function = 'Researcher'
              and emp > "+tmp+" order by emp";
  return classQuery;
}
```

La requête est construite de manière incrémentale à partir de la dernière valeur de clé de transformation pour la paire (*rEmployee*, `Researcher`).
C'est la fonction `objPart0` qui contient la transaction de création et de mise à jour des objets dans la base.

```
function objPart0(n:integer) {
  // initialisations
  TRANSACTION;
  Tant que le nombre d'objets à créer n'est pas atteint
      récupérer un n-uplet
      Si (ce n-uplet est nul)
        finished0=true;break;
      récupérer la valeur de la clé de tranformation
      courante dans ResearcherKeyval
      rootObj=createObj0_Researcher(ResearcherKeyval);
      Si (rootObj n'est pas nul)
        ResearcherComputeAtts0(rootObj);
  FinTant que
  COMMIT;
}
```

Pour calculer les attributs d'un chercheur créé dans `Bloc0`, une fonction `ResearcherComputeAtts0` est générée. Cette fonction fait appel à la fonction `ResearcherAtomicAssign0_name` puisque c'est le seul attribut de la classe `Researcher`.

```
function ResearcherAtomicAssign0_name() {
  // initialisations
  i=fieldPos(classRes,"empName");
  strcpy(self->name,classRow[i]);
}
```

S'il s'agit d'un attribut complexe, comme c'est le cas pour le bloc de calcul de l'attribut `participants` de la classe `Conf` calculé dans le bloc2 de la représentation par défaut, la fonction générée est donnée ci-dessous. Afin de simuler le regroupement par projet des chercheurs participant à un même projet, la condition de la boucle `Tant que` est utilisée. Elle signifie que tant que la valeur du projet est égale à la valeur courante, il s'agit de chercheurs appartenant au même projet (sachant que les n-uplets ont été préalablement triés par projet). Ainsi, aucun regroupement n'est effectué au préalable. `rootObj` est l'objet conférence à modifier. `innerSet` contient l'ensemble des chercheurs créés à travers le même objet `rootObj` et qui seront assignés à son attribut `participants`.

```
function ConfComplexAssign2_participants_Researcher() {
  // initialisations
  j=fieldPos(classRes,"conf");
  Tant qu'il s'agit du même objet à modifier
      innerObj=createObj2_Researcher();
      Si (innerObj n'est pas nul)
        innerSet+=set(innerObj)
      Si (n-uplet courant est nul)
        finished2=true;break;
      Passer au n-uplet suivant
  FinTant que
  rootObj->participants=innerSet;
}
```

7.4 Environnement de contrôle du chargement

Cet environnement permet à l'utilisateur de lancer les programmes de chargement générés à l'issue de la phase d'optimisation. Le contrôle consiste à (i) vérifier l'enchaînement des programmes, (ii) garder trace du dernier n-uplet traité pour chaque paire (relation, classe) et enfin (iii) gérer l'utilisation des tables de correspondance. Reprenons l'exemple des chercheurs et conférences. Nous donnons une description des racines de persistance générées pour la sauvegarde de points de reprise.

Racine	Type	Description
host	string	adresse IP de la machine sur laquelle tourne le serveur relationnel
currentTuple	integer	n-uplet courant
ConfKeyval	integer	dernière valeur courante de clé de transformation pour `Conf`
cor_Conf_rConf	[key:integer,obj:Conf]	table de correspondance de *rConf*,`Conf`
ResearcherKeyval	integer	dernière valeur courante de clé de transformation pour `Researcher`
cor_Researcher_rEmployee	[key:integer,obj:Conf]	table de correspondance de *rEmployee*,`Researcher`
finished0	boolean	indicateur pour bloc 0
finished1	boolean	indicateur pour bloc 1
finished2	boolean	indicateur pour bloc 2

Exemples d'instructions de chargement
Voici quelques exemples de programmes que l'utilisateur peut écrire. La fonction `migStart()` initialise les variables de l'environnement de contrôle et crée un client relationnel. La fonction `migFinish()` ferme la connection.

```
{
migStart();
migBloc0(0);
migBloc1(50);
migFinish();
}
```

Par convention, le chiffre 0 donné en paramètre d'un programme `migBloc` signifie, l'exécuter entièrement. Dans cet exemple, l'utilisateur demande à migrer la totalité des conférences et 50 chercheurs. Les valeurs des racines de persistance `ConfKeyval` et `ResearcherKeyval` ainsi que les tables de correspondance `cor_Conf_rConf` et `cor_Researcher_rEmployee` sont mises à jour en conséquence.

```
{
migStart();
migBloc0(20);
migBloc1(10);
migAll();
migFinish();
}
```

Dans cet exemple, le système crée 20 conférences et refusera éventuellement de charger les chercheurs tel que spécifié dans l'instruction `migBloc1(10)` si jamais la classe `Conf` n'est pas totalement peuplée. `migAll` migre le reste.

```
{
migStart();
migAll();
migFinish();
}
```

Enfin, cet exemple est un raccourci syntaxique à :

```
{
migStart();
migBloc0(0);
migBloc1(0);
migBloc2(0);
migFinish();
}
```

qui signifie «migrer la totalité de la base».

Chapitre 8

Tests et performances

Ce chapitre contient la validation de notre travail à travers des tests comparatifs de plusieurs exemples et programmes de chargement. Cette validation se situe à deux niveaux. D'abord, nous validons l'approche d'optimisation, ensuite les programmes de chargement générés et le modèle de coût.

8.1 Validation de l'approche d'optimisation

La validation de l'approche consiste à valider les étapes linéaire et énumérative de la phase d'optimisation.

– *La validation de l'étape linéaire* consiste à montrer, à travers des exemples, la réalisation des contraintes physiques.

– *La validation de l'étape énumérative* montre que, du point de vue du temps d'exécution, l'algorithme en largeur d'abord est plus intéressant que l'algorithme en profondeur d'abord. Elle compare ensuite les deux algorithmes pour déterminer lequel est plus intéressant par rapport à la taille du schéma objet. Ce point définit le seuil à partir duquel, pour des raisons d'espace, il est nécessaire d'utiliser l'algorithme en profondeur. Il est utile de déterminer ce seuil afin de choisir l'algorithme à appliquer en fonction du nombre de classes et d'attributs dans les classes.

8.1.1 Validation de l'étape linéaire

Nous vérifions que la représentation algébrique générée à la suite de cette étape réalise bien les contraintes physiques. Aucune autre optimisation n'est effectuée. La phase énumérative n'est pas enclenchée. Le modèle de coût n'est pas utilisé. Nous affichons d'abord la représentation par défaut et ensuite la représentation qui satisfait les contraintes pour chaque exemple. Pour cela, nous avons développé une interface java permettant d'afficher une représentation algébrique sous forme d'arbre. Rappelons que la réalisation d'une contrainte de regroupement d'objets sur disque repose essentiellement sur la création des objets à regrouper dans le même bloc.

Premier exemple. Cet exemple définit les classes `Researcher` et `Project` et montre la réalisation d'une contrainte de regroupement des objets d'une même classe et d'une contrainte d'ordre dans la création des objets de cette classe.

```
create class      Researcher                          create class      Project
from relations    rEmployee                           from relations    rProject
with attributes   name:string is empName;             with attributes   pName:string is prjName;
                  salary:real is bonus;                                 town:string is address;
                  function:string is function;
where             function='researcher'

create link       Project,Researcher
with attributes   boss:Researcher in class Project
where             boss=emp;
```

Avec la contrainte,

```
class        Project
order by     prjDpt
cluster on   class
```

La représentation par défaut générée contient trois blocs : (i) un bloc qui crée les chercheurs et leurs attributs atomiques, (ii) un bloc qui crée les projets et leurs attributs atomiques et un bloc qui calcule le lien `boss` de chaque projet. La phase linéaire réordonne les blocs en commençant par celui qui crée les projets dans lequel le tri demandé est inséré. Nous remarquons également l'utilisation de la fonction de création «new» dans les blocs de peuplement des projets et des chercheurs et l'utilisation de «getnew» dans le bloc de calcul de l'attribut `boss`.

Deuxième exemple. Il contient les classes `Researcher`, `Project` et une nouvelle classe de conférences `Conf` ainsi que les liens `manages` et `attends`. Il montre la satisfaction de la contrainte ci-dessous :

```
class   Researcher
if      salary ≤ 5000
cluster on all
```

```
create class     Conf                              create link      Researcher,Conf
from relations   rConf                             from relations   rEmployee,rConf,rResConf;
with attributes  cName:string is confName;         with attributes  attends:set(Conf)
                 place:string is confPlace;                         in class Researcher
                 date:string is confDate;          where            conf=rConf and emp=cEmp;
where            confTopic='Mixed'

create link      Project,Researcher
with attributes  manages:Project in class Researcher
where            boss=emp;
```

Alors que la représentation par défaut peuple les chercheurs, les projets, les conférences et calcule les liens `manages` et `attends`, la représentation optimisée génère d'abord un premier bloc qui crée les chercheurs dont le salaire est inférieur à 5000 avec le projet qu'ils gèrent et les conférences auxquelles ils ont assisté. Les blocs qui suivent complètent le reste de la base.

Troisième exemple. Dans celui-ci, les classes `Researcher`, `Project` et `Conf` sont conservées, le lien `manages` est augmenté du lien inverse `boss` pour constituer une association, une nouvelle classe `Dept` et un liens `department` sont définis. La classe `Dept` est définie à partir des relations *rDeptRocq* et *rDeptSophia*. Un regroupement plus profond est demandé.

```
class        Researcher
cluster on   manages(department),attends *
```

```
create class     Dept
from relations   rDeptRocq,rDeptSophia
with attributes  dName:string is rName  in relation rDeptRocq,
                                 sName  in relation rDeptSophia;
```

Le lien `department` défini ci-dessous sera créé dans la classe `Project` mais ne sera calculé que pour les départements situés à «Sophia».

```
create link      Dept,Project
from relations   rDeptSophia,rProject;
with attributes  department:Dept in class Project
where            prjDpt=sDpt;
```

L'étape linéaire génère d'abord un bloc qui satisfait la contrainte demandée soit un bloc qui crée les chercheurs avec les conférences auxquelles ils ont assisté et calcule les valeurs atomiques de chaque conférence créée (*). Ce bloc attribue à chaque chercheur le projet qu'il gère (si celui-ci existe) à travers son attribut `manages` et à chaque projet créé ainsi, il attribue son département. Les blocs qui suivent sont respectivement, le bloc qui crée le reste des départements de «Sophia» (fonction «condnew»), le bloc qui crée tous les départements de «Rocquencourt» (fonction «new»), le bloc qui crée le reste des conférences, le bloc qui créé le reste des projets (fonction «condnew») et enfin, le bloc qui calcule le lien `department` entre les projets et les départements situés à «Rocquencourt» (fonction «getnew»).

Quatrième exemple. Ce dernier exemple montre un cas d'héritage combiné à un regroupement à travers un attribut complexe. Nous conservons les classes`Researcher` et `Conf` et définissons `ConfDB`, `ConfAI` et `ConfMIN` comme sous-classes de `Conf`. Le lien `attends` est augmenté de l'attribut `participants` pour constituer une association. La contrainte ci-dessous est demandée :

```
class         Researcher
cluster on    attends
```

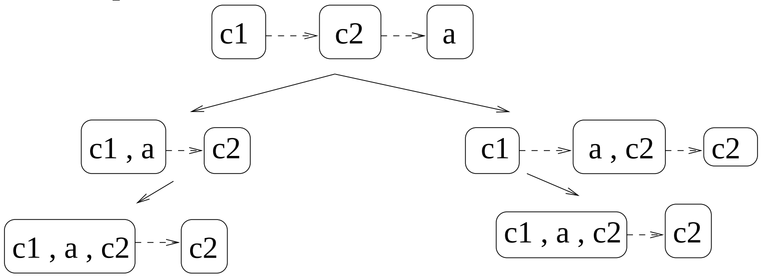
```
create class     ConfDB                       create class      ConfAI
inherits         Conf                         inherits          Conf
from relations   rConf                        from relations    rConf
extension        DBconfs type list            extension         AIconfs type list
where            confTopic='Database'         where             confTopic='Intelligence'

create class     ConfMIN                      create link       Researcher,Conf
inherits         Conf                         from relations    rEmployee,rConf,rResConf;
from relations   rConf                        with attributes   attends:set(Conf)
extension        MINconfs type list                             in class Researcher
where            confTopic='Datamining'                         participants:set(Researcher)
                                                                in class Conf
                                              where             conf=rConf and emp=cEmp;
```

La représentation par défaut est constituée de 10 blocs. Les 5 premiers peuplent les 5 classes. Ensuite, il y a un bloc qui calcule l'attribut `participants` pour chacune des classes `Conf`, `ConfDB`, `ConfAI` et `ConfMIN`. Enfin, un bloc est généré pour le calcul de l'attribut `attends` des chercheurs. Cet attribut est calculé avec l'opérateur «+=».
La représentation générée à la suite de la phase linéaire contient d'abord un bloc qui crée les chercheurs avec leur attribut `attends` et le reste des blocs après.

8.1.2 Validation de l'étape énumérative

Les tests effectués pour cette validation comparent les temps d'exécution et la mémoire nécessaires à chaque algorithme de recherche. Ils sont totalement indépendants de la taille des bases relationnelles sous-jacentes et dépendent de la taille du schéma objet. Le modèle de coût est fixé de sorte à ne privilégier aucun de ses paramètres (tous les paramètres ont un coefficient égal à 1). Ceci assure que l'algorithme de recherche sera appliqué dans tous les cas. Afin de ne pas influencer les résultats des tests, aucune heuristique ne limite l'espace de recherche. Pour avoir une idée de ce qui est généré, on suppose un exemple contenant deux classes et un attribut complexe entre ces classes. L'arbre de recherche est alors :



Premiers tests. La première série de tests part d'un exemple simple contenant deux classes `Researcher` et `Conf` avec un attribut complexe `participants1` qui représente l'ensemble des chercheurs qui assistent à une conférence. Cet exemple se complexifie à chaque étape en rajoutant les attributs `participants2, participants3, ...`. Nous faisons tourner les algorithmes en largeur et en profondeur sur chacun de ces exemples. La mémoire donnée est la mémoire maximale utilisée par les deux algorithmes.

exemple 1	en largeur	en profondeur
mémoire maximale (en octets)	249064	273064
temps d'exécution	1 sec 268 msec	977 msec

exemple 2	en largeur	en profondeur
mémoire maximale (en octets)	325520	340528
temps d'exécution	4 sec 852 msec	5 sec 999 msec

exemple 3	en largeur	en profondeur
mémoire maximale (en octets)	929744	423608
temps d'exécution	1 min 32 sec 740 msec	8 min 41 sec 565 msec

Nous commençons à voir que la mémoire consommée à la quatrième étape de l'algorithme en largeur est plus de deux fois la mémoire maximale de l'algorithme en profondeur.

exemple 4	en largeur	en profondeur
mémoire maximale (en octets)	6199792	2982416
temps d'exécution	49 min 20 sec 712 msec	1hs 16 min 30 sec

Nous nous arrêtons à cet exemple qui contient 4 attributs complexes. Bien que le temps que prend l'algorithme en profondeur prenne plus de temps à s'exécuter que l'algorithme en largeur, la mémoire consommée par ce dernier est 2 fois plus importante. Regardons maintenant la deuxième série de tests.

Deuxièmes tests. Dans cette série, nous partons du même exemple et à chaque attribut complexe `participants`, nous associons son inverse `attends` pour constituer une association. À chaque fois, nous étendons l'exemple à deux associations puis à trois etc.

exemple 1	en largeur	en profondeur
mémoire maximale (en octets)	487504	343568
temps d'exécution	20 sec 959 msec	44 sec 453 msec

exemple 2	en largeur	en profondeur
mémoire maximale (en octets)	10328768 et il s'arrête	5803112
temps d'exécution		1hs 57 min

L'algorithme en largeur s'arrête par manque d'espace. Ces deux séries de tests nous montrent que malgré le fait que l'algorithme en profondeur dure plus longtemps que l'algorithme en largeur, il est indispensable de l'utiliser dans le cas où le schéma objet est constitué de plus de 4 attributs complexes ou de plus de 2 associations. Bien entendu, s'il n'existe aucun attribut complexe dans le schéma objet, c'est l'algorithme en largeur qui est utilisé.

Intuitivement, sachant qu'un niveau dans l'espace de recherche contient toutes les possibilités d'une première fusion partant des attributs complexes du schéma objet, le volume d'un niveau grossit en fonction de ce nombre d'attributs. L'algorithme de recherche en largeur d'abord consomme beaucoup de mémoire dans ce cas. Également, lorsque le graphe de composition du schéma objet est profond (une classe qui en référence une autre qui à son tour en référence une autre etc), c'est l'algorithme en profondeur qui consomme beaucoup de mémoire.

Conclusion. Nous avons vu que malgré l'utilisation de l'algorithme en profondeur, l'espace de recherche croît considérablement. Nous pensons qu'il serait intéressant d'utiliser des heuristiques pour réduire la mémoire consommée.
Une heuristique possible serait de limiter la profondeur de l'arbre de recherche. Cette contrainte peut être facilement appliquée à une recherche en largeur dans laquelle l'arbre de recherche est généré niveau par niveau. Dans ce cas, certaines solutions ne seraient pas exploitées.
Une autre heuristique serait de limiter les nombre d'attributs complexes à partir desquels les fusions sont appliquées, par exemple, en ne traitant qu'un seul des attributs qui constituent une association. Ainsi, sachant que les attributs `attends` et `participants` forment une association, on pourrait traiter toutes les possibilités de fusion en partant de l'attribut `attends` puisque l'on sait qu'une réécriture possible est de le fusionner avec l'attribut `participants`.
Une autre heuristique serait de combiner les poids attribués aux paramètres du modèle de coût avec une profondeur de l'arbre de recherche. Par exemple, sachant que l'on veut optimiser le temps de communication et que pour cela, on sait qu'au delà d'un certain nombre de

fusions, on ne rencontrera jamais de solution plus intéressante que les solutions courantes, on peut s'arrêter. Bien sûr cela exigerait une étude plus approfondie de la relation entre les réécritures définies et des paramètres du modèle de coût. Dans ce qui suit, nous tentons, travers quelques tests, de tirer des conclusions au cas par cas.

8.2 Validation des programmes générés et du modèle de coût

Afin de réaliser des tests, nous avons généré une base de données relationnelle dont le schéma est le suivant :

rEmployee[emp:int,empName:char(20),salary:real,bonus:int,function:char(20)]
rProject[prj:int,prjName:char(20),boss:int,address:char(20),prjDpt:int]
rConf[conf:int,confName:char(20),confPlace:char(20),confDate:int,confTopic:char(20)]
rResConf[cEmp:int,rConf:int]

La taille de chaque relation est donnée dans le tableau suivant :

relation	taille (en Moctets)	nombre de n-uplets
rEmployee	7	100000
rProject	4	50000
rConf	3.6	10
rResConf	10	550000

La base de données relationnelle fait environ 25 méga octets. Afin de «bénéficier» de ce volume, nous ne posons aucune condition pour le peuplement des classes dérivées de ces relations (par exemple tous les employés sont chercheurs). Dans la relation *rResConf*, la moitié des chercheurs sont associés à 10 conférences, tous les autres ont assisté à seulement une conférence.

Dans ce qui suit, nous commençons par étudier chacun des paramètres du modèle de coût séparément. Pour cela, il suffit de pondérer de 1 le paramètre étudié et de 0 les autres. Dans un premier temps, nous regardons sur des exemples quelles sont les représentations algébriques générées. Dans un deuxième temps, nous comparons le temps de chargement total des programmes générés et des programmes issus de la représentation par défaut.
Tous les exemples choisis sont bien évidemment sans contrainte.
Pour réaliser le chargement, il suffit de lancer dans l'ordre, les programmes `migBloci()` générés pour chaque bloc `i`. Afin de bénéficier de la possibilité de lancer un programmes avec un paramètre «nombre d'objets à créer à la fois» et de libérer ainsi la mémoire régulièrement, nous avons écrit une boucle simple pour chaque programme. Par exemple, si le bloc 1 crée tous les chercheurs et calcule leurs attributs atomiques, nous utilisons la boucle ci-dessous :

```
run body {
 migStart(host);
   o2 integer i;
   for (i=1;i<=100;i++)
     migBloc1(1000);
 migFinish();
}
```

Avec cela, le système va exécuter plusieurs transactions qui créent chacune 1000 objets jusqu'à ce que les chercheurs soient complètement créés et leurs attributs atomiques calculés. La mémoire temporaire est ainsi régulièrement libérée. Si une interruption intervient, seule la dernière transaction lancée est annulée.

8.2.1 Privilégier le temps de communication

En pondérant le coût de communication d'une représentation algébrique de 1 et les autres coûts de 0, on pourrait s'attendre à ce que l'optimiseur produise en résultat la représentation par défaut puisque c'est elle qui minimise les jointures (voir chapitre 6). L'exemple que nous utilisons contient les classes `Researcher` et `Conf` et le lien `attends`. L'optimiseur produit deux blocs. Le premier bloc crée tous les chercheurs avec leurs attributs atomiques et leur attribut `attends` en calculant les attributs atomiques des objets de la classe `Conf` créés à travers l'attribut `attends`. Le deuxième bloc créé le reste des conférences et leur assigne leurs attributs atomiques. Partant de cet exemple, nous générons les programmes qui correspondent à la représentation par défaut et ceux qui correspondent à la représentation optimisée et comparons les temps de communication et de chargement effectifs des deux scénarios. Les temps sont donnés en secondes.

	temps de communication	temps de chargement
représentation par défaut	2838.67	2h 1mn
représentation optimisée	410.23	1h 33mn

8.2.2 Privilégier le temps de traitement relationnel

Nous savons que notre stratégie de recherche part du principe selon lequel le système relationnel est le mieux placé pour effectuer les opérations relationnelles. Par conséquent, l'optimisation de ce temps revient dans notre cas à (i) privilégier la factorisation pour éviter de réévaluer certaines requêtes relationnelles et à (ii) réduire le nombre d'itérations sur la même relation. En y réfléchissant un peu, ce dernier point pourrait être réalisé en effectuant certaines fusions. On pourrait alors se demander, étant donnés deux blocs : l'un qui calcule un attribut complexe et l'autre son inverse, est ce que c'est plus intéressant de fusionner les deux blocs en un seul (pour réduire le nombre d'itérations sur les relations sources) ou est ce que c'est plus intéressant de faire une factorisation ?

Nous avons choisi de prendre un exemple qui contient la classe `Researcher` et la classe `Conf` ainsi que l'association `attends/participants`. Malheureusement, nous n'avons pas pu vérifier la question posée plus haut puisque nous n'avons pas implanté la factorisation. Nous nous sommes contentés alors de vérifier les temps de traitement relationnel de la représentation par défaut et celui de la représentation optimisée. La représentation optimisée produit deux blocs : un premier bloc qui crée tous les chercheurs avec leurs attributs atomiques et qui calcule leur attribut `attends` ainsi que les valeurs des attributs atomiques de chaque conférence créée et un deuxième bloc qui crée le reste des conférences et assigne à toutes les conférences leurs attributs atomiques ainsi que leur attribut `participants`. Les résultats sont les suivants :

	temps de traitement relationnel	temps de chargement
représentation par défaut	3787.58	4h 10mn
représentation optimisée	3719.74	3h 16mn

Plusieurs améliorations peuvent être apportées afin d'optimiser le temps de traitement relationnel. Tout d'abord, nous avons réalisé que lorsque le volume des données chargées est important, il est impossible de se contenter, pour la partie relationnelle, de la seule condition définie jusque là dans la clause «where». En effet, le chargement étant incrémental, nous gardions comme valeur, la dernière valeur de l'argument de la fonction de Skolem qui crée les objets d'un bloc. La requête SQL générée par l'appel suivant au même bloc consistait à augmenter le filtre d'une classe de la condition «argument de la fonction de création supérieur à la dernière valeur traitée». Le système sélectionnait alors tous les n-uplets de la relation ou les relations (lorsqu'il s'agissait d'une jointure) qui satisfaisaient la requête demandée. Lorsque les relations sous-jacentes sont très importantes, un gros volume de données peut alors être chargé sans pour autant qu'on en ait besoin. L'amélioration apportée à cela est d'avoir une borne supérieure dans la condition qui devient alors : «argument de la fonction de création supérieur à la dernière valeur traitée et inférieur à la dernière valeur traitée + le nombre d'objets à créer». Le nombre d'objets à créer est donné par l'utilisateur. Bien sûr, cette condition suppose une distribution uniforme des n-uplets d'une relation. Dans nos exemples de tests, le numéro d'un employé est incrémenté de 1 d'un n-uplet à l'autre.

C'est bien évident que l'utilisation d'index au niveau relationnel améliorerait le temps de traitement d'une requête. De plus, l'utilisation d'informations sémantiques supplémentaires permettrait par exemple d'éviter d'évaluer certaines jointures. Une autre amélioration possible serait de tenir compte de l'évaluation de requêtes pour peupler des classes qui héritent les unes des autres. La classe mère et ses sous-classes sont toujours définies à partir des mêmes relations sources. Les prédicats permettant de les peupler sont disjoints. Au lieu d'évaluer une requête par classe, il peut être possible de la réécrire en une requête dont le prédicat serait construit par disjonction des prédicats de la classe mère et de ses sous-classes. L'information selon laquelle un n-uplet serait considéré plutôt qu'un autre (pour savoir dans quelle classe un objet doit être créé) doit alors être enregistrée dans un attribut relationnel qui sera testé à chaque fois qu'un objet doit être créé.

8.2.3 Privilégier le temps de traitement objet

Il s'agit, dans ce cas, de privilégier l'accès séquentiel à l'accès aléatoire aux objets de la base cible et également, de minimiser le nombre d'entrées/sorties en accédant à un même objet le minimum de fois possible. Le premier point est réalisé par l'introduction de tris dans les blocs. Le même tri est introduit dans le bloc créant les objets d'une classe et dans celui qui calcule leur valeur de façon à ce que l'accès à ces objets se fasse dans l'ordre dans lequel ils ont été créés ce qui minimise les défauts de page. Cette remarque est valable aussi bien pour l'accès disque que pour l'accès aux tables de correspondance. Le deuxième point revient à regrouper la création d'un objet et le calcul de sa valeur (attributs atomiques et attributs complexes) dans un même bloc de sorte à n'accéder à l'objet qu'une seule fois.

L'exemple que nous prenons contient les classes `Researcher`, `Conf` et `Project` ainsi que les liens `attends` et `manages` dans la classe `Researcher`. La représentation optimisée génère trois blocs : un premier bloc qui crée les projets et calcule leurs attributs atomiques, un deuxième bloc créant les conférences avec leurs attributs atomiques et enfin un dernier bloc qui crée les chercheurs et calcule tous leurs attributs. Ainsi, l'accès aux chercheurs est optimisé. Regardons les coûts engendrés :

	temps de traitement objet	temps de chargement
représentation par défaut	5770.34	3h 28mn
représentation optimisée	3880.56	2h 12mn

En réalisant ce test, nous avons réalisé que certaines nouvelles optimisations pourraient être introduites. Sachant que tous les objets d'une classe ont été créés dans des blocs précédents, une optimisation possible était d'utiliser des fonctions «getnew» pour récupérer l'identifiant d'un objet de la table de correspondance appropriée et éviter pour cela de tester l'existence de cet objet. Dans ce cas, nous sommes sûrs que les valeurs atomiques de l'objet (s'il y en a) ont été calculées auparavant. Par conséquent, à chaque fois que c'est l'opération «getnew» qui est utilisée dans un bloc et même si le calcul des attributs atomiques est demandé, celui-ci est occulté. On pourrait même faire mieux. En rajoutant un attribut booléen dans chaque table de correspondance, on pourrait enregistrer l'information selon laquelle les attributs atomiques déjà calculés ou pas. Ceci éviterait de recalculer les valeurs atomiques d'un objet. Une autre solution serait d'être capable de vérifier, à partir de l'identifiant d'un objet, si ses valeurs atomiques ont été calculées ou pas. Seulement, dans le système O_2, la notion de valeur nulle n'existe pas.

8.2.4 Privilégier le volume de stockage

Pour vérifier ce paramètre, nous avons choisi de prendre un exemple contenant les classes `Researcher` et `Conf` ainsi que le lien `attends` défini dans `Researcher`. La représentation optimisée est constituée de deux bloc : un bloc qui crée tous les chercheurs avec leurs attributs atomiques et calcule aussi la valeur de leur attribut complexe `attends` et un bloc qui crée le reste des conférences et leurs attributs atomiques. Sachant que la taille d'un entier dans le système O_2 est de 4 octets et que celle d'un identifiant d'objet est de 8 octets, les tailles des tables de correspondance sont :

	volume de stockage (en kilos octets)
représentation par défaut	80
représentation optimisée	28.5

Nous voyons bien que le volume de stockage est optimisé puisque la table (*rEmployee*,`Researcher`) est supprimée.
Certains systèmes offrent la notion d'identifiants logiques. Celle-ci permet de ne créer un objet (identifiant physique) que lorsque toutes les informations le concernant sont disponibles. Elle pourrait être exploitée pendant le chargement. Le système O_2 n'offre pas cette possibilité. L'utilisation de nos tables de correspondance permet de s'abstraire des particularités du système à objets cible utilisé.

8.2.5 Compromis entre les différents paramètres

Dans cette deuxième partie de tests, nous avons choisi de travailler sur deux exemples. Dans le premier cas, nous avons repris l'exemple utilisé dans le paragraphe 8.2.3 et avons pondéré tous les paramètres du modèle de coût à 1. Cet exemple a pour objet de comparer le temps que prend le chargement par défaut et le chargement réalisant un compromis entre tous les paramètres du modèle de coût. La représentation optimisée génère trois blocs : un bloc qui crée les conférences avec leurs attributs atomiques, un deuxième bloc qui crée les projets avec leurs attributs atomiques et un dernier bloc qui crée les employés et calcule leurs attributs atomiques ainsi que leurs liens `manages` et `attends`. Nous vérifions si les programmes issus de la représentation optimisée offrent le meilleur compromis entre ces différents paramètres.

	communication	relationnel	objet	temps global
représentation par défaut	2954.79	2650.15	5770.34	3h 14mn
représentation optimisée	490.49	2644.16	3880.56	2h 06mn

Dans le deuxième exemple, nous reprenons l'exemple du paragraphe 8.2.1 avec la contrainte physique suivante :

```
class        Researcher
cluster on   attends
```

Cet exemple a pour objet de vérifier un regroupement sur disque demandé par l'utilisateur lors du chargement. Pour cela, nous exécutons une requête OQL sur la base générée par les programmes par défaut et celle générée par les programmes optimisés. La requête est choisie de sorte à bénéficier du regroupement sur disque demandé :
`select c.confName from r in Researchers, c in r.attends`. Les résultats sont alors :

	temps d'exécution de la requête
base par défaut	154.7
base optimisée	118.8

Conclusion

Nous avons présenté un travail qui porte sur le développement de techniques d'optimisation du chargement en masse dans une base de données.

Réalisations.

En premier lieu, nous avons étudié les problèmes posés par le chargement en masse et l'état de l'art en la matière. Nous en avons déduit qu'un bon système de chargement doit (i) tenir compte de contraintes exprimées sur la base de données cible, (ii) se composer de plusieurs parties pouvant être exécutées indépendamment, (iii) avoir une durée paramétrable et surtout (iv) offrir des techniques d'optimisation adaptées qui optimisent le temps global du chargement.
Nous avons ensuite réfléchi au choix d'un modèle de données qui permet de représenter les types de données qui nous intéressent à savoir, les données de type relationnel et de type objet du fait que l'on s'intéresse à charger des données dans une base relationnelle ou à objets, mais également des données provenant de fichiers ASCII. Dans la pratique, ce type de fichiers est souvent utilisé pour le chargement. Nous avons fini par opter pour le modèle IQL auquel nous avons ajouté la notion de collection ordonnée d'éléments.
Ensuite, nous avons travaillé sur l'expression déclarative du chargement. Pour cela, nous avons utilisé une algèbre objet et des fonctions de création et d'assignation de valeurs des entités d'une base de données. En utilisant l'algèbre, chaque programme de chargement correspond à une expression algébrique. Nous avons alors défini une nouvelle classe d'équivalences algébriques qui exploitent l'effet de bord de chaque expression algébrique sur la base cible. Ces équivalences nous permettent de réécrire des programmes en d'autres programmes équivalents.

En second lieu, nous avons validé notre approche en l'appliquant au cas du chargement de données relationnelles dans un système à objets. Nous avons développé un prototype permettant d'exprimer des transformations relationnel-objet à travers un langage de vues. Ce langage permet aussi d'exprimer des contraintes d'organisation physique que l'on aimerait voir réalisées sur la base à objets.
Une fois les transformations et contraintes données, le système se charge de les traduire dans l'algèbre à des fins d'optimisation. Nous avons alors développé une stratégie de recherche et un modèle de coût dont l'objectif est d'optimiser le temps de chargement et de tenir compte de la décomposition de ce processus en différentes parties et de l'organisation des données

cibles. La stratégie de recherche applique d'abord une succession de réécritures algébriques en vue de satisfaire les contraintes exprimées sur la base cible et tente ensuite d'optimiser le temps et les ressources allouées au chargement. Elle se base sur un modèle de coût pour comparer les différentes solutions et en choisir la meilleure. Par défaut, les paramètres du modèle sont pondérés de 1. L'utilisateur peut attribuer d'autres poids à ces paramètres. La prise en compte du découpage en transactions est faite dans le modèle de coût. En effet, lorsque deux solutions ont le même coût (somme des coûts de traitement, de communication et du volume de stockage), le système privilégie celle qui a le plus grand nombre de blocs.
Nous avons écrit un générateur de code qui prend en entrée une représentation optimisée (dans ce cas, celle qui est générée par la phase d'optimisation) et produit les transactions correspondantes. L'utilisateur peut alors lancer chaque transaction plusieurs fois en spécifiant le nombre d'objets à créer à chaque fois. Si le chargement est interrompu, seule la dernière transaction non validée est défaite.

Prototype et extensions.

Notre prototype constitue une plate-forme d'essais sur laquelle nous avons effectué quelques tests de validation et de performance. Pour réaliser les tests de validation, nous avons choisi plusieurs scénarii qui offrent différentes possibilités d'optimisation et qui ont permis de valider les deux étapes de l'optimisation : l'étape linéaire qui correspond à la réalisation des contraintes physiques sur les objets de la base cible et l'étape énumérative dans laquelle un algorithme en largeur-d'abord ou en profondeur-d'abord est appliqué. Cette première phase de test nous a permis de nous rendre compte que les algorithmes proposés doivent être optimisés en temps soit en développant d'autres algorithmes ou en appliquant les heuristiques que nous proposons et qui permettent de réduire l'espace de recherche.
La deuxième phase de test a porté sur l'étude des performances des programmes de chargement générés. Nous avons pondéré chacun des paramètres du modèle de coût de manière indépendante des autres. Par exemple, pour tester le temps de communication, nous avons pondéré le coût de communication d'une représentation algébrique de 1 alors que les autres coûts étaient pondérés de 0. Dans chacun des cas, nous avons testé les coûts de chargement des programmes générés par la représentation par défaut et de ceux générés par la représentation optimisée. Nous nous sommes ensuite intéressés à attribuer un poids de 1 aux différents coûts (compromis) et avons comparé dans ce cas les programmes par défaut et les programmes optimisés. Enfin, le dernier test avait pour objet de vérifier le regroupement des objets de la base chargées en comparant le temps que prend une requête sur une base chargée avec les programmes par défaut et sur une base chargée en utilisant les programmes optimisés.

L'architecture du prototype facilite son adaptation au chargement d'autres types de données en greffant, par exemple, les modules d'optimisation, de génération de code et l'environnement de contrôle de la migration à un système existant qui nécessite un chargement de données efficace (intégration de données hétérogènes, entrepôts de données, ...). Cette adaptation nécessiterait de (i) traduire les primitives du langage de conversion (ou de transformation), utilisé dans ce système, dans notre algèbre et de laisser l'optimiseur faire son travail, (ii) d'adapter le code généré aux systèmes source et cible et (iii) d'estimer le modèle de coût en fonction des possibilités offertes par les systèmes source et cible. Notre stratégie de recherche étant générique, c'est dans le modèle de coût que les spécificités des systèmes source et cible devront être codées.

Perspectives.

Nous pensons que les techniques d'optimisation développées dans le cadre de cette thèse peuvent s'adapter à la construction d'entrepôts de données. L'état de l'art en la matière est pauvre. Seulement un travail préliminaire à l'Université de Stanford propose des algorithmes de chargement dans les entrepôts de données qui tiennent compte de l'interruption de ce processus [WJHM98]. Dans l'industrie, DB2 et Teradata fournissent des outils de chargement qui tentent de minimiser la quantité de travail fourni dans le cas d'une interruption du processus de chargement. L'inconvénient dans ces systèmes est que cette optimisation est effectuée en dernier lieu (i.e. après avoir importé et restructuré les données sources). Nous pensons qu'appliquer à ce cas des techniques d'optimisation développées dans le cadre de cette thèse constitue une piste de recherche intéressante.

Une autre direction de recherche serait d'appliquer ces techniques à la migration de fichiers du génome humain dans une base de données. Les applications qui traitent de ce type de données pourraient bénéficier des possibilités offertes à l'utilisateur sur l'expression de l'organisation des données sur disque afin d'optimiser les accès futurs aux données.

Nous avons aussi envisagé d'étudier la propagation des mises à jour effectuées sur les données sources pendant le chargement. Nous pensons que du fait que nos techniques facilitent le chargement incrémental des données, il est possible d'envisager la prise en compte des mises à jour sur la base source pendant le chargement. En effet, nous pouvons imaginer d'effectuer une décomposition «intelligente» du chargement en fonction des «catégories de mises à jour» effectuées sur la base source. Les programmes générés seront alors utilisés non seulement pour le peuplement de la base cible mais aussi pour sa maintenance.

Enfin, en exploitant les possibilités de réorganisation des données sur disque offertes par le langage *RelOO*, notre système peut être utilisé pour réorganiser les données dans une base existante. Les systèmes de base de données offrent un moyen de réorganiser les données déjà existantes. Il serait intéressant d'étudier de plus près les performances de tels systèmes et de les comparer avec le système *RelOO*. Cette étude permettrait d'améliorer le système *RelOO* dans cette direction.

Bibliographie

[Aa93] J. Albert and al. Automatic importation of relational schemas in Pegasus. In *Intl. workshop on Research Issues on Data Engineering: Interoperability in Multidatabase System*, 1993.

[AB91] S. Abiteboul and A. Bonner. Objects and Views. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 238–247, 1991.

[ABD+94] T. Atwood, D. Barry, J. Duhl, J. Eastman, G. Ferran, D. Jordan, M. Loomis, and D. Wade. *The Object Database Standard: ODMG-93, Release 1.2*. Morgan Kaufmann, San Francisco, 1994. http://www.odmg.org.

[AK89] S. Abiteboul and P. Kanellakis. Object Identity as a Query Language Primitive. In *Proc. ACM SIGMOD Symp. on the Management of Data*, volume 18, pages 159–173, Portland, Oregon, June 1989. ACM.

[AY97] S. Amer-Yahia. From Relations to Objects: The RelOO prototype. In *ICDE conference – Industrial session*, Birmingham, England, April 1997.

[AYBdS97] S. Amer-Yahia, P. Brèche, and C. Souza dos Santos. Object Views and Updates. *Ingénierie des Systèmes d'Information*, 5(1):63–89, April 1997. Also in BDA'96.

[AYCD98] S. Amer-Yahia, S. Cluet, and C. Delobel. Bulk Loading Techniques for Object Databases and an Application to Relational Data. In *Proc. of Intl. Conf. on Very Large Data Bases*, NY, USA, August 1998.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-Oriented Database System — The Story of O_2*. Morgan Kaufmann, San Mateo, California, 1992.

[Ber91] E. Bertino. A View Mechanism for Object-Oriented Databases. In *Proc. of Intl. Conf. on Extending Data Base Technology*, pages 136–151, Vienna, March 1991.

[BGD97] A. Behm, A. Geppert, and K.R. Dittrich. On the Migration of Relational Schemas and Data to Object-Oriented Database Systems. In *Proc. of Intl. Conf. on Re-Technologies in Information Systems*, Klagenfurt, Austria, December 1997.

[CBB+97] R.G.G. Cattel, D. Barry, D. Bartels, M.Berler, J. Eastman, S. Gammerman, D. Jordan, A. Springer, H. Strickland, and D. Wade. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, San Francisco, 1997. http://www.odmg.org.

[CBS94] R.H.L. Chiang, T.M. Barron, and V.C. Storey. Reverse Engineering of Relational Databases: Extraction of an EER Model from a Relational Database. *Data and Knowledge Enginnering*, 12:107–142, 1994.

[CDSS98] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your mediators need data conversion! In *Proc. ACM SIGMOD Symp. on the Management of Data*, Seattle, Washington, June 1998.

[CHS+95] 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*, pages 124–131, Los Alamitos, California, March 1995.

[CM94] S. Cluet and G. Moerkotte. Classification and Optimization of Nested Queries in Object Bases. In *Bases de Données Avancées*, pages 331–349, 1994.

[CP82] K. Steiglitz C.H. Papadimitriou. *Combinatorial Optimization: Algorithms and Complexity.* Prentice–Hall, 1982.

[DA97] S. Davidson and A.S.Kosky. WOL: A Language for Database Transformations and Constraints. In *Proc. of Intl. Conf. on Data Engineering*, pages 55–65, UK, April 1997.

[DCa94] S. B. Davidson, A. S. KoskyM. Carey, and al. Shoring up persistent applications. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 383–394, 1994.

[DdST95] C. Delobel, C. Souza dos Santos, and D. Tallot. Object Views of Relations. In *Proceedings of the 2nd International Conference on Applications of Databases - ADB '95*, San José, California, December 1995.

[DK97] S. B. Davidson and A. S. Kosky. Wol: A language for database transformations and constraints. In *Proc. of Intl. Conf. on Data Engineering*, pages 55–65, Birmingham, UK, April 1997.

[Exe97] F. Exertier. ROBIN: Generating Object-Oriented and WEB Interfaces for Relational Databases. Technical report, Bull, January 1997. White paper.

[Fa87] D. Fishman and al. IRIS: an Object-Oriented Database Management System. In *ACMTOIS*, volume 5, January 1987.

[FFK+98] M. Fernandez, D. Florescu, J. Kang, A. Y. Levy, and D. Suciu. Catching the Boat with Strudel: Experiences with a Web-Site Management System. In *Proc. ACM SIGMOD Symp. on the Management of Data*, Seattle, Washington, June 1998.

[Fon97] J. Fong. Converting Relational to Object-Oriented Databases. *SIGMOD Record 1997*, 26(1), March 1997.

[FP] F. Fallouh and E. Pichat. D'un Éditeur de Données Complexes à des Schémas de Bases de Données Sémantiques et Objet. Technical report, Université Claude Bernard, Villeurbanne, France.

[Fre87] J.C. Freytag. A rule-based view of query optimization. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 1987.

[FV95] C. Fahrner and G. Vossen. Transforming Relational Database Schemas into Object-Oriented Schemas acoording to ODMG-93. In *Proc. of Intl. Conf. on Deductive and Object-Oriented Databases (DOOD)*, Singapore, December 1995.

[Gem] Gemstone. *http://www.gemstone.com.*

[Gol89] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison–Wesley, 1989.

[Hai95a] Jean-Luc Hainaut. Database Reverse Engineering - Problems, Techniques and CASE tools. In *Conference on Advanced Inf. Syst. Engineering*, Jvaskla, Finland, June 1995. Tutorial.

[Hai95b] Jean-Luc Hainaut. Transformation-based Database Engineering. In *Proc. of Intl. Conf. on Very Large Data Bases*, Zurich, Switzerland, 1995. Tutorial.

[HMWJZ98] H.Garcia-Molina, W.J.Labio, J.L.Wiener, and Yue Zhuge. Distributed and parallel computing issues in data warehousing. Technical report, Stanford University, USA, June 1998. http://www-db.stanford.edu/warehousing/warehouse.html.

[IK84] T. Ibaraki and T. Kameda. Optimal Nesting for Computing N-relational joins. In *ACM Trans. on Database Systems*, volume 9, September 1984.

[IK90] Y. E. Ioannidis and Y. Kang. Randomized algorithms for Optimizing Large Join Queries. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 312–321, Atlantic city, NJ, may 1990.

[INC93] Persistence Software INC. Bridging the Gap between Relational Data and Object Oriented Development. Technical report, Persistence Software INC., September 1993.

[Ioa97] Y.E. Ioannidis. Query Optimization. In *Encyclopedia of Computer Science*, 1997.

[IUT98] Tadashi Iijima, Shirou Udoguchi, and Motomichi Toyama. Applying Database Publishing Technology to the Migration from a Relational Database to an Object-Oriented Database. Technical report, Information and Computer Science - Keio University, 1998.

[IW87] Y. Ionnidis and E. Wong. Query optimization by simulated annealing. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 1987.

[JK84] M. Jarke and J. Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

[JK89] P. Johannesson and K. Kalman. A Method for Translating Relational Schemas into Conceptual Schemas. In *Entity Relationship*, pages 5–26, 1989.

[JSZ96] 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.

[Jun] Jung-Lee. *http://www.junglee.com*.

[Kel94] A. M. Keller. Penguin: Objects for Programs, Relations for Persistence. Technical report, Stanford University, April 1994. Submitted.

[KFL95] I. Kwan, J. Fong, and Q. Li. A Methodology for Reenginnering Relational to Object Oriented Database Systems via an Extended E-R Model. In *Proc. of the Intl. Conf. on Data and Expert Systems Applications (DEXA)*, pages 152–161, UK, September 1995.

[Kos96] A. S. Kosky. *Transforming Databases with Recursive Data Structures*. PhD thesis, University of Pennsylvania, 1996.

[Leb93] 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.

[LV91] R. Lanzelotte and P. Valduriez. Extending the Search Strategy in a Query Optimizer. In *Proc. of Intl. Conf. on Very Large Data Bases*, Barcelone, Espagne, 1991.

[MCS84] M.V. Mannino, P. Chu, and T. Sager. Statistical Profile Estimation in Databse Systems. *ACM Computing Surveys*, 20(3):192–221, September 1984.

[MGG95] 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.

[Mic94] Microsoft. *Open DataBase Connectivity software development kit manual – version 2.0*, 1994. http://www.ddodbc.com.

[MIR93] R. Miller, Y. Ioannidis, and R. Ramakrishnan. The Use of Information Capacity in Schema Integration and Translation. In *Proc. of Intl. Conf. on Very Large Data Bases*, Dublin, Ireland, 1993.

[MR96] J.A. Makowsky and E.V. Ravve. Translation Schemes and the Fundamental Problem of Database Design. In *Entity Relationship*, pages 271–285, Toronto, Canada, October 1996. Frederick H. Lochovsky.

[MZ98] T. Milo and S. Zohar. Using Schema Matching to Simplify Heterogeneous Data Translation. In *Proc. of Intl. Conf. on Very Large Data Bases*, New York, August 1998.

[O2T96] O2Technology. *O2DBaccess User Manual Release 4.6*, September 1996.

[Obj] Objectstore. *http://www.odi.com.*

[Ont96] Ontos Inc. *Ontos Object Integration Server*, 1996. http://www.ontos.com.

[Ora] Oracle. *http://www.oracle.com.*

[Ora96] Oracle. *Oracle 7 Server Utilities*, 1996. Release 7.3.

[Pac91] Hewlett Packard. OpenODB: Facilitating Change. Technical report, Hewlett Packard, September 1991.

[PB94] W. Premerlani and R. Blaha. An Approach of Reverse Engineering of Relational Databases. *Comm. of the ACM*, 5, May 1994.

[Pea91] P. Pearson. *The Genome Data Base (GDB), A Human Genome Mapping Repository*, 1991.

[Pet96] Jean-Marc Petit. *Fondements pour un processus réaliste de rétro–conception de bases de données relationnelles.* PhD thesis, Université de Claude Bernard, Lyon, France, October 1996.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proc. of Intl. Conf. on Data Engineering*, pages 251–260, Taipei, Taiwan, March 1995.

[RH96] S. Ramanathan and J. Hodges. Reverse Engineering Relational Schemas to Object-Oriented Schemas. Technical Report MSU-960701, Mississipi State University, Mississipi, 1996.

[RH97] S. Ramanathan and J. Hodges. Extraction of Object-Oriented Structures from Existing Relational Databases. *SIGMOD Record 1997*, 26(1), March 1997.

[RP] Xavier Roche and Yann Philippot. *HTTrack.* http://www.ensicaen.ismra.fr/ roche/.

[Run92] E. Rundensteiner. MultiView: A Methodology for Supporting Multiple Views in Object-Oriented Databases. In *Proc. of Intl. Conf. on Very Large Data Bases*, Vancouver, 1992.

[Sa79] P.G. Selinger and al. Access path selection in a rdbms. In *Proc. ACM SIGMOD Symp. on the Management of Data*, Boston, May 1979.

[SG88] A. Swami and A. Gupta. Optimization of large join queries. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 1988.

[SLT91] M. H. Scholl, C. Laasch, and M. Tresch. Updatable Views in Object-Oriented Databases. In *Proc. of Intl. Conf. on Deductive and Object-Oriented Databases (DOOD)*, 1991.

[TN92] M. Tsangaris and J. Naughton. On the Performance of Object Clustering Techniques. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 144–153, USA, June 1992. ACM.

[TRV96] 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*, pages 449–457, Hong Kong, May 1996.

[Ull88] J. D. Ullman. *Database and Knowledge base systems*. Computer Science Press, 1988.

[VA95] M.W.W. Vermeer and P.M.G. Apers. Reverse engineering of relational database applications. In *Object Oriented and Entity Relationship*, Australia, December 1995.

[VMa] VMark. *http://www.o2tech.fr*.

[Wie95] J.L. Wiener. *Algorithms for Loading Object Databases*. PhD thesis, University of Wisconsin - Madison, 1995.

[WJHM98] W.J.Labio, J.L.Wiener, and H.Garcia-Molina. Algorithms for resuming interrupted warehouse loads. Technical report, Stanford University, USA, February 1998.

[WN95] J. L. Wiener and J. F. Naughton. OODB Bulk Loading Revisited: The Partitioned-List Approach. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 30–41, Zurich, Switzerland, 1995.

[WN96] J. L. Wiener and J. F. Naughton. Incremental Loading of Object Databases. Technical report, University of Wisconsin, 1996.

[YL89] H. Yoo and S. Lafortune. An intelligent search method for query optimization by semijoins. In *Proc. IEEE Intl. Conf. on Trans. Knowledge and Data Eng.*, volume 1(2), pages 226–237, June 1989.

Annexe A

Syntaxe complète du langage *RelOO*

Nous donnons la syntaxe du langage *RelOO* en utilisant la notation EBNF. La barre (|) représente un choix, ([]) représente une construction optionnelle. Les mots clès du langage sont entre quotes. Soient les points suivants :

1. si le schéma objet existe déjà, il est complété,
2. si le nom de la base de données à objets n'est pas donné, son nom par défaut est `b_`*schema_name*,
3. si la clause de l'extension d'une classe n'est pas donnée, son extension par défaut est *class_name*`s` et son type est `set(`*class_name*`)`. Si la clause `none` est donnée, la classe n'a pas d'extension et ne peut être peuplée qu'à partir d'autres classes.
4. *query* est un prédicat de définition de classe ou d'attribut.
   Pour une classe :
   - si le prédicat n'est pas donné ou s'il est à `true`, tous les n-uplets de la relation source sont sélectionnés,
   - s'il est à `false`, aucun n-uplet n'est sélectionné
   - sinon, c'est un prédicat simple sur des attributs relationnels

   Pour un attribut :
   - si le prédicat n'est pas donné, une valeur par défaut est assignée à l'attribut,
   - des fonctions simples sur des attributs relationnels,
5. Les attributs complexes représentant des liens sont créés à travers la clause «`create link`». Pour créer des associations ODMG, un lien et son inverse doivent être donnés dans la même commande.

mapping_file ::= "`object schema`" *name* "`from`" *relational_schema_name*
  ["`object base`" *base_name*]
  [*command_list*]
  "`end object schema`"
  [*constraint_list*]

```
command_list ::= command command_list
              |  command

command ::= class_clause
         |  links_clause

class_clause ::= ”create class” class_name
                 [”inherit” class_name]
                 ”from relations” relations
                 [”with attributes” attribute_definitions]
                 [”extension” extension_clause]
                 [”where” list_queries]
                 ”end”

relations ::= relation ”,” relations
           |  relation

relation ::= relation_name
          |  relation_name ”[” attribute_names ”]”

attribute_names ::= attribute_name ”,” attribute_names
                 |  attribute_name

attribute_definitions ::= attribute_definition ”;” attribute_definitions
                       |  attribute_definition

attribute_definition ::= attribute_name ”:” atomic_type ”is” list_queries
                      |  attribute_name ”:” object_type ”isref” attribute_names
                      |  attribute_name ”:” att_type

list_queries ::= queries
              |  ”"” query ”"”

queries ::= att_query ”,” list_queries
         |  att_query

att_query ::= ”"” query ”"” ”in relation” relation_name

att_type ::= atomic_type
          |  object_type

extension_clause::= extension_name
                 |  “none”

links_clause ::= ”create link” classes
                 [”from relations” link_relations]
                 ”with attributes” link_attributes
                 ”where” “"” queries ”"”
                 ”end”
```

classes ::= *class_name* "," *class_name*

link_relations ::= *link_relation* ";" *link_relations*
| *link_relation*

link_relation ::= *relation_name* "," *relation_name* "," *relation_name*
| *relation_name* "," *relation_name*
| *relation_name*

link_attributes ::= *link_attribute* "‘," *link_attribute*
| *link_attribute*

link_attribute ::= *attribute_name* ":" *type_expression* ["`in class`" *class_name*]

type_expression ::= *att_type*
| "`set(`" *att_type* "`)`"
| "`list(`" *att_type* "`)`"

constraint_list ::= *constraint* ","*constraint_list*
| *constraint*

constraint ::= "`class`" *class_name*
["`order`" ["`asc`"|"`desc`"] *attribute_names*]
["`cluster on`" ⋆ | *cluster_attributes*]
["`if`" "‘"" *queries* """]

cluster_attributes ::= *cluster_attribute* "," *cluster_attributes*
| *cluster_attribute*

cluster_attribute ::= *attribute_name*[⋆]"("*cluster_attributes*")"
| *attribute_name*[⋆]

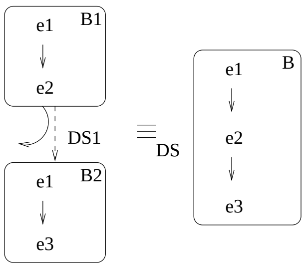
Annexe B

Preuves des équivalences algébriques

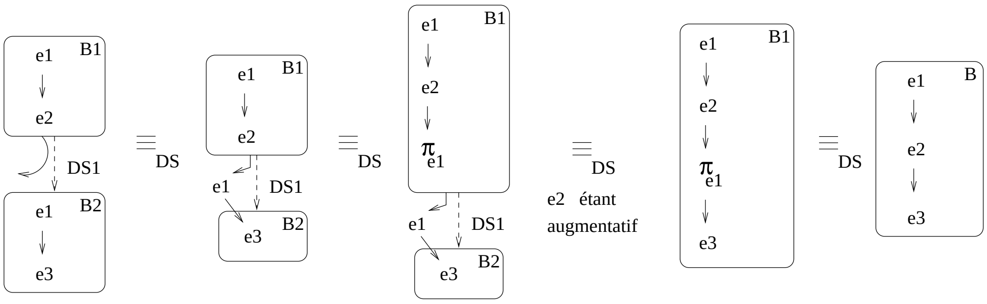
La majorité de nos équivalences reposent sur le fait que deux expressions algébriques sont équivalentes si l'application des mêmes opérations Map sur le résultat algébrique de chacune d'elles a le même effet de bord sur la base cible. Ceci est rendu possible par les propriétés d'idempotence et d'insensibilité à la valeur nulle des fonctions de création et de mise à jour des entités cibles. En réalité, les différences acceptées entre les résultats algébriques de deux expressions équivalentes sont qu'une même donnée ou qu'un même n-uplet apparaisse avec un nombre de fois différent dans les deux résultats ou que des valeurs nulles soient présentes dans un résultat et pas dans un autre. Nos équivalences reposent alors sur des équivalences standard qui tolèrent la présence de valeurs redondantes ou de valeurs nulles. Leur preuve revient alors à vérifier que le flux de données de deux expressions algébriques sont différents aux valeurs nulles ou redondantes près.

B.1 Équivalences inter-blocs

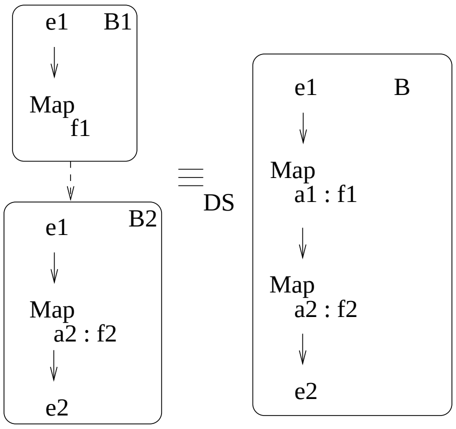
Nous nous intéressons à prouver les équivalences qui transforment deux blocs contenant une expression algébrique en commun en un seul. Sur le schéma ci-dessous, nous voulons prouver que B_1 suivi de B_2 est équivalent à B. L'idée principale de cette réécriture est que l'application d'une expression e_2 sur le résultat de l'évaluation d'une expression e_1 n'altère pas le contenu de e_1 qui sera ensuite passé en entrée à une expression e_3. Les opérations effectuées dans e_2 ne doivent pas affecter le contenu de e_1. En réalité, ceci est toujours vrai lorsque e_2 ne contient que des opérations Map. Celles-ci assurent que le flux de données en entrée reste inchangé. En effet, ces opérations sont augmentatives : $Map_{a:f}(e)$ met le résultat de l'application de f sur chaque élément de e_1 dans l'attribut a. Ainsi, si e_1 contient au départ des n-uplets de la forme $[a_1 : \tau_1, \ldots, a_n : \tau_n]$, le résultat de $Map_{a:f}(e_1)$ est un ensemble de n-uplets de la forme
$[a : \tau, a_1 : \tau_1, \ldots, a_n : \tau_n]$ où τ est le type du résultat de $f([a_1 : \tau_1, \ldots, a_n : \tau_n])$. Le contenu de e_1 reste inchangé. En projetant ensuite sur les attributs de e_1 : $(atts(e_1))$, on récupère bien tout e_1. e_1 n'étant pas jeté à la fin du bloc B_1, e_3 peut lui être appliquée.



c'est ce que nous voulons prouver



Ainsi, si nous reprenons l'équivalence suivante,

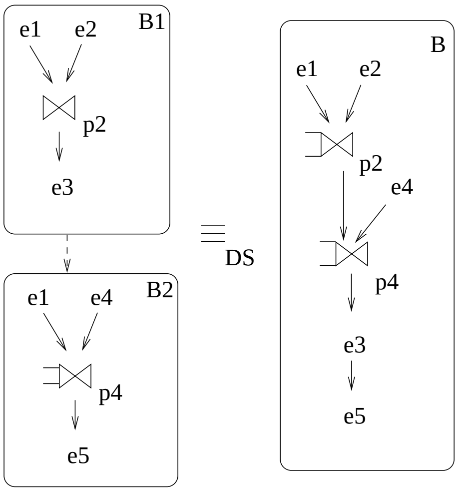


Nous pouvons voir qu'en partie gauche :
$Map_{f_1}(e_1) = \{f_1(t) | t \in e_1\}$ est d'abord effectuée
$Map_{a_2:f_2}(e_1) = \{[a_2 : f_2(t)] \circ t | t \in e_1\}$ est ensuite effectuée
En partie droite :

$$Map_{a_2:f_2}(Map_{a_1:f_1}(e_1)) = \{[a_2 : f_2(t)] \circ t | t \in Map_{a_1:f_1}(e_1)\} \text{ est effectuée.}$$
$$= \{[a_2 : f_2(t)] \circ t | t \in \{[a_1 : f_1(t)] \circ t | t \in e_1\}\}$$

Étant donné que $Map_{a_1:f_1}(e_1)$ n'altère pas le résultat de e_1 (elle augmente e_1 d'un attribut a_1 contenant l'évaluation de f_1 sur chacun de ses n-uplets), f_2 va porter sur tout n-uplet t dans l'ensemble $\{[a_1 : f_1(t)] \circ t | t \in e_1\}$ i.e. sur tout t de e_1.

Prenons maintenant un exemple plus complexe, celui illustré ci-dessous :



Sachant que :

- e_2 et e_4 ne portent pas sur les mêmes données sources.
- e_3 et e_5 ne contiennent pas toutes les deux des opérations *Group*.
- si e_3 et e_5 exécutent des opérations sur e_1 alors ce sont les mêmes opérations.

Regardons comment s'exprime : $((e_1 \sqsupset\!\bowtie_{p_2} e_2) \sqsupset\!\bowtie_{p_4} e_4)$

$$
\begin{aligned}
e_1 \sqsupset\!\bowtie_{p_2} e_2 = & \{t_1 \circ t_2 | t_1 \in e_1, t_2 \in e_2, p_2\} \\
& \cup \\
& \{t_1 \circ null | t_1 \in e_1, \neg\exists t_2 \in e_2, p_2\} \\
& = Exp_1 \\
Exp_1 \sqsupset\!\bowtie_{p_4} e_4 = & \{t_1 \circ t_2 \circ t_4 | t_1 \in e_1, t_2 \in e_2, t_4 \in e_4, (p_2 \wedge p_4)\} \\
& \cup \\
& \{t_1 \circ null \circ t_4 | t_1 \in e_1, t_4 \in e_4, (p_4 \wedge \neg\exists t_2 \in e_2 | p_2)\} \\
& \cup \\
& \{t_1 \circ t_2 \circ null | t_1 \in e_1, t_2 \in e_2, (p_2 \wedge \neg\exists t_4 \in e_4 | p_4)\} \\
& \cup \\
& \{t_1 \circ null \circ null | t_1 \in e_1, (\neg\exists t_2 \in e_2 | p_2 \wedge \neg\exists t_4 \in e_4 | p_4)\} \\
& = Exp_2
\end{aligned}
$$

Par suite,

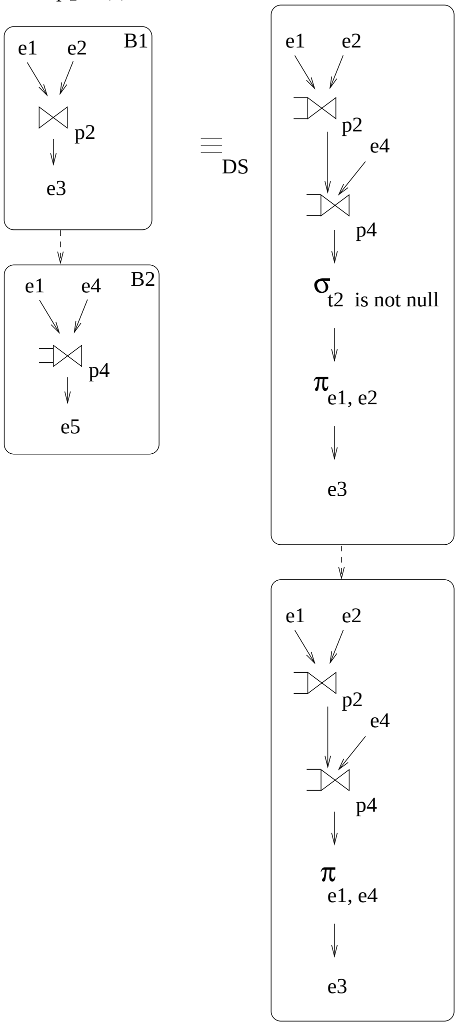
$$
\begin{aligned}
\sigma_{t_2 \neq null}(Exp_2) = & \{t_1 \circ t_2 \circ t_4 | t_1 \in e_1, t_2 \in e_2, t_4 \in e_4, (p_2 \wedge p_4)\} \\
& \cup \\
& \{t_1 \circ t_2 \circ null | t_1 \in e_1, t_2 \in e_2, (\neg\exists t_4 \in e_4 | p_4)\} \\
& = Exp_3
\end{aligned}
$$

donc,

$$
\begin{aligned}
\pi_{e_1,e_2}(Exp_3) = & \{t_1 \circ t_2 | t_1 \in e_1, t_2 \in e_2, (p_2 \wedge (t_4 \in e_4 \wedge \neg\exists t_4 \in e_4) | p_4)\} \\
& = \{t_1 \circ t_2 | t_1 \in e_1, t_2 \in e_2, p_2\} \\
& = e_1 \bowtie_{p_2} e_2
\end{aligned}
$$

Nous avons donc l'équivalence standard :
$e_1 \bowtie_{p_2} e_2 \equiv \pi_{e_1,e_2}(\sigma_{t_2 \neq null}((e_1 \sqsupset\!\bowtie_{p_2} e_2) \sqsupset\!\bowtie_{p_4} e_4))$.

Si $\sigma_{t_2 \neq null}$ n'est pas appliquée, ce serait équivalent à e_1 ⟕ $_{p_2} e_2$. Par suite : e_4 ⟕ $_{p_4} e_1 \equiv \pi_{e_1,e_4}(e_4$ ⟕ $_{p_4}(e_1$ ⟕ $_{p_2} e_2))$. Nous pouvons alors écrire :



La projection peut être enlevée (en appliquant une équivalence intra-bloc) et sachant que e_3 vérifie l'insensibilité à la valeur nulle et qu'elle ne porte pas sur e_4, nous pouvons enlever la sélection $\sigma_{t_2 \neq null}$.

En posant, $((e_1 \bowtie_{p_2} e_2)$ ⟕ $_{p_4} e_4) = e$, on retombe sur l'équivalence inter-blocs prouvée précédemment. Cette équivalence reste vraie tant que l'ensemble des opérations de e_3 portant sur e_1 sont comprises dans e_5. En effet, après l'application de l'équivalence inter-blocs, e_3 va s'appliquer sur tous les n-uplets de e_1 (du fait de la jointure externe sur e_1). Si e_3 contient par exemple une opération de création qui utilise ces n-uplets, il faut que cette opération apparaisse déjà dans e_5 pour que l'effet sur la base de la partie droite de l'équivalence soit le même.

B.2 Équivalences intra-bloc

B.2.1 Insérer et pousser une projection

Cette équivalence revient à prouver que :
$Map_{f(\vec{a})}(e) \equiv_{DS} Map_{f(\vec{a})}(\pi_{\vec{a}}(e))$
Sachant que :
$Map_{f(\vec{a})}(e) = \{f(t.\vec{a}))|t \in e\}$ et que $\pi_{\vec{a}}(e) = \{t.\vec{a}|t \in e\}$, nous avons
$Map_{f(\vec{a})}(\pi_{\vec{a}}(e)) = \{f(t)|t \in \pi_{\vec{a}}(e)\} = \{f(t')|t' \in \{t.\vec{a}|t \in e\}\}$
En utilisant la propriété d'idempotence de f, on a $Map_{f(\vec{a})}(e) = \{f(t.\vec{a})|t \in e\}$
Les deux expressions sont alors DS-équivalentes. L'élimination des duplicats par la projection ne change pas l'effet de ces fonctions sur la base cible. L'opération Map va alors s'appliquer sur les mêmes données.

B.2.2 Inverser un Map et une jointure

Nous prouvons que :
$e3(Map_{a:f}(e_1) \bowtie_p e_2) \equiv_{DS} e3(Map_{a:f}(e_1 \sqsupset\!\!\bowtie\, _p e_2))$
sachant que f porte sur le résultat de e_1 et que si e_3 porte sur e_1, alors elle ne contient que f. En effet, à partir du moment où la jointure externe ($\sqsupset\!\!\bowtie$) «laisse passer» en partie gauche, tous les n-uplets de e_1, il ne faut pas que e_3 exécute des opérations sur e_1 autres que f. e_3 peut bien sûr contenir des opérations sur e_2.

$$\begin{aligned} Map_{a:f}(e_1) \bowtie_p e_2 &= \{[a : f(t_1)] \circ t_1 | t_1 \in e_1\} \bowtie_p e_2 \\ &= \{[a : f(t_1)] \circ t_1 \circ t_2 | t_1 \in e_1, t_2 \in e_2, p(t_1, t_2)\} \end{aligned}$$

D'autre part,

$$\begin{aligned} e_1 \sqsupset\!\!\bowtie\, _p e_2 = \{&t_1 \circ t_2 | t_1 \in e_1, t_2 \in e_2, p(t_1, t_2)\}) \\ &\cup \{t_1 \circ null | t_1 \in e_1, \neg\exists t_2 \in e_2, p(t_1, t_2)\}) \end{aligned}$$

Par suite :

$$\begin{aligned} Map_{a:f}(e_1 \sqsupset\!\!\bowtie\, _p e_2) = \{&[a : f(t_1 \circ t_2)] \circ t_1 \circ t_2 | t_1 \in e_1, t_2 \in e_2, p(t_1, t_2)\} \\ &\cup \{[a : f(t_1 \circ null)] \circ t_1 \circ null | t_1 \in e_1, \neg\exists t_2 \in e_2, p(t_1, t_2)\} \end{aligned}$$

Sachant que f ne porte que sur les éléments de e_1, nous pouvons écrire :

$$\begin{aligned} Map_{a:f}(e_1 \sqsupset\!\!\bowtie\, e_2) = \{&[a : f(t_1)] \circ t_1 \circ t_2 | t_1 \in e_1, t_2 \in e_2, p(t_1, t_2)\} \\ &\cup \{[a : f(t_1)] \circ t_1 \circ null | t_1 \in e_1, \neg\exists t_2 \in e_2, p(t_1, t_2)\} \end{aligned}$$

L'opération $Map_{a:f}$ est alors effectuée sur tous les n-uplets de e_1 dans les deux cas.
Les deux expressions sont donc équivalentes.

B.2.3 Inverser un Map et un $Group$

Nous voulons prouver que :
$e_2(Group_{g;\vec{a_2};\vec{a_3}}(Map_{a:f(\vec{a_1})}(e_1))) \equiv_{DS} e_2(Map_{a:f(\vec{a_1})}(Group_{g;\vec{a_2}\cup\vec{a_1};\vec{a_3}}(e_1))$
sachant que :

- $\exists r, \vec{a_1} \cup \vec{a_2}$ est un sous-ensemble des attributs de la relation r.

– $\vec{a_1} \cap \vec{a_3} = \emptyset$

– $a \notin \vec{a_1} \cup \vec{a_2} \cup \vec{a_3}$

Pour cela, nous vérifions d'abord que les valeurs des attributs de $\vec{a_1}$, $\vec{a_2}$ et $\vec{a_3}$ sont conservées :

1. $\pi_{\vec{a_1}}(Group_{g;\vec{a_2}\cup\vec{a_1};\vec{a_3}}(e_1)) = \pi_{\vec{a_1}}(e_1)$

2. $\pi_{\vec{a_2}}(Group_{g;\vec{a_2}\cup\vec{a_1};\vec{a_3}}(e_1)) = \pi_{\vec{a_2}}(e_1)$

3. $\pi_{\vec{a_3}}(Group_{g;\vec{a_2}\cup\vec{a_1};\vec{a_3}}(e_1)) = \pi_{\vec{a_3}}(e_1)$

Prouvons que :
$\pi_{\vec{a_1}}(Group_{g;\vec{a_2}\cup\vec{a_1};\vec{a_3}}(e_1)) = \pi_{\vec{a_1}}(e_1)$

$$\begin{aligned}\pi_{\vec{a_1}}(Group_{g;\vec{a_2}\cup\vec{a_1};\vec{a_3}}(e_1)) &= \pi_{\vec{a_1}}\{[t_1.\vec{a_2}] \circ [t_1.\vec{a_1}] \circ [g : G] | t_1 \in e_1,\\ &G = \{t_2.\vec{a_3} | t_2 \in e_1, t_2.\vec{a_2} = t_1.\vec{a_2}, t_2.\vec{a_1} = t_1.\vec{a_1}\}\}\\ &= \{[t_1.\vec{a_1}] | t_1 \in e_1\}\\ &= \pi_{\vec{a_1}}(e_1)\end{aligned}$$

De la même manière, nous prouvons que :
$\pi_{\vec{a_2}}(Group_{g;\vec{a_2}\cup\vec{a_1};\vec{a_3}}(e_1)) = \pi_{\vec{a_2}}(e_1)$

Prouvons maintenant que :
$\pi_{\vec{a_3}}(Group_{g;\vec{a_2}\cup\vec{a_1};\vec{a_3}}(e_1)) = \pi_{\vec{a_3}}(e_1)$

$$\begin{aligned}\pi_{\vec{a_3}}(Group_{g;\vec{a_2}\cup\vec{a_1};\vec{a_3}}(e_1)) &= \pi_{\vec{a_3}}\{[t_1.\vec{a_2}] \circ [t_1.\vec{a_1}] \circ [g : G] | t_1 \in e_1,\\ &G = \{t_2.\vec{a_3} | t_2 \in e_1, t_2.\vec{a_2} = t_1.\vec{a_2}, t_2.\vec{a_1} = t_1.\vec{a_1}\}\}\\ &= \pi_{\vec{a_3}}(g) | g = \{t_2.\vec{a_3} | t_2 \in e_1, t_2.\vec{a_2} = t_1.\vec{a_2}, t_2.\vec{a_1} = t_1.\vec{a_1}\}\\ &= \pi_{\vec{a_3}}(g) | g = \{t_2.\vec{a_3} | t_2 \in e_1\}\\ &= \{t_2.\vec{a_3} | t_2 \in e_1\}\\ &= \pi_{\vec{a_3}}(e_1)\end{aligned}$$

Nous vérifions également que les contraintes de regroupement ne sont ni plus fortes ni moins fortes ce qui revient à prouver que :
$e_2(Group_{g;\vec{a_2};\vec{a_3}}(e_1)) = e_2(Group_{g;\vec{a_2}\cup\vec{a_1};\vec{a_3}}(e_1))$
$e_2(Group_{g;\vec{a_2};\vec{a_3}}(e_1)) = e_2(t_3) | t_3 \in \{[t_1.\vec{a_2}] \circ [g : G] | t_1 \in e_1,$
$G = \{t_2.\vec{a_3} | t_2.\vec{a_2} = t_1.\vec{a_2}\}\}$
et,
$e_2(Group_{g;\vec{a_2}\cup\vec{a_1};\vec{a_3}}(e_1)) = e_2(t_3) | t_3 \in \{[t_1.\vec{a_2}] \circ [t_1.\vec{a_1}] \circ [g : G] | t_1 \in e_1,$
$G = \{t_2.\vec{a_3} | t_2 \in e_1, t_2.\vec{a_2} = t_1.\vec{a_2}, t_2.\vec{a_1} = t_1.\vec{a_1}\}\}$
Sachant que e_2 ne porte pas sur les valeurs de $\vec{a_1}$ et avec la condition que e_2 contient des fonctions vérifiant la propriété d'idempotence, les deux expressions sont équivalentes.

B.2.4 Remplacer un *Group* par un +=

Nous voulons prouver que :
$Map_{condnew(a_1).a=Map_{condnew(a_2)(g)}}(Group_{g;\vec{a_1};\vec{a_2}}(e_1))$ est équivalent à
$Map_{condnew(a_1).a+=condnew(a_2)}(e_1)$
sachant que
$a_1 \in \vec{a_1}$ et $a_2 \in \vec{a_2}$.
L'idée de cette équivalence est de prouver que toutes les valeurs des attributs de e_1 après application du *Group* sont conservées de manière à pouvoir le supprimer et aussi de prouver que pour toute valeur $v \in \pi_{\vec{a_1}}$, $condnew(v).a$ est au complet dans les deux cas.
Il suffit donc de prouver que :

1. $\pi_{a_1}(Group_{g;\vec{a_1};\vec{a_2}}(e_1)) = \pi_{a_1}(e_1)$

$$\begin{aligned}\pi_{a_1}(Group_{g;\vec{a_1};\vec{a_2}}(e_1)) &= \pi_{a_1}(\{t_1.\vec{a_1} \circ [g : G] | t_1 \in e_1, G = \{t_2.\vec{a_2} | t_2 \in e_1, t_2.\vec{a_1} = t_1.\vec{a_1}\}\}) \\ &= \{t.a_1 | t \in \{t_1.\vec{a_1} \circ [g : G] | t_1 \in e_1, G = \{t_2.\vec{a_2} | t_2 \in e_1, t_2.\vec{a_1} = t_1.\vec{a_1}\}\}\} \\ &= \{t_1.a_1 | t_1 \in e_1\}\end{aligned}$$

2. $\pi_{a_2}(Group_{g;\vec{a_1};\vec{a_2}}(e_1)) = \pi_{a_2}(e_1)$

$$\begin{aligned}\pi_{a_2}(Group_{g;\vec{a_1};\vec{a_2}}(e_1)) &= \pi_{a_2}(\{[t_1.\vec{a}] \circ [g : G] | t_1 \in e_1, G = \{t_2.\vec{a_2} | t_2 \in e_1, t_2.\vec{a_1} = t_1.\vec{a_1}\}\}) \\ &= \pi_{a_2}([g : G] | G = \{t_2.\vec{a_2} | t_2 \in e_1\}) \\ &= \pi_{a_2}(\{t_2.\vec{a_2} | t_2 \in e_1\}) = \{t_2.a_2 | t_2 \in e_1\} \\ &= \pi_{a_2}(e_1)\end{aligned}$$

3. et que si on note $o = condnew(v)$ avec $v \in \pi_{a_1}$, alors : $o.a = Map_{condnew(a_2))}$ est équivalent à $o.a$+=$\{condnew(a_2)\}$ pour toute valeur de a_2 correspondant à v.
   Pour tout o :
   $o.a = Map_{condnew(a_2))} = \{condnew(t.a_2) | t \in e_1, t.\vec{a_1} = v\}$
   $o.a$+=$condnew(a_2) = \cup\{condnew(t.a_2) | t \in e_1, t.\vec{a_1} = v\}$
   Toutes les valeurs contenues dans la collection sont soit affectées comme un ensemble soit calculées et insérées une à une. Ceci ne change en rien le contenu de $o.\vec{a_1}$ pour tout o.

   On peut aussi écrire :

$$\begin{aligned}Map_{o.a+=\{f(a_2)\}}(e_1) &= \{t_1.o.a+ = \{f(t_1.\vec{a_2})\} | t_1 \in e_1\} \\ &= \{t_1.o.a+ = \{f(t_2.a_2)\} | t_1 \in e_1, t_2 \in e_1, t_2.\vec{a_2} = t_1.\vec{a_2}\} \\ &= \{t_1.o.a = f(t_2.a_2) | t_2.\vec{a_2} = t_1.\vec{a_2}, t_2 \in e_1 | t_1 \in e_1\} \\ &= \{t_1.o.a = Map_{f(a_2)}(g) | t_1 \in e_1, g = \{t_2.\vec{a2} | t_2.\vec{a_2} = t_1.\vec{a_2} | t_2 \in e_1\}\} \\ &= Map_{o.a=Map_{f(a_2)}(g)}(e_1)\end{aligned}$$

B.2.5 Fragmenter un bloc

Nous prouvons que :
$Map_f(e)$ est DS-équivalent à $Map_f(\sigma_p(e))$ suivi de $Map_f(\sigma_{\neg p}(e))$
Si on note : $DS = Map_f(e) = \{f(t)|t \in e\}$ et
$DS1 = Map_f(\sigma_p(e)) = \{f(t)|t \in e, p(t)\}$ et $DS2 = Map_f(\sigma_{\neg p}(e)) = \{f(t)|t \in e, \neg p(t)\}$
alors : $DS1 \cup DS2 = \{f(t)|t \in e, p(t) \vee \neg p(t)\}$
par suite : $DS = DS1 \cup DS2$. Ceci peut être généralisé à plus de deux prédicats avec la condition que leur disjonction soit «true».

Résumé

Ce travail porte sur l'étude de techniques d'optimisation de programmes de chargement en masse dans une base de données relationnelle ou à objets. Dans la première partie, nous considérons le problème dans sa généralité. Nous décrivons le modèle de données sur lequel est fondée notre approche. Ce modèle permet de représenter les types relationnel, objet mais aussi des fichiers texte ayant une grammaire simple. Nous présentons ensuite une algèbre qui permet d'exprimer les transformations source-cible et qui sert de support à l'optimisation. Notre but étant d'optimiser les programmes qui peuplent une base de données, nous avons défini une nouvelle classe d'équivalences algébriques qui tiennent compte des effets de bord sur la dite base.
Dans la deuxième partie de cette thèse, nous validons notre approche par le développement d'un prototype, appelé *RelOO*, pour la migration de données relationnelles vers un système à objets. *RelOO* offre un langage convivial et déclaratif pour spécifier des transformations relationnel-objet mais également des contraintes physiques de regroupement et d'ordonnancement des objets sur disque. Nous montrons comment les transformations exprimées dans ce langage sont traduites dans l'algèbre. Notre stratégie de recherche applique des équivalences algébriques afin de réaliser les contraintes physiques d'une part et de réécrire les programmes de chargement en programmes plus efficaces d'autre part. Cette efficacité est mesurée par un modèle de coût qui prend en compte le temps de communication entre les systèmes source et cible, le temps de traitement sur chacun de ces systèmes et enfin le volume de stockage temporaire nécessité par le chargement. Le résultat de l'optimisation est donné à un générateur de code qui produit des transactions paramétrées. Chaque transaction utilise l'interface ODBC pour charger des données de n'importe quelle base relationnelle vers le système à objets O_2. Enfin, nous avons effectué des tests de validation et de performance sur le prototype *RelOO*.

Du fait de la généricité de notre approche, nous pensons que nos techniques d'optimisation peuvent être utilisées dans plusieurs domaines où le volume des données à charger est important (matérialisation de vues, construction d'entrepôts de données, chargement de bases du génome à partir de fichiers, ...).

Mots-clés : chargement déclaratif de données en masse, réécritures algébriques, effets de bord, stratégie de recherche, modèle de coût.

Bulk Loading Techniques for Databases and an Application to Relational-to-Object Migration

Abstract

We present novel optimization techniques for loading data into a relational or an object database. The first part of the work is concerned with the general problem. We first describe a data model on which our approach is based. This model captures relational, objet models and simple grammars of text files. Then, we describe a formal language that is in fact an algebra to express source-target transformations. This algebra provides a framework for designing, in a declarative and flexible way, loading programs. In our case, it is used to apply a new kind of equivalences that takes into account side-effects on the target database. We validate our approach on the relational to object migration case. This is the subject of the second part of the work that describes our prototype. In this part, we first present a simple declarative end-user language to express relational to object transformations as well as physical properties (clustering, sorting) that one wants to achieve on the object database. The system is then in charge of translating user specifications into the algebra to perform optimization. Our search strategy tries to achieve user requests and generate efficient loading programs. Efficiency is measured using a cost model that considers communication time between source and target systems, processing time on both of them and buffering and storage space. Once the optimization is finished, loading programs are generated. Each of these programs is a parameterized transaction that can be launched for a limited period of time. The targets of our prototype are any ODBC compliant relational system and the O_2 system. Finally, we ran validation and performance tests.

We believe that our optimization techniques can be applied to many applications that aim to migrate huge amount of data into a database (view materialization, data warehousing, loading genome files into a database, ...).

Key-words : declarative bulk loading, algebraic rewriting, side-effects, search strategy, cost model.