UNIVERSITE PARIS I

THESE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE PARIS I

Discipline : Informatique

présentée et soutenue publiquement

par

Fanny WATTEZ

le 10 janvier 2001

Titre :

Optimisation de requêtes sur des structures d'arbres : application à l'objet et à XML

Directeurs de thèse :

Véronique BENZAKEN et Sophie CLUET

JURY

M.	Claude DELOBEL	Président
M.	Omar BOUCELMA	Rapporteurs
M.	Patrick VALDURIEZ	
Mme.	Véronique BENZAKEN	
Mme.	Sophie CLUET	Examinatrices
Mme.	Colette ROLLAND	

Remerciements

Je remercie Sophie Cluet, pour toute l'aide qu'elle m'a donnée, pour m'avoir tant appris et pour avoir lu et relu si attentivement mon manuscrit. Elle a toujours été très compréhensive et humaine envers moi. Je lui en suis très reconnaissante.

Je remercie Omar Boucelma et Patrick Valduriez pour avoir accepté d'être mes rapporteurs.

Je remercie Véronique Benzaken, Claude Delobel et Colette Rolland. Ils me font l'honneur de faire partie de mon jury.

Je remercie Serge Abiteboul pour m'avoir accueillie au projet Verso.

Je remercie Guy Ferran, notamment, pour les conseils qu'il m'a donnés pour réaliser le banc d'essais sur O_2.

Je remercie Christian Chaplais, Olivier Descout, Christian Fiegel et Jean-Luc Schmidt pour m'avoir aidée au sein de la société Ardent Software.

Je remercie Vincent Aguilera, Pierangelo Veltri, Dan Vodislav et Till Westmann avec qui j'ai pris plaisir à travailler au sein de l'action de recherche Xyleme. Je remercie tout particulièrement Vincent pour sa disponibilité et ses multiples explications : il m'a beaucoup appris.

Je remercie toutes les personnes que je n'ai pas encore citées et que j'ai côtoyées au sein de l'équipe Verso et de l'action de recherche Xyleme. L'ambiance a été très chaleureuse.

Je remercie Yves Lechevallier pour le temps qu'il m'a consacré. Les discussions que nous avons eues ont été très instructives.

Je remercie mes parents et ma soeur. Sans eux, je ne serai pas là aujourd'hui.

Table des matières

I Etat de l'art **13**

1 Le standard OQL **17**
1.1 Le modèle de données ODMG . . . 17
1.1.1 Données et types . . . 18
1.1.2 La notion de classe et de hiérarchie de classes . . . 18
1.1.3 La persistance . . . 20
1.2 OQL . . . 20
1.2.1 Fonctions d'arité 0 . . . 21
1.2.2 Fonctions d'arité 1 . . . 21
1.2.3 Fonctions d'arité 2 . . . 21
1.2.4 Fonctions d'arité n . . . 21
1.2.5 Les itérateurs et la composition de fonctions . . . 22

2 Le système O2 **25**
2.1 Organisation physique du système O2 . . . 25
2.1.1 La hiérarchie système, volume, fichier, extent et page . . . 25
2.1.2 La notion de Rid . . . 26
2.1.3 Format d'un enregistrement . . . 26
2.1.4 Taux de remplissage d'une page . . . 27
2.1.5 Exemple . . . 27
2.2 Architecture client-serveur du système O2 . . . 28
2.2.1 Schéma . . . 28
2.2.2 Que se passe-t'il au niveau du processus serveur? . . . 29
2.2.3 Que se passe-t'il au niveau du processus client? . . . 29
2.2.4 Serveur de pages ou serveur d'objets? . . . 29

3 Optimisation **31**
3.1 Algèbre objet . . . 33
3.1.1 Base utilisée . . . 33
3.1.2 Les opérateurs de l'algèbre . . . 33
3.1.3 Exemples de réécritures algébriques . . . 36
3.2 Algorithmes de jointure . . . 40
3.2.1 Au moins une des relations à joindre tient en MC . . . 42
3.2.2 Aucune des deux relations à joindre ne tient en MC . . . 45
3.2.3 Conclusion . . . 52
3.3 Stratégies de recherche . . . 53
3.3.1 Les stratégies de recherche énumératives . . . 53
3.3.2 Les stratégies aléatoires . . . 54

II Requêtes sur structures hiérarchiques : étude d'un système industriel 55

1 Paramètres du banc d'essais 59
1.1 Choix du schéma . . . 59
1.2 Machine utilisée, déroulement des tests en architecture client-serveur . . . 60
1.3 Paramètres concernant la taille de la base . . . 60
1.4 Paramètres concernant le stockage physique des données . . . 61
1.4.1 Placement aléatoire . . . 61
1.4.2 Cluster par classe . . . 61
1.4.3 Cluster de composition . . . 61

2 Expérimentateurs en herbe 65
2.1 Créer une base de grande taille . . . 65
2.1.1 Indexer, oui, mais au bon moment . . . 66
2.1.2 Influence des transactions . . . 66
2.1.3 Influence de la taille des caches serveur et client . . . 66
2.1.4 Cependant 67
2.2 Disposer, à tout moment, de toutes les bases de test . . . 68
2.3 Comment stocker les résultats? . . . 69
2.4 Quoi mesurer? . . . 69
2.5 Quelle méthodologie utiliser pour effectuer les tests? . . . 71

3 Sur les accès associatifs dans O_2 73
3.1 Le mécanisme d'adressage dans O_2 . . . 73
3.1.1 Les identificateurs . . . 73
3.1.2 Les Handles . . . 73
3.2 Hacher des objets . . . 74
3.3 La mystérieuse performance des index . . . 75
3.4 Les Handles ne sont pas optimaux . . . 77
3.5 Comment mieux gérer les objets en mémoire centrale . . . 77

4 Sur la jointure et la navigation dans des structures hiérarchiques 81
4.1 Les algorithmes . . . 82
4.1.1 Naviguer des parents vers les enfants (NL) . . . 82
4.1.2 Naviguer des enfants vers les parents (NOJOIN) . . . 82
4.1.3 Hacher les parents et effectuer la jointure (PHJ) . . . 82
4.1.4 Hacher les enfants et effectuer la jointure (CHJ) . . . 83
4.2 Résultats avec les clusters par classe . . . 84
4.3 Résultats avec les clusters de composition . . . 84
4.4 Conclusion . . . 86

III Requêtes sur structures hiérarchiques : étude d'un système natif XML 91

1 Le standard XML 95
1.1 Un exemple de document XML . . . 96
1.2 Une DTD pour ce document . . . 96

2 Interrogation de données XML dans Xyleme 101
2.1 Un exemple de requête dans Xyleme . . . 102
2.1.1 Choix concernant le langage de requêtes . . . 104
2.1.2 Processeur de requêtes et architecture . . . 104

3 Les index plein-texte dans Xyleme **109**

3.1 Les codages 110

3.1.1 Codage préfixe-postfixe 110

3.1.2 Codage bit-structuré 112

3.1.3 Comparaison des codages 114

3.2 Conclusion 115

4 Evaluation d'un FtiScan **117**

4.1 Définition d'un FtiScan 117

4.2 La problématique du FtiScan 119

4.2.1 Quelques exemples simples de FtiScan 120

4.2.2 Problème posé 122

4.3 Solution proposée 122

4.3.1 Choix d'un algorithme de jointure 122

4.3.2 Choix d'une heuristique 122

4.4 L'heuristique 123

4.4.1 Propriétés des jointures en termes de cardinalité 123

4.4.2 Description de l'heuristique 125

A Proposition d'algorithme **141**

A.1 Hypothèses 141

A.2 Principes de l'algorithme 141

A.3 Ses avantages 142

Introduction

Problématique

Jusqu'à la fin des années 90, les données sur support informatique étaient presque exclusivement bien structurées, stockées dans des systèmes de gestion de bases de données (SGBDs). Aujourd'hui, le Web est un support d'information extrêmement important (plusieurs teraoctets de données en constante augmentation). A l'opposé des systèmes traditionnels, il propose des données peu ou pas structurées.

Ces deux technologies se sont ignorées jusqu'à ce qu'un nouveau standard apparaîsse : XML. Ce langage est en train de déclencher une véritable révolution. XML est un format de représentation des données que le W3C propose comme standard d'échange entre applications sur Internet. De par sa simplicité et ses bonnes propriétés, il a séduit la majorité des acteurs de l'information et on le retrouve maintenant dans des applications Internet comme le commerce électronique, mais, également, dans des applications plus traditionnelles comme, par exemple, les systèmes intégrés d'information. Le stockage des données XML devient maintenant un sujet préoccupant et des SGBDs, qu'ils soient relationnels, objets ou relationnels-objets, proposent des solutions.

Une chose est sûre : quel que soit le format des données, il reste primordial de savoir les interroger efficacement. Cette thèse étudie cette question. Plus précisément, elle étudie comment optimiser des requêtes sur des données en forme d'arbre. Dans la première partie, nous considérons des arbres stockés dans le SGBD à objets O_2. Dans la seconde, nous nous intéressons à l'interrogation de documents XML. Les volumes de données, dans ce second cas, sont plus importants et nous étudions comment utiliser des index pour permettre une interrogation efficace à grande échelle.

Etude sur un système industriel

Les SGBDs à objets, malgré dix ans d'existence, n'ont pas percé. Comment expliquer cet échec? Deux arguments nous semblent raisonnables. D'une part, il faut dire que les concurrents - les SGBDs relationnels - sont de taille. D'autre part, les performances de ces systèmes ont été souvent critiquées. Pourquoi? Il faut bien voir que ceux-ci offrent un système de types qui est équivalent à celui des langages de programmation orientés-objet. Ils permettent de créer des données qui se réfèrent les unes les autres. Une base objet peut vraiment être vue comme un graphe. Dans ce contexte, l'atout, c'est de pouvoir naviguer d'un objet à un ou plusieurs autres au moyen d'un langage impératif orienté-objet et, ce n'est plus, comme dans le relationnel, de parcourir efficacement de très gros ensembles de nuplets grâce à une requête. Les concepteurs des bases à objets se sont donc efforcés de favoriser la navigation, au détriment des accès associatifs et des performances du langage de requêtes.

Au sein de la division objet de la société Ardent Software (ancienne O_2 Technology), nous avons étudié les performances du système O_2 en réalisant un banc d'essais. Notre contribution est de montrer ses faiblesses concernant les gros accès associatifs et de proposer des solutions pour y remédier. Nous avons montré que la taille des objets O_2 en mémoire est beaucoup trop importante pour permettre de bons accès associatifs. De plus, le système perd beaucoup trop de temps en allocations et désallocations.

Notre contribution est aussi d'expliquer les avantages et les inconvénients de la navigation par rapport aux jointures pour évaluer des requêtes sur des bases stockant des structures arborescentes. Nous avons, pour cela, réalisé un banc d'essais. Nous étudions l'influence de plusieurs paramètres : taille de la base, cardinalité de la relation 1:n, sélectivité des prédicats de sélection, organisation physique des données. Une des conclusions principales que nous avons tirée de nos expériences est que la navigation simple est sûrement profitable aux algorithmes sophistiqués de jointure lorsque les données conservent leur forme d'arbre sur le disque.

Etude sur un système natif XML

Nous avons étudié comment indexer de gros volumes de documents XML afin de répondre rapidement à de nombreuses interrogations concurrentes sur leur structure.

Ce travail a été réalisé dans le cadre de l'action de recherche Xyleme. L'objectif de cette action est de concevoir et d'implanter un prototype d'entrepôt de données capable d'exploiter toutes les données XML qui peuvent être trouvées sur le Web. La priorité est de proposer des services de haut niveau qui sont difficiles ou impossibles à supporter avec les technologies utilisées actuellement sur le Web. En particulier, sont considérées, des requêtes plus complexes que les recherches par mot clé, une intégration de données en fonction de leur sémantique et une acquisition sophistiquée des pages sur le Web avec détection des modifications.

Le contexte est ici vraiment différent de l'optimisation de requêtes dans le contexte objet. La taille des données interrogées est très importante, potentiellement, tout le Web. Celles-ci sont réparties. Les utilisateurs sont nombreux, exigeants et, en particulier, impatients.

Pour pouvoir répondre rapidement à des requêtes portant sur la structure d'un grand nombre de documents, quelle technique adopter? Notre contribution est de proposer une méthode d'indexation plein-texte de documents XML. Traditionnellement, un index plein-texte permet de connaître la liste des documents dans lesquels un mot apparaît. Vues les requêtes que nous souhaitons évaluer, ceci est insuffisant. C'est pourquoi à chaque mot indexé correspond une information - nous dirons un codage - qui indique non seulement le document dans lequel il se trouve mais aussi sa position dans ce document. Pour garantir de bonnes performances, certaines propriétés doivent être satisfaites. (1) D'abord, le codage doit être dense. Il doit permettre de stocker le maximum d'informations nécessaires dans le minimum de place. (2) Ensuite, le coût de la mise à jour des index ne doit pas être prohibitif. En effet, Xyleme recueille ses données du Web. Celles-ci sont constamment en changement. Par conséquent, les index risquent de subir très fréquemment des modifications. (3) Enfin, n'oublions pas que les index permettent, pour évaluer une requête, de minimiser le nombre de lectures sur disque. Très fréquemment, les informations récoltées dans l'index servent à accéder aux données sur disque. Les informations à stocker dans l'index sont donc très liées à la manière d'accéder aux données sur disque. Il doit y avoir correspondance.

Notre contribution est aussi de montrer comment, une fois les documents indexés, nous évaluons les requêtes. Chaque requête est, en fait, évaluée grâce à un arbre de jointures. Le problème est donc de savoir comment les réordonner. Nous appliquons pour cela une heuristique qui minimise la somme des cardinalités des résultats intermédiaires de chaque jointure.

Plan

Cette thèse comporte trois parties.

Tout d'abord, l'état de l'art présente le vocabulaire utilisé et donne quelques définitions utiles à la compréhension concernant les SGBDs à objets. Nous décrivons brièvement le modèle ODMG, le standard OQL, le système O_2 en particulier, et l'optimisation de requêtes (algèbre objet utilisée, algorithmes de jointure, stratégies de recherche). XML, quant à lui, sera présenté au début de la troisième partie.

Ensuite, la deuxième partie relate le banc d'essais que nous avons réalisé sur le SGBD à objets O_2. Le chapitre 1 décrit les paramètres des expériences. Le chapitre 2 montre les problèmes que nous avons rencontrés lors de sa réalisation et les solutions que nous avons adoptées. Le chapitre 3 montre les faiblesses des accès associatifs dans O_2 et propose des solutions pour y remédier. Le chapitre 4 compare les performances de la navigation et de la jointure dans des bases réelles. En annexe, nous proposons un algorithme de jointure. Malheureusement, nous n'avons pas eu le temps d'étudier ses performances.

Enfin, la troisième partie traite de l'indexation des documents XML. Le chapitre 1 explique le standard XML sur des exemples. Le chapitre 2 situe notre travail dans l'action de recherche Xyleme. Le chapitre 3 expose comment nous indexons des documents XML. Le chapitre 4 décrit comment nous évaluons une requête une fois des documents XML indexés.

Première partie

Etat de l'art

Introduction

Dans cette partie, nous allons présenter le vocabulaire utilisé et donner quelques définitions utiles à la compréhension concernant les SGBDs à objets. Le standard XML, quant à lui, sera décrit au début de la troisième partie.

Le chapitre 1 décrit brièvement le modèle ODMG et le standard OQL sur des exemples.

Le chapitre 2 détaille certaines caractéristiques du système O_2. Notamment, il explique l'organisation physique et l'architecture client-serveur de celui-ci.

Le chapitre 3 traite de l'optimisation de requêtes en général. D'abord est décrite une algèbre objet. Ensuite sont expliqués les principes de certains algorithmes de jointures par boucles imbriquées, par tri-fusion et par hachage de la littérature. Enfin sont exposées des stratégies de recherche.

Chapitre 1

Le standard OQL

Les systèmes de gestion de bases de données relationnels (SGBDRs), sont fondés sur le concept d'indépendance logique-physique. Au niveau logique, l'utilisateur voit les données qu'il manipule comme des tables. Au niveau physique, les données sont stockées sous forme d'enregistrements dans des fichiers.

Comment un utilisateur peut-il créer des applications? Deux langages sont à sa disposition : les langages de programmation traditionnels (tels C, SmallTalk, C++, Java), langages impératifs et le langage de requête du SGBDR dont le standard est SQL, langage déclaratif. Les deux langages sont nécessaires : l'un pour les calculs ou manipulations complexes tels que récursion ou affichage, l'autre pour l'extraction et la mise à jour des données.

L'existence du langage déclaratif est justifiée par la constatation suivante. Si les applications étaient uniquement écrites en langage impératif, elles seraient trop dépendantes du stockage physique des données, en particulier, par exemple, de l'existence d'index. Ce qui signifie que, à chaque modification du stockage physique des données, les applications devraient être réécrites.

Les fondateurs des SGBDRs ont choisi de proposer à l'utilisateur d'écrire une application sous forme d'un programme en langage impératif dans lequel sont inclus des requêtes. Cette inclusion n'est pas sans poser de problèmes vu que les deux langages n'ont pas le même système de types.

Cependant, cet état de fait, bien que justifié dans les SGBDRs, l'est-il autant dans les SGBDs à objets (SGBDOs)? Les SGBDOs [Ard, Gem, Obja, Objb, Ont, Poe, Ver] sont nés au début des années 90 suite à l'apparition des langages orientés objets (Smalltalk, C++, puis, maintenant, Java) et suite à des besoins nouveaux tels que les applications géographiques. En effet, les données stockées dans un SGBDO possèdent les mêmes types que ceux proposés par les langages de programmation orientée objet. L'existence d'un langage de requêtes dans un SGBDO se trouve-t'elle remise en cause? Une programmation toute impérative devient-elle justifiée? En fait, pas vraiment, et c'est pourquoi l'*Object Database Management Group* (ODMG) [1] propose un standard de langage de requêtes pour SGBDOs : OQL [Clu91, Clu99, Clu98]. L'objectif recherché est de conserver l'indépendance logique-physique, clé de voûte des SGBDs. Néanmoins, ce langage de requêtes ne s'est pas imposé dès le départ car le système O_2 est resté jusqu'au milieu des années 90 le seul SGBDO à le proposer. En fait, c'est l'émergence des applications distribuées sur le Web qui l'a propulsé. Partant du principe qu'un langage de requêtes est un excellent outil pour échanger des données entre systèmes, OQL et SQL sont devenus des standards de l'*Object Management Group* (OMG) [2].

Ce chapitre [3] comporte deux parties. Nous décrirons d'abord le modèle ODMG, puis, le standard OQL.

1.1 Le modèle de données ODMG

Nous allons présenter le modèle ODMG ([Cat97]) sur un exemple de schéma (figure 1.1) et sur un exemple de base (figure 1.2) qui lui est associée. Ce modèle reprend les concepts qui ont été introduits dans

1. Organisme de normalisation pour les SGBDOs dépendant de l'*Object Management Group* (OMG) et qui s'est constitué en 1991
2. Organisme de normalisation pour la programmation distribuée par objets
3. Pour écrire ce chapitre, nous nous sommes fortement inspirés de [Clu99].

[ABD+89].

Un schéma comporte deux parties. La première présente les types des données et la seconde présente les points d'entrée. Les points d'entrée d'un schéma ont deux fonctions. D'une part, comme leur nom l'indique, ils permettent d'accéder aux données stockées dans une base associée à celui-ci. D'autre part, comme dans tout système qui implante la persistance par atteignabilité (cela signifie que toute donnée accessible via une donnée persistante est persistante), ils définissent les données persistantes de la base. Chaque point d'entrée s'appelle un nom. Le schéma sur lequel nous illustrons les concepts possède, par exemple, le nom Les_Villes. De plus, comme v11 est persistant, toutes les données que je peux atteindre grâce à lui sont persistantes. Donc, la valeur de l'attribut nom, la chaîne de caractères “Paris”, est persistante, les lieux culturels lc1, m2, lc3 sont persistants, le pays pa1 est persistant. Dans notre exemple de schéma, toutes les racines de persistance sont soit des objets soit des ensembles d'objets. En fait, le type d'une racine de persistance est n'importe quel type permis par le modèle de données.

1.1.1 Données et types

Dans le modèle ODMG, une donnée est représentée soit sous la forme d'une valeur, soit sous la forme d'un objet. Un objet encapsule des données (c'est la valeur de l'objet) et un comportement. Ce comportement est défini par un ensemble de méthodes (programmes). De plus, un objet possède un identificateur appelé oid (*object identifier*) qui est géré, par le système, de façon transparente pour l'utilisateur. Cet identificateur permet, entre autres, qu'un objet soit référencé plusieurs fois. On parle alors de partage d'objets.

A chaque valeur est associé un type. Les types sont obtenus par application récursive de constructeurs (ensemble (set) noté `{.}`, multi-ensemble (bag) noté `{{.}}`, liste (list) noté <.>, n-uplet (struct) noté [.]) sur des types atomiques (entier (integer), réel (real), caractère (char), booléen (boolean: true ou false), des types préconstruits (chaîne de caractères (string), ...)) et des classes définies par l'utilisateur.

Donnons quelques exemples. pa1 est un objet alors que [nom:"France", villes:{v11,v12,v13}] est une valeur. Plus précisément, c'est la valeur encapsulée dans l'objet pa1.

Egalement, illustrons le partage d'objets. Cette notion est une des notions fondamentales de l'ODMG. Comme vous pouvez le voir, l'objet pa1 apparaît à la fois comme valeur de l'attribut pays de l'objet v11 et comme élément de l'ensemble Les_Pays qui est un point d'entrée.

1.1.2 La notion de classe et de hiérarchie de classes

Une classe est un ensemble d'objets ayant le même type et le même comportement. Les valeurs associées aux objets d'une classe possèdent le même type. Le comportement d'un objet d'une classe est défini par un ensemble de méthodes (petits programmes).

Par exemple, la classe LieuCulturel est un ensemble d'objets dont le type de la valeur encapsulée est le nuplet [nom:string,ville:Ville] et dont le comportement est défini par la méthode affiche qui, pour un lieu culturel donné, affiche à l'écran toutes ses caractéristiques.

De plus, il existe bien souvent une hiérarchie entre les classes d'un schéma. Une classe est souvent définie par référence à une ou plusieurs autres classes. On dit qu'une classe hérite d'une ou de plusieurs classes.

Par défaut, une classe hérite toujours de la classe Object, sommet de la hiérarchie. C'est le cas des classes LieuCulturel, Ville, Pays, Peinture et Peintre. Sinon, une classe peut hériter d'une ou de plusieurs autres classes. C'est le cas pour la classe Musee qui hérite de la classe LieuCulturel. Nous n'avons pas d'exemple d'héritage multiple.

Définissons maintenant plus précisément la notion d'héritage. La classe B peut hériter de la classe A à deux conditions. D'une part, le type de la valeur encapsulée à chaque objet de la classe B est un sous-type du type de la valeur encapsulée à chaque objet de la classe A. D'autre part, un objet de la classe B connaît les méthodes qui définissent le comportement d'un objet de la classe A (certaines méthodes de la classe A peuvent d'ailleurs être redéfinies dans la classe B). La classe B peut éventuellement posséder d'autres méthodes.

Pour être plus clair, donnons une définition de la relation de sous-typage. Nous la représentons par le symbole $\prec$. La relation de sous-typage est définie comme suit: (1) Si $T_i \prec T'_i$ pour i de 1 à n et n$\leq$m, alors $[a_1 : T_1, ..., a_n : T_n, ..., a_m : T_m] \prec [a_1 : T'_1, ..., a_n : T'_n]$, (2) Si $T \prec T'$, alors $set(T) \prec set(T')$, $bag(T) \prec bag(T')$, $list(T) \prec list(T')$.

Classes	**Types**
LieuCulturel	nom: string ville: Ville pays: Pays Method affiche()
Musee	Inherits LieuCulturel peintures_exposees: set(Peinture) prix_normal: real prix_reduit: real jours_de_fermeture: list(string) Method affiche() Method verifie_reduction():boolean
Ville	nom: string lieux_culturels: list(LieuCulturel) pays: Pays Method ajoute_lieu_culturel(lc:LieuCulturel):Ville
Pays	nom: string villes: set(Ville) Method ajoute_ville(v:Ville):Pays
Peinture	nom: string peintre: Peintre expose_a: Musee prete_par: Musee date_acquisition: string achat_ou_don: string Method modifie_lieu_exposition(l:LieuCulturel):Peinture
Peintre	nom: string prenom: string date_naissance: string nationalite: Pays expose_a: set(Musee) a_peint: set(Peinture) Method ajoute_peinture(t:Peinture):Peintre

Points d'entrée	**Types**
Les_Lieux_Culturels	set(LieuCulturel)
Les_Musees	set(Musee)
Les_Villes	set(Ville)
Les_Pays	set(Pays)
Les_Peintures	set(Peinture)
Les_Peintres	set(Peintre)
Le_Louvre	Musee
National_Gallery	Musee
Paris	Ville
Chardin	Peintre

FIG. 1.1 – *Un schéma objet*

```
Les_Pays = {pa1,pa2}
Les_Villes = {v11,v12,v13,v21,v22}
Paris = v11
Les_Lieux_Culturels = {lc1,m2,lc3,lc4,m5}
Les_Musees = {m2,m5}
Le_Louvre = m2
...
...
pa1 : [nom:"France", villes:{v11,v12,v13}]
pa2 : [nom:"Angleterre",villes:{v21,v22}]
v11 : [nom:"Paris",lieux_culturels:<lc1,m2,lc3>,pays:pa1]
lc1 : [nom:"Olympia",ville:v11]
```

FIG. 1.2 – *Une base objet (partielle) associée*

Ainsi, le type de la valeur encapsulée dans chaque objet de la classe Musee est

```
[ nom:string,
  ville:Ville,
  pays:Pays,
  peintures_exposees:set(Peinture),
  prix_normal:real,
  prix_reduit:real,
  jours_de_fermeture:list(string) ]
```

Le comportement d'un objet de la classe Musee est défini par deux méthodes. La première, la méthode affiche, même si elle existe déjà dans la classe LieuCulturel est redéfinie dans la classe Musee. C'est un exemple simple de redéfinition où seul le code de la méthode est modifiée. En effet, la méthode affiche dans la classe Musee affiche toutes les caractéristiques d'un musée et non pas seulement les attributs nom, ville et pays. Cependant, une méthode peut être redéfinie à la fois en changeant le code associé et en raffinant les paramètres par sous-typage. La seconde, la méthode verifie_reduction, est une nouvelle méthode. Elle retourne vrai si la réduction accordée est d'au moins 10%.

1.1.3 La persistance

Dans le modèle ODMG, la persistance est gérée par atteignabilité. Comme nous le disions plus haut, les points d'entrée sont les points de départ de la persistance. Par exemple, dans la base présentée dans la figure 1.2, toutes les données que je peux atteindre à partir des points d'entrée sont persistantes. Prenons par exemple le point d'entrée Les_Villes. v11 désigne un objet qui est persistant puisque élément du point d'entrée Les_Villes. Puisque v11 est persistant, toutes les données que je peux atteindre grâce à lui sont persistantes. Donc, la valeur de l'attribut nom, la chaîne de caractères "Paris", est persistante, les lieux culturels lc1,m2,lc3 sont persistants, le pays pa1 est persistant.

1.2 OQL

Parmi les caractéristiques principales d'OQL qui le distinguent de SQL, citons le fait que OQL est fonctionnel. Autrement dit, toute requête est une fonction dont les arguments peuvent être le résultat d'autres fonctions/requêtes.

Comme nous l'avons fait pour le modèle ODMG, nous allons présenter progressivement OQL par un ensemble d'exemples : fonctions d'arité 0, d'arité 1, d'arité 2, d'arité n pour passer ensuite aux itérateurs et opérateurs de tri et de regroupement.

1.2.1 Fonctions d'arité 0

Donnons trois exemples :
1
"Chardin"
Le_Louvre

1 est une requête qui retourne l'entier 1. **"Chardin"** est une requête qui retourne la chaîne de caractères "Chardin". Enfin, **Le_Louvre** est une requête qui retourne l'objet de la classe Musee stocké dans la racine de persistance Le_Louvre.

1.2.2 Fonctions d'arité 1

Donnons deux exemples :
Le_Louvre.prix_normal
count(Le_Louvre.peintures_exposees)

La première requête retourne la valeur de l'attribut prix_normal de l'objet stocké dans la racine de persistance Le_Louvre. Cette requête est, en fait, la composition de deux fonctions. La première, Le_Louvre, nous l'avons vue plus haut, retourne l'objet de la classe Musee stocké dans la racine de persistance Le_Louvre. La seconde, '.prix_normal', étant donné cet objet, retourne la valeur de l'attribut prix_normal encapsulé dans celui-ci.

La seconde requête donnée en exemple retourne le nombre de peintures exposées au Louvre. Elle illustre la fonction **count** qui prend comme argument une collection[4] et qui retourne le nombre d'éléments dans cette collection.

1.2.3 Fonctions d'arité 2

Donnons les exemples suivants :
1 + 3.5
count(Le_Louvre.peintures_exposees) > 600
Le_Louvre in Paris.lieux_culturels
Paris.lieux_culturels[0]

Le première requête illustre la fonction d'arité 2 '+' qui prend en argument deux nombres (ici, un entier (1) et un réel (3.5)) et retourne leur somme (le réel 4.5). Notez la conversion de type. La deuxième requête retourne un booléen qui vaut vrai s'il y a plus de 600 peintures exposées au Louvre, faux sinon. Elle illustre l'opérateur '>'. La troisième requête retourne également un booléen qui vaut vrai si Le Louvre est bien présent dans les lieux culturels de Paris. Cette requête s'évalue en trois temps : (1) la requête **Le_Louvre** s'effectue (nous l'avons déjà vue en exemple), (2) la requête **Paris** puis la requête **Paris.lieux_culturels** s'effectue et, (3) la requête avec **in** est évaluée. La quatrième requête retourne le premier lieu culturel dans la liste des lieux culturels de Paris. Elle illustre l'opérateur '[]' qui, étant donné un entier i et une liste l, retourne l'élément de l à la position i.

1.2.4 Fonctions d'arité n

La requête

```
struct(nom : Le_Louvre.nom,
       ville : Le_Louvre.ville.nom,
       reduction : Le_Louvre.prix_reduit / Le_Louvre.prix_normal)
```

retourne un nuplet avec trois attributs nom, ville et réduction qui, respectivement, sont le nom du Louvre, la ville où il se trouve et la réduction accordée sur le prix d'entrée de ce musée.

4. Une collection est un multi-ensemble ou un ensemble ou une liste

1.2.5 Les itérateurs et la composition de fonctions

Exemple 1

```
select m
from m in Les_Musees
where m.prix_normal < 50
```

Cette requête retourne les musées stockés dans la racine de persistance Les_Musees dont le prix, sans réduction, est inférieur à 50.

Exemple 2

```
select struct(nom:m.nom,prix_normal:m.prix_normal)
from m in Les_Musees
where m.prix_normal < 50
  and not(``lundi'' in m.jours_de_fermeture)
```

Cette requête retourne le nom et le prix d'entrée , sans réduction, des musées qui ne sont pas fermés le lundi et dont le prix d'entrée sans réduction est inférieur à 50 Francs. Elle montre que la clause where peut contenir toute requête dont le résultat est un booléen. La clause select précise l'information qui est retournée par la requête.

Exemple 3 La requête suivante montre que la collection parcourue dans la clause from peut être le résultat d'une fonction, dans notre cas, **Le_Louvre.peintures_exposees**.

```
select t.date_acquisition
from t in Le_Louvre.peintures_exposees
where t.peintre.nom = ``Chardin''
```

Cette requête retourne les dates d'acquisition des tableaux de Chardin exposés au Louvre.

Exemple 4 Maintenant, montrons une requête qui est une jointure.

```
select t1.prete_par
from t1 in Le_Louvre.peintures_exposees,
     t2 in National_Gallery.peintures_exposees
where t1.prete_par = t2.prete_par
```

Cette requête retourne les musées qui ont prêté à la fois un tableau au Louvre et au moins un tableau à la National Gallery.

Exemple 5 Donnons maintenant un exemple de jointure dépendante.

```
select m.nom
from m in Les_Musees,
     t in m.peintures_exposees
where m.prix_normal < 50
  and t.peintre.nom = ``Chardin''
```

Cette requête retourne les musées dont le prix d'entrée est inférieur à 50 et qui expose au moins un tableau peint par Chardin. La jointure effectuée est appelée jointure dépendante. En effet, regardons la clause from. Le second ensemble parcouru **m.peintures_exposees** dépend du premier **Les_Musees**.

Exemple 6 Montrons maintenant qu'un itérateur peut être inclus dans une requête à n'importe quel endroit.

```
select struct(nom:m.nom,
              peintres_exposes: select distinct t.peintre.nom
                                from t in m.peintures_exposees)
from m in Les_Musees
where count(m.peintures_exposees) > 500
  and not(``mardi'' in m.jours_de_fermeture)
```

Cette requête retourne, pour chacun des musées qui exposent plus de 500 peintures et qui ne sont pas fermés le mardi, les différents peintres exposés. Notez l'utilisation de **distinct**.

Exemple 7 Enfin, terminons en illustrant les tris et regroupements.

```
select t.peintre
from t in Le_Louvre.peintures_exposees
order by t.peintre.nom
```

Cette requête retourne, dans une liste, les peintres exposés au Louvre. Cette liste est triée sur leur nom.

```
select *
from t in Le_Louvre.peintures_exposees
group by t.date_acquisition
```

Cette requête retourne un multi-ensemble de tuples de type [date_acquisition:string,partition:set(Peinture)]. Pour chaque date d'acquisition, l'attribut partition contient toutes les peintures acquises à cette date.

```
select date_acquisition,quantite:count(partition)
from t in Le_Louvre.peintures_exposees
group by t.date_acquisition
```

Par rapport à la précédente requête, nous montrons ici qu'il est possible d'appliquer une fonction au multi-ensemble partition. Pour une date donnée, l'attribut quantité contient le nombre de peintures acquises à cette date.

Chapitre 2

Le système O2

Les SGBDs permettent au programmeur de manipuler des données en faisant presque abstraction du système de gestion de fichiers qui permet leur implantation. L'utilisateur ne voit plus une donnée sous forme d'un enregistrement dans un fichier mais plutôt comme une donnée dont il connaît la structure.

Un des défis de ces systèmes est de savoir stocker de manière optimale (au sens performance d'accès) les données qu'un utilisateur veut rendre persistantes par son aide.

Plus la technologie des SGBDs évolue, plus l'utilisateur possède une vision de haut niveau des données qu'il y stocke. C'est donc au SGBD d'opérer une correspondance logique-physique efficace.

L'objectif de ce chapitre est d'expliquer (i) comment le système O_2 gère la correspondance logique-physique et (ii) l'architecture client-serveur (comment sont effectuées les requêtes, que signifient caches serveur et client).

2.1 Organisation physique du système O2

Un concept fondamental des systèmes de gestion de bases de données est l'indépendance logique-physique. Un utilisateur d'un SGBD a besoin, essentiellement, de connaître la structure des données qu'il veut faire persister pour les manipuler. Le vocabulaire employé par un programmeur d'applications bases de données n'est pas : stocker n octets sur la page p du fichier f, mais plutôt, rendre persistant dans la transaction t la donnée de type T dans la base b.

L'objectif de cette section est d'expliquer les règles adoptées, dans le système O_2, pour stocker des données sur disque. Ces règles ne sont pas exhaustives mais permettent de comprendre la complexité (notamment le nombre d'entrées/sorties) des algorithmes que nous allons présenter par la suite.

2.1.1 La hiérarchie système, volume, fichier, extent et page

Vision de l'utilisateur

Un utilisateur travaille sous O_2 par l'intermédiaire d'un système. Vu de l'extérieur, un système n'est rien d'autre qu'un répertoire. Une fois initialisé, il contient, par défaut, un ensemble de volumes; chaque volume est un fichier du répertoire. Les volumes créés à l'initialisation du système sont les volumes dédiés à la gestion ACID des transactions (le *journal* **log** et l'*ombre* **shadow**), un volume dédié à la gestion des schémas (le *catalogue* **catalog**) et un volume dédié, par défaut, au stockage de données persistantes (le *volume par défaut* **default volume**).

Afin de stocker des données persistantes, l'utilisateur crée une ou plusieurs bases. Une base est associée à un schéma. Le schéma contient la description de la structure des données de la base, la base contient les données elles-mêmes. Les données d'une base sont, par défaut, stockées dans le volume par défaut. Sinon, charge à l'utilisateur de créer un ou plusieurs autres volumes, appelés *volumes utilisateurs*, dans lesquels il va créer une ou plusieurs bases. Une base peut contenir des données stockées dans plusieurs volumes. Un volume peut contenir une ou plusieurs bases.

Gestion interne

Maintenant, voyons, en nous situant à un plus bas niveau, càd. au niveau de la couche logicielle O_2Store, ce qu'est un volume.

Un *volume*, quand il n'est pas étendu [1], est un seul fichier du système d'exploitation utilisé. Mais, c'est aussi plusieurs *fichiers O_2Store*. Un *fichier O_2Store* est, quant à lui, une liste chaînée d'*extents*. Chaque *extent* est un ensembles de huit *pages* contigues sur disque. La taille d'une page est de 4 Kilooctets. Nous verrons dans la suite comment la page est utilisée par les processus serveur et client.

2.1.2 La notion de Rid

Une donnée est stockée, physiquement, par un (ou plusieurs, nous verrons pourquoi dans la suite) enregistrements. Un enregistrement est une suite contigue d'octets. Une page peut contenir un ou plusieurs enregistrements.

Le **Rid** associé à un objet ou à une valeur est l'adresse physique de l'enregistrement correspondant sur disque. **Rid** signifie record identifier (en français, identificateur persistant d'objet). C'est une structure de huit octets composée :

- d'un identificateur de volume sur deux octets
- d'un identificateur de page au sein de ce volume sur quatre octets
- d'un nombre, appelé *slot number*, sur deux octets qui représente la position relative de l'enregistrement dans la page

Pourquoi avoir choisi des identificateurs physiques? Pour des raisons d'efficacité. En effet, si les identificateurs étaient logiques, un tableau, persistant et devant être accessible simultanément par plusieurs utilisateurs, gérerait les correspondances entre adresses physiques et identificateurs logiques. Ainsi, pour accéder à un enregistrement, il faudrait toujours interroger cette table qui serait, potentiellement, très grande. En effet, il y aurait une entrée pour chaque objet de la base.

2.1.3 Format d'un enregistrement

La façon avec laquelle est stockée une donnée dépend de son type. Ci-dessous, nous expliquons progressivement, en partant des types de base, comment s'effectue la correspondance logique-physique.

- **Les valeurs de type atomique sauf les chaînes de caractères :** Elles sont stockées sous forme d'une suite d'octets.
- **Les chaînes de caractères :** La taille de la chaîne de caractères détermine la manière avec laquelle elle est stockée.
  - Si cette taille est inférieure à 15 caractères, alors la chaîne est stockée comme une suite d'octets
  - Sinon, si la taille reste inférieure à celle d'une page, alors c'est un pointeur vers la suite d'octets correspondante
  - Sinon (la taille de la chaîne dépasse la taille d'une page), elle est stockée dans une structure particulière appelée *Long Data Item*
- **Les valeurs de type nuplet :** A une valeur de type nuplet correspond un enregistrement qui est la concaténation de la représentation physique de chaque attribut du nuplet.
- **Les valeurs de type collection :** Le stockage d'une valeur de type collection (liste, multi-ensemble, ensemble) dépend de son type et du nombre d'éléments dans celle-ci.
  - Si les éléments d'une collection occupent moins d'une page, la collection est stockée dans un enregistrement O_2Store. La façon avec laquelle est stocké chaque élément dépend de son type.

1. Le système O_2 donne la possibilité à l'utilisateur d'étendre un volume après qu'il a été créé. Plusieurs fichiers du système d'exploitation utilisé constitue un volume une fois qu'il est étendu

  - Si la totalité de la place prise par les éléments d'une collection dépasse la taille d'une page :
    - un multi-ensemble correspond à un fichier séquentiel O_2Store
    - un ensemble correspond à un arbre B standard
    - une liste correspond à un arbre B positionnel

- **Les objets :** A la valeur encapsulée dans un objet correspond un enregistrement. Le Rid de l'objet est l'adresse de cet enregistrement.

  Les Rids permettent le partage d'objets. En effet, supposons que le même objet soit, par exemple, un attribut de deux valeurs différentes de type nuplet. Dans ce cas, dans l'enregistrement correspondant a chacune de ces valeurs, l'objet est représenté par son Rid.

2.1.4 Taux de remplissage d'une page

Comme tout SGBD, O_2 ne remplit pas les pages utilisées à 100%. La stratégie est plutôt de remplir chaque page à 70%. Pourquoi laisser de la place disponible dans une page? La taille d'un enregistrement, surtout dans le modèle objet, évolue beaucoup au cours du temps. En effet, alors que le modèle relationnel ne permet de rendre persistants que des nuplets dont chaque attribut est une valeur de type atomique (la taille d'un enregistrement correspondant à un tel nuplet est donc fixe), le système de type du modèle objet est plus général. Par conséquent, la taille d'un enregistrement correspondant à une donnée persistante est très variable. Quand la taille d'un enregistrement augmente, il existe deux possiblités: soit il reste assez de place dans la page courante pour opérer cette mise à jour, soit il ne reste pas assez de place et l'enregistrement est déplacé dans une autre page qui pourra le contenir entièrement. Laisser un peu de place disponible dans une page permet donc d'éviter les déplacements d'enregistrements lors de modifications de ceux-ci.

Quand un enregistrement est déplacé d'une page p1 vers une autre page p2, la stratégie adoptée est la *forward-marker technique*. Le Rid de l'enregistrement reste identique. Cependant, dans la page p1, l'enregistrement est remplacé par un pointeur vers sa nouvelle adresse dans la page p2. Par conséquent, l'accès à l'enregistrement nécessite deux E/S : une première pour accéder à la page p1 et lire le pointeur, une seconde pour accéder à la page p2 et lire l'enregistrement.

2.1.5 Exemple

Illustrons la correspondance logique-physique par un exemple. Soit le schéma suivant :

```
class Medecin type tuple (
    id_medecin:integer,
    specialite:string,
    nom:Nom,
    clients:set(Patient));

class Patient type tuple (
    id_patient:integer,
    nom:Nom,
    medecin_traitant:Medecin);

type Nom:tuple(
    nom_de_famille:string,
    prenom:string);

name Les_Medecins:set(Medecin);
name Les_Patients:set(Patient);
```

Dans la suite, Rid(x) est le Rid de l'enregistrement correspondant à l'entité x. Supposons que nous rendons persistant le médecin de numéro 100, de spécialité “ophtalmologie”, de nom “Guy MARTIN”, qui possède 3 clients (il vient d'ouvrir son cabinet!). Un de ces clients s'appelle "Jean DUPOND" et possède le numéro 1500.

L'enregistrement correspondant au médecin, noté m, est :

100	"ophtalmologie"	Rid(nom)	Rid(clients)

Rid(nom) est l'adresse de l'enregistrement suivant :

"MARTIN"	"Guy"

Rid(clients) est l'adresse de l'enregistrement suivant :

Rid(pa1)	Rid(pa2)	Rid(pa3)

où pa1, pa2 et pa3 désigne les trois objets de la classe Patient qui sont les clients de Guy MARTIN. Remarquons que c'est parce que l'attribut clients est un ensemble ne contenant que 3 éléments qu'il est stocké de cette manière.

L'enregistrement correspondant au client Jean DUPOND (noté pa1) est :

1500	Rid(nom)	Rid(m)

avec Rid(nom), qui est l'adresse de l'enregistrement suivant :

"DUPOND"	"Jean"

2.2 Architecture client-serveur du système O2

Le système O_2 a une architecture client-serveur. Pour insérer, supprimer ou modifier des données persistantes d'un système, il est nécessaire de lancer deux processus : un processus serveur et un processus client. Quel est le rôle de chacun de ces processus?

2.2.1 Schéma

La figure 2.1 donne une vision globale de l'architecture du système O_2. Dans la suite, nous décrirons principalement comment s'effectue l'accès aux données, une fois qu'elles sont créées. En effet, c'est l'aspect interrogation des données qui nous a le plus interessés.

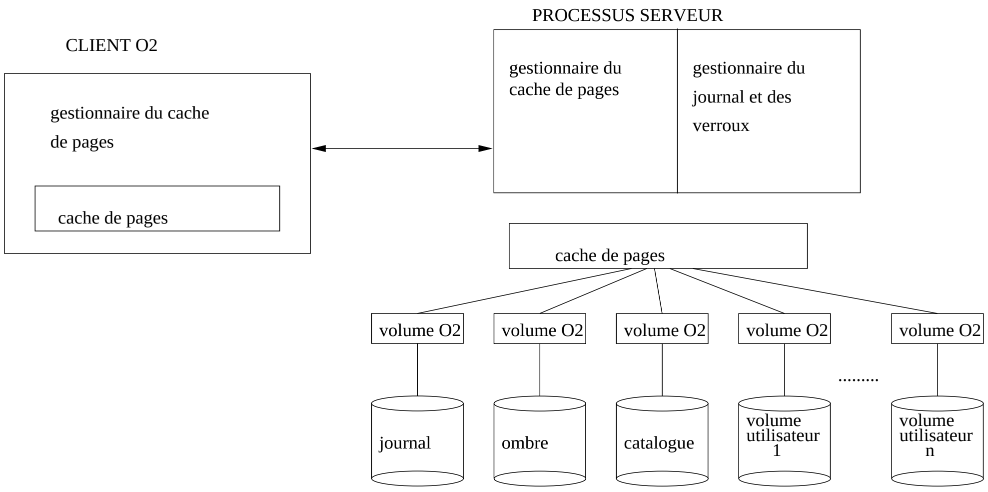


FIG. 2.1 – *Architecture client-serveur du système O_2*

Expliquons donc ce qui se passe au niveau des processus serveur et client.

2.2.2 Que se passe-t'il au niveau du processus serveur?

Pour manipuler des données stockées sur disque, le serveur peut être amené à lire des pages en provenance du disque (Attention! l'unité de lecture/écriture entre le disque et le processus serveur est la page). Mais, également, lors de la création de données persistantes, il doit supporter l'allocation de nouvelles pages, de nouvelles extents et de nouveaux fichiers. Enfin, le serveur gère la concurrence d'accès et les mécanismes de reprise sur panne.

Comme tout processus qui se respecte, le processus serveur possède un cache, un cache de pages, dont la taille par défaut est 4Mo. L'utilisateur peut donner la taille qu'il désire au cache du processus serveur.

2.2.3 Que se passe-t'il au niveau du processus client?

Le processus client et le processus serveur communiquent entre eux au moyen de RPC (Remote Procedure Call). Quand le processus client veut accéder à une page, il la demande au processus serveur par un RPC et le processus serveur lui envoie également par l'intermédiaire d'un RPC. Toutes les pages que le processus client lit sont uniquement des pages qu'il lit en provenance du processus serveur. Celles-ci sont stockées dans le cache du processus client dont la taille par défaut est 4Mo. L'utilisateur peut donner la taille qu'il désire au cache du processus client.

Il faut bien voir que, avec cette architecture, c'est au niveau du client que sont exécutées les requêtes et les appels de méthodes.

2.2.4 Serveur de pages ou serveur d'objets?

Accéder à une donnée revient à accéder à la page[2] dans laquelle elle est stockée. La gestion ACID des transactions est assurée par la pose de verrous. Deux types de verrous existent: les verrous en lecture (le client ne fait que lire la donnée) ou les verrous en écriture (le client modifie la donnée).

Supposons qu'un client accède (en lecture ou en écriture) à un objet. Deux stratégies sont aujourd'hui adoptées par les SGBDs. Soit le verrou est posé sur la page auquelle l'objet accédé appartient (le serveur porte alors le nom de serveur de pages), soit il est posé uniquement sur l'objet (le serveur porte alors le nom de serveur d'objets).

Examinons les avantages et inconvénients des deux approches. Un serveur de pages est efficace car le nombre de communications entre client et serveur est réduit. Cependant, la gestion de la concurrence peut laisser à désirer puisque, si un client accède en écriture à un objet d'une certaine page, c'est la page toute entière qui est interdite d'accès à tous les autres clients. Il est clair que l'avantage d'un serveur de pages est l'inconvénient d'un serveur d'objets et vice versa.

Le système O_2 est implanté avec un serveur de pages. Cependant, quand plusieurs clients demandent à poser des verrous sur des objets différents d'une même page, le logiciel passe à un verrouillage sur chaque objet.

2. ou, quelquefois, à plusieurs pages si la donnée accédée est stockée sur plusieurs pages

Chapitre 3

Optimisation

Trouver, en un temps raisonnable, un algorithme rapide pour évaluer une requête : voilà ce pourquoi sont conçus les interprètes-optimiseurs de langages de requêtes. En effet, l'objectif n'est pas, forcément, de trouver l'algorithme le plus rapide et d'y passer un temps considérable mais, plutôt, de trouver assez rapidement un algorithme proche de l'optimal.

La rapidité d'exécution d'un algorithme est jugée à l'aide de deux critères: nombre d'entrées/sorties (E/S) et nombre d'instructions élémentaires nécessaires à son évaluation.

Il peut paraître aisé de trouver un algorithme pour évaluer une requête. En effet, il suffit de transformer les clauses from en boucles imbriquées, les sélections en conditions, ... Cependant, ceci mène souvent à un algorithme qui est loin d'être optimal.

La question qui se pose est la suivante: sur quelle représentation de la requête travailler pour pouvoir l'optimiser? En effet, une requête n'est rien d'autre qu'une chaîne de caractères qui, représentée telle qu'elle, ne signifie rien. La réponse à cette question a été donnée par le fondateur des langages de requêtes Ted Codd en 1970 : travailler sur une expression algébrique. L'algèbre qui est la base de toutes celles qui sont apparues par la suite est décrite dans [Cod70, Cod79]. Les opérateurs de l'algèbre constituent les briques de base (jointure, sélection, projection, différence et union) à partir desquelles sont construites les expressions algébriques.

Partant de ce principe, la tâche d'un interprète-optimiseur se décompose en plusieurs sous-tâches, ce que nous montrons dans la figure 3.1. (1) L'analyse syntaxique transforme une requête chaîne de caractères en un arbre syntaxique; cette étape a pour but de vérifier si la syntaxe de la requête est cohérente avec celle du langage. (2) La construction de graphe consiste à représenter chacune des opérations qui apparaît dans la requête sous forme d'un graphe; ceci permet de vérifier si toutes les opérations sont bien typées et permet de construire, par défaut, une expression algébrique correspondant à la requête. Par abus de langage, une expression algébrique est appelée plan d'évaluation ou encore plan logique. (3) C'est à ce moment (enfin!) que commence l'optimisation proprement dite. A partir de l'expression algébrique trouvée précédemment, la phase d'optimisation consiste à trouver l'algorithme optimal (ou plan d'exécution) pour évaluer la requête. Un plan d'exécution est un plan d'évaluation dans lequel chaque opérateur algébrique connaît l'algorithme avec lequel il est évalué physiquement. (4) Enfin, l'évaluation de la requête peut s'effectuer en évaluant le plan d'exécution trouvé précédemment.

L'optimisation, processus à deux niveaux, logique et physique. L'optimisation est un processus qui fonctionne à deux niveaux : logique et physique, les deux étant intimement liés. L'optimisation, au niveau logique, consiste à réécrire l'expression algébrique de départ en des expressions algébriques supposées moins coûteuses. En effet, l'algèbre, si elle est bien conçue, possède des équivalences utiles comme, par exemple, l'associativité des jointures.

L'optimisation, au niveau physique, consiste à trouver, pour chaque opérateur logique (ou groupe d'opérateurs logiques) l'algorithme optimal. Donnons un exemple. Par exemple, supposons que la requête que l'interprète-optimiseur doit effectuer est la suivante: retourner tous les peintres dont le nom commence par la lettre 'C'. Comment faire? Est-il préférable d'utiliser un index ou ne pas utiliser d'index?

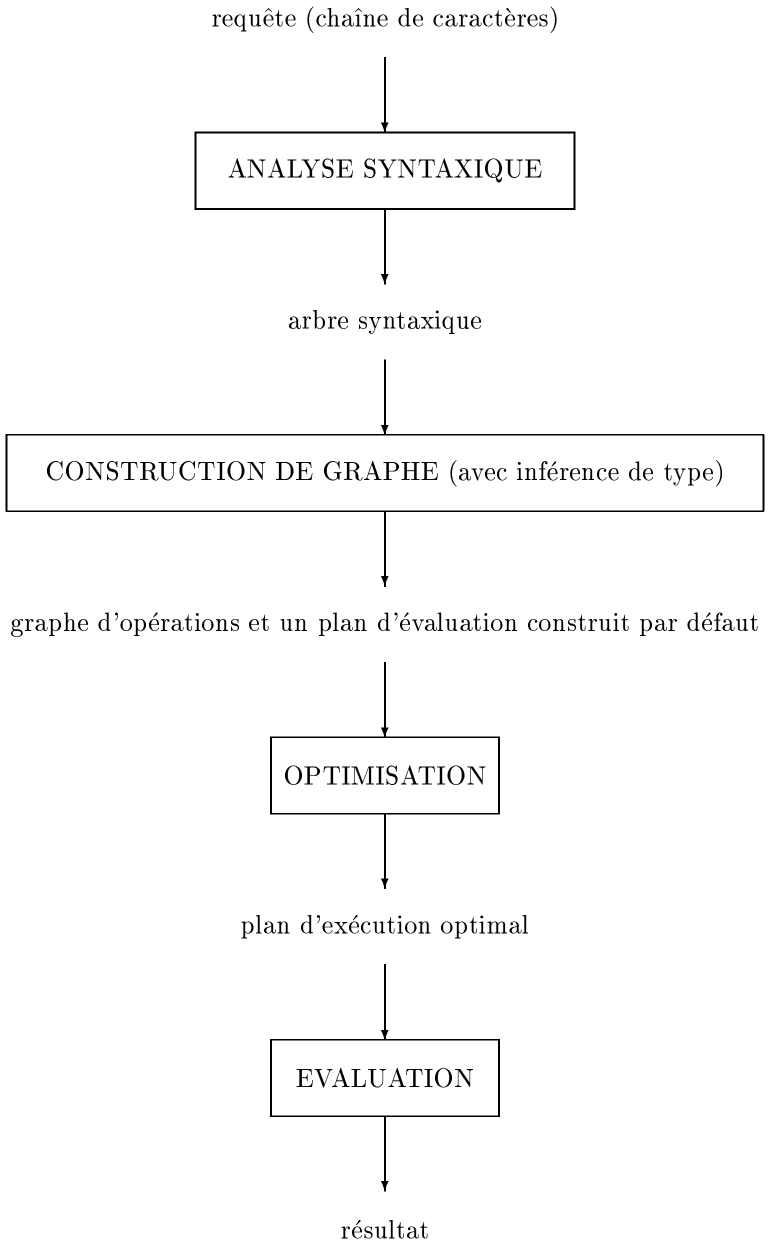


FIG. 3.1 – *Tâches successives d'un interprète-optimiseur*

Les_Pays	nom	villes
pa1	France	{v1,v4,v7,v8}
pa2	Allemagne	{v2,v5,v6}
pa3	Angleterre	{v3,v9,v10}

Les_Peintres	nom	prénom	date_naissance	nationalité	exposé_a	a_peint
p1	Chardin	Jean-Siméon	1699	pa1	{m2,m3}	{t1,t5,t7}
p2	Dürer	Albrecht	1471	pa2	{m3,m4,m1}	{t2,t3}
p3	Turner	William	1775	pa3	{m1,m2}	{t4,t6,t8}
p4	Delacroix	Eugène	1798	pa1	{m1,m2,m3}	{t9,t10}

FIG. 3.2 – *Base de référence*

L'objectif de ce chapitre est triple. Dans la première section, nous présentons l'algèbre objet [CM93] avec laquelle nous travaillons. Il existe de nombreuses algèbres objet dédiées à l'optimisation objet [SZ89, SO90, VD91, YO91]. Celle que nous décrivons est une extension de [KM94]. Une de ses caractéristiques les plus notables est qu'elle est presque relationnelle et que seul un opérateur permet de faire la transition objet/relationnel. Nous illustrons ensuite, sur un exemple, quelques règles d'équivalence spécifiques à l'objet et montrons comment elles permettent d'optimiser une requête. Dans la deuxième section, nous présentons quelques algorithmes de jointure. Nous présentons ceux-ci dans le contexte relationnel mais, nous le verrons dans la suite, ceux-ci sont aisément utilisables dans le contexte objet. Enfin, dans la troisième section, nous parlons brièvement de quelques stratégies de recherche d'optimisation présentes dans la littérature. En effet, pour ne pas être trop lent, l'optimiseur utilise une stratégie de recherche qui le guide dans l'utilisation des règles d'équivalence et dans le choix des algorithmes associés à chaque opération algébrique.

3.1 Algèbre objet

L'algèbre objet que nous présentons ici est définie dans [CM93]. Les algèbres objet ont été beaucoup étudiées. Comme ce n'est pas, à proprement parler, le sujet de cette thèse, nous allons présenter ici juste ce qu'il y a de nécessaire pour comprendre la suite. Par conséquent, l'objectif de cette section est juste d'expliquer chacun des opérateurs de l'algèbre que nous utilisons par l'exemple et d'appliquer quelques règles d'équivalence algébrique.

3.1.1 Base utilisée

Pour illustrer les opérateurs de l'algèbre sur des exemples, reprenons le schéma présenté dans le chapitre 1 (cf figure 1.1). Considérons la base suivante qui est une base différente de celle qui y est présentée. Toutes les racines de persistance sont de type ensemble (noté { . }).

Les_Pays = {pa1,pa2,pa3}
Les_Peintres = {p1,p2,p3,p4}
Les_Musees = {m1,m2,m3,m4}
Les_Peintures = {t1,t2,t3,t4,t5,t6,t7,t8,t9,t10}

La figure 3.2 représente les racines de persistance Les_Pays et Les_Peintres.

3.1.2 Les opérateurs de l'algèbre

L'algèbre comporte, entre autres, les opérateurs suivants : l'opérateur Map, l'opérateur de jointure Join (ainsi que tous ses dérivés, notamment les opérateurs de jointure dépendante D-Join, de semi jointure Semi-Join et de jointure externe Outer-Join), l'opérateur de sélection Select, l'opérateur de projection Proj et l'opérateur de regroupement Group. Ces opérateurs peuvent prendre en entrée des collections (ensembles, listes ou multi-ensembles). Nous allons ici, pour simplifier, les présenter en travaillant uniquement sur des ensembles. Pour tous les opérateurs à l'exception de Map, ces ensembles sont des ensembles de nuplets. Map prend en entrée soit un ensemble de nuplets, soit un ensemble de non-nuplets (par exemple, des objets). Ils retournent tous des ensembles.

Dans la suite, nous adopterons les notations suivantes. e, ainsi que e_i, désigne une expression algébrique. Si elle se trouve à droite d'une égalité, e désigne l'évaluation de l'expression algébrique désignée par e. Si elle se trouve à gauche, e est la définition de l'expression algébrique. a, a_1, ..., a_k, ..., a_n désigne un attribut. p est un prédicat. $\mathcal{A}(e) = \{a_1, ..., a_n\}$ si e est un nuplet, ou un ensemble de nuplets, de type $[a_1 : \tau_1, ..., a_n : \tau_n]$, avec a_i de type τ_i. Enfin, A désigne un ensemble d'attributs. Si t est un nuplet, $t.a$ désigne la valeur de l'attribut a de t et $t.A$ désigne le nuplet composé des attributs de A et dont les valeurs sont celles des attributs correspondants de t.

L'opérateur Map Cet opérateur est important car il permet, entre autres, de transformer des ensembles de non-nuplets (par exemple, des ensembles d'objets) en ensembles de nuplets. Ces derniers peuvent alors être l'entrée d'autres opérateurs. Il existe sous trois formes syntaxiques différentes.

1. Si e est une expression algébrique désignant un ensemble, $e[a] = \{[a : x], x \in e\}$. L'exemple le plus fréquent d'utilisation de cet opérateur est de prendre, comme expression algébrique e, un point d'entrée du schéma sur lequel on pose une requête. Prenons Les_Peintres.

$$Les\_Peintres[pe] = \{[pe : p1], [pe : p2], [pe : p3], [pe : p4]\}$$

   L'ensemble de nuplets obtenu peut ainsi être l'entrée d'un autre opérateur de l'algèbre, ce qui n'est pas le cas du point d'entrée Les_Peintres puisqu'il n'est pas un ensemble de nuplets.

2. Si e est un ensemble de nuplets de type τ et f une fonction définie sur τ, alors

$$Map_f(e) = \{f(t), t \in e\}$$

   Autrement dit, cet opérateur applique la fonction f à chaque élément de e. Par exemple,

$$Map_{pe.nom}(Les\_Peintres[pe]) = \{\text{“}Chardin'', \text{“}Dürer'', \text{“}Turner'', \text{“}Delacroix''\}$$

3. Si e est un ensemble de nuplets de type τ et f une fonction définie sur τ, alors

$$Map_{a:f}(e) = \{t \circ [a : f(t)], t \in e\}$$

   où $\circ$ est l'opérateur de concaténation entre deux nuplets. Par exemple,

$$\begin{aligned}&Map_{nomP:pe.nom} Les\_Peintres[pe] = \\ &\quad \{[pe : p1, nomP : ''Chardin''], [pe : p2, nomP : ''Dürer''], [pe : p3, nomP : ''Turner''], \\ &\quad [pe : p4, nomP : ''Delacroix'']\}\end{aligned}$$

Les opérateurs de jointure Soient e_1 et e_2, des ensembles de nuplets et p un prédicat.

L'opérateur de jointure $\bowtie$

$$e_1 \bowtie_p e_2 = \{t_1 \circ t_2 | t_1 \in e_1, t_2 \in e_2, p(t_1, t_2)\}$$

Par exemple,

$$\begin{aligned}&Les\_Peintres[pe] \bowtie_{pe.nationalite=pa} Les\_Pays[pa] = \\ &\quad \{[pe : p_1, pa : pa_1], [pe : p_2, pa : pa_2], [pe : p_3, pa : pa_3], [pe : p_4, pa : pa_1]\}\end{aligned}$$

L'opérateur de semi jointure SemiJoin ($\rtimes$ ou $\ltimes$)

$$e_1 \rtimes_p e_2 = \{t_1 | t_1 \in e_1, \exists t_2 \in e_2, p(t_1, t_2)\}$$

$\ltimes$ est définie de la même manière en permutant le rôle de e_1 et de e_2 dans l'expression de droite.

L'opérateur de jointure externe OuterJoin

$$\begin{aligned} & e_1 ⟕_p e_2 = \\ & \quad \{t_1 \circ t_2 | t_1 \in e_1, t_2 \in e_2, p(t_1, t_2)\} \cup \\ & \quad \{t_1 \circ t_3 \; | t_1 \in e_1, \neg \exists t_2 \in e_2, p(t_1, t_2), \mathcal{A}(t_3) = \mathcal{A}(e_2), \forall a \in \mathcal{A}(e_2), t_3.a = NULL\} \end{aligned}$$

L'opérateur de jointure dépendante D-Join On effectue une jointure dépendante entre deux ensembles quand l'évaluation du second ensemble dépend de l'évaluation du premier ensemble. Ainsi,

$$e_1 < e_2 >= \{t_1 \circ t_2 | t_1 \in e_1, t_2 \in e_2(t_1)\}$$

Par exemple,

$$\begin{aligned} & Les\_Peintres[pe] < pe.a\_peint[t] >= \\ & \quad \{[pe : p_1, t : t_1], [pe : p_1, t : t_5], [pe : p_1, t : t_7], [pe : p_2, t : t_2], \ldots, [pe : p_4, t : t_{10}]\} \end{aligned}$$

L'opérateur de sélection Select σ Si e est un ensemble de nuplets et p est un prédicat, alors

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

Cet opérateur sélectionne tous les nuplets de e qui satisfont le prédicat p. Les variables du prédicat p sont des attributs de chaque nuplet t. Par exemple,

$$\sigma_{pe.nom \ like \ ``C*''}(Les\_Peintres[pe]) = \{[pe : p_1]\}$$

L'opérateur de projection π Si e est un ensemble de nuplets, alors

$$\pi_{a_1, \ldots, a_k}(e) = \{[a_1 : t.a_1, ..., a_k : t.a_k] | t \in e\}$$

Attention! Puisque cet opérateur retourne un ensemble de nuplets, les redondances sont éliminées. Par exemple,

$$\pi_{nationalite}(Les\_Peintres[pe]) = \{[nationalite : pa_1], [nationalite : pa_2], [nationalite : pa_3]\}$$

Notons enfin que l'opérateur de projection est en fait un cas particulier de l'opérateur Map. En effet,

$$\pi_{a_1, \ldots, a_k}(e) \equiv Map_{[a_1:\tau_1, \ldots, a_k:\tau_k]}(e)$$

L'opérateur de regroupement Group Γ Si e est un ensemble de nuplets de type τ, g un nom d'attribut, f une fonction définie sur τ, $A\theta$ une séquence d'attributs et d'opérateurs, alors

$$\begin{aligned} & \Gamma_{g;A\theta;f}(e) = \\ & \quad \{t_1.A \circ [g : G] | t_1 \in e, G = f(\{t_2 | t_2 \in e, t_2.A\theta t_1.A\})\} \end{aligned}$$

Prenons un exemple. A n'est qu'un seul attribut, nationalite, et θ, l'égalité $=$. La fonction f est l'identité, id.

$$\begin{aligned} & \Gamma_{g;nationalite=;id}(Les\_Peintres[pe]) = \\ & \quad \{[nationalite : pa_1, g : \{p_1, p_4\}], [nationalite : pa_2, g : \{p_2\}], [nationalite : pa_3, g : \{p_3\}]\} \end{aligned}$$

3.1.3 Exemples de réécritures algébriques

Dans cette section, nous allons illustrer, sur un exemple de requête, quelques équivalences algébriques (on parle aussi de règles d'équivalences algébriques ou, encore, de transformations algébriques) spécifiques à l'objet.

Exemple de requête OQL

Considérons la requête OQL, notée Q, suivante :

```
select m.nom,t.peintre.nom
from   m in Les_Musees,
       t in m.peintures_exposees
where  t.peintre.nationalite = m.pays
       and t.peintre.nom like "C*"
```

Cette requête retourne les peintres dont le nom commence par la lettre 'C' et qui sont exposés dans au moins un musée de leur pays d'origine. Le nom de ces peintres et chacun des noms des musées satisfaisants sont retournés par la requête.

Etant donnés les opérateurs que nous avons vus précédemment, la figure 3.3 représente une expression algébrique associée à la requête Q sous forme d'arbre.

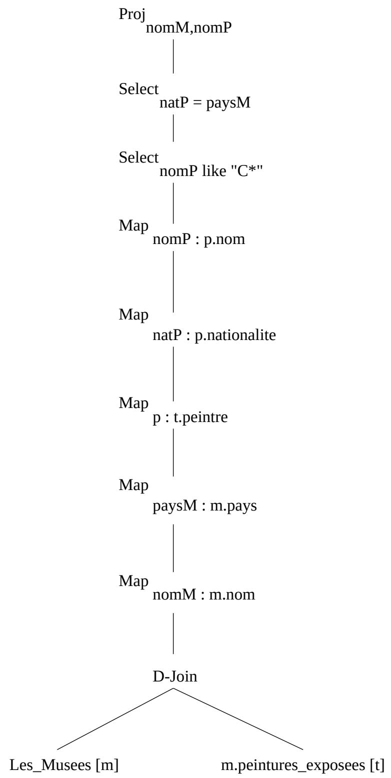


FIG. 3.3 – *Expression algébrique relative à la requête Q sous forme d'arbre*

Première transformation : D-Join en jointure

Constat Evaluons cet arbre algébrique. L'algorithme qui vient naturellement à l'esprit pour ce faire est le suivant :

```
Pour chaque m dans Les_Musees Faire /*boucle B1*/
  Pour chaque t dans m.peintures_exposees Faire /*boucle B2*/
    nomM := m.nom;
    paysM := m.pays;
    p := t.peintre;
    natP := p.nationalite;
    nomP := p.nom;
    Si (natP = paysM et nomM like "C*") Alors
      Résultat += {[nomM,nomP]}
    Fsi
  Fait
Fait
```

Cet algorithme est-il susceptible d'être performant? Examinons ce qui se passe en termes d'entrées/sorties (E/S) pour évaluer la requête.

La boucle B1 consiste à parcourir l'ensemble d'identificateurs d'objets stockés dans la racine de persistance Les_Musees. Comme nous le disions dans le chapitre précédent, la taille d'un identificateur est petite (8 octets). Cela n'occasionne donc qu'un nombre raisonnable d'E/S.

Ensuite, pour chacun des musées m, la boucle B2 consiste, d'abord, à évaluer la valeur de l'attribut peintures_exposees de m (ce qui conduit à lire la page contenant l'enregistrement relatif à m du disque en MC) et, ensuite, à parcourir cet ensemble d'identificateurs d'objets. Même remarque que ci-dessus, comme la taille d'un identificateur est petite (8 octets), cela n'occasionne qu'un nombre raisonnable d'E/S.

Puis, pour chacune de ces peintures t,

- l'instruction *p := t.peintre* nécessite d'obtenir la valeur de l'attribut peintre de t (ce qui conduit à lire la page contenant l'enregistrement relatif à t du disque en MC)

- les instructions *nomP := p.nom* et *natP := p.nationalite* nécessitent de lire la page contenant l'enregistrement relatif à l'objet p.

- les instructions *paysM := m.pays* et *nomM := m.nom* n'occasionnent pas d'E/S supplémentaires car la page contenant l'enregistrement relatif au musée m est déjà en MC.

Récapitulons et concentrons nous sur les E/S sur les pages qui contiennent des enregistrements relatifs à des musées et/ou à des peintures. Puisque la racine de persistance Les_Musees est représentée sur disque par un ensemble trié d'identificateurs d'objets (cf le chapitre 2, section 2.1.3), chacune des pages contenant au moins un enregistrement relatif à un musée est lue une fois et une seule.

Une fois qu'une page relative à un musée m est chargée en MC, obtenir chacun des t ne nécessite qu'un nombre raisonnable d'E/S (parcours d'un ensemble d'identificateurs d'objets, donc, d'un ensemble de petite taille). Par contre, évaluer l'attribut peintre d'un t nécessite de lire la page contenant l'enregistrement relatif à t.

Les E/S occasionnées dépendent alors de la façon avec laquelle les objets sont stockés sur disque.

Premier cas de figure : les peintures sont regroupées sur disque avec le musée dans lequel elles sont exposées. Dans ces conditions, chaque page contenant au moins un enregistrement relatif à une peinture est lue une fois et une seule.

Second cas de figure : le placement physique des peintures est tout à fait aléatoire sur disque par rapport à celui des musées. Il est alors très probable qu'une page contenant des peintures sera lue de nombreuses fois et cela risque d'être très coûteux.

Autre algorithme possible De multiples algorithmes permettent d'évaluer la requête Q. Nous allons ici en décrire un autre en expliquant l'équivalence algébrique utilisée.

L'idée est la suivante. Regardons le schéma que nous utilisons. Chaque objet de la classe Musée connaît les peintures qui y sont exposées grâce à l'attribut peintures_exposées. Inversement, chaque objet de la classe Peinture connaît le musée qui lui correspond grâce à l'attribut exposé_à.

Tirons partie de cela. Pour évaluer la requête Q, il est possible d'effectuer la jointure dans l'autre sens, càd., en parcourant en premier tous les objets de la classe Peinture et, pour chacun de ces objets, en allant lire le musée qui lui correspond. Ceci suppose que le SGBDO utilisé supporte la notion d'extension. L'extension d'une classe A, notée Ext_As, est l'ensemble des objets de cette classe. En l'occurrence, ici, nous effectuons la jointure entre Ext_Peintures et Les_Musees.

Cet algorithme est-il plus performant que le précédent? Difficile de répondre ... Un argument semble en sa faveur. Le parcours de l'extension Ext_Peintures permet de ne lire chaque page contenant au moins un enregistrement relatif à des peintures qu'une fois et une seule. De plus, comme nous allons le voir dans la section suivante, la jointure pourrait être rendue performante en choisissant un bon algorithme de jointure, par exemple, un algorithme de jointure par hachage.

Equivalences algébriques utilisées La transformation que nous venons d'effectuer utilise les deux équivalences algébriques suivantes.

$$\begin{aligned} & e(e_1 < e_2[x] >) \equiv e(e_1 \bowtie_{x \in e_2} Ext\_Cs[x]) \ ssi \\ & \quad x \ est \ de \ type \ C \ et \ e \ contient \ une \ sélection \ nil - éliminante \ sur \ x \end{aligned}$$

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

$$\begin{aligned} & e_1 \bowtie_{y \in x.a} e_2 \equiv e_1 \bowtie_{y.b = x} e_2 \ ssi \\ & \quad x \ est \ un \ attribut \ d'un \ nuplet \ de \ e_1 \ de \ type \ C_1, \\ & \quad y \ est \ un \ attribut \ d'un \ nuplet \ de \ e_2 \ de \ type \ C_2, \\ & \quad C_1.a \ est \ inverse \ de \ C_2.b \end{aligned}$$

Si nous appliquons ces deux équivalences à l'expression algébrique de la figure 3.3, nous obtenons l'expression algébrique suivante :

$$\begin{aligned} & Les\_Musees[m] < m.peintures\_exposees[t] > \equiv \\ & \quad Ext\_Peintures[t] \bowtie_{t.expose\_a = m} Les\_Musees[m] \end{aligned}$$

Elle est représentée sur la figure 3.4.

Deuxième transformation : expression de chemin en jointure

La première réécriture consistait à éviter de lire de nombreuses fois chacune des pages contenant au moins un enregistrement relatif à une peinture. Malgré tout, dans l'expression algébrique obtenue, comme l'accès aux pages contenant des enregistrements relatifs à des peintres se fait de façon aléatoire, chaque page contenant des enregistrements relatifs à des objets de la classe Peintre est susceptible d'être lue plusieurs fois.

Equivalence algébrique utilisée Pour éviter cela, nous utilisons une autre équivalence algébrique. Nous allons transformer l'expression algébrique représentée sur la figure 3.4 en celle représentée sur la figure 3.5 grâce à l'équivalence algébrique suivante.

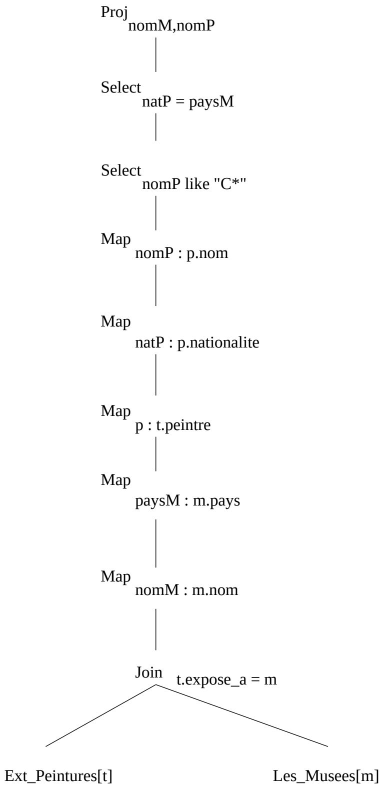


FIG. 3.4 – *jointure dépendante en jointure*

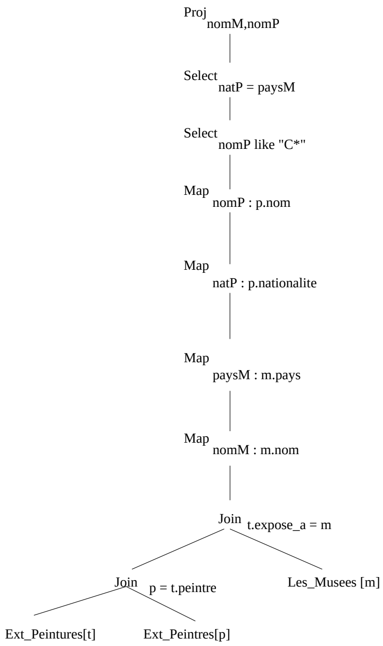


FIG. 3.5 – *expression de chemin en jointure*

$e(Map_{att:y.a}(Alg\_Exp)) \equiv e(Alg\_Exp \bowtie_{att=y.a} Ext\_As[att])$
ssi
le type de $y.a$ est de la classe A et e contient une sélection nil-éliminante sur att.

Dans ce cas, la jointure entre Ext_Peintures et Ext_Peintres peut s'effectuer grâce, par exemple, à un algorithme de hachage, ce qui permet d'éviter de lire plusieurs fois chaque page contenant des peintres.

On peut aussi pousser la sélection avant la jointure Nous pouvons également transformer l'arbre algébrique de la figure 3.5 en l'arbre algébrique de la figure 3.6 en poussant la sélection $\sigma_{nomP\ like\ ``C*''}$ avant la jointure entre Ext_Peintures et Ext_Peintres.

Dans notre cas, cela vaut le coup de pousser la sélection avant la jointure. En effet, comme nous le disions plus haut, en choisissant un bon algorithme (par exemple, un algorithme par hachage), nous pouvons effectuer la jointure en ne lisant qu'une seule fois chaque page contenant des peintres et chaque page contenant des peintures. De plus, en poussant la sélection sur les peintres avant la jointure, nous restreignons le nombre de pages contenant des peintres accédées.

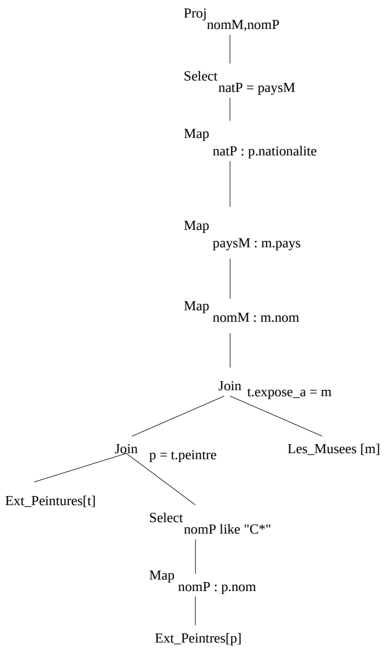


FIG. 3.6 – *pousser la sélection*

On transforme encore ...

La figure 3.7 montre une expression algébrique correspondant à la requête Q obtenue en appliquant successivement différentes équivalences algébriques. Notamment, cette expression a été obtenue en transformant une expression de chemin en jointure avec Ext_Pays. Egalement, les jointures sont réordonnées.

Il est intéressant de noter que cette expression ressemble beaucoup aux expressions rencontrées dans le relationnel. Ainsi, toutes les techniques qui existent dans le relationnel peuvent être utilisées.

3.2 Algorithmes de jointure

Comment effectuer, de manière efficace, la jointure entre deux relations R et S? Voilà une question qui fait, depuis maintenant près de trente ans, l'objet de nombreux travaux de recherche et qui continue d'être étudiée. Pourquoi un tel engouement? simplement parce que cette opération est la plus courante et également la plus coûteuse en terme de performance.

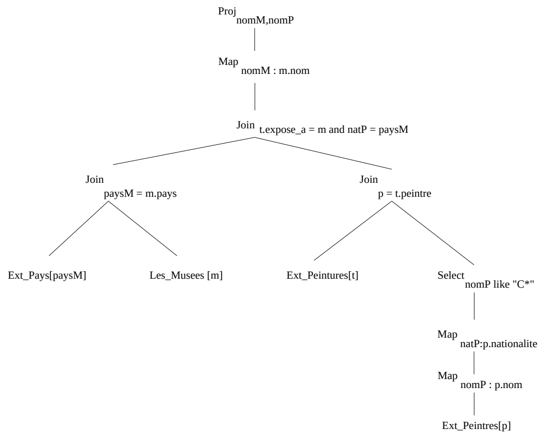


FIG. 3.7 – *encore après plusieurs transformations*

De nombreux algorithmes de jointure ont été proposés dans la littérature. Le plus simple est l'algorithme de jointure par boucles imbriquées. Cependant, même si cet algorithme s'avère le plus rapide quand l'un des ensembles joints tient en mémoire centrale, notée MC, (le nombre d'entrées/sorties (E/S) nécessaires à l'exécution de celui-ci est la somme des nombres de pages occupées par chaque relation jointe), ses performances deviennent désastreuses quand aucun des deux ensembles joints ne tient en mémoire centrale.

La taille des données stockées et interrogées au moyen d'un SGBD ne cesse de croître. Même si l'augmentation de la taille des données va de pair avec l'augmentation de la taille de la mémoire centrale, il reste fondamental de trouver des algorithmes qui restent efficaces même quand la taille des données interrogées est beaucoup plus importante que la taille de la mémoire centrale.

C'est pourquoi la recherche dans ce domaine a rivalisé d'ingéniosité pour inventer de nouveaux algorithmes de jointure. Successivement sont apparus les algorithmes de jointure à base de tri-fusion (tiens, si on triait les relations avant de les joindre?) puis les algorithmes de jointure à base de hachage (tiens, si on hachait les relations avant de les joindre?).

Ces algorithmes ont tous un point commun : ils cherchent à optimiser le nombre d'entrées/sorties nécessaires à leur exécution. Autrement dit, même quand aucune des relations jointes ne tient en MC, ils tendent à ce que chaque page de chaque relation jointe soit lue un minimum de fois, l'idéal étant une fois et une seule.

Dans ce chapitre, nous allons expliquer les principes des algorithmes de jointure par boucles imbriquées, par tri-fusion et par hachage [Gar99, Gra93, Ull88, Ull89, GMUW00]. Les hypothèses que nous prenons sont les suivantes. Nous nous situons dans le contexte relationnel - mais ces algorithmes sont parfaitement utilisables dans le contexte objet - et effectuons la jointure entre deux relations R et S. r (resp. s) étant un tuple de R (resp. S), le prédicat de jointure est de la forme r.a $< op >$ s.b. Les algorithmes de hachage ne peuvent être utilisés que pour des équijointures, càd. des jointures dont le prédicat est une égalité. $|R|$ (resp. $|S|$) désigne le nombre de pages de la relation R (resp. S) et $|R| \leq |S|$. $|M|$ est le nombre de pages dans la mémoire centrale. Deux fonctions de hachage sont utilisées dans la suite : la première, h_1, sert à partitionner R (resp. S), la seconde, h, sert à construire des tables de hachage. Elles sont appliquées aux attributs de jointure. La taille, en nombre de pages, de la table de hachage relative à R est plus grande que $|R|$. Dans la suite, nous approximerons la taille de la table de hachage de R par $|R| * F$ avec $F \geq 1$.

Un paramètre qui détermine la façon avec laquelle est effectuée une jointure est : quand on effectue une jointure par boucles imbriquées, est-ce qu'au moins une des relations tient en MC? quand on effectue une jointure par tri-fusion ou par hachage, est-ce que la structure de données (par exemple, table de hachage) relative à une des relations jointes tient en MC?

Nous expliquerons donc les algorithmes de jointure en distinguant ces deux situations. Dans la deuxième situation, afin que les explications soient plus claires, nous allons décrire les algorithmes quand seulement deux passes sont nécessaires. Pour cela, nous supposerons que $\sqrt{|S| * F} \leq |M|$. Nous verrons plus loin pourquoi cette hypothèse est utile. En tous cas, cette hypothèse n'est pas trop restrictive quant à la taille des relations jointes. Prenons un exemple. Soient 1 page = 4 Ko, F = 1.2 et $|M| = 1000$ pages (la taille de la MC est donc de 4 Mo). Dans ces conditions, la taille de la relation S est donc inférieure ou égale à 3.3 Go. Nous voyons ainsi que les algorithmes que nous allons décrire sont opérationnels pour joindre vraiment de grosses relations.

3.2.1 Au moins une des relations à joindre tient en MC

Algorithme de jointure par boucles imbriquées Nous supposons ici que R, étant la plus petite des deux relations jointes, tient en MC, càd. $|R| \leq |M|$.

Utilisation de la MC Voyons d'abord, avant de décrire l'algorithme, comment la MC est utilisée.

1. Un groupe de $|R|$ pages est utilisé pour charger R depuis le disque vers la MC. Appelons ce groupe R_{MC}
2. 1 page, notée P_S, est utilisée pour lire,page par page, S en MC
3. 1 page de la MC, notée P_{res}, sert à construire le résultat de la jointure. Un couple (r,s) qui appartient au résultat de la jointure est d'abord écrit dans P_{res} avant d'être écrit sur disque

Par conséquent, pour être plus précis, la condition pour que cet algorithme soit utilisable est, non pas $|R| \leq |M|$ mais, plutôt, $|R| \leq |M| - 2$.

Algorithme

```
charger R depuis le disque vers la MC
Tant que non(fin de S) faire
  lire une page non encore lue de S dans P_S
  Pour chaque s dans P_S Faire
    Pour chaque r dans R_MC Faire
      Si r joint avec s Alors
        écrire (r,s) dans P_res
        Si P_res est pleine Alors
          écrire P_res sur disque
        Fsi
      Fsi
    Fait
  Fait
Fait
```

Entrées/sorties Dans cette situation, le coût en nombre d'E/S est idéal. En effet, puisque la relation R est chargée une seule fois en MC, chaque page de R est lue une fois et une seule du disque vers la MC. Egalement, chaque page de S est lue une fois et une seule du disque vers la MC puisque la relation S est parcourue une seule fois.

Algorithme de jointure par tri-fusion Nous supposons, pour décrire cet algorithme et pour simplifier, que les deux relations R et S tiennent en MC, càd. $|R| \leq |M|$ et $|S| \leq |M|$.

Première étape: tri La première étape consiste à obtenir, sur disque, chacune des relations R et S triées sur l'attribut de jointure. La figure 3.8 montre l'entrée et la sortie de cette étape.

Comment s'effectue le tri d'une relation? Par exemple, prenons la relation R (le même traitement est appliqué à la relation S).

1. Lire toute la relation R en MC. C'est possible puisque, d'après les hypothèses, $|R| \leq |M|$.

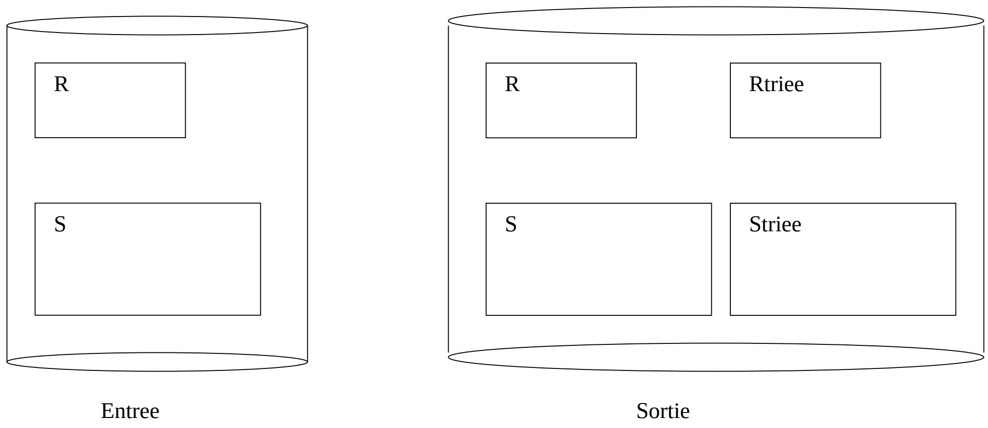


FIG. 3.8 – *Entrée et sortie de la 1ère étape du tri-fusion*

2. Avec un algorithme de tri permettant de trier un ensemble tenant en MC (un des plus connus est le tri rapide), trier l'ensemble en MC.

3. Ecrire le résultat du tri sur disque. On obtient R_{triee}.

Seconde étape: fusion Le résultat de la jointure est alors obtenu en effectuant la fusion (en anglais, merging) entre R_{triee} et S_{triee}. Supposons, pour simplifier, que le prédicat de jointure porte sur l'égalité des clés de chacune de ces relations. La clé de chaque relation est l'attribut c. Dans ces conditions, 3 pages seulement suffisent pour effectuer cette fusion.

- une page, appelée P_R, pour lire successivement chaque page de R en MC
- une page, appelée P_S, pour lire successivement chaque page de S en MC
- une page, appelée P_{res} où sont écrits successivement tous les couples (r,s) appartenant au résultat de la jointure. Dès que P_{res} est pleine, elle est écrite sur disque.

L'algorithme est le suivant :

```
r = R.tupleSuivant();
s = S.tupleSuivant();
Tant que non((fin de R) ou (fin de S)) Faire
  Tant que (r.c < s.c) Faire
    r = R.tupleSuivant();
  Fait

  Si (r.c = s.c) Alors
    Tant que (r.c = s.c) Faire
      ajouteRésultat(r,s);
      r = R.tupleSuivant();
    Fait

    s = S.tupleSuivant();
  Fsi

  Tant que (r.c > s.c) Faire
    s = S.tupleSuivant();
  Fait
Fait
```

Détaillons les fonctions *tupleSuivant* et *ajouteRésultat*. La fonction *tupleSuivant*, appliquée à la relation R (resp. S), consiste en l'algorithme suivant :

Si P_R est vide ou P_R a déjà été entièrement parcourue Alors
  Si R a dejà été entièrement parcourue Alors
    retourner fin
  Sinon
    lire une nouvelle page de R dans P_R
    retourner le premier tuple de R dans P_R
  Fsi
Sinon
  retourner le tuple suivant de R dans P_R

La fonction *ajouteRésultat* consiste en l'algorithme suivant :

écrire (r,s) dans P_{res}
Si P_{res} est pleine Alors
  l'écrire sur disque
Fsi

Entrées/sorties Examinons le nombre d'E/S nécessaires au déroulement de cet algorithme. Supposons qu'aucune des deux relations n'est triée. Pour trier R, chaque page de R est lue du disque vers la MC puis écrite de la MC vers le disque. Il en est de même pour trier la relation S. Ensuite, pour joindre les deux relations triées, chaque page de R (resp. S) est lue une fois du disque vers la MC. Nous ne comptons pas le nombre d'E/S nécessaires pour écrire le résultat de la jointure sur disque.

Ainsi, lorsqu'aucune des deux relations n'est triée, le coût en nombre d'E/S de l'algorithme de tri-fusion est trois fois plus important que celui de l'algorithme par boucles imbriquées. 3 E/S sont effectuées sur chaque page de R (resp. S). L'une des deux raisons suivantes peut expliquer l'utilisation de cet algorithme. Cet algorithme vaut le coup d'être utilisé car le résultat de la jointure est trié sur l'attribut de jointure (ce qui permet d'éliminer les duplicats) ou, alors, parce que l'une des deux relations à joindre (ou, idéalement, les deux relations) est déjà triée.

Algorithme de jointure par hachage (appelé Classic Hash Join) Deux conditions sont nécessaires pour que cet algorithme soit performant. Il est , bien sûr, utilisable dans d'autres conditions mais se révélera non performant pour cause de swap. D'une part, il faut que la table de hachage relative à la relation R (la plus petite des relations jointes) tienne dans $|M| - 2$ pages. Ainsi, puisque la taille de la table de hachage relative à R est approximée par $|R| * F$, il faut que $|R| * F \leq |M| - 2$. En effet, nous allons le voir ci-après, cet algorithme construit la table de hachage relative à R en MC en utilisant la fonction de hachage h. 2 pages de la MC doivent rester libres. Le reste est dédiée à contenir la table de hachage relative à R. Une première page, notée P_S, sert à lire successivement chaque page de S en MC. La seconde page, notée P_{res}, sert à construire le résultat au fur et à mesure.

L'algorithme est le suivant :

1. Construire, en utilisant la fonction de hachage h, la table de hachage HT_R relative à R. Pour ce faire, l'algorithme est le suivant:

   Tant que non(fin de R) Faire
     lire une page non encore lue de R en MC
     Pour chacun des tuples r de cette page Faire
       calculer h(r);
       inserer r dans HT_R;
     Fait
   Fait

2. Tant que non(fin de S) Faire
   Lire une page de S en MC dans P_S
   Pour chacun des tuples s appartenant à P_S Faire
   calculer h(s)
   sonder HT_R pour récupérer tous les tuples r qui joignent avec s
   Pour chacun de ces tuples r Faire
   écrire (r,s) dans P_{res}
   Si P_{res} est pleine Alors
   écrire P_{res} sur disque
   Fsi
   Fait
   Fait
   Fait

Entrées/sorties Examinons le nombre d'E/S nécessaires au déroulement de cet algorithme. Construire la table de hachage relative à R en MC nécessite de lire une fois chaque page de la relation R. La jointure en elle-même nécessite de lire chaque page de la relation S une fois également. Cet algorithme est donc aussi peu coûteux en nombre d'E/S que l'est l'algorithme par boucles imbriquées.

3.2.2 Aucune des deux relations à joindre ne tient en MC

Algorithme de jointure par boucles imbriquées Il existe deux versions de cet algorithme : une version dans laquelle R, la plus petite relation, est parcourue en premier, une autre version où S est parcourue en premier. Nous avons choisi de décrire cet algorithme quand R est parcourue en premier.

Utilisation de la MC La MC est utilisée comme suit :

1. $|M| - 2$ pages, notées R_{MC}, sont utilisées pour charger R en MC par paquets de $|M| - 2$ pages
2. 1 page, notée P_S, est utilisée pour lire S en MC page par page
3. 1 page de la MC, notée P_{res}, sert à construire le résultat de la jointure

Algorithme

Tant que non(fin de R) Faire
lire au maximum $|M| - 2$ pages de R dans R_{MC}
Tant que non(fin de S) Faire
lire une nouvelle page de S dans P_S
Pour chaque r dans R_{MC} Faire
Pour chaque s dans P_S Faire
Si r joint avec s Alors
écrire (r,s) dans P_{res}
Si P_{res} est pleine Alors
écrire P_{res} sur disque
Fsi
Fsi
Fait
Fait
Fait
Fait

Entrées/Sorties Dans cet algorithme, la relation R est chargée en MC par paquets d'au maximum $|M| - 2$ pages. Mais, elle est parcourue entièrement une fois et une seule. Par conséquent, chaque page de R est lue une fois et une seule. Par contre, la relation S est parcourue entièrement à chaque fois que $|M| - 2$ pages de R sont chargées en MC. Ainsi, chaque page de S est lue $\lceil \frac{|R|}{|M|-2} \rceil$ fois.

Par conséquent, plus la relation R est grande par rapport à la MC, plus la relation S est lue et donc plus le nombre d'E/S est grand et l'algorithme lent. Nous allons voir que les autres algorithmes que nous présentons s'évertuent à faire mieux que ça.

Quand la relation S est parcourue en premier, les rôles de R et de S permutent.

Algorithme de jointure par tri-fusion Il existe de nombreuses variantes de l'algorithme de jointure par tri-fusion. Dans celle que nous expliquons, la première étape consiste à trier chacune des deux relations jointes sur l'attribut de jointure. Nous allons voir que ce tri s'effectue progressivement, en générant des sous-ensembles triés de R et de S de plus en plus gros. Ces sous-ensembles triés s'appellent des monotonies (en anglais, runs).

La seconde étape consiste alors à fusionner les deux relations triées.

Première étape : tri des deux relations Voyons comment trier R (resp. S). Supposons que la taille de la MC est $|M| + 1$ pages et que $|M| = 4$ pages. La figure 3.9 montre les deux premières étapes de ce tri. Les monotonies obtenues font $|M|^2$ pages.

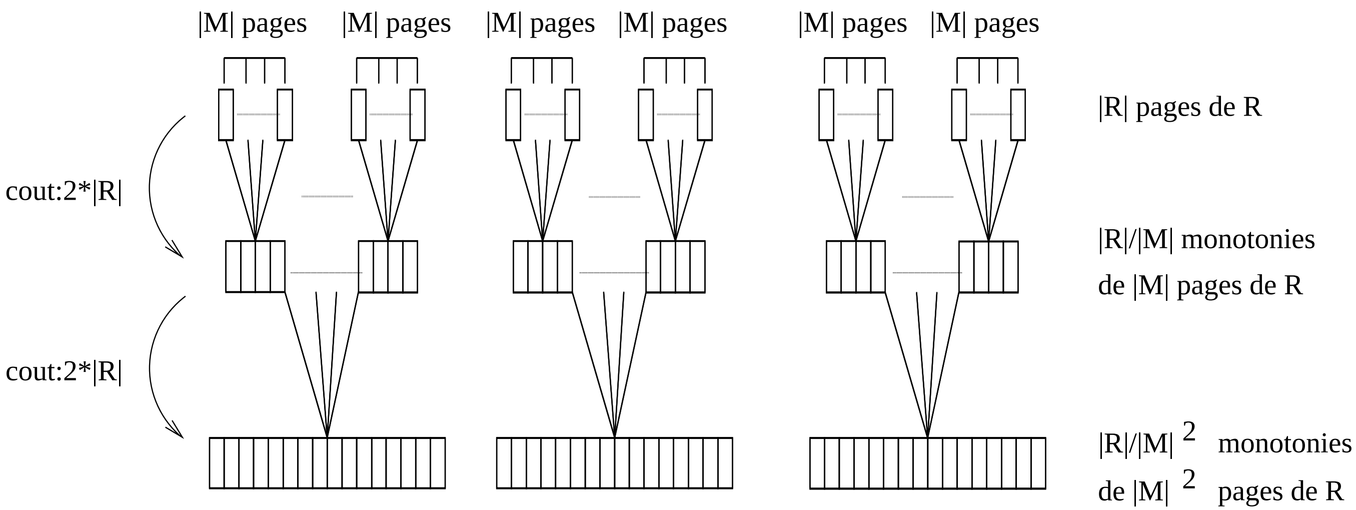


FIG. 3.9 – *Obtenir des monotonies*

La première étape consiste à obtenir autant de monotonies de $|M|$ pages qu'il y a de sous-ensembles de $|M|$ pages dans R. Pour obtenir une monotonie de $|M|$ pages, il suffit de prendre $|M|$ pages de R, de les charger en MC et, à l'aide d'un algorithme de tri, de trier ces pages en MC et de réécrire sur disque les $|M|$ pages triées. Le coût de cette première étape est donc : $2 * |R|$. Chaque page de R est lue du disque vers la MC puis écrite de la MC vers le disque.

La seconde étape consiste à faire la même chose mais cette fois en considérant les $|R|/|M|$ monotonies de $|M|$ pages créées à l'étape précédente. Le coût en nombre d'E/S de cette étape est aussi $2 * |R|$.

Ainsi, pour trier R (resp. S), il faut itérer autant de fois qu'il est nécessaire pour obtenir, au final, une seule monotonie de $|M|^{log_{|M|}|R|}$. Le coût pour trier R (resp. S) est $2 * |R| * log_{|M|}|R|$ (resp. $2 * |S| * log_{|M|}|S|$).

Seconde étape : fusion Ensuite, il ne reste plus qu'à fusionner les deux ensembles triés relatifs à R et à S. Pour ce faire, 3 pages de la MC suffisent :

- une première page pour charger successivement chaque page de l'ensemble trié relatif à R
- une deuxième page pour charger successivement chaque page de l'ensemble trié relatif à S
- une troisième page où seront écrits successivement chaque couple (r,s) appartenant au résultat de la jointure avant leur écriture sur disque

Cette étape nécessite donc, si nous ne comptons pas le coût pour écrire le résultat de la jointure sur disque, que chaque page de R (resp. S) soit lue une fois.

Nombre d'Entrées/Sorties Le nombre d'E/S est : $2 * |R| * log_{|M|}|R| + 2 * |S| * log_{|M|}|S| + |R| + |S|$.

Algorithme de jointure par hachage simple (Simple Hash Join) Cet algorithme est valable quand $|R| * F > |M| - 2$. Autrement dit, la taille de la table de hachage relative à R est strictement supérieure à $|M| - 2$.

L'algorithme de jointure par hachage simple se déroule comme suit :

1. Soit $P = min(|M| - 2, |R| * F)$. Etant donnée la fonction de hachage h, choisir un sous-ensemble X de valeurs retournées par h. Ce sous-ensemble est choisi de telle sorte que le nombre de tuples de R tels que h(r) est dans X tient dans $\frac{P}{F}$ pages. Parcourir chaque tuple de R. Pour chacun des tuples r de R, calculer h(r). Si h(r) est dans X, insérer r dans une table de hachage HT en MC (HT tient en MC d'après ce que nous avons dit avant). Sinon, recopier r sur disque. La figure 3.10 montre le résultat obtenu à la fin de cette étape.

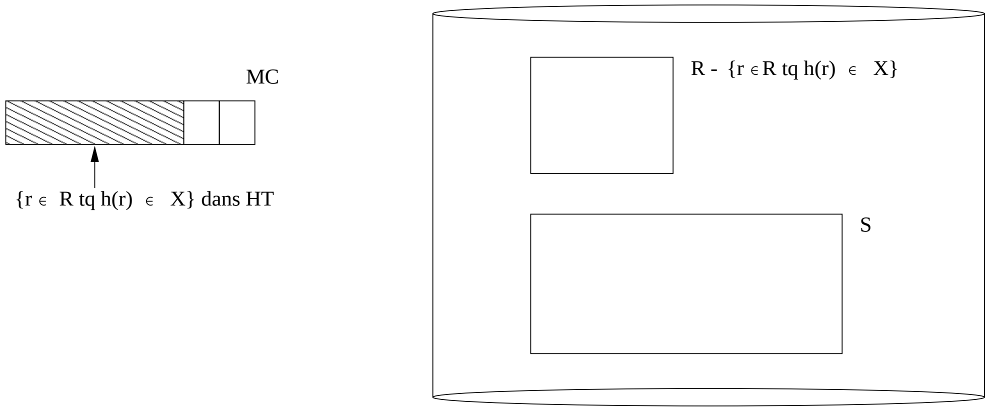


FIG. 3.10 – *Sortie de la 1ère étape du hachage simple*

2. Parcourir alors la relation S. Pour chacun des tuples s de S, calculer h(s). Sonder la table de hachage HT. Si s joint avec des tuples r appartenant à HT, écrire chaque couple (r,s) dans P_{res}. A chaque fois que P_{res} est pleine, l'écrire sur disque. Si s ne joint avec aucun tuple r appartenant à HT, écrire s sur disque. La figure 3.11 montre le résultat obtenu à la fin de cette étape.

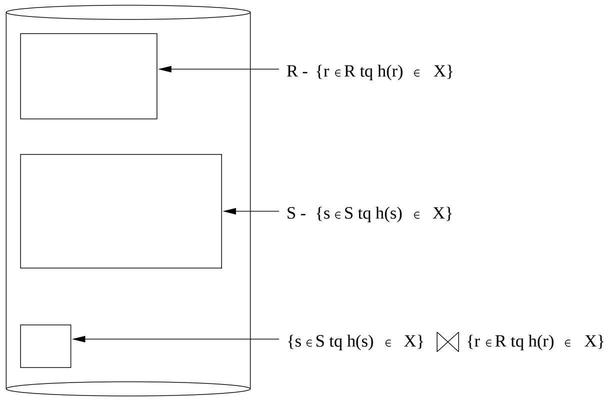


FIG. 3.11 – *Sortie de la seconde étape du hachage simple*

3. Répéter les étapes (1) et (2) en remplaçant R et S par les sous-ensembles de R et de S qui ont été écrits sur disque dans la passe précédente. L'algorithme s'arrête quand aucun tuple de R n'est écrit sur disque lors de l'étape (1).

Nombre d'Entrées/Sorties Estimons le coût en nombre d'E/S de cet algorithme.

Le nombre de passes (ou itérations) nécessaires à son exécution est $\lceil \frac{|R|*F}{|M|-2} \rceil$.

Calculons d'abord le nombre d'E/S opérées sur les pages de la relation R, noté $nbES(R)$. Il est inférieur ou égal à $\sum_{k=0}^{\lceil \frac{|R|*F}{|M|-2} \rceil -1} (2k+1) \times (\frac{|M|-2}{F})$. En effet, à chaque itération, c'est au maximum $\frac{|M|-2}{F}$ pages de R qui sont jointes. A la fin de la $(k+1)$ième itération, $(2k+1)$ E/S ont été effectuées sur ce paquet de pages.

Le même raisonnement est valable pour les pages de S. Cependant, le nombre de pages de S jointes lors d'une itération varie d'une itération à l'autre. Appelons $n_k(S)$ le nombre de pages de S qui joignent avec les $\frac{|M|-2}{F}$ pages de R appartenant à HT à la $(k+1)$ième itération. Le nombre d'E/S opérées sur les pages de la relation S, noté $nbES(S)$, est inférieur ou égal à $\sum_{k=0}^{\lceil \frac{|R|*F}{|M|-2} \rceil -1} (2k+1) \times n_k(S)$.

Le nombre d'E/S est donc inférieur ou égal à $nbES(R) + nbES(S)$. Par conséquent, plus la relation R est grande par rapport à la mémoire centrale, plus cet algorithme devient lent.

Algorithme de hachage GRACE (GRACE Hash Join) Cet algorithme est décrit dans [KTMO83]. L'idée qui se situe derrière cet algorithme, ainsi que l'algorithme de hachage HYBRID que nous présentons ensuite, a été exposée dans [Bab79, Goo81]. Elle est la suivante. Pour effectuer la jointure entre deux relations R et S qui ne tiennent pas en MC, pourquoi ne pas, d'abord, partitionner R (resp. S) en n partitions R_1, …, R_n (resp. S_1, …, S_n) et , ensuite, effectuer la jointure entre R_1 et S_1, R_2 et S_2, etc. Le partitionnement de chaque relation est intéressant car il permet d'obtenir des ensembles plus petits (l'idéal est qu'ils tiennent en MC). Par conséquent, chaque jointure entre chaque R_i et chaque S_i peut s'évaluer efficacement.

Décrivons maintenant plus précisément l'algorithme GRACE.

1. La première étape de cet algorithme consiste à partitionner R (resp. S) en $|M|$ sous-ensembles de taille équivalente R_1, …, $R_{|M|}$ (resp. S_1, …, $S_{|M|}$). Pourquoi avoir choisi $|M|$? Il y a deux arguments pour répondre à cette question. Le premier, c'est que la table de hachage relative à chaque R_i tient en MC. En effet, chaque R_i étant de taille équivalente, $| R_i |= \frac{|R|}{|M|}$. Ainsi, $|HT_{R_i}| = \frac{|R|*F}{|M|}$. Comme $|R| < |S|$ et $\sqrt{|S|*F} \leq |M|$, $|HT_{R_i}| < |M|$. Le second argument provient de la manière avec laquelle le partitionnement est effectué. Il est effectué en utilisant une fonction de hachage h_1, choisie avec beaucoup

de précaution. Chaque partition R_i est $\{r \in R | h_1(r) = i\}$. Pourquoi prendre tant de précaution? L'objectif du partitionnement est d'obtenir des partitions de taille équivalente, ce qui n'est pas facilement garanti. Quelquefois, on obtient ce que l'on veut, d'autres fois, non. Nous n'allons pas rentrer dans les détails. Pour simplifier, nous supposons que nous utilisons une fonction de hachage $h_1 : R \rightarrow [1, |M|]$ et qu'elle a été choisie de façon à obtenir des partitions de R de taille équivalente. Voyons juste comment ce partitionnement utilise h_1. La MC est constituée de $|M|$ pages M_i. L'algorithme de partitionnement est le suivant :

```
Pour chaque tuple r de R Faire
  calculer h1(r)
  Si h1(r) = i Alors
    écrire r dans Mi
    Si Mi est pleine Alors
      écrire Mi sur disque
    Fsi
  Fsi
Fait
```

Le partitionnement s'effectue en une passe. Petite remarque : il serait plus correct de dire que la taille de la mémoire centrale est $|M| + 1$ pages. En effet, pour effectuer le partitionnement, une page de la MC est utilisée pour charger successivement chaque page de R. Les $|M|$ autres pages servent à contenir, chacune, les tuples appartenant à chacune des $|M|$ partitions créées.

2. La seconde étape consiste alors à effectuer successivement la jointure entre R_1 et S_1, puis la jointure entre R_2 et S_2, etc. L'algorithme utilisé pour effectuer la jointure entre chaque R_i et chaque S_i tire partie du fait que la table de hachage relative à chaque R_i tient en MC. En effet, nous présentons ici une version de l'algorithme de hachage GRACE dans laquelle un algorithme de hachage est utilisé pour effectuer la jointure entre chaque R_i et chaque S_i ([WKO+84]). Dans la version originelle de l'algorithme de hachage GRACE ([KTMO83]), un algorithme de tri-fusion est utilisé. Notons que tout autre algorithme de jointure pourrait être utilisé. La jointure entre R_i et S_i s'effectue de la manière suivante :

   (a) parcourir R_i. Construire, avec la fonction de hachage h (Attention! h est différente de h_1), la table de hachage relative à R_i, HT_{R_i}.

   (b) parcourir S_i. Pour chaque tuple s de S_i, cacluler h(s). Sonder HT_{R_i} pour récupérer les tuples r de R_i qui joignent avec s. Ajouter chaque couple (r,s) au résultat.

Entrées/Sorties Le nombre d'E/S dans l'algorithme GRACE est constant quelque soit la taille de la MC. En effet, le nombre d'E/S relatif à chaque page de R (resp. S) est 3 :

- 1 lecture et 1 écriture : pour partitionner une relation (que ce soit R ou S), il est nécessaire de lire puis d'écrire chaque page de celle-ci.

- 1 lecture : pour faire la jointure entre chaque R_i et chaque S_i, il est nécessaire de lire chaque page de R_i et de S_i du disque vers la MC.

Algorithme de hachage HYBRID (HYBRID Hash Join) Cet algorithme, décrit dans [WKO+84], est une variante de l'algorithme GRACE.

L'objectif, ici, est d'améliorer la performance de l'algorithme GRACE. Le constat est le suivant. Actuellement, la taille des MCs ne cesse d'augmenter et leur prix de diminuer. Supposons que nous disposons vraiment d'une grande MC. L'algorithme GRACE conduit à partitionner R en $|M|$ sous-ensembles. L'objectif est d'obtenir des partitions suffisamment petites pour que la table de hachage relative à chaque partition de R tienne en MC. Cependant, vu que la taille de la MC dont nous disposons est vraiment grande, il n'était peut-être pas utile de partitionner R en autant de sous-ensembles. L'algorithme de hachage HYBRID choisit donc de partitionner R en $B + 1$ sous-ensembles, $B < |M|$.

1. Comme nous venons de le dire, la première étape consiste à partitionner la relation R en $B + 1$ sous-ensembles $R_0, \ldots, R_B$. D'après [WKO+84], B est choisi égal à $max(0, \frac{|R|*F-|M|}{|M|-1})$. Effectuer ce partitionnement consiste à trouver une fonction de hachage h_1 de telle sorte que (1) la table de hachage relative à R_0 tienne dans $|M| - B$ pages, (2) $R_1, \ldots, R_B$ sont de taille équivalente et (3) la table de hachage relative à chaque R_i, $i > 0$ tienne en MC. Le lecteur trouvera toutes les explications dans [WKO+84]. La figure 3.12 montre comment est utilisée la MC. Chaque page $M_1, \ldots, M_B$ sert de tampon mémoire pour la construction des partitions $R_1, \ldots, R_B$. $|M| - B$ pages sont réservées pour contenir la table de hachage relative à R_0.

FIG. 3.12 – *Utilisation de la MC dans l'algorithme HYBRID*

L'algorithme est le suivant :

Pour chaque tuple r de R Faire
  calculer h_1(r)
  Si (h_1(r) = 0) Alors
    calculer h(r) /* ATTENTION! h est différente de h_1 */
    insérer r dans HT_{R_0}
  Sinon /*h_1(r) = i*/
    écrire r dans M_i
    Si M_i est pleine Alors
      l'écrire sur disque
    Fsi
  Fsi
Fait

La figure 3.13 montre le résultat obtenu à la fin de cette étape.

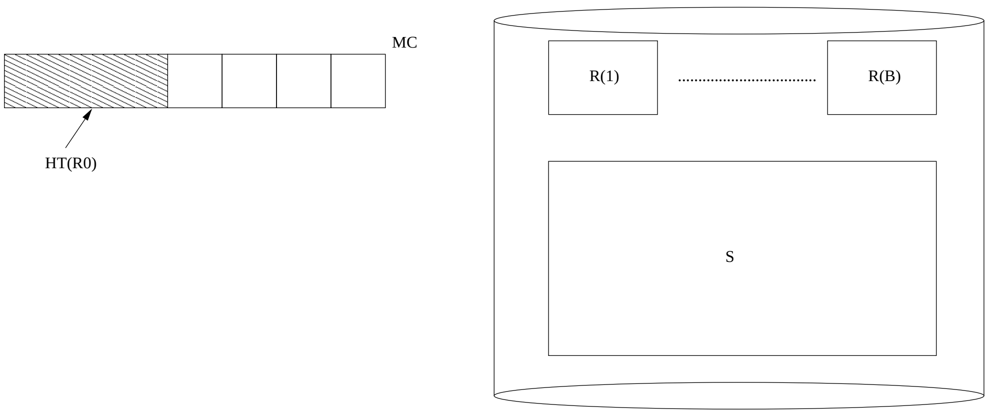


FIG. 3.13 – *Sortie de la 1ère étape de l'algorithme HYBRID*

2. La seconde étape consiste à partitionner S en $B + 1$ sous-ensembles. Cependant, la partition S_0 n'est pas écrite sur disque. En effet, la jointure entre R_0 et S_0 est effectuée immédiatement. L'algorithme est le suivant :

   Pour chaque tuple s de S Faire
     calculer $h_1(s)$
     Si $(h_1(s) = 0)$ Alors
       calculer h(s)
       sonder HT_{R_0}
       Pour chacun des tuples r de HT_{R_0} qui joignent avec s Faire
         (r,s) appartient au résultat
       Fait
     Sinon /* $h_1(s) = i$*/
       écrire s dans M_i
       Si M_i est pleine Alors
         l'écrire sur disque
       Fsi
     Fsi
   Fait

   La figure 3.14 montre le résultat obtenu à la fin de cette étape.

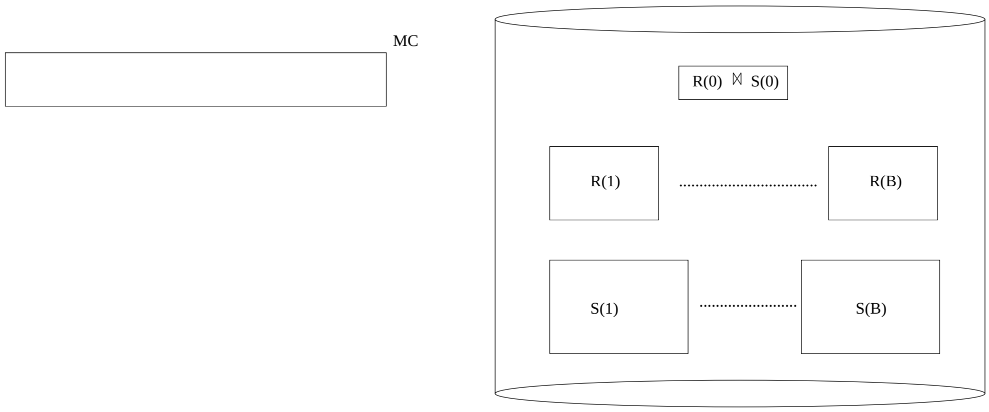


FIG. 3.14 – *Sortie de la seconde étape de l'algorithme HYBRID*

3. Effectuer chaque jointure entre chaque R_i et chaque S_i. Chaque jointure est effectuée en construisant la table de hachage relative à R_i en MC en utilisant la fonction de hachage h. Pour chaque tuple de S_i, h(s) est calculé pour sonder la table de hachage relative à R_i et obtenir les tuples r de R_i qui joignent avec s.

Entrées/Sorties Avec l'algorithme HYBRID, le nombre d'E/S diminue quand la taille de la MC augmente. En effet, plus la MC est grande, plus la taille de R_0 peut être importante et, donc, plus on économise d'E/S. C'est en cela qu'il peut être intéressant par rapport à l'algorithme GRACE.

3.2.3 Conclusion

Nous avons décrit quelques-uns des algorithmes de jointure de référence de la littérature. Bien sûr, il en existe beaucoup d'autres. Citons, par exemple, celui décrit dans [HWM98] qui a comme particularité d'exploiter le placement physique des données jointes, en particulier, le fait qu'elles sont "presque" ordonnées sur disque.

Le lecteur doit sans doute se poser la question suivante : quel algorithme est-il optimal, et dans quelle situation? Quand faut-il choisir l'un, quand faut-il choisir l'autre?

Essayons de donner des éléments de réponse à cette question. Il faut bien voir que les algorithmes par hachage GRACE et HYBRID ne sont efficaces que si les partitions relatives à chacune des relations jointes sont de taille équivalente. Une chose est sûre, la distribution des données a beaucoup d'influence. En effet, quand les données sont uniformément distribuées, il y a de fortes chances d'obtenir des partitions de taille équivalente et, donc, d'avoir des performances optimales. Quand ce n'est pas le cas, il est plus risqué d'utiliser ces algorithmes.

Par conséquent, on peut dire que, dans le cas général, si la mémoire centrale est petite, l'algorithme par hachage GRACE a de très grandes chances d'être le plus rapide. Par contre, quand la mémoire centrale devient grande, peut-être vaut-il mieux jouer la sécurité en optant pour l'algorithme par boucles imbriquées. En effet, celui-ci possède un grand avantage par rapport à ses concurrents : son coût en E/S n'est pas optimal mais on peut être sûr de ses performances.

3.3 Stratégies de recherche

L'optimisation de requêtes, nous l'avons vu, est un processus à deux niveaux, logique et physique. Comme nous l'avons vu précédemment, optimiser une requête au niveau logique consiste à la considérer comme une expression algébrique et à la transformer progressivement en utilisant des règles d'équivalence algébrique [CD92, CM93]. L'optimisation au niveau physique consiste, quant à elle, à trouver, parmi tous les plans d'exécution[1] qui correspondent à une expression algébrique donnée, celui de coût minimal ou, plutôt, un plan d'exécution proche du plan optimal. Nous reviendrons sur ce point dans la suite. Deux plans d'exécution sont comparés vis à vis d'une fonction de coût [GGT95]. Attention! Le coût associé à un plan d'exécution n'est pas forcément le coût réel en nombre d'E/S et d'instructions élémentaires de l'algorithme sous-jacent. Il faut plutôt le voir comme une mesure de comparaison qui ne représente pas nécessairement la réalité.

L'ensemble des plans d'exécution possibles, pour une requête complexe, peut être très grand. Cet ensemble est appelé **espace des plans**. Par conséquent, la question suivante se pose. Si, pour une requête donnée, l'optimiseur énumère tous les plans d'exécution possibles pour obtenir le plan d'exécution optimal, le temps passé à l'optimisation risque d'être plus grand que la durée nécessaire à l'évaluation de la requête elle-même. Optimiser efficacement une requête revient donc à trouver un bon compromis entre le temps passé à l'optimisation et le temps de l'évaluation [LV91]. Un optimiseur utilise, pour ce faire, une stratégie de recherche, càd. une tactique, qui lui permet de trouver une solution optimale ou proche de l'optimal sans avoir à examiner tout l'espace des plans [SG88, Gar99]. L'ensemble des plans examinés par un optimiseur est appelé **espace de recherche**.

De nombreuses stratégies de recherche existent dans la littérature. Dans ce chapitre, nous allons décrire en bref - ce n'était pas, à proprement parler l'objet de notre travail - certaines d'entre elles. Traditionnellement, on distingue deux familles de stratégies de recherche : les stratégies énumératives et les stratégies aléatoires. Celles-ci peuvent utiliser des heuristiques afin d'accélérer l'optimisation. En effet, dans le cas d'un interprète, la rapidité de l'optimisation est critique.

3.3.1 Les stratégies de recherche énumératives

Comme leur nom l'indique, ces stratégies considèrent, ou plutôt énumèrent, des plans d'exécution possibles pour évaluer la requête à optimiser.

Cette stratégie peut être exhaustive. Dans ce cas, l'espace de recherche est exactement l'espace des plans, càd. que l'optimiseur examine tous les plans d'exécution possibles pour évaluer la requête. Cependant, même si cette stratégie peut être concevable pour une requête simple (la cardinalité de l'espace des plans dans ce cas n'est pas très grande), elle s'avère inadéquate quand la requête est compliquée.

Plusieurs types d'algorithmes peuvent implanter une stratégie énumérative : recherche en largeur d'abord, en profondeur d'abord ou, encore, recherche du meilleur d'abord.

Dans les optimiseurs commerciaux, c'est la programmation dynamique [SAC+79] qui est utilisée. Cette stratégie construit au fur et à mesure le plan d'exécution optimal. Cette technique découle de la recherche

1. un plan d'exécution est une expression algébrique dans laquelle chaque opération connaît l'algorithme qui l'évalue

opérationnelle. Le principe est que toute sous-politique d'une politique optimale est optimale. Prenons un exemple. Supposons que nous voulons trouver le plan d'exécution optimal pour effectuer n jointures. Le plan d'exécution optimal pour effectuer k jointures est la meilleure façon de joindre une des relations avec la meilleure façon de joindre les k−1 autres. Ainsi, progressivement, cette stratégie énumère la meilleure façon d'accéder à chaque relation jonte, puis, les meilleures façons pour effectuer la jointure entre tous les couples de relations possibles parmi les n à joindre, puis, entre trois relations, etc, ainsi de suite, jusqu'à obtenir le plan optimal.

3.3.2 Les stratégies aléatoires

L'algorithme de programmation dynamique devient très coûteux pour déterminer la meilleure façon de joindre un nombre important de relations, si l'on considère tous les arbres de jointure possibles. La complexité est de l'ordre de 2^N où N est le nombre de relations. Il n'est donc que difficilement applicable au delà d'une dizaine de relations [SG88]. L'objectif des stratégies de recherche aléatoires est donc, dans le cas où les requêtes sont compliquées (ce qui est souvent le cas dans le modèle objet), de réduire l'espace de recherche. Pour cela, elles vont s'intéresser plus particulièrement, mais pas exclusivement, aux transformations menant à des plans moins coûteux.

Le recuit simulé Cette technique s'appelle, en anglais, *simulated annealing* [IW87].

Cette stratégie part d'un plan donné choisi aléatoirement. Puis, des transformations sont successivement appliquées aux plans considérés. Cette stratégie est originale car, étant donné un plan à un instant donné, les transformations qui lui sont appliquées mènent, bien sûr, à des plans moins coûteux et, également, à des plans plus coûteux. Cette exploration est justifiée car elle permet d'explorer un espace de plans plus large. De plus, elle est contrôlée. En effet, plus l'algorithme se déroule, moins il est probable que la stratégie examine des plans moins performants que les précédents. Comment cette probabilité est-elle calculée? A chaque instant de son déroulement, l'algorithme utilise un paramètre appelé température, **T**. Au début, cette température est très grande. Puis, elle diminue au fur et à mesure au cours du temps. Quand la température vaut T, la probabilité d'examiner des plans plus coûteux est $e^{-\frac{1}{T}}$. Ainsi, lorsque le système gèle (la température devient égale à 0), les seuls plans examinés ne peuvent être que des plans moins coûteux que les précédents.

La stratégie par amélioration itérative Cette stratégie part de n plans choisis au hasard ([NSS86]). Sur chacun de ces n plans, la stratégie applique successivement des transformations (par exemple, en poussant les restrictions et projections ou encore en choisissant les index adéquats), ce qui amène à n plans optimalement locaux. Le plan retenu est le plus rapide de ces n plans.

La stratégie à base d'algorithme génétique Cette stratégie découle des algorithmes génétiques ([BFI91, Gol89]). Le principe est le suivant. En partant d'une population de plans , la stratégie la reproduit par mutation et par croisement et conserve, dans chaque génération, les meilleurs éléments. La stratégie converge ainsi vers une population contenant la solution optimale ou presque.

Deuxième partie

Requêtes sur structures hiérarchiques : étude d'un système industriel

Introduction

L'étude[2] que nous décrivons ci-après a débuté en octobre 1997. Elle s'est effectuée au sein de la division objet de la société Ardent Software (ancienne O_2 Technology). Quand nous l'avons commencée, notre objectif était d'améliorer les performances de l'interprète-optimiseur OQL qui était alors implanté dans ce système, notamment, pour évaluer des requêtes sur des données au format d'arbre. Notre souhait était d'améliorer l'existant sans tout réécrire.

Pour ce faire, nous avons réalisé un banc d'essais de requêtes OQL. Notre but était surtout de comprendre l'influence du placement physique des données et de la taille de la base sur les performances de différents algorithmes. Dans le chapitre 1, nous décrirons l'environnement de travail que nous avons choisi. Nous y donnerons le schéma des bases de test. Nous expliquerons comment nous avons utilisé le système O_2 et les paramètres que nous avons fait varier. Nous donnerons le type de requête testée ainsi que les algorithmes étudiés dans le chapitre 4.

Les expériences nous ont montré que les performances des accès associatifs dans O_2 n'étaient pas optimales. Dans le chapitre 3, nous allons expliquer pourquoi et proposer des solutions pour y remédier.

Elles nous ont également permis de comparer les performances de la navigation par rapport aux jointures pour évaluer une requête sur des données au format d'arbre. C'est l'objet du chapitre 4. Nous allons voir, notamment, que la navigation est sûrement préférable aux algorithmes sophistiqués de jointure quand les données conservent leur forme d'arbre sur disque.

Les enseignements que nous avons tirés de nos expériences sont intéressants car généraux. Ils s'appliquent aux systèmes purement objets qui, par rapport aux systèmes relationnels, ont l'avantage de représenter, de la même façon, les objets en mémoire et sur disque. Ils mettent en valeur certains points forts et points faibles de ce type de systèmes. Notamment, ils montrent que ceux-ci ont, avant tout, été conçus pour favoriser la navigation et que les performances des accès associatifs en pâtissent. Néanmoins, nous allons le voir, cette faiblesse n'est pas inhérente au modèle de données.

Au bout de deux années, ce travail s'est arrêté en dépit de notre volonté. En effet, à la fin du mois d'août 1999, la direction de la société Ardent Software a décidé de licencier la quasi totalité du personnel travaillant dans sa division objet. Nous n'avons donc pas pu modifier l'optimiseur en fonction des résultats que nous avions obtenus.

2. Cette étude a fait l'objet d'une publication dans la session industrielle de la conférence SIGMOD 2000 [WCB+00]

Chapitre 1

Paramètres du banc d'essais

A l'origine, la motivation de notre travail était d'améliorer les performances de l'optimiseur OQL implanté dans le SGBDO O_2. L'optimiseur en place appliquait des heuristiques pour trouver le plan d'exécution optimal relatif à une requête. Ceci menait parfois à des plans d'exécution optimaux, d'autres fois à des plans très peu performants, voire très mauvais. Depuis plusieurs années, l'optimiseur n'évoluait que de la façon suivante : un client se plaignait de la mauvaise performance d'une de ses requêtes, une analyse était effectuée afin de comprendre comment pallier à cela tout en conservant les améliorations antérieures et la généricité de l'optimiseur. Après cette analyse, soit l'optimiseur était amélioré, soit le client était invité à reformuler sa requête.

Notre objectif était donc de concevoir un optimiseur plus facilement améliorable avec stratégie de recherche et modèle de coût. Dans cette perspective, les premières questions que nous nous sommes posées sont les suivantes : quelles sont les statistiques à maintenir dans un SGBDO? Que peut être un modèle de coût en adéquation avec celles-ci? La réponse à ces questions est loin d'être immédiate et, d'ailleurs, après notre étude, nous ne sommes pas en mesure d'y répondre. En effet, un SGBDO permet à l'utilisateur de créer des données de types complexes. De ce fait, comme nous l'avons dit dans le chapitre 2, il est difficile de connaître le placement physique des données sur disque. Contrairement au relationnel qui ne manipule que des nuplets dont on connaît la taille maximale, l'objet permet de manipuler des collections dont il est impossible de connaître la taille par avance.

Afin d'apporter des éléments de réponse à ces questions, nous avons décidé de réaliser un banc d'essais (benchmark, en anglais) de requêtes OQL. Plus précisément, nous nous sommes interessés dans un premier temps à des requêtes très typiques, qui consistent à interroger des structures arborescentes. Nous décrirons ci-après en détail la structure des données que nous avons interrogées. L'histoire a voulu que nos expériences sur O_2 s'achèvent alors.

L'objectif de ce chapitre est de décrire l'environnement de travail que nous avons choisi pour réaliser nos expériences. En bref, nous avons évalué des requêtes OQL représentatives (jointure dépendante, sélection) dans des conditions différentes avec des algorithmes différents. Nous ne parlerons pas ici des requêtes que nous avons choisies de tester ni des algorithmes comparés. Nous parlerons uniquement des paramètres physiques et logiques que nous avons faits varier.

1.1 Choix du schéma

Toutes les bases sur lesquelles nous avons effectué des tests sont associées au même schéma. Il est fortement réduit. En effet, comme l'indique la figure 1.4, il ne contient que deux classes Médecin et Patient reliées entre elles par une relation 1-n : chaque médecin soigne plusieurs patients, un patient est soigné par un unique médecin. Il est inspiré du schéma Derby [Der97], schéma de test de performances pour bases à objets.

Typiquement, une base de test peut se représenter sous forme d'arbre. Représentons chaque objet par un noeud. La racine de l'arbre possède comme enfants tous les médecins de la base. Chaque médecin possède comme enfants tous ses patients.

Expliquons certains des attributs utilisés. L'attribut *med_id* (resp. *pat_id*) d'un médecin (resp. patient) est son numéro d'identification. Chaque médecin (resp. patient) a, bien sûr, un numéro d'identification différent. L'attribut *entier_aleatoire* d'un patient est un entier qui sert à attribuer aléatoirement un médecin à celui-ci. Nous reviendrons sur son utilité dans la suite.

1.2 Machine utilisée, déroulement des tests en architecture client-serveur

La machine sur laquelle les tests ont été effectués était une Sparc 20 avec 128 Mo de mémoire centrale, un disque dur SCSI de 2 Go et un processeur de fréquence égale à 100 MHz. Le système d'exploitation utilisé était Solaris 2.6. A part la base de données elle-même, tous les fichiers étaient gérés sous AFS (Andrew File System [AFS97]). Cela avait un coût non négligeable que nous n'avons pas pu évaluer. Egalement, tous les tests ont été lancés en mode graphique : le gestionnaire de fenêtres utilisé était twm, ce qui consommait un peu de mémoire.

Le SGBDO O_2, comme nous le disions au chapitre précédent, possède une architecture client-serveur. Nous avions donc prévu d'évaluer des requêtes OQL dans différentes conditions :

- soit le client et le serveur tournent sur la même machine, soit ils tournent sur deux machines différentes.
- soit la requête est évaluée à froid (le cache client et le cache serveur sont vides avant que la requête ne s'effectue), soit elle est évaluée à chaud (caches non vides avant que la requête ne s'effectue).
- la taille du cache serveur ainsi que celle du cache client peuvent varier.

Finalement, par manque de temps, nous avons évaluer les requêtes toujours dans les mêmes conditions :

- Client et serveur sont lancés sur une même machine, la machine de test
- Chaque requête est évaluée à froid
- La taille du cache serveur est égale à 4 Mo (1000 pages de 4 Ko). La taille du cache client est égale à 32 Mo (8000 pages de 4 Ko)

1.3 Paramètres concernant la taille de la base

Dans le schéma, chaque médecin est associé à n patients et chaque patient est associé à un unique médecin. La taille d'une base est fixée par deux paramètres : le nombre de médecins et le nombre moyen de clients par médecin. Nous avons considéré deux tailles :

- la première dans laquelle il y a 2000 médecins et une moyenne de 1000 clients par médecin
- la seconde dans laquelle il y a 1 million de médecins et une moyenne de 3 clients par médecin

La taille réelle des bases que nous avons créées variait entre 300 Mo et 800 Mo. Parlons brièvement de la taille des objets que nous avons manipulés. Nous comptons 4 octets par entier, 8 octets par identificateur d'objet, 16 caractères pour les chaînes de caractères et quelques octets supplémentaires dû au système. Quelque soit le nombre moyen de clients par médecin, la taille d'un objet de la classe Patient était d'environ 60 octets. Quand il y avait une moyenne de 3 clients par médecin, la taille d'un objet de la classe Médecin était de 120 octets. Elle était un peu plus petite quand il y avait une moyenne de 1000 clients par médecin. En effet, dans ce cas, l'attribut *clients* de chaque médecin avait, en moyenne, une taille de $8 * 1000$ octets, càd. 8 Ko. Comme cette taille était supérieure à celle d'une page, cet ensemble était stocké de façon séparée.

```
@r1    "Gaston Lagaffe"  .......  {r7,r22,r50}
@r2    "Boule"  ......  r15
@r3    "Lucky Luke"  ......  r13
.
.
@r7    "Obelix"  ......  r1
.
.
@r13   "Fantasio"  ......  {r3,r100,r214}
.
.
```

FIG. 1.1 – *Placement aléatoire*

1.4 Paramètres concernant le stockage physique des données

Comme nous l'avons dit plus haut, les bases que nous avons considérées contiennent uniquement des objets de deux classes : la classe Médecin et la classe Patient.

Notre objectif était de créer des bases ressemblant le plus possible à celles que l'on peut rencontrer dans la vie courante et de montrer comment le placement physique des données interrogées perturbe la performance de tel ou tel algorithme.

1.4.1 Placement aléatoire

Nous avons d'abord pensé à créer des bases contenant des objets de chacune des deux classes, placés de façon tout à fait aléatoire les uns par rapport aux autres. Ceci peut sans doute être le cas d'une base réelle en supposant que celle-ci est peuplée de façon tout à fait aléatoire. La figure 1.1 montre les premières pages d'une telle base dans le cas où chaque médecin a une moyenne de trois patients. Dans la suite, nous illustrerons nos propos en gardant cette hypothèse. De plus, les adresses physiques relatives à chaque objet seront préfixées par le caractère '@'. Rappelons également que, dans le cas 1-3, la valeur de l'attribut *clients* d'un médecin - un ensemble de trois Rids[1] - est stockée dans l'enregistrement correspondant à celui-ci.

1.4.2 Cluster par classe

Ensuite, nous avons pensé à un mode de placement qui ressemble à celui rencontré dans le modèle relationnel. Dans cette politique, la base est constituée de deux fichiers : l'un contient tous les médecins, l'autre, tous les patients. Dans chacun de ces fichiers, les objets médecins (resp. patients) sont rangés en fonction de leur attribut *med_id* (reps. *pat_id*) qui peut être vu comme une clé. De plus, une des caractéristiques du placement physique, ici, est que les médecins et les patients sont reliés entre eux de façon aléatoire. La position physique des patients d'un médecin n'est pas déterminée en fonction de la sienne. De plus, les patients d'un médecin sont placés les uns par rapport aux autres tout à fait aléatoirement. La figure 1.2 montre cette politique de placement.

1.4.3 Cluster de composition

Le troisième et dernier placement que nous avons considéré a consisté à regrouper chaque médecin avec tous ses patients. La base peut être vue comme un unique fichier dans lequel tous les patients d'un médecin

1. La notion de Rid a été définie dans le chapitre sur le système O_2 de la première partie

```
@m1   "Fantasio"   ......  {p14,p22,p50}
@m2   "Gaston Lagaffe"   ......  {p1,p44,p100}
.
.
.
.
.
```

```
@p1   "Boule"   ......  m2
.
.
@p14   "Obelix"   ......  m1
.
.
.
```

FICHIER CONTENANT LES MEDECINS

FICHIER CONTENANT LES PATIENTS

FIG. 1.2 – *Cluster par classe*

```
@c1   "Fantasio"   ......  {c2,c3,c4}
@c2   "Boule"   ......  c1
@c3   "Lucky Luke"   ,.....  c1
@c4   "Obelix"   ......  c1
@c5   "Gaston Lagaffe"  ......  {c6,c7,c8}
.
.
.
```

FIG. 1.3 – *Cluster de composition*

sont directement après l'enregistrement de celui-ci. Le placement précédent, appelé cluster par classe, à condition que l'ordre dans lequel sont stockés les patients respecte celui dans lequel sont stockés les médecins, peut être vu comme une variante de celui-ci.

Cette politique de placement peut être spécifiée dans O_2 mais elle n'est pas garantie. En effet, le système fait de son mieux pour la satisfaire en fonction du programme qui crée les objets. C'est pour cette raison que nous avons préféré le programmer à la main.

Classes	**Types**
Medecin	nom: string med_id: integer adresse: string specialite: string cabinet: string clients: set(Patient)
Patient	nom: string pat_id: integer age: integer sexe: char entier_aleatoire: integer num: integer medecin_traitant: Medecin

Noms	**Types**
Medecins	set(Medecins)
Patients	set(Patient)

FIG. 1.4 – *Le schéma des bases de test*

Chapitre 2

Expérimentateurs en herbe

C'est le premier banc d'essais que nous avons réalisé. Nous l'avons donc commencé, conscients de notre inexpérience mais aussi du fait que *c'est en forgeant qu'on devient forgeron!* Deux années de travail nous ont permis de réaliser seulement un dixième des tests que nous avions prévu d'effectuer. Pourquoi? Comme tout novice, nous avons commis de nombreuses erreurs. Dans ce chapitre, nous relatons notre expérience. Nous expliquons les problèmes que nous avons successivement rencontrés et comment nous y avons remédié.

2.1 Créer une base de grande taille

Avant de débuter notre banc d'essais, nous avions l'habitude de créer des bases O_2 de petites tailles, juste quelques dizaines de Megaoctets. Ces programmes de chargement s'effectuaient rapidement (quelques minutes) et une seule transaction était nécessaire.

Quand nous avons commencé à créer nos premières bases de test, nous pensions ne pas rencontrer de problèmes particuliers. Nous nous attendions juste à des temps de chargement plus longs que ceux auxquels nous étions habitués. Notamment, nous pensions, par exemple, que le temps de chargement d'une base de un Gigaoctet était dix fois plus long que le temps de chargement d'une base de 100 Megaoctets.

Et bien, nos préjugés se sont révélés inexacts. Nous allons, ci-après, expliquer pourquoi et les enseignements fort intéressants que nous avons tirés de nos expériences.

Nous avons avancé par paliers en créant des bases de plus en plus grosses. Au départ, nous créions nos bases successives au moyen d'une seule transaction et tout se passait fort bien. Cependant, lors de la création d'une base, le programme de chargement s'est brusquement arrêté, le message suivant apparaissant à l'écran : *la mémoire est pleine!* D'où venait le problème? Notre programme de chargement avait-il des fuites mémoire, càd. que nous allouions des pointeurs sans jamais les désallouer? Nous avons donc, dans un premier temps, vérifié ce point. Alors que nous nous étions assurés que notre gestion de la mémoire était bonne, le problème persistait.

En fait, la raison était que nous créions un nombre trop important d'objets dans notre unique transaction. En effet, voyons ce qui se passait. Pendant la durée d'une transaction, toutes les données créées résident, dans la limite de la place disponible, en mémoire centrale et, aussi, dans la zone de swap. Dans notre cas, le nombre d'objets créés devenait si grand que toute la mémoire se retrouvait, à un moment donné, pleine. D'où le message.

Pour résoudre ce problème, nous avons donc découpé notre programme de chargement en plusieurs transactions de taille équivalente. Là encore, nous avons tâtonné. Combien d'objets créer par transaction pour éviter un swap trop important? Nous avons donc commencé par créer un petit nombre (une centaine) d'objets par transaction. Cependant, le nombre de transactions, donc de commits devenait si important que le programme de chargement durait une éternité, càd. un peu plus d'une journée. Nos contraintes étaient les suivantes : ne pas effectuer trop de commits mais en faire suffisamment pour que, lors de l'exécution d'une transaction, peu ou pas de swap ne s'effectue. En effet, il faut bien voir qu'un commit est une opération coûteuse. Elle correspond à l'écriture sur disque des données crées, modifiées ou supprimées dans la transaction courante. Cette écriture est d'ailleurs, pour des raisons de performances, retardée dans les SGBDs. En demandant conseil aux experts du système O_2, nous avons opté pour 10000 objets par transaction.

Une fois ce problème résolu, enfin, nous avions réussi à créer une base d'un Gigaoctet qui contenait quatre millions d'objets (un million de médecins et trois millions de patients). Oui, mais, le chargement durait quand même une bagatelle de douze heures... Nous étions un peu dubitatifs quant à la durée et, pour vérifier, nous avons demandé aux experts que nous pouvions questionner la durée nécessaire au peuplement d'une base d'un Gigaoctet. La réponse a été nette et sans bavure : une heure! Il n'y avait plus de doutes. Notre programme de chargement n'était pas du tout optimal. Mais, pourquoi donc? Voilà un problème qu'il est plus facile de poser que de résoudre ... Nous donnons, ci-après, les éléments de réponse que nous avons trouvés.

2.1.1 Indexer, oui, mais au bon moment

Les bases que nous avons peuplées possédaient des index. En effet, les tests de requêtes que nous voulions effectuer nécessitaient leur existence.

A ce propos, nous avions toujours entendu dire qu'il était plus efficace d'indexer une collection une fois qu'elle était peuplée plutôt que de mettre à jour l'index à chaque création d'objet. Cela est souvent vrai, mais, pas pour le premier index.

En effet, afin de gérer proprement les index, le système O_2 enregistre, pour chaque objet, les index auxquels il appartient (nous expliquerons pourquoi dans le chapitre suivant). Cette information est stockée sur disque dans l'en-tête de l'objet. Quand un objet devient persistant, et qu'il appartient à une collection indexée, le système crée un en-tête permettant de stocker une information à propos de huit index[1]. Par contre, si l'objet n'est pas indexé, aucun espace n'est réservé dans l'en-tête pour stocker l'information concernant les index auxquels il pourrait appartenir dans la suite. Par conséquent, quand nous indexions une collection après l'avoir entièrement peuplée, le système était contraint de déplacer les enregistrements correspondant à chaque objet de celle-ci afin d'y ajouter l'information sur les index. C'est une opération qui est très coûteuse surtout quand elle concerne une collection contenant entre un et trois millions d'objets. De plus, cela détruisait le placement physique des objets que nous tentions de fixer.

2.1.2 Influence des transactions

La gestion des transactions par le système prend un temps loin d'être négligeable. Quand plusieurs transactions clientes effectuent des opérations sur une même base, chacune doit avoir, à chaque instant, une vision cohérente des données. Pour garantir cela, le système pose des verrous en lecture/écriture sur les données auxquelles les différentes transactions accèdent. Cela permet d'éviter, par exemple, qu'une application lise une donnée en même temps qu'une autre la modifie. Ces poses de verrous sont coûteuses en termes de performances et, dans le contexte de notre utilisation, inutiles. En effet, quand nous peuplions une base, nous étions les seuls à travailler sur notre machine de test et les bases étaient créées, à chaque fois, en partant de zéro.

C'est pourquoi la plupart des systèmes de gestion de bases de données permettent de lancer des programmes de chargement sans transactions. De cette façon, le système ne gère aucun journal et ne pose aucun verrou en lecture/écriture. C'est ce que nous avons fait, ce qui nous a également permis d'améliorer notre temps de chargement.

2.1.3 Influence de la taille des caches serveur et client

Evidemment, la taille des caches serveur et client influence également beaucoup la durée de chargement. Par défaut, la taille de ces caches est fixée, par le système, à 4 Megaoctets.

Dans notre utilisation, quelles tailles était-il préférable de donner à ces caches? Rappelons leur rôle. Le cache serveur stocke et manipule des pages en provenance du disque : lire une page du disque vers le cache serveur coûte une E/S. Le cache client, quant à lui, stocke et manipule des pages en provenance du cache serveur pour l'application cliente, càd. le programme de chargement. La communication entre caches

1. Cela peut être étendu si nécessaire.

client et serveur se fait par RPCs[2]. Rappelons aussi notre environnement de travail. Nous disposions de 128 Megaoctets de mémoire centrale; seul un client tournait; aucun journal n'était géré.

Nous voulions optimiser les performances de notre programme de chargement, donc, minimiser le nombre d'E/S du disque vers le cache serveur et minimiser le nombre de RPCs entre le cache serveur et le cache client.

Minimiser le nombre d'E/S revenait à augmenter la taille de l'un des caches, qu'il soit client ou serveur.

Pour minimiser le nombre de RPCs, nous avons choisi d'agrandir la taille du cache client. La taille du cache serveur était de 4 Megaoctets (celle par défaut) et celle du cache client, de 32 Megaoctets. Pourquoi? Il faut bien voir que, lors de son déroulement, notre programme de chargement accédait, de manière aléatoire, aux mêmes pages plusieurs fois (notamment quand les médecins et les patients étaient reliés entre eux, nous le verrons dans la suite). Si le cache client était plus petit que le cache serveur, beaucoup de défauts de pages se produiraient dans le cache client. En effet, les pages accédées resteraient dans le cache serveur. Ceci aurait augmenté le nombre de RPCs. C'est ce que nous voulions éviter.

2.1.4 Cependant ...

Une fois ces améliorations mises en place, notre programme de chargement durait encore cinq heures et non une seule. Comment expliquer cela? Trois raisons nous semblent appropriées :

Problème de matériel Notre machine n'était certainement pas la meilleure, sur le plan performances. La fréquence du processeur était de 100 MHz. En tous cas, elle devait certainement être moins performante que celle utilisée pour effectuer un chargement d'un Gigaoctet de données en une heure.

Problème de langage Notre programme de chargement était écrit en O_2C, le premier langage à être implanté dans le système O_2. Nous l'avions choisi pour sa convivialité. Ceux qui ont été proposés ensuite sont le C++ et le Java. Le compilateur de O_2C était un peu vieux et, par conséquent, loin d'être optimal. Nous aurions eu de meilleures performances avec l'interface C++ ou, encore mieux, avec l'API[3] d'O_2.

Contraintes de placement Dans les bases de test que nous avons créées, nous imposions un certain placement physique des données. Ceci avait un coût non négligeable. Prenons le cas d'une base contenant un million de médecins. Chaque médecin a, en moyenne, trois patients et les objets sont regroupés sur disque suivant la classe à laquelle ils appartiennent. Le placement physique est donc ce qu'on appelle un cluster par classe.

Comment imposer ce placement physique? Dans le système O_2, il était possible de spécifier, à la déclaration de la base, les clusters. Le système faisait alors de son mieux pour satisfaire ce qui lui était demandé. Malheureusement, cette solution s'est révélée inadéquate puisque, notamment, rien n'était garanti.

Nous avons donc écrit un programme de chargement spécifique pour chaque placement physique. D'abord, pour garantir le regroupement des objets sur disque suivant la classe à laquelle ils appartiennent, nous avons déclaré deux volumes. L'un servait à contenir tous les médecins, l'autre, tous les patients. Voyons maintenant en quoi l'algorithme consistait.

1. Etape 1 : création de tous les médecins : Chaque médecin, souvenez-vous, possède un attribut *med_id*. Cet attribut a une valeur unique pour chaque médecin. Dans notre exemple, elle est comprise entre un et un million (nombre de médecins) et elle représente (en tous cas, elle peut être interprétée comme telle) la localisation physique de l'enregistrement correspondant au médecin.

2. Etape 2 : création de tous les patients : De la même façon que pour les médecins, chaque patient possède un attribut *pat_id* de valeur comprise entre un et trois millions (nombre de patients) qui peut être également vu comme représentant la localisation physique de l'enregistrement correspondant. De plus, afin de relier de façon aléatoire[4] un médecin à une moyenne de trois patients, chaque patient possède un attribut *entier_aléatoire*. La valeur de cet attribut est, comme son nom l'indique, un entier

2. Remote Procedure Calls
3. Application Programming Interface
4. Le placement sur disque des patients ne correspond pas au placement sur disque des médecins

aléatoire compris entre un et un million qui représente en fait la valeur de l'attribut *med_id* du médecin de ce patient. Nous avons utilisé la fonction **lrand48** pour générer des nombres aléatoires.

3. Etape 3 : Une fois ceci fait, il nous faut maintenant relier entre eux médecins et patients. Cette opération peut être coûteuse surtout si elle est mal implantée. En effet, dans une première version de notre programme de chargement, médecins et patients étaient reliés entre eux de la façon suivante. Nous disposions d'un index Patients.entier_aléatoire. L'algorithme était donc le suivant :

```
Pour chaque medecin m dans Les_Medecins Faire
  sesPatients = select pa from pa un Les_Patients
              where pa.entier_aleatoire = m.med_id

  Pour chaque pa dans sesPatients Faire
    m->ajoutePatient(pa)
    pa->assigneMedecin(m)
  Fait

Fait
```

Un commit était réalisé toutes les 10000 itérations. Comme il y avait un million de médecins, la boucle s'effectuait un million de fois, d'où un million de requêtes interrogeant à chaque fois le même index. Ceci ralentissait beaucoup notre programme.

Nous avons donc optimisé cela de la manière suivante. Plutôt que d'utiliser un index, nous utilisions deux listes : l'une (listeDesMedecins : list(Medecin)) servait à stocker tous les médecins, l'autre (listeDesPatients : list(Patient)), tous les patients. A chaque création d'un objet, celui-ci était inséré dans la liste correspondante. La position d'un objet dans chaque liste correspondait donc à sa date de création. Ainsi, relier médecins et patients consistait en l'algorithme suivant :

```
Pour i de 1 a 3 millions Faire
  pa = listeDesPatients[i];
  p = listeDesMedecins[pa->entier_aleatoire];
  p->ajoutePatient(pa);
  pa->assigneMedecin(p);
Fait
```

Comme dans la version précédente, un commit était effectué tous les 10000 patients traités. Cette solution s'est révélée meilleure vu (1) que l'accès au ième élément d'une liste est rapide et (2) que la taille des listes était petite, de l'ordre de quelques dizaines de Mégaoctets seulement.

2.2 Disposer, à tout moment, de toutes les bases de test

En octobre 1997, quand nous avons commencé notre banc d'essais, nous disposions d'une machine performante pour l'époque : une Sparc 20. Cependant, elle n'offrait que 500 Megaoctets de disque dur. Cela semblait insuffisant vu que nous voulions effectuer des tests sur des bases d'environ un Gigaoctet. Nous nous sommes donc procurés un disque dur SCSI de deux Gigaoctets. C'était le plus gros disque disponible sur le marché à cette époque et cela nous semblait largement suffisant.

Et bien, en fait, non! Expliquons pourquoi.

Au début de notre travail, nous étions convaincus qu'il fallait mieux réaliser toute une batterie de tests et, ensuite seulement, analyser les résultats. Ainsi, voilà comment nous procédions. Nous créions d'abord une base de tests. Sur celle-ci, nous effectuions tous les tests de requêtes que nous avions imaginés. Puis, une fois ces tests réalisés, comme nous n'avions pas assez d'espace disque pour stocker simultanément deux bases, nous détruisions la précédente pour créer la suivante. Ainsi de suite, ...

Or, il s'est avéré, après notre première batterie de tests, que nous avions d'autres idées d'expériences et que nous étions dans l'obligation de recréer toutes les bases que nous venions de détruire. Ces créations et destructions successives de bases nous ont fait perdre un temps précieux.

Par conséquent, à l'avenir, si nous avions à refaire un banc d'essais, nous prendrions un disque dur suffisamment grand pour contenir simultanément toutes les bases de test.

2.3 Comment stocker les résultats?

Au début de nos expériences, les résultats que nous obtenions étaient stockés dans des fichiers et nous les analysions à la main. Cette manière de faire, bien qu'adoptée par tous les expérimentateurs que nous avons interrogés, se révèle inadéquate quand le nombre de tests est grand. En effet, nous avions réussi à collecter quelques 5000 résultats. Dans ces conditions, les fichiers résultats que nous analysions devenaient trop nombreux, les noms que nous leur avions attribués étaient tout sauf clairs et nous devions les éditer à la main pour trouver l'information cherchée.

En cherchant comment nous pouvions améliorer les choses, nous nous sommes dits : pourquoi ne pas utiliser une base de données pour stocker les résultats de nos tests? Ce choix s'est révélé être le bon. La figure 2.3 montre le schéma objet de la base dans laquelle nous stockions les résultats. Un objet de la classe Stat était créé à chaque fois qu'une expérience était effectuée. Une expérience consistait en (1) une requête (attribut *requête*) sur (2) une base de données (attribut *base*) dont (3) la politique de placement des objets sur disque était fixée (attribut *cluster*). Cette requête était évaluée grâce à un certain algorithme (attribut *algo*) et en fixant certains paramètres sur la configuration du système (attribut *système*).

Une fois que les résultats étaient stockés dans la base d'expériences, il était beaucoup plus facile de les analyser. Le langage de requêtes permettait notamment d'extraire l'information recherchée. Egalement, en vue d'analyser les résultats grâce à un logiciel d'analyse de données (tel SICLA ou SPAD-N), il était désormais possible de transformer le résultat de nos requêtes sur les données expérimentales en fichiers compréhensibles par ce type de logiciel. Ceci représente un gain de temps non négligeable.

2.4 Quoi mesurer?

Les tests que nous avons réalisés consistaient à évaluer une requête dans des conditions fixées : placement physique des données interrogées, taille de celles-ci, algorithme, sélectivité des prédicats.

Quelles étaient les données que nous pouvions collecter à chaque expérience, comment les obtenir et, surtout, quelles étaient celles qu'il était utile d'analyser et d'exploiter?

Nous voulions comparer les performances de plusieurs algorithmes, expliquer pourquoi l'un était plus lent que l'autre. Vue l'architecture client-serveur du système O_2, il nous semblait utile de collecter différentes données :

1. le nombre d'E/S du disque vers le cache serveur
2. le nombre de pages qui sont lues du cache serveur vers le cache client
3. le nombre de RPCs entre le cache serveur et le cache client ainsi que la taille totale, en Mégaoctets, de ces RPCs
4. le taux d'échecs dans le cache client, càd. : sur le nombre de fois que le client a voulu accéder à une page dans son cache, le nombre de fois qu'il a dû relire la page en question en provenance du cache serveur
5. le taux d'échecs dans le cache serveur, càd : sur le nombre de fois que le serveur a voulu accéder à une page dans son cache, le nombre de fois qu'il a dû relire la page en question en provenance du disque

mais aussi, bien sûr, de distinguer temps processeur et temps écoulé.

Finalement, après de nombreux tests, nous avons compris que le temps écoulé était finalement bien représentatif d'une expérience. Dans la plupart des cas, il évoluait de la même manière que le nombre de RPCs et d'E/S. Quand ce n'était pas le cas, il y avait toujours une bonne raison, par exemple, un hachage sur une très grosse table provoquant beaucoup de swapping.

class Stat
    numtest: integer,
    requête: Requête,
    base: {(Extension)},
    cluster: string,
    algo: string,
    système: Système,
    DéfautsDePagesCC: integer, /*nombre de défauts de pages dans le cache client*/
    TempsEcoulé: real,
    /*Temps écoulé entre le début et la fin de la requête*/
    nombreRPCs: integer,
    /*nombre de RPCs entre le cache client et le cache serveur*/
    tailleTotaleRPCs: integer,
    /*taille totale (en Mo) des messages entre le cache client et le cache serveur*/
    pagesLuesD2CS: integer,
    /*nombre de pages qui sont lues du disque vers le cache serveur*/
    pagesLuesCS2CC: integer,
    /*nombre de pages qui sont lues du cache serveur vers le cache client*/
    tauxEchecsCC: integer,
    /*sur le nombre de fois que le client a demandé à lire une page dans son cache, nombre de fois où la page n'y était pas et où le client a dû recharger la page*/
    tauxEchecsCS: integer
    /*sur le nombre de fois que le serveur a demandé à lire une page dans son cache, nombre de fois où la page n'y était pas et où le serveur a dû recharger la page*/

class Requête:
    aFroid: boolean, /*la requête est-elle évaluée après un arrêt du serveur?*/
    typeProjection: string, /*type de la projection*/
    sélectivité: integer, /*sélectivité sur les extensions interrogées*/
    texte: string /*le texte de la requête*/

class Extension
    nomClasse: string, /*l'extension est sur cette classe*/
    taille: integer, /*cardinalité de l'extension*/
    associations: { [extension: Extension,
        cardinalitéAssociation: integer] }
    /*toutes les associations entre l'extension et les autres extensions dans la base*/

class Système
    tailleCacheServeur: integer, /*la taille du cache serveur*/
    tailleCacheClient: integer, /*la taille du cache client*/
    mêmeStationDeTravail: boolean
    /*le client et le serveur tournent-ils sur la même machine?*/

FIG. 2.1 – *Un schéma pour stocker les résultats des tests*

2.5 Quelle méthodologie utiliser pour effectuer les tests?

Réaliser des expériences peut vite devenir laborieux et rébarbatif. C'est pourquoi un minimum de rigueur et de méthode est requis. Encore faut-il savoir quelle méthodologie suivre ...

En effet, après avoir réfléchi aux tests que nous voulions réaliser, nous étions partis dans l'idée qu'il fallait mieux réaliser toute une batterie de tests et, ensuite seulement, les analyser.

Nous voilà partis... Après un peu moins de deux mois de travail, nous nous sommes retrouvés avec une quantité assez importante de résultats et nous peinions à les analyser. Par quoi fallait-il commencer? Quels graphiques tracer? Que voulions-nous montrer? Que pouvions-nous conclure?

Finalement, nous n'avons rien pu tirer de ces premiers tests. En effet, nous pensons vraiment que réaliser beaucoup de tests sans les analyser au fur et à mesure est une erreur. Beaucoup de temps est gaspillé pour analyser des expériences sans interêt. Nous sommes convaincus que, par contre, la méthodologie suivante est plus efficace. Faire peu de tests à la fois; les analyser au fur et à mesure et, surtout, avant de réaliser un ensemble de tests, se demander ce qu'ils vont permettre de montrer ou , plus précisément, réfléchir à la question à laquelle il pourrait apporter des éléments de réponse.

Chapitre 3

Sur les accès associatifs dans O_2

Deux expériences que nous avons réalisées nous ont permis de comprendre vraiment comment les objets étaient gérés par O_2 en mémoire centrale. Nous expliquons ici en quoi elles consistent et les enseignements fort intéressants que nous en avons tirés. En conclusion, nous expliquons comment le système O_2 pourrait mieux gérer les objets en mémoire centrale.

Ces résultats peuvent sembler un peu dénués d'interêt puisque le système O_2 n'est plus vendu actuellement. Cependant, les observations que nous avons pu faire sur O_2 sont tout à fait celles que nous aurions pu faire en étudiant un autre système. C'est en cela qu'elles sont intéressantes.

3.1 Le mécanisme d'adressage dans O_2

Dans le système O_2, une donnée est représentée, en interne, de deux façons : soit par des Rids (records identifiers ou identificateurs persistants d'objets), soit par des Handles (en français, poignées).

3.1.1 Les identificateurs

Les identificateurs sont des identificateurs physiques. Autrement dit, l'identificateur associé à un objet ou à une valeur est l'adresse physique de l'enregistrement correspondant sur disque. Comme nous l'avons vu précédemment dans le chapitre décrivant le système O_2, un enregistrement se situe toujours dans une page d'un certain volume. Par conséquent, un Rid est une structure de huit octets composée [Tec] :

- d'un identificateur de volume sur deux octets
- d'un identificateur de page au sein de ce volume sur quatre octets
- d'un nombre, appelé *slot number*, sur deux octets, qui, de façon indirecte identifie un enregistrement dans une page.

Rappelez-vous, ce sont les @p_i dans les figures 1.1, 1.2, 1.3.

Pourquoi avoir choisi des identificateurs physiques? Pour des raisons d'efficacité. En effet, si les identificateurs étaient logiques, un tableau, persistant et devant être accessible simultanément par plusieurs utilisateurs, gérerait les correspondances entre adresses physiques et identificateurs logiques. Ainsi, pour accéder à un enregistrement, il faudrait toujours interroger cette table qui serait, potentiellement, très grande. En effet, il y aurait une entrée pour chaque objet de la base.

3.1.2 Les Handles

Une Handle, quant à elle, sert à gérer un enregistrement en mémoire centrale. C'est en fait un pointeur vers une structure de données, appelée *dope vector*, qui représente un objet en mémoire centrale.

Ces pointeurs ont été inventés pour gagner de la place lors de l'exécution d'applications. En effet, quand plusieurs Handles sur le même objet existent, la structure, appelée *dope vector*, est partagée par tous ces

```
Obtenir les Rids des patients tq pat_id < k1 grace a un index placant

Pour tous ces Rids Faire  {

    Obtenir un Handle et lire l'objet correspondant

    Hacher l'objet par la valeur de son attribut medecin_traitant
      /*la table de hachage contient des entrees de la forme (medecin,{patient, ...})*/}

Obtenir les Rids des medecins tq med_id < k2 grace a un index placant

Pour tous ces Rids Faire  {

    Sonder la table de hachage pour y obtenir les patients correspondants

   Ajouter chaque couple [docteur,patient] a l'ensemble resultat  }
```

FIG. 3.1 – *Principe de l'algorithme de hachage testé*

pointeurs. Petite précision : pour gérer proprement la mémoire, le système comptabilise, pour chaque *dope vector*, le nombre de pointeurs qui le référencent. Quand ce compteur est égal à 0, cette structure est désallouée.

3.2 Hacher des objets

Nous le verrons dans le chapitre suivant, les algorithmes de jointure que nous avons comparés utilisent des tables de hachage. Par conséquent, une des premières questions que nous nous sommes posée est la suivante : quelle forme adopter pour stocker un objet dans une table de hachage? une Handle ou un Rid? Pour répondre à cette question, nous avons comparé deux versions d'un même algorithme de jointure par hachage. Son principe est décrit dans la figure 3.1. Il joint les patients dont l'identificateur pat_id est inférieur à k_1 avec les médecins dont l'identificateur med_id est inférieur à k_2 et retourne un ensemble de couples [médecin,patient]. Il est inspiré de l'algorithme de jointure *pointer-based algorithm* décrit dans [SC90].

Pourquoi avoir choisi cet algorithme? Nous le verrons après, ce n'est pas l'algorithme optimal, mais il illustre bien ce qu'implique, en termes de performances, l'utilisation de Rid ou Handle pour représenter des objets hachés. En effet, comme il nécessite de hacher la plus grande collection, càd. la collection Patients, il utilise la plus grande table de hachage.

La figure 3.2 relate les résultats de l'expérience que nous avons réalisée sur une base contenant 2000 médecins et 1000 clients par médecin. Ces objets sont stockés de façon aléatoire sur disque. **HJHD** (resp. **HJRID**) représente l'algorithme dans lequel des Handles (resp. Rids) sont utilisés pour représenter les patients dans la table de hachage. Les sélectivités sont exprimées en pourcentages[1]. Les temps sont exprimés en secondes : c'est le temps chronomètre nécessaire pour évaluer chaque algorithme. Pour un algorithme donné, à une sélectivité donnée, la valeur indiquée dans la colonne *ratio de temps* représente le rapport entre

1. Dans la suite, nous dirons que la sélectivité diminue si elle passe de 10% à 90%. Autrement dit, une forte sélectivité signifie que le poucentage est petit.

Sélectivité sur les patients	Sélectivité sur les médecins	Algorithme	Ratio de temps	Temps (en sec)
10	10	HJRID	1	206.07
		HJHD	1.10	226.85
10	90	HJHD	1	351.28
		HJRID	1.03	364.57
90	10	HJRID	1	1964.28
		HJHD	2.82	5501.74
90	90	HJRID	1	16233.79
		HJHD	2.11	34311.17

FIG. 3.2 – *Jointure par hachage: Rids ou Handles?*

le temps de celui-ci et le temps du meilleur algorithme dans cette situation. Prenons les deux premières lignes du tableau. Elles indiquent que, quand la sélectivité sur la collection Patients et la sélectivité sur la collection Médecins sont égales à 10%, **HJRID** est l'algorithme le plus rapide et **HJHD** est 1.1 fois plus lent. La jointure retourne une collection possédant entre 20000 et 162000 éléments en fonction des sélectivités.

Egalement, remarquons que 1.8 millions d'objets de la classe Patient sont hachés quand la sélectivité sur la collection Patients est égale à 90%. La performance des deux versions de l'algorithme devient alors désastreuse. En effet, nous n'utilisons pas d'algorithme par hachage GRACE ou HYBRID [WKO+84] dont nous avons expliqué les principes précédemment.

Que nous enseigne cette expérience? Comme nous pouvons le voir, stocker des objets dans une table de hachage sous forme de Rid améliore les performances par un facteur de 2 à 3 quand la collection hachée est grande (90% des patients sont sélectionnés). Pourquoi? Nous pouvons, raisonnablement, supposer qu'un Rid est plus petit en taille qu'une Handle et, donc, à des sélectivités fixées, la table de hachage utilisée par **HJHD** est plus grande que celle utilisée par **HJRID**. Ainsi, plus le nombre d'entrées dans la table de hachage est grand, plus il y a du swapping et plus l'algorithme devient lent.

3.3 La mystérieuse performance des index

Après avoir réalisé un certain nombre de tests sur les algorithmes de jointure, nous nous sommes aperçus de la chose suivante. Lorsqu'aucun index n'était utilisé pour faire les sélections, les algorithmes étaient plus performants que lorsque des index étaient utilisés et ceci même quand la sélectivité était élevée, càd. égale à 10%. Nous avons donc décidé de comprendre pourquoi et nous avons réalisé que nous utilisions des index non plaçants.

Rappelons plus précisément ce que signifie un index plaçant. Indexer une collection d'objets sur un attribut revient à construire un arbre B [BJP98]. Dans cet arbre, seules les feuilles contiennent les rids des objets indexés. Les autres noeuds de l'arbre permettent, étant donnée une valeur, d'accéder très rapidement (chaque noeud traversé nécessite une E/S) aux feuilles qui contiennent les rids des objets recherchés . Un index est plaçant quand le placement physique des objets indexés correspond à l'ordre de la clé sur laquelle ils sont indexés. Plus concrètement, parcourir séquentiellement tous les Rids stockés dans une feuille d'un arbre B revient à accéder à une ou plusieurs pages placées sur disque de façon contigue. Autrement dit, chaque page concernée est lue une fois et une seule.

Afin de vérifier que la perte de performance s'expliquait par l'utilisation d'index non plaçants, nous avons testé la requête suivante:

```
select pa.age
from pa in Patients
where pa.num < k
```

sur une base contenant 2 millions de patients. Pour chaque patient, l'attribut num est un entier aléatoire compris entre l'entier 1 et 2000,000. Sur la base, il existe un index Patients.num non plaçant. Les résultats,

Sélectivité	**Temps (sec) avec index**	**Nb de pages lues avec index**	**Temps (sec) pas d'index**	**Nb de pages lues pas d'index**
0.1	27.24	1961	802.15	54618
0.5	101.84	8680	814.90	54612
1	147.26	16451	815.17	54615
5	947.72	77431	821.41	54610
10	1824.83	153709	1352.79	54608
30	5376.24	458383	1467.75	54610
60	10724.57	915233	1641.24	54613
90	16177.83	1372629	1908.24	54612

FIG. 3.3 – *Tester les index non plaçants*

Sélectivité sur les patients	**Index plaçant + Tri**	**Pas d'index**
10	343.49	1352.79
30	591.49	1467.75
60	1015.52	1641.24
90	1648.62	1908.24

FIG. 3.4 – *Sans index ou avec index non plaçant + tri?*

résumés dans le tableau de la figure 3.3, montrent bien qu'utiliser un index non plaçant pour effectuer une sélection n'est valable que pour de très fortes sélectivités. Ici, d'ailleurs, vu le grand nombre d'objets traités, pour une sélectivité de 5%, il n'est déjà plus valable d'utiliser un index non plaçant pour effectuer la sélection.

Expliquons pourquoi. D'abord, regardons le nombre d'E/S nécessaires pour effectuer la requête. On constate que ce nombre est toujours identique, quelque soit la sélectivité, quand aucun index n'est utilisé. Ceci est normal vu qu'évaluer la sélection de cette manière consiste à parcourir séquentiellement chaque page dans laquelle se trouve un ou plusieurs enregistrements correspondant à des objets de la classe Patient. Par contre, quand un index est utilisé, plus la sélectivité diminue, plus le nombre d'E/S augmente. Il devient supérieur au nombre d'E/S nécessaires pour effectuer la sélection sans index dès que la sélectivité est supérieure ou égale à 5%. En effet, effectuer la sélection grâce à un index non plaçant consiste :

1. à parcourir les feuilles de l'index possédant les Rids des objets satisfaisant le prédicat de sélection.

2. puis, pour chaque Rid obtenu, à lire la page contenant l'enregistrement correspondant afin d'évaluer la projection. Dans notre cas, puisque l'index est non plaçant, l'accès à ces pages se fait de façon aléatoire. Donc, une même page est susceptible d'être lue de nombreuses fois.

Ensuite, nous nous sommes demandés : que se passe t'il si on effectue toujours la sélection grâce à un index non plaçant mais si, cette fois, on trie les Rids sélectionnés dans l'index avant de lire chacune des pages associées à ces Rids? La figure 3.4 montre les résultats de ce nouveau test. Grande surprise! Quelque soit la sélectivité, effectuer la sélection grâce à un index non plaçant associé à un tri est plus rapide qu'effectuer la sélection sans index. Nous étions alors très étonnés de ce résultat surtout quand la sélectivité vaut 90%. En effet, dans ce cas, il faut plus d'E/S pour effectuer la sélection avec l'index associé au tri que pour effectuer la sélection sans index. Pourquoi? Avec l'index, il faut d'abord parcourir les pages feuilles de l'index qui contiennent les Rids des objets satisfaisant le prédicat de sélection. Avec une sélectivité de 90%, ce sont presque toutes les pages feuilles de l'index qui sont parcourues. Ensuite, vraisemblablement, ce sont toutes les pages de la collection Patients qui sont accédées puisque ce sont presque tous les patients qui sont sélectionnés et que le placement physique des patients sur disque est aléatoire. De plus, nous effectuons un tri de 1.8 millions de Rids.

Nous avons, par conséquent, vérifié à nouveau les résultats concernant la sélection sans index. Regardons la première ligne de la figure 3.3. Quand la sélectivité vaut 0.1%, comme le temps pour lire une page du disque vers la mémoire centrale est 10 millisecondes et que le nombre de pages lues est 54618, 546 secondes

ouvrir le parcours sur Patients
pour chaque Rid r **retourné par le parcours**
    obtenir la Handle h
    si obtenir_att(h**, num) > k**
        ajouter obtenir_att(h**, age) au résultat**
    déréférencer h
	ouvrir le parcours d'index sur (Patients, num > k)
	pour chaque Rid r **retourné par le parcours d'index**
	    ajouter r **à la Table** T
	trier T **sur les Rids**
	pour chaque r **dans** T
	    obtenir la Handle h
	    ajouter obtenir_att(h**, age) au résultat**
	    déréférencer h

FIG. 3.5 – *Sélection standard et sélection avec index et tri*

	Parcours standard	**Parcours d'index avec tri**
Entrées/Sorties		+ Lire les pages de l'index
CPU	+ obtenir&déréférencer 200,000 Handles	+ trier 1.8M Rids (8 octets chacun)
CPU	+ comparer 2M d'entiers	

FIG. 3.6 – *Sélection standard et sélection avec index et tri : comparaison en termes de coût*

sont dédiées à ces lectures. Ainsi, puisque la durée de la sélection dans ce cas est 802 secondes, 256 secondes semblent nécessaires pour construire la collection résultat. Cela semble vraiment beaucoup trop vu qu'elle ne contient que 2000 entiers.

3.4 Les Handles ne sont pas optimaux

Pour trouver une explication à cela, nous avons donc eu besoin de regarder, à un plus bas niveau, comment la sélection était effectuée. La figure 3.5 montre le code pour exécuter (case de gauche) la sélection sur les patients sans index et (case de droite) la sélection sur les patients avec index non plaçant associé à un tri. h étant une Handle, l'instruction "déréférencer h" consiste à libérer la structure pointée par celle-ci[2] si l'objet référencé par h n'est pas utilisé par ailleurs.

Ensuite, la figure 3.6 analyse les différences entre ces deux algorithmes quand la sélectivité vaut 90%. Nous pouvons remarquer que la sélection sans index nécessite de créer et de déréférencer autant de Handles qu'il y a d'objets dans la collection Patients tandis que la sélection avec index et tri nécessite seulement de créer et de déréférencer autant de Handles qu'il y a d'objets sélectionnés.

Cette comparaison, ainsi que les résultats précédents, montrent que l'utilisation des Handles est chère en termes de performances des accès associatifs.

3.5 Comment mieux gérer les objets en mémoire centrale

Le SGBDO O_2 [Ard] propose, en effet, des fonctionnalités que d'autres systèmes de bases de données à objets ne proposent pas. Notamment, il propose: (1) le modèle ODMG [Cat97] dans son intégralité, ce qui inclut des valeurs complexes arbitraires et la persistance par attachement, (2) des index sur des collections arbitraires, càd. pas seulement sur des extensions, (3) la gestion de versions d'objets et (4) l'évolution de classe dynamique. Tout cela ainsi que le fait que les objets peuvent être partagés implique que, quand un objet est manipulé en mémoire centrale, la structure des données nécessaire pour le représenter contient non seulement la donnée encapsulée dans l'objet mais aussi des informations sur le système proprement dit.

Pour illustrer cela, prenons la fonctionnalité qui permet d'indexer des collections arbitraires. Prenons un exemple. Supposons que deux collections existent. La première, **LesPatients**, contient tous les objets de la classe Patient. La seconde, **LesPatientsParisiens**, contient tous les objets de la classe Patient qui

2. En fait, ce n'est pas forcément effectué de suite

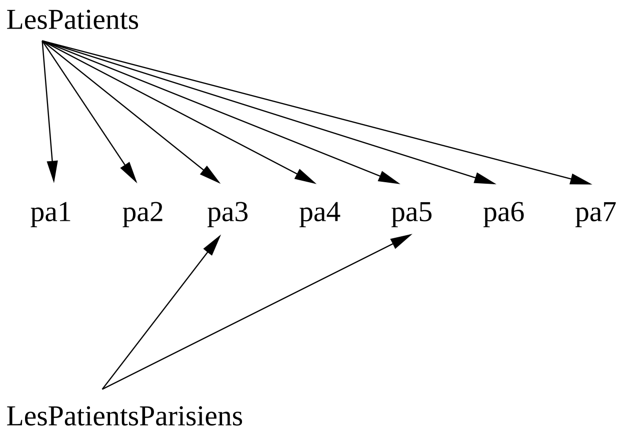


FIG. 3.7 – *Les collections LesPatients et LesPatientsParisiens*

habitent Paris. Supposons, de plus, qu'il existe deux index : l'un, **LesPatients.medecin_traitant**, l'autre, **LesPatientsParisiens.medecin_traitant**. La figure 3.7 montre ces deux collections.

Supposons maintenant qu'un médecin part à la retraite et que ces patients étaient les objets pa_1, pa_3 et pa_6. Il faut donc mettre à jour les deux index. Comment le savoir? Une première solution consisterait à parcourir tous les index qui indexent des patients et, pour chacun d'eux, vérifier si l'un des objets pa_1, pa_3 ou pa_6 y est indexé. Cette solution est loin d'être optimale. Dans le système O_2, plutôt, la structure que nous appelons *dope vector* contient, entre autres, l'ensemble des index auxquels il appartient. De cette façon, le système connaît immédiatement les index qu'il faut mettre à jour. Les autres informations sont les versions, le type exact (à cause de l'héritage), la persistance (à cause de la persistance par attachement), etc. Certaines de ces informations sont stockées sur disque, d'autres sont nécessaires seulement quand l'objet est en mémoire centrale.

Maintenant, dressons une liste non exhaustive de toutes les informations dans un *dope vector* en mémoire. Notons que certaines de ces informations sont communes à d'autres systèmes à objets.

1. Quelques bits indiquent si l'objet est indexé, persistant, en mémoire, détruit, etc ...

2. Un pointeur réfère une structure possédant des informations sur le type exact de l'objet. Cette structure est partagée par tous les objets de ce type.

3. Un pointeur réfère la liste des index auquel l'objet appartient. Cela est nécessaire, comme nous l'avons expliqué plus haut, puisqu'il se peut qu'un objet soit indexé via plusieurs collections et modifié via plusieurs chemins.

4. Un compteur contient le nombre de pointeurs vers l'objet en mémoire. Quand ce nombre est égal à 0, l'espace mémoire utilisé par la structure de données peut être réutilisée

5. Un pointeur vers une structure qui représente la version à laquelle l'objet appartient.

6. Des informations dressent l'historique de la modification de schéma relatif à la classe auquelle l'objet appartient.

Toutes ces informations additionnées, la taille de cette structure est de 60 octets. Ces octets doivent être alloués, modifiés et libérés à chaque fois qu'il est nécessaire de le faire. La liste ci-dessus n'est pas exhaustive et n'a cessé de grossir au fil des ans. Aucune étude n'a analysé son impact sur la performance. La raison en est simple. De gros *dope vectors* sont mauvais, essentiellement, pour des accès associatifs à froid. Or, les benchmarks objets, où O_2 se situe parmi les meilleurs, analysent avec attention la performance des applications qui requièrent de naviguer de façon aléatoire au travers d'objets résidant en mémoire. En effet, à notre connaissance, il n'existe pas de benchmark objet où sont étudiées les performances du processeur de requêtes. Par exemple, le banc d'essais 007 [CWN93] a pour but de comparer les performances des systèmes orientés objets non différentes stratégies d'évaluation de différentes requêtes. Notamment, il considère la navigation au travers de structures hiérarchiques mais n'étudie pas différents algorithmes de jointure pour effectuer cette navigation. Par conséquent, dans O_2, l'allocation des Handles a été optimisée pour favoriser la navigation en mémoire centrale. Notamment, un *dope vector* est suffisamment grand pour que le système puisse modifier l'objet rapidement sans avoir à le fixer en mémoire centrale. De plus, la destruction des Handles est retardée le plus possible pour éviter des allocations et des libérations de mémoire non nécessaires.

Peut-on et, surtout, comment améliorer cette situation? Nous proposons ci-après plusieurs solutions.

1. Nous pensons que les informations à stocker dans chaque *dope vector* devraient être déterminées en fonction de l'application. Ainsi, un banc d'essais pertinent pourrait nous enseigner les informations qui pourraient y être supprimées sans pour autant nuire aux performances des applications standard.

2. Les *dope vectors* ne représentent pas seulement des objets mais aussi des valeurs complexes ou, encore, des chaînes de caractères. La plupart des informations qui y sont stockées sont inutiles pour manipuler les littéraux (pas de partage, pas de version, etc). Une première tâche consisterait donc à manipuler les littéraux avec des structures plus petites. Par exemple, une hiérarchie de classes pourrait être créée afin de représenter les *dope vectors*. De plus, certaines informations qui y sont stockées concernent seulement quelques objets en particulier mais, cependant, sont données à tous. Cette hiérarchie permettrait également de mieux manipuler les objets.

3. Une connexion plus étroite entre l'optimiseur de requêtes et la couche système de O_2 permettrait de gérer les *dope vectors* plus efficacement. Plutôt que d'allouer, modifier, libérer ces structures objet par objet, il serait possible de le faire pour des paquets d'objets.

Chapitre 4

Sur la jointure et la navigation dans des structures hiérarchiques

Dans ce chapitre, nous étudions une requête OQL typique dans laquelle on parcourt une structure hiérarchique (autrement dit, une structure d'arbre). L'objectif de notre travail a été de comparer la performance de différents algorithmes pour évaluer celle-ci dans différentes situations (taille de la base, organisation physique des données dans la base, sélectivité des prédicats de sélection, type de la projection). Notre but a été de réaliser ces tests sur des bases ressemblant le plus possible à des bases réelles et de comprendre quels facteurs influencaient la performance de tel algorithme.

La requête que nous avons testée est définie sur le schéma de la figure 1.4. C'est la suivante :

```
select  f(m,pa)
from    m in Medecins, pa in m.clients
where   pa.pat_id < k1 and m.med_id < k2
```

où **f(m,pa)** est une sous-requête ayant comme argument un objet de la classe Medecin et un objet de la classe Patient.

Avant de présenter les algorithmes dont nous avons étudié la performance relative, quelques remarques sont nécessaires :

1. Souvenons-nous, nous l'avons dit précédemment, toutes les requêtes sont exécutées à froid, càd. que les caches serveur et client sont vides au début de l'exécution de la requête.

2. Nous stockons toujours, dans les tables de hachage, les éléments nécessaires pour construire **f(m,pa)**, càd., soit des objets, soit des entiers, soit des chaînes de caractères.

3. A chaque fois que cela est possible, nous utilisons un index pour évaluer chacune des sélections : d'une part, sur pa.pat_id, d'autre part, sur m.med_id. Ces deux index sont des index plaçants et stockent seulement des identificateurs d'objets dans leurs feuilles (càd. pas de propriétés d'objets).

4. Dans nos tests, **f(m,pa)** correspond à la construction d'un tuple avec deux attributs. Nous avons effectué des tests avec **f(m,pa)** = [**m.nom,pa.age**]. Pourquoi avoir choisi de projeter des propriétés d'objets plutôt que des objets? En fait, la sous-requête **f(m,pa)** influence beaucoup la durée d'exécution des différents algorithmes. Supposons, en effet, que nous projetons des objets. Quand un algorithme par hachage est utilisé, les pages contenant les objets de la collection non hachée ne sont pas lues. Par contre, quand un algorithme de navigation est utilisé, chacune des pages de chacune des collections jointes est parcourue. C'est donc pour que toutes les pages contenant les objets sélectionnés soient lues au moins une fois que nous avons choisi cette projection.

4.1 Les algorithmes

Nous comparons quatre algorithmes, deux sont fondés sur la navigation et deux sur le hachage. Nous avions commencé à tester des algorithmes à base de tri mais ils se sont avérés plus lents que les algorithmes à base de hachage. Nous avons donc décidé de les éliminer de cette première étude. Les algorithmes sont les suivants :

4.1.1 Naviguer des parents vers les enfants (NL)

L'algorithme est :

Pour tous les médecins m tq. m.med_id < k2 Faire /* parcours d'index */
    Pour tous les patients pa de m Faire /* navigation */
        Si pa.pat_id < k1 Alors
            Ajouter f(m,pa) au résultat

Remarquons que seul un index peut être utilisé pour évaluer cet algorithme puisque les patients sont accédés via leurs médecins. C'est un grand handicap pour cet algorithme puisque la collection des patients est la plus grande des deux (jusqu'à 1000 fois plus grande). De plus, autre remarque, alors que l'accès aux médecins se fait de façon séquentielle, l'accès aux patients se fait (i) de façon aléatoire quand la base interrogée possède un cluster par classe ou quand les objets sont placés de façon aléatoire (pas de cluster) et (ii) de façon séquentielle quand la base interrogée possède un cluster de regroupement. Il apparaît donc clair que (i) constitue un autre grand handicap (dès lors que c'est la plus grande collection qui est accédée de façon aléatoire) tandis que (ii) est un réel plus pour l'algorithme.

4.1.2 Naviguer des enfants vers les parents (NOJOIN)

L'algorithme est :

Pour tous les patients pa tq. pa.pat_id < k1 Faire { /* parcours d'index */
    Obtenir la valeur de l'attribut médecin_traitant de pa, noté m /* navigation */
    Si m.med_id < k2 Alors
        Ajouter f(m,pa) au résultat }

Ici encore, comme précédemment, seulement un index peut être utilisé. Cependant, cette fois, c'est l'index sur la plus grande collection et, donc, l'handicap est moins important. Cependant, remarquons que, puisqu'un médecin est associé à 3 (resp. 1000) patients, la condition sur m.med_id est vérifiée 3 (resp. 1000) fois. Maintenant, concernant la manière avec laquelle les collections sont parcourues, la collection des patients (la plus grande collection) est toujours parcourue de façon séquentielle. En ce qui concerne les médecins, il faut distinguer 3 situations. (i) Si le placement est aléatoire ou s'il existe un cluster par classe, l'accès aux médecins se fait de façon aléatoire. (ii) Quand chaque médecin est regroupé avec ses 3 clients, l'accès aux médecins se fait de façon séquentielle. (iii) Quand chaque médecin est regroupé avec ses 1000 clients, la page contenant l'enregistrement relatif à un médecin est différente de celle contenant son millième patient (il y a environ 50 patients par page). Cependant, il y a de grandes chances pour que la page contenant le médecin reste en mémoire le temps du parcours des pages contenant chacun de ses patients. En effet, tous les patients sont accédés en une fois.

Cet algorithme s'appelle NOJOIN car la jointure est cachée derrière la navigation.

4.1.3 Hacher les parents et effectuer la jointure (PHJ)

L'algorithme est :

Hacher tous les médecins m tq. m.med_id < k2 sur leurs identificateurs /* parcours d'index */
/* Une entrée, dans la table de hachage, est un couple (r,i) où
    1) r est le rid d'un médecin sélectionné
    2) i est l'information concernant ce médecin qui est utilisée pour construire f(m,pa).

Algorithme	**Nombre de médecins**	**Nombre d'enfants**	**Sélectivité sur les patients**	**Sélectivité sur les médecins**	**Taille table de hachage (Mo)**
PHJ	2000	1:1000		10	0.0128
PHJ	2000	1:1000		90	0.1152
PHJ	1000000	1:3		10	6.4
PHJ	1000000	1:3		90	57.6
CHJ	2000	1:1000	10		1.72
CHJ	2000	1:1000	90		14.52
CHJ	1000000	1:3	10		62.4
CHJ	1000000	1:3	90		81.6

FIG. 4.1 – *Approximation de la taille de la table de hachage*

Dans notre cas, c'est la valeur de son attribut nom */

Pour tous les patients pa tq. pa.pat_id < k1 Faire { /* parcours d'index */
Obtenir l'information sur son médecin traitant en sondant la table de hachage
Ajouter f(m,pa) au résultat }

Quelque soit la politique de placement des objets dans la base interrogée, cet algorithme utilise les index définis sur chacune des collections et accède aux objets de chaque collection de façon séquentielle. Cependant, nous pouvons noter que, si la politique de placement est un cluster de composition, accéder à tous les patients amène à accéder à tous les médecins et vice versa.

Naturellement, la taille de la table de hachage dépend de la sélectivité sur les patients. La figure 4.1 donne une approximation de la taille de la table de hachage pour cet algorithme et le suivant. Rappelons que la MC dont nous disposons est de 128 Mo dont 36 Mo sont utilisés pour les caches de O_2. Quelques autres Mo (il est difficile d'en évaluer le nombre) sont utilisés par le gestionnaire de fenêtres. Etant donné cela, nous pouvons en déduire que du swapping va se produire dans le cas 1:3, quand 90% des médecins est sélectionné. En effet, dans ce cas, la table de hachage ne tient plus en MC. Nous n'avons pas considéré d'algorithmes à base de hachage HYBRID ([WKO+84]) pour optimiser cela. Enfin, remarquons que, en termes d'instructions, cet algorithme est plus coûteux que les deux précédents.

4.1.4 Hacher les enfants et effectuer la jointure (CHJ)

L'algorithme est :

Hacher tous les patients pa tq. pa.pat_id < k1 sur leur attribut médecin_traitant /* parcours d'index */
/* Une entrée de la table de hachage est un couple (r,li) où
1) r est le rid d'un médecin
2) li est une liste d'information concernant chacun des patients sélectionnés
de ce médecin. Dans notre cas, cette liste est la liste des valeurs de
l'attribut age de chacun des patients sélectionnés. Chacune de ces informations
sert donc à construire chacun des f(m,pa) appartenant au résultat. */

Pour tous les médecins m tq. m.med_id < k2 Faire { /* parcours d'index */
Obtenir l'information sur ses patients préalablement sélectionnés
en sondant la table de hachage
Pour chacun des patients pa obtenu en sondant la table de hachage Faire
Ajouter f(p,pa) au résultat }

Cet algorithme est une légère variation de l'algorithme Pointer-based join de [SC90].

Il a les mêmes caractéristiques que le précédent concernant l'utilisation des index et la séquentialité des accès. La différence vient de la taille de la table de hachage qui contient des enfants plutôt que des parents

Sélectivité sur les patients	**Sélectivité sur les médecins**	**Algorithme**	**Ratio de temps**	**Temps (sec)**
10	10	PHJ	1	89.83
		CHJ	1.12	101.05
		NOJOIN	1.40	125.90
		NL	15.79	1418.56
10	90	CHJ	1	154.09
		PHJ	1	154.57
		NOJOIN	1.24	191.51
		NL	80.03	12331.96
90	10	PHJ	1	925.07
		NOJOIN	1.36	1266.31
		CHJ	1.42	1320.69
		NL	1.63	1509.19
90	90	PHJ	1	1913.80
		CHJ	1.02	1956.35
		NOJOIN	1.20	2315.62
		NL	7.01	13423.38

FIG. 4.2 – *Un fichier par classe,* 2×10^3 *Médecins,* 2×10^6 *Patients*

(càd. potentiellement, 3 à 1000 fois plus d'éléments). Par conséquent, nous pouvons voir que cette table est trop grande dans le cas 1:3 quelque soit la sélectivité sur les patients.

4.2 Résultats avec les clusters par classe

La figure 4.2 (resp. 4.3) montre les résultats que nous avons obtenus avec une association 1:1000 (resp. 1:3) sur une base où sont stockés, dans un même fichier, tous les objets d'une même classe.

Regardons d'abord l'association 1:1000. Ce n'est pas surprenant, les algorithmes de jointure par hachage sont les plus performants et *NL* est très lent sauf quand très peu de médecins sont sélectionnés. Ce qui est plus surprenant, c'est que *NOJOIN* est comparable aux algorithmes de hachage. Pourquoi? Dans cet algorithme, c'est la plus petite collection(càd. les médecins) qui est accédée de façon aléatoire. Or, quand l'association est 1:1000, cette collection ne possède que 2000 éléments. Par conséquent, cela n'a pas d'incidence sur les performances.

Les choses sont évidemment différentes quand l'association est 1:3 puisqu'il y a 1 million de médecins (et 3 millions de patients). *NOJOIN* devient très mauvais et comparable aux algorithmes de jointure par hachage, seulement quand ceux-ci réclament trop de place mémoire. Nous remarquons aussi que la performance de *CHJ* et de *PHJ* dépend naturellement de la taille de la table de hachage nécessaire à leur exécution.

Bien que nous ne montrons pas ici les résultats des tests que nous avons effectués sur des bases où le placement des objets est aléatoire, nous pouvons ajouter que ce placement multiplie le temps d'évaluation des algorithmes par un facteur de 1.5 à 2 et favorise les mêmes algorithmes (voir figure 4.6).

4.3 Résultats avec les clusters de composition

La figure 4.4 (resp. 4.5) montre les résultats que nous avons obtenus avec une association 1:1000 (resp. 1:3) sur une base avec cluster de composition.

Cette fois, nous pouvons voir que naviguer est beaucoup plus rapide que hacher. De plus, si nous comparons ces résultats avec ceux de la section précédente, nous constatons que la politique de placement par cluster de composition améliore nettement les performances sauf dans un cas : quand beaucoup de médecins et très peu de patients sont sélectionnés. Ce n'est pas une surprise puisqu'utiliser la navigation dans ces

Sélectivité sur les patients	**Sélectivité sur les médecins**	**Algorithme**	**Ratio de temps**	**Temps (sec)**
10	10	PHJ	1	365.72
		CHJ	1.10	402.38
		NOJOIN	9.70	3550.62
		NL	12.48	4566.06
10	90	CHJ	1	1286.18
		NOJOIN	2.93	3777.10
		PHJ	4.44	5723.28
		NL	31.96	41119.29
90	10	PHJ	1	2676.37
		NL	1.77	4738.09
		CHJ	3.53	9457.91
		NOJOIN	11.70	31318.05
90	90	NOJOIN	1	34708.13
		NL	1.26	43850.03
		PHJ	1.27	44188.33
		CHJ	1.69	58963.71

FIG. 4.3 – *Un fichier par classe,* 10^6 *Médecins,* 3×10^6 *Patients*

Sélectivité sur les patients	**Sélectivité sur les médecins**	**Algorithme**	**Ratio de temps**	**Temps (sec)**
10	10	NL	1	92.78
		NOJOIN	10.36	961.88
		CHJ	10.47	971.84
		PHJ	10.56	980.42
10	90	NL	1	923.84
		PHJ	1.12	1042.16
		CHJ	1.16	1078.47
		NOJOIN	1.18	1090.98
90	10	NL	1	155.17
		PHJ	7.50	1164.97
		CHJ	7.87	1221.29
		NOJOIN	8.40	1303.90
90	90	NL	1	1665.51
		PHJ	1.14	1898.97
		CHJ	1.19	1993.88
		NOJOIN	1.20	2006.76

FIG. 4.4 – *Cluster de composition,* 2×10^3 *Médecins,* 2×10^6 *Patients*

Sélectivité sur les patients	Sélectivité sur les médecins	Algorithme	Ratio de temps	Temps (sec)
10	10	NL	1	165.97
		NOJOIN	8.82	1465.20
		PHJ	9.43	1566.68
		CHJ	9.84	1634.72
10	90	NOJOIN	1	1572.40
		NL	1.11	1749.50
		CHJ	2.02	3181.43
		PHJ	5.14	8090.45
90	10	NL	1	280.53
		PHJ	6.88	1932.78
		NOJOIN	7.08	1988.82
		CHJ	17.79	4993.11
90	90	NL	1	2709.16
		NOJOIN	1.22	3332.08
		PHJ	3.78	10251
		CHJ	3.97	10761.14

FIG. 4.5 – *Cluster de composition,* 10^6 *Médecins,* 3×10^6 *Patients*

conditions conduit à lire de nombreuses pages contenant des patients qui ne sont pas sélectionnés (surtout dans le cas 1:1000).

Pour des raisons similaires, il est très probable qu'il sera plus lent d'effectuer une sélection sur une base avec un cluster de composition. Cependant, remarquons aussi que ceci est dû principalement à la manière avec laquelle O_2 implante les clusters de composition. En effet, c'est la navigation aléatoire parmi quelques objets particuliers d'une application que cette implantation favorise et non pas les accès associatifs. Une autre possibilité, comme dans [CL90], serait de stocker les patients et les médecins dans deux fichiers séparés. De plus, leur placement dans chacun de ces fichiers serait guidé par l'association qui les relie : les premiers objets dans le fichier contenant les patients seraient les patients du premier médecin dans le fichier contenant les médecins. Avec cette politique de placement, les sélections simples et les algorithmes de hachage auraient le même comportement qu'avec un cluster par classe tandis que la performance des algorithmes *NOJOIN* et *NL* resteraient telles qu'elles sont ici.

4.4 Conclusion

En guise de conclusion, la figure 4.6 donne le meilleur algorithme ainsi que son temps d'évaluation dans les différents cas que nous avons étudiés.

Quelles conclusions pouvons-nous tirer? Evaluer la jointure avec un algorithme par boucles imbriquées sur une base avec cluster de composition est la solution qui, dans la majorité des cas, est la plus performante.

Certes, mais, le cluster par composition est-il facile à obtenir et surtout à garantir dans la réalité? Cela dépend des applications.

Si la base est peuplée de manière aléatoire, ce qui est très fréquent dans la réalité, le placement des enregistrements sur disque le sera aussi. Charge alors à l'administrateur de remettre de l'ordre dans les données : tâche bien délicate et surtout coûteuse ...

Par contre, il est certain que, dans certaines applications type gestion de clients et de leurs commandes, les données jointes sont créées dans un certain ordre. Un client sera créé avant ses commandes. Dans ce genre d'application, il est plus aisé d'avoir la maîtrise du placement des objets sur disque.

Nous pouvons également constater qu'une organisation physique avec cluster par classe est avantageuse, dans la majorité des cas, par rapport à un placement aléatoire. Cette organisation physique possède un avantage certain par rapport à la composition : elle est beaucoup plus facile à garantir.

Rel méd: pat	**Sél. pat.**	**Sél. med.**	**Meilleur Algo Org. Aleat.**	**Temps Pour Org. Aleat.**	**Meilleur Algo Cluster Par Classe**	**Temps Pour Cluster Par Classe**	**Meilleur Algo Cluster Comp.**	**Temps Pour Cluster Comp.**
1:1000	10	10	PHJ	158.67	PHJ	89.83	NL	92.78
	10	90	CHJ	279.88	CHJ	154.09	NL	923.84
	90	10	PHJ	1419.87	PHJ	925.07	NL	155.17
	90	90	CHJ	2617.10	PHJ	1913.80	NL	1665.51
1:3	10	10	PHJ	277.24	PHJ	365.72	NL	165.97
	10	90	CHJ	1884.61	CHJ	1286.18	NOJOIN	1572.40
	90	10	PHJ	2216.87	PHJ	2676.37	NL	280.53
	90	90	NL	41954.19	NOJOIN	34708.13	NL	2709.16

FIG. 4.6 – *Les meilleurs algorithmes*

Enfin, il serait certainement intéressant d'étudier ces algorithmes quand la relation n'est plus 1:n mais n:m. Les résultats seront peut-être totalement différents.

En annexe, nous proposons un autre algorithme pour effectuer la jointure que nous avons étudiée. Malheureusement, les circonstances ont fait que nous n'avons pas pu l'étudier en détail.

Petite remarque pour faire le lien avec ce qui suit. Pour stocker des documents, le cluster de composition est certainement une organisation physique plus naturelle. En effet, un document se représente naturellement comme un arbre et, à ce titre, peut être plus facilement stocké, sur disque, sous forme d'arbre.

Conclusion

Les enseignements que nous avons tirés de nos expériences sur le système O_2 sont intéressants car généraux. Ils mettent en valeur certains points forts et points faibles des systèmes à objets en général par rapport aux systèmes relationnels.

D'une part, nous avons expliqué que les performances des accès associatifs dans O_2 n'étaient pas optimales. Il faut bien voir, en effet, que l'atout, dans un système à objets, est de naviguer, au sein d'une base, d'un objet vers un ou plusieurs autres au moyen d'un langage impératif orienté-objet. Ce n'est plus, comme dans le relationnel, d'effectuer très rapidement de gros accès associatifs grâce à une requête. Ces systèmes ont donc été conçus pour favoriser la navigation au détriment des accès associatifs et des performances du langage de requêtes. La taille d'un objet en mémoire est très importante, trop pour permettre des accès associatifs performants. De plus, le système perd beaucoup trop de temps en allocations et désallocations. Pour améliorer la situation, nous avons proposé des solutions. Les représentations des objets en mémoire pourraient par exemple être allouées et désallouées par paquets. Dans O_2, les chaînes de caractères ainsi que les valeurs complexes non partageables sont représentées en mémoire comme des objets alors que de nombreuses informations, bien que pertinentes pour les objets, sont inutiles à stocker pour ce genre de données. Enfin, la représentation des données en mémoire devrait être plus souple et s'adapter aux besoins des applications qui les manipulent.

D'autre part, nous avons comparé les performances de la navigation par rapport aux jointures pour évaluer des requêtes sur des bases stockant des structures arborescentes. Nous avons étudié l'influence de plusieurs paramètres : taille de la base, cardinalité de l'association 1:n, sélectivité des prédicats de sélection, organisation physique des données. Une des principales conclusions que nous avons tirée est que la navigation simple est sûrement profitable aux algorithmes sophistiqués de jointure lorsque les données conservent leur forme d'arbre sur disque. Ici encore, nous tirons partie de la richesse du modèle de données dans un système à objets et du fait qu'un arbre au niveau logique peut tout à fait être stocké physiquement comme un arbre.

Dans la partie suivante, nous allons continuer à étudier les performances de requêtes sur des structures arborescentes dans un système où, cette fois, les données, des documents XML, donc, des arbres, sont vraiment toujours stockées sous forme d'arbre. Le contexte est différent : la taille des données est très importante et les utilisateurs très nombreux. Nous nous sommes intéressés à minimiser le nombre de lectures sur disque au moyen de structures de données : des index plein-texte.

Troisième partie

Requêtes sur structures hiérarchiques : étude d'un système natif XML

Introduction

Nous allons maintenant étudier comment indexer de gros volumes de documents XML afin de répondre rapidement à de nombreuses interrogations concurrentes sur leur structure. Une fois les documents indexés, nous allons également montrer comment nous évaluons une requête.

Ce travail[1] a été réalisé dans le cadre de l'action de recherche Xyleme menée au projet Verso et en collaboration avec de nombreux autres groupes de recherche. L'objectif de cette action est de concevoir et d'implanter un prototype d'entrepôt de données capable d'exploiter toutes les données XML qui peuvent être trouvées sur le Web. Ce standard connaît en effet de plus en plus de succès, ce qui laisse à penser qu'il remplacera peut-être HTML ou, en tous cas, prendra une place importante sur le Web. La priorité est de proposer des services de haut niveau qui sont difficiles ou impossibles à supporter avec les technologies utilisées actuellement sur le Web. En particulier, sont considérées, des requêtes plus complexes que les recherches par mot clé, une intégration des données en fonction de leur sémantique et une acquisition sophistiquée des pages sur le Web avec détection des modifications. Xyleme peut, grossièrement, être vu comme une énorme base de données où sont répliquées, potentiellement, toutes les données XML du Web.

Il existe plusieurs façons de stocker des données XML : le relationnel, l'objet ou le natif. Nous avons choisi le système natif Natix [KM00] développé à l'Université de Mannheim. Deux raisons nous ont conduits à faire ce choix. D'une part, comme les données sont stockées sur disque sous forme d'arbre, l'accès aux éléments gagne en rapidité. D'autre part, les données, une fois stockées, occupent moins de place sur disque que les données réelles.

Même si, comme précédemment, les requêtes portent sur des données au format d'arbre, le contexte est ici vraiment différent de l'optimisation de requêtes dans le contexte objet. Les données étant nombreuses, elles sont distribuées dans autant de disques que nécessaire. La structure des données n'est pas stricte comme elle peut l'être dans un SGBD. Seule indication? Les DTDs et, n'oublions pas, un document XML peut très bien exister sans que sa structure soit définie dans une DTD. Le nombre d'utilisateurs simultanés est très important. Ils sont exigeants et, en particulier, impatients.

Plutôt que d'étudier l'évaluation par accès associatif comme dans le système O_2, nous nous sommes intéressés à découvrir de nouvelles structures de données. Nous allons le voir, nous avons choisi d'indexer les données dans des index plein-texte.

Nous montrons ensuite comment, une fois les documents indexés, une requête est évaluée.

Cette partie comporte quatre chapitres. Le chapitre 1 explique le standard XML. Notamment, il montre que le format d'un document XML n'est rien d'autre qu'un arbre. Le chapitre 2 situe notre travail dans l'action de recherche Xyleme. Le chapitre 3 expose comment nous indexons des documents XML. Le chapitre 4 décrit comment nous évaluons une requête une fois des documents XML indexés.

1. Ce travail a fait l'objet d'une publication au *workshop* ACM-SIGIR 2000 [ACV+00]

Chapitre 1

Le standard XML

Comme son grand-frère SGML, abréviation de *Standard Generalized Markup Language*, dont il est issu, XML [Gro00b], abréviation de *eXtensible Markup Language*, permet de créer des documents qui ont une structure mais celle-ci est beaucoup moins stricte, rigide que celle imposée par les SGBDs traditionnels. En effet, ce sont des balises, définies par l'utilisateur lui-même, qui structurent un document en indiquant le début et la fin des éléments qui le composent. De plus, XML peut, avant tout, être considéré comme un format d'échange très efficace entre applications. C'est sa simplicité et ses bonnes propriétés qui séduisent la majorité des acteurs de l'information. La preuve! On le retrouve maintenant dans des applications Internet comme le commerce électronique mais également dans des applications plus traditionnelles comme, par exemple, les systèmes intégrés d'information. Le stockage des données XML est maintenant un sujet préoccupant et des SGBDs, qu'ils soient relationnels, objets ou relationnels-objets, proposent des solutions.

XML est le fruit d'une collaboration d'industriels et de chercheurs au sein du *World Wide Web Consortium* (W3C). Depuis 1998, il est un des standards proposés par ce consortium . L'histoire a, en fait, commencé en juin 1996 avec la mise en route d'un groupe de travail. Son objectif? Tirer partie des enseignements de SGML, norme ISO de 1986, pour établir un formalisme d'échange de données sur le Web qui soit facile à utiliser par le grand public et qui dépasse les limites imposées par HTML. Le SGML était alors très utilisé pour réaliser de gros systèmes documentaires dans des domaines comme l'industrie de l'armement, l'automobile, ... Cependant, trois défauts sautaient aux yeux. (1) Il était trop complexe pour séduire le grand public. (2) La structure d'une donnée SGML devait être obligatoirement définie. (3) C'est une norme fort complexe, HyTime (norme ISO de 1992), qui définissait les mécanismes de liens.

XML est donc né pour pallier à ces inconvénients. Il comporte de nombreux atouts. Il est simple à utiliser, assez intuitif. A la différence d'HTML, il sépare le contenu d'un document de sa mise en page. Les données deviennent donc indépendantes des logiciels qui permettent de les visualiser (par exemple, des navigateurs), donc, plus pérennes. Comme nous le disions plus haut, l'utilisateur peut, avec XML, fractionner un document à sa convenance, en utilisant les balises qu'il désire. Il n'est pas obligé, comme avec HTML, d'utiliser un ensemble restreint de balises, qui plus est, trop générales, sans vraiment de signification et, par conséquent, difficiles à manipuler. Les balises apparaîssant dans un document peuvent - ce n'est pas obligatoire - être définies dans une structure type appelée DTD (abréviation de *Document Type Definition*). A ce titre, on peut d'ailleurs présager que des DTDs seront bientôt définies par différentes communautés (presse, chimie, droit, etc) pour uniformiser le format des données échangées et les rendre plus compréhensibles. En outre, XML offre un mécanisme de liens qui correspond aux besoins des utilisateurs (XLink [Gro98b], XPointer [Gro00a]).

Dans ce chapitre, notre objectif n'est pas d'expliquer le langage XML dans son intégralité. Pour une description détaillée, nous invitons le lecteur à consulter [ABS99, Mic99]. Nous allons juste donner un exemple d'utilisation de XML. Pour montrer que ce langage, au même titre qu'un SGBD à objets, permet de représenter des données au format d'arbre, nous allons représenter grâce à lui, comme nous l'avons fait précédemment avec le modèle objet (cf le chapitre 1 de la partie II), des médecins et des patients. Chaque médecin soigne plusieurs patients. Chaque patient est soigné par un unique médecin.

1.1 Un exemple de document XML

Il y a deux catégories de documents XML : les documents bien formés et les documents valides. Un document est bien formé s'il obéit aux règles syntaxiques du langage. Cela implique qu'il sera traité correctement, par exemple, par des parseurs XML. Un document est valide s'il est bien formé et si les balises qu'il contient sont définies dans une structure type, appelée DTD (*Document Type Definition*). Pourquoi distinguer ces deux notions? Tout d'abord, il faut bien voir qu'un document, même s'il n'est que bien formé, peut tout à fait être manipulé par des éditeurs spécialement conçus pour XML. Cependant, définir une DTD et créer des documents valides par rapport à elle comporte des avantages. D'une part, une DTD est utile car réutilisable. D'autre part, avec XML, la mise en page d'un document est définie dans, ce qu'on appelle, une feuille de style. Quand une DTD existe, une seule feuille de style est nécessaire pour mettre en page tous les documents valides par rapport à elle.

La figure 1.1 montre un document bien formé, noté d. Il contient des médecins et des patients. Comme vous pouvez le constater, ce sont des balises ouvrantes (flot de caractères entouré de $<$ et de $>$) et fermantes (flot de caractères entouré de $</$ et de $>$) qui indiquent le début et la fin des éléments qui composent le document. Par exemple, le nom d'un médecin est entouré des balises $< nomMed >$ et $< /nomMed >$. Remarquons que cet exemple est très simple et qu'il n'illustre pas les multiples possibilités offertes par XML, notamment, pour représenter des liens.

Voyons maintenant, globalement, comment ce document est construit.

Tout d'abord, tout document XML peut posséder - c'est facultatif mais conseillé - un prologue. Dans notre exemple, il se restreint à la première ligne et il indique uniquement la version de XML utilisée (ici, 1.0). Quand un document est valide par rapport à une DTD, le prologue, alors obligatoire, sert à indiquer celle-ci.

Ensuite, nous trouvons le contenu proprement dit du document. Pour qu'un document soit bien formé, il doit contenir un arbre d'éléments. Cela est vrai à deux conditions. (1) Tout élément enfant d'un autre élément doit être entièrement inclus dans celui-ci. Par exemple, dans le document d, un élément **patient** est entièrement inclus dans un élément **patients**. (2) Il y a un seul élément qui contient tous les autres. Cet élément est appelé l'élément racine du document. L'élément racine du document d est l'élément **annuaire**.

Enfin, dernière chose, cet exemple montre qu'un élément peut être caractérisé par un ou plusieurs attributs. Chaque attribut caractérisant un élément apparaît dans la balise d'ouverture de celui-ci sous la forme *attribut = 'valeur'*. L'élément **medecin** (resp. **patient**) est caractérisé par un seul attribut **med_id** (resp. **pat_id**). Pourquoi, dans un document, faire figurer une information comme valeur d'un attribut plutôt que dans un élément? Cela dépend des applications qui vont l'utiliser. Par exemple, notons que (1), par défaut, une valeur d'attribut n'est pas visualisée par un navigateur XML sauf si une feuille de style le spécifie et (2) une valeur d'attribut, à la différence d'un élément, n'est pas structurée.

1.2 Une DTD pour ce document

Une DTD permet de définir, nous l'avons dit plus haut, la structure d'un document valide par rapport à elle. En l'occurrence, elle définit les éléments qui seront utilisés ainsi qu'éventuellement, leurs attributs.

Que peut contenir un élément? Il peut contenir :

- un ou des éléments fils spécifiés
- des données représentées par un flot de caractères
- un mélange de données et d'éléments fils spécifiés
- un mélange arbitraire de données et d'éléments, en fait, n'importe quel contenu respectant la syntaxe XML

Il peut également être caractérisé par un ou plusieurs attributs.

La figure 1.2 montre une DTD par rapport à laquelle le document d est valide. Expliquons maintenant chacune des lignes.

```
<? xml version = ''1.0'' ?>
<annuaire>

    <medecin med_id = '1'>
      <nomMed> Gaston Lagaffe </nomMed>
      <adresse> 10, rue Mozart - 75016 Paris </adresse>
      <specialite> dermatologue </specialite>
      <cabinet> etage 2, porte 202 </cabinet>
      <patients>
        <patient pat_id = '400'>
          <nomPat> Boule </nomPat>
          <age> 10 </age>
          <sexe> M </sexe>
          <entier_aleatoire> 2 </entier_aleatoire>
          <num> 34 </num>
        </patient>
        <patient pat_id = '180'>
          <nomPat> Obelix </nomPat>
          <age> 40 </age>
          <sexe> M </sexe>
          <entier_aleatoire> 5 </entier_aleatoire>
          <num> 47 </num>
        </patient>
      </patients>
    </medecin>

    <medecin med_id = '2'>
      <nomMed> Fantasio </nomMed>
      <adresse> 43, rue Lavoisier - 91000 Evry </adresse>
      <cabinet> etage 6, porte 601 </cabinet>
      <patients>
        <patient pat_id = '24'>
          <nomPat> Lucky Luke </nomPat>
          <age> 37 </age>
          <sexe> M </sexe>
          <entier_aleatoire> 3 </entier_aleatoire>
          <num> 28 </num>
        </patient>
        <patient pat_id = '97'>
          <nomPat> Tintin </nomPat>
          <age> 35 </age>
          <sexe> M </sexe>
          <entier_aleatoire> 4 </entier_aleatoire>
          <num> 88 </num>
        </patient>
      </patients>
    </medecin>


</annuaire>
```

FIG. 1.1 – *Un document XML d*

Définition des éléments

Chaque élément est défini par une ligne dont la syntaxe est la suivante :

```
<!ELEMENT nom-element definition-de-ses-sous-elements >
```

Prenons des exemples et voyons ce qu'ils signifient.

Tout d'abord, prenons l'élément patient. Ses sous-éléments sont définis dans une séquence. Cela signifie que chaque instance d'un élément *patient* comporte comme sous-éléments les éléments *nomPat*, *age*, *sexe*, *entier_aleatoire* et *num*. Dans un document valide, chaque élément *patient* doit contenir tous ces sous-éléments. De plus, dans celui-ci, l'ordre dans lequel ils apparaîssent doit correspondre à celui donné dans la DTD.

Les caractères '?', '*' et '+' peuvent annoter chacun des sous-éléments dans la séquence.

- Si un '?' annote un sous-élément, cela signifie que la présence de celui-ci est optionnelle dans un document. C'est la cas du sous-élément *specialite* dans l'élément *medecin*. Notons d'ailleurs que le second médecin dans le document d n'est pas caractérisé par une spécialité.
- Si un '*' annote un sous-élément, cela signifie que l'élément parent peut contenir 0, 1 ou n tel sous-élément. C'est le cas de l'élément *annuaire* qui peut contenir 0, 1 ou n sous-éléments *medecin*.
- Si un '+' annote un sous-élément, cela signifie que l'élément parent est caractérisé par au moins un tel sous-élément. C'est le cas de l'élément *patients* qui est caractérisé par au moins un élément *patient*.

Aussi, n'oublions pas qu'un élément peut être caractérisé par un flot de caractères. C'est le cas, par exemple, de l'élément *nomMed*. Le mot clé **#PCDATA**, abréviation de *Parsed Character Data*, est alors utilisé.

Enfin, malheureusement, l'exemple ne le montre pas, le caractère '|' peut remplacer la virgule dans la séquence où sont définis les sous-éléments d'un élément. Nous pourrions par exemple avoir :

```
<!ELEMENT x (#PCDATA | y | z)*>
<!ELEMENT y (#PCDATA)>
<!ELEMENT z (#PCDATA)>
```

Cela signifie qu'une instance de l'élément x peut contenir, dans un ordre quelconque, des flots de caractères, des éléments y et des éléments z, par exemple,

```
<x> une chaine de caracteres <z> un element z </z> une autre chaine de
caracteres <y> un element y </y> <z> un autre element z </z> </x>
```

Définition des attributs

Chaque attribut d'un élément est défini par une ligne dont la syntaxe est la suivante :

```
<!ATTLIST nom-element nom-attribut type-attribut declaration-de-defaut>
```

Dans notre exemple, l'élément *medecin* (resp. *patient*) possède un seul attribut *med_id* (resp. *pat_id*).

Le mot clé **CDATA** indique que la valeur de l'attribut est une chaîne de caractères. Le mot clé **#REQUIRED** indique que l'attribut *med_id* (resp. *pat_id*) sera obligatoirement présent dans chaque instance d'élément *medecin* (resp. *patient*) et que la valeur de cet attribut devra toujours être indiquée.

Beaucoup d'autres possibilités sont offertes. Nous ne les décrirons pas ici.

```
<? xml version = '1.0' ?>
<!ELEMENT annuaire medecin*>
<!ELEMENT medecin (nomMed, adresse, specialite?,cabinet,patients)>
<!ATTLIST medecin med_id CDATA #REQUIRED>

<!ELEMENT patients patient+>
<!ELEMENT patient (nomPat,age,sexe,entier_aleatoire,num)>
<!ATTLIST patient pat_id CDATA #REQUIRED>

<!ELEMENT nomMed (#PCDATA)>
<!ELEMENT adresse (#PCDATA)>
<!ELEMENT specialite (#PCDATA)>
<!ELEMENT cabinet (#PCDATA)>
<!ELEMENT nomPat (#PCDATA)>
<!ELEMENT age (#PCDATA)>
<!ELEMENT sexe (#PCDATA)>
<!ELEMENT entier_aleatoire (#PCDATA)>
<!ELEMENT num (#PCDATA)>
```

FIG. 1.2 – *Une DTD par rapport à laquelle d est valide*

Chapitre 2

Interrogation de données XML dans Xyleme

Nous avons vu précédemment comment optimiser, dans le SGBD à objets O_2, l'évaluation d'une requête sur des données au format d'arbre. Cette requête était posée par un seul utilisateur à la fois et entièrement évaluée sur une même machine. De plus, les données interrogées étaient stockées sur un seul disque. Enfin, leur typage était strict. Dans ce contexte, les techniques d'optimisation reposaient sur l'utilisation d'algorithmes exploitant la façon avec laquelle les données sont stockées sur disque et/ou utilisant des structures de données telles que table de hachage.

Nous nous intéressons maintenant à l'optimisation de requêtes portant sur la structure de données au format XML. Ce travail a été entrepris au sein de l'action de recherche Xyleme [Xyl], menée au projet Verso [1], dont l'objectif était de concevoir et d'implanter un prototype d'entrepôt de données capable d'exploiter toutes les données XML qui peuvent être trouvées sur le Web. Xyleme peut, grossièrement, être vu comme une énorme base de données où sont répliquées, potentiellement, toutes les données XML du Web.

Même si, comme précédemment, le format des données interrogées est toujours un arbre, le contexte est ici tout à fait différent. Les données étant nombreuses, potentiellement toutes les données du Web, celles-ci sont distribuées dans autant de disques que nécessaire. La structure des données n'est pas stricte comme elle peut l'être dans les SGBDs. Seule indication? Les DTDs et, n'oublions pas, un document XML peut tout à fait exister sans que sa structure soit définie dans une DTD. En dépit de cela, nous voulons être capable, dans Xyleme, de répondre rapidement à des millions de requêtes simultanées portant sur la structure des documents. Une requête peut par exemple être : parmi tous les documents stockés dans Xyleme, quels sont ceux qui possèdent un élément *société* sous lequel se trouve, à n'importe quelle profondeur, un élément *produit* qui contient la chaîne *"Xyleme"*?

Dans ces conditions, quelle technique d'optimisation adopter? Peut-on encore exploiter la position des données interrogées sur disque, utiliser des algorithmes sophistiqués construisant des structures de données intermédiaires? Non, certainement pas. Notre objectif, ici encore, est de minimiser le nombre de lectures sur disque. Pour ce faire, nous avons choisi la solution adoptée par les moteurs de recherche tels que Altavista [Alt] ou Google [Goo]. Nous indexons, dans autant d'index plein-texte que nécessaire, les données stockées dans Xyleme. Chacun de ces index doit nous permettre de récupérer des documents correspondant à une certaine structure. Pourquoi des index plein-texte? Il faut bien voir que, dans les SGBDs, les index permettent d'accéder rapidement à des données de structure identique. Dans notre cas, les structures des documents indexés sont très variées.

La taille de chacune de ces structures est critique. En effet, vu le grand nombre d'utilisateurs simultanés, chacune d'elles doit tenir dans la mémoire centrale d'une machine. De cette façon, aucune entrée/sortie n'est

1. François Bancilhon, ancien directeur de O_2 Technology et le projet Verso sont à l'initiative de l'action de recherche Xyleme. D'autres groupes de recherches provenant, entre autres, de l'Université de Mannheim, de l'Université d'Orsay et du CNAM ont collaboré. La priorité était de proposer des services de haut niveau qui sont difficiles ou impossibles à supporter avec les technologies utilisées actuellement sur le Web. En particulier, ont été considérées, des requêtes plus complexes que les recherches par mot clé, une intégration de données en fonction de leur sémantique et une acquisition sophistiquée des pages sur le Web avec détection des modifications.

select n,a
from pe in culture//peinture, n in pe/nom, a in? pe/artiste
where pe/musee contains "Amsterdam"

FIG. 2.1 – *Une requête Q*

occasionnée quand un index est interrogé.

Nous verrons, dans le chapitre suivant, comment nous avons adapté l'indexation plein-texte pour répondre à nos besoins. Un index plein-texte n'est en effet rien d'autre qu'un tableau qui associe, à chaque mot indexé, l'ensemble des documents dans lequel il apparaît. Il est donc spécialement conçu pour répondre efficacement à des recherches par mot clé. Par contre, nous voulons être capable de vérifier, en interrogeant un index, que telle balise est descendante (ou fille) de telle autre dans un document. En effet, reprenons la requête citée précédemment. Pour l'évaluer, nous voulons être capable de vérifier, en interrogeant un index, que, dans un document, un élément *produit* est sous un élément *société* et que la chaîne *"Xyleme"* est sous un élément *produit*.

Avant de parler de cela, profitons de ce chapitre pour expliquer l'architecture de Xyleme. Nous allons, pour cela, prendre un exemple de requête et montrer comment elle est évaluée par le processeur de requêtes. Nous allons le voir, dans Xyleme, une requête porte sur une collection de documents qui peut être le résultat d'une union de plusieurs collections plus petites. Les données sont réparties sur disque. Chaque partition correspond à une collection. Chaque collection est indexée séparément. A une collection correspondent un ou plusieurs index[2].

Pour évaluer une requête, ce sont donc ce ou ces index qui sont interrogés.

L'algèbre utilisée par le processeur de requêtes travaille sur des arbres. Notamment, un opérateur, noté **PatternScan**, permet de récupérer, dans une collection d'arbres (ou forêt), tous les documents correspondant à une certaine structure spécifiée dans un *arbre de motif*. C'est cet opérateur qui est évalué en interrogeant le ou les index correspondant à la collection. Nous verrons la problématique de cette interrogation dans le dernier chapitre.

2.1 Un exemple de requête dans Xyleme

2. NB : une collection, dans Xyleme, n'est pas indexée par défaut. Elle l'est si l'utilisateur, plus précisément, l'administrateur, le décide.

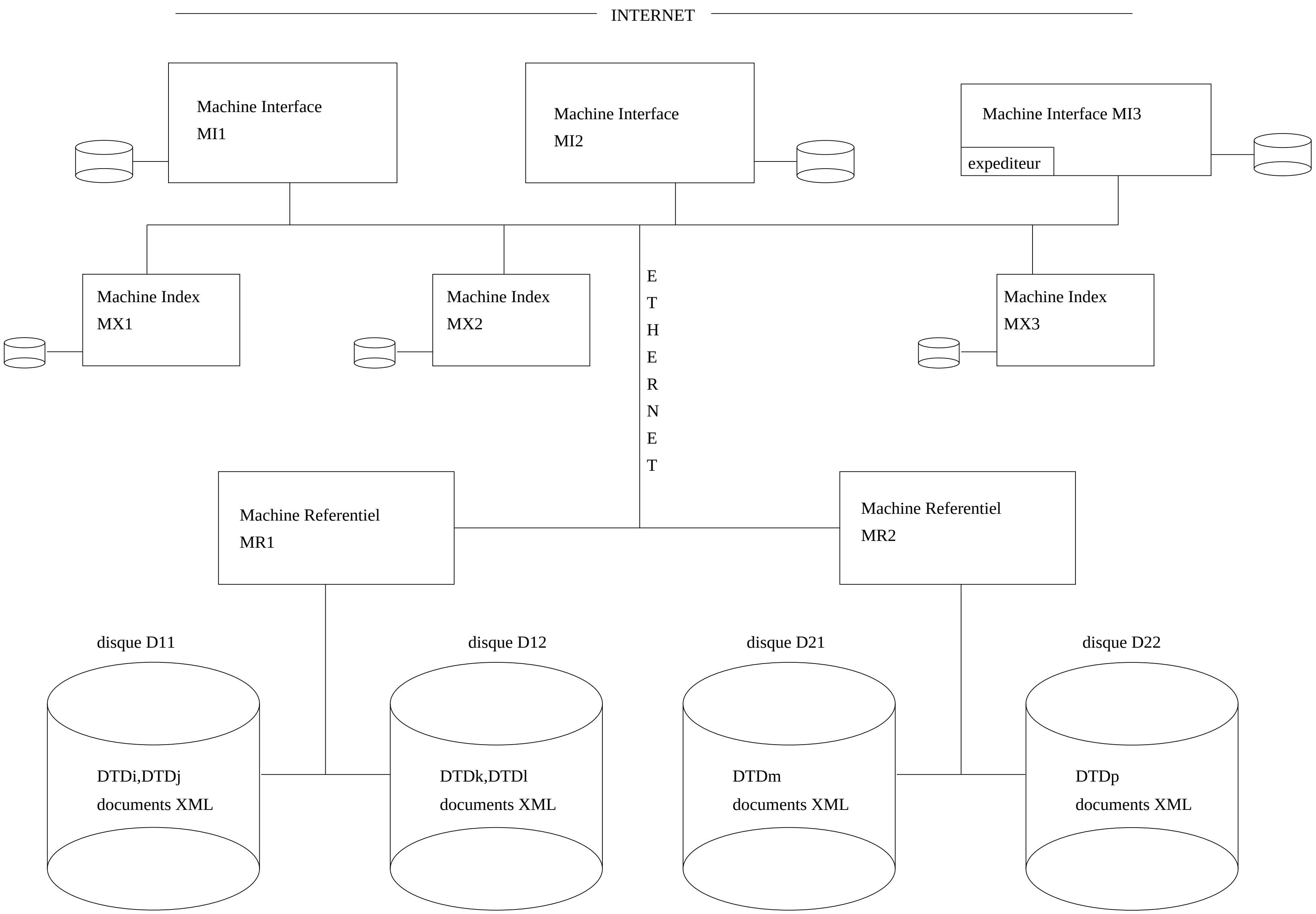


FIG. 2.2 – *Les différents groupes de machines dans Xyleme*

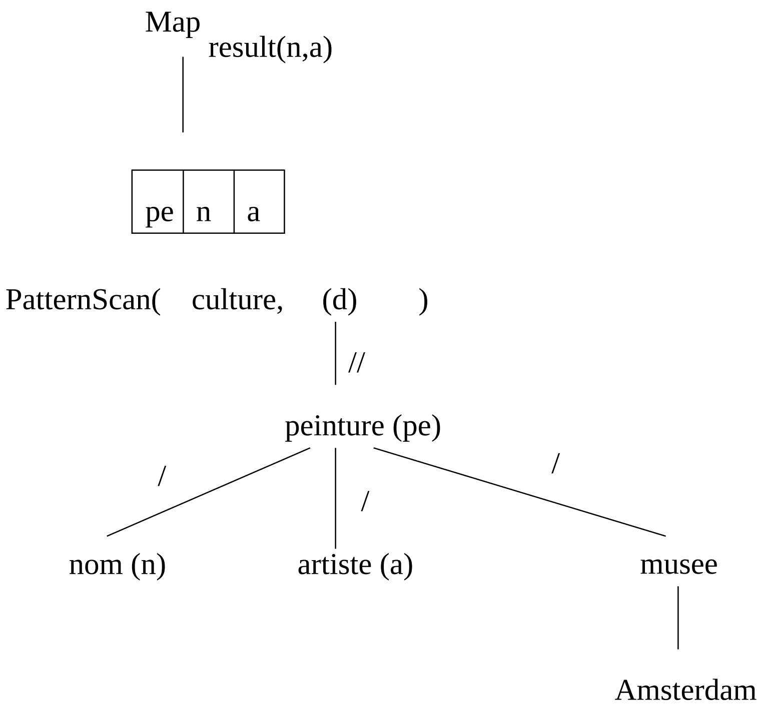


FIG. 2.3 – *Une expression algébrique*

2.1.1 Choix concernant le langage de requêtes

Le langage de requêtes de Xyleme est une extension d'OQL [Cat97, Clu98]. Il a été étendu pour évaluer des requêtes sur la structure des données interrogées comme le permet XQL [RLS98]. Il comporte, à ce titre, les opérateurs / et // qui, respectivement, signifient *estFils* et *estDescendant*.

Une requête porte sur une collection qui peut être le résultat d'une union de plusieurs collections plus petites. Des documents sont regroupés dans une collection parce qu'ils ont la même thématique, notamment, parce que leurs DTDs se "ressemblent".

La figure 2.1 montre une requête autorisée par Xyleme, notée **Q**. Elle porte sur la collection de documents *culture*. Comme son nom l'indique, tous les documents qui appartiennent à cette collection ont trait à la culture.

Cette requête recherche, dans les documents de cette collection, le nom des peintures exposées dans un musée de "Amsterdam" et, s'ils sont indiqués, le (ou les) artistes qui les ont peintes. Plus précisément, **Q** récupère, pour tous les documents de la collection *culture* qui contiennent un élément *peinture* qui contient lui-même un sous-élément *musee* qui contient la chaîne "Amsterdam", la valeur du sous-élément *nom* des éléments *peinture* sélectionnés et, s'ils existent, la valeur de chacun de ses sous-éléments *artiste*[3]. Notez la signification de l'opérateur **in?**. Quand aucun sous-élément *artiste* n'est contenu dans un élément *peinture* sélectionné, seule la valeur de l'élément *nom* de cet élément *peinture* est retournée par la requête. Quand un ou plusieurs éléments *artiste* sont contenus dans un élément *peinture* sélectionné, il y a autant d'éléments dans le résultat que d'artistes pour cette peinture. Chaque élément du résultat indique la valeur de l'élément *nom* ainsi que la valeur d'un élément *artiste* de l'élément *peinture* sélectionné.

2.1.2 Processeur de requêtes et architecture

Architecture

3. Nous supposons que : (1) seul un élément *nom* peut être sous-élément d'un élément *peinture*, (2) plusieurs éléments *artiste* peuvent être sous-éléments d'un élément *peinture*

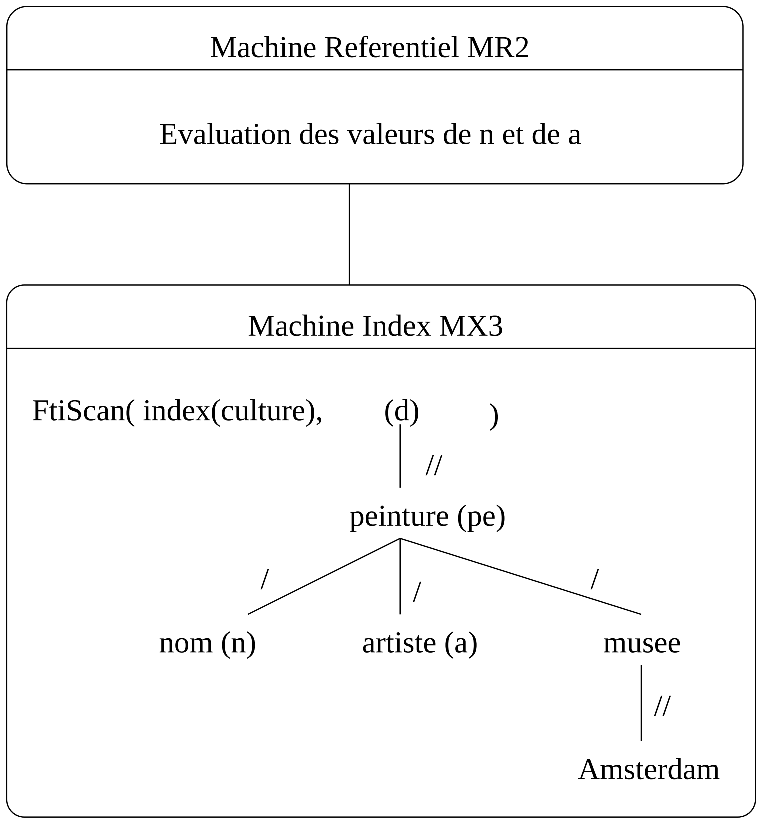


FIG. 2.4 – *Un plan d'exécution distribué*

La figure 2.2 montre l'architecture de Xyleme. Les machines, des PCs tournant sous Linux, sont représentées par des rectangles. La figure montre deux machines référentiels MR_1 et MR_2. Chacune possède deux disques.

Comme vous pouvez le constater, les documents sont regroupés sur disque suivant la DTD par rapport à laquelle ils sont valides. Quand des documents relatifs à plusieurs DTDs sont stockés sur le même disque, c'est qu'ils ont trait à la même thématique. Les documents d'une collection interrogeable par le langage de requêtes sont donc regroupés sur disque. Quand cela est suffisant, ils sont tous stockés dans la mémoire secondaire d'une seule machine référentiel. Sinon, ils sont distribués dans la mémoire secondaire d'autant de machines référentiel que nécessaire.

Supposons que la collection **culture** regroupe les documents valides par rapport aux structures-type DTD_i et DTD_j. Comme cela est représenté dans la figure, elle serait stockée entièrement dans le disque D_{11} de la machine référentiel MR_1. Autre possibilité, supposons que cette collection regroupe les documents valides par rapport aux structures-type DTD_i, DTD_j, DTD_k, DTD_l et DTD_m. Dans ce cas, comme cela est représenté dans la figure, la mémoire secondaire de deux machines référentiels serait nécessaire pour stocker cette collection : les deux disques de MR_1 et le disque D_{21} de MR_2.

Les index, quant à eux, résident dans la mémoire centrale de machines, appelées machines index. Trois machines index (MX_1, MX_2 et MX_3) sont représentées sur la figure.

Supposons qu'une collection soit indexée. Par conséquent, tous les documents de celle-ci qui sont stockés dans la mémoire secondaire d'une machine référentiel sont indexés :

- Si cela est suffisant, dans un seul index résidant dans la mémoire centrale d'une machine index. Un ou plusieurs index peuvent résider dans la mémoire centrale d'une machine index.

- Dans le cas contraire, dans autant d'index que nécessaire, chacun résidant dans la mémoire centrale de différentes machines index. Chacun indexe autant de documents que possible.

Reprenons la collection *culture* et supposons qu'elle est indexée. Prenons le cas où les documents de cette collection sont stockés dans les deux disques de MR_1 et dans le disque D_{21} de MR_2. Comme cela est représenté dans la figure, il se pourrait que deux index, chacun résidant dans la mémoire centrale de MX_1 (resp. MX_2), soient nécessaires pour indexer tous les documents de cette collection stockés dans la mémoire secondaire de MR_1 et qu'un autre index, dans la mémoire centrale de MX_3, indexe les documents de cette collection qui sont stockés dans la mémoire secondaire de MR_2.

Pourquoi avoir fait ce choix? Reprenons la requête **Q**. Si les documents appartenant à la collection *culture* étaient stockés, de façon aléatoire, dans différents disques de différentes machines référentiel et si la politique d'indexation restait la même, que se passerait-il?

D'une part, le nombre d'index à interroger pour évaluer la requête **Q** serait très important. Si beaucoup d'utilisateurs posaient simultanément une requête sur la collection *culture*, alors, de multiples interrogations simultanées sur de multiples index s'effectueraient. Ceci défavoriserait les autres requêtes qui, simultanément, interrogeraient d'autres collections.

D'autre part, pour accéder aux valeurs des éléments *nom* et *artiste*, des lectures sur disque sont nécessaires. Même si l'interrogation des index permet d'en minimiser le nombre, il n'en reste pas moins que ces lectures s'effectueraient sur de nombreux disques différents. Ceci serait loin d'être efficace.

Avec notre solution, pour évaluer la requête **Q**, nous n'interrogeons qu'un nombre restreint d'index et, quand des lectures sur disque s'effectuent, elles se font sur la mémoire secondaire d'un nombre restreint de machines référentiel. Seules les machines index qui contiennent les index relatifs à la collection interrogée travaillent. Cela ne défavorise donc pas les performances des requêtes simultanées portant sur les autres collections.

Evaluation de la requête

Voyons maintenant plus précisément comment la requête **Q** est évaluée.

Supposons, pour simplifier, que tous les documents de la collection *culture* soient dans un des disques de MR_2, qu'elle est indexée, et qu'un seul index, résidant dans la mémoire centrale de la machine MX_3 indexe toute cette collection.

Tout d'abord, la requête, une fois posée par l'utilisateur, n'est qu'une chaîne de caractères. Elle est traitée par le programme, appelé **expéditeur**, sur la machine interface MI_3. Celui -ci, comme son nom l'indique, l'envoie sur une des machines interface qui travaille le moins à ce moment là, pour être traitée par le processeur de requêtes.

Supposons que la machine interface choisie soit MI_1. Sur cette machine, le processeur de requêtes transforme la requête, d'abord, en un arbre syntaxique, puis, en une expression algébrique, appelée plan d'évaluation.

Nous avons adapté l'algèbre de [CACS94] pour traiter des collections d'arbres plutôt que d'objets. En effet, un document XML n'est rien d'autre qu'un arbre (par exemple, un arbre DOM [Gro98a]). Comme dans [CCM96, CCS00], nous avons introduit un opérateur qui permet de sélectionner, parmi une collection d'arbres (nous dirons une forêt), ceux qui correspondent à une certaine structure spécifiée dans un *arbre de motif.* Cet opérateur est appelé **PatternScan**.

La figure 2.3 montre l'expression algébrique relative à la requête **Q** qui est construite par le processeur de requêtes.

1. Le premier opérateur **PatternScan** recherche, parmi tous les documents de la collection *culture* ceux-qui correspondent à l'arbre de motif donné en paramètre. Il retourne un ensemble de n-uplets. Chaque attribut d'un n-uplet de cet ensemble correspond à chaque variable apparaîssant dans l'arbre de motif (ici, pe, n et a). La variable d ne sert qu'à symboliser la racine de cet arbre.

2. Le second opérateur **Map** applique une fonction à tous les éléments qu'elle reçoit en entrée. Dans cet exemple, cette fonction est **result(n,a)**. Elle construit un élément de nom *result* dont la valeur est donnée par **n** et, quand il existe, par **a**. Le résultat de la requête est un document XML contenant autant d'élément *result* qu'il y a de couples **(n,a)**.

Enfin, le processeur de requêtes, après optimisation, transforme le plan d'évaluation en un plan d'exécution. Celui-ci est distribué. Il est indiqué dans la figure 2.4.

Tout d'abord, le **PatternScan** est évalué sur la machine index MX_3 en interrogeant l'index relatif à la collection *culture*. L'opération physique est appelée **FtiScan**. Elle interroge l'index adéquat avec le même arbre de motif.

Ensuite, pour chaque n-uplet retourné par le FtiScan, la valeur des sous-éléments *nom* et *artiste* (s'ils sont présents) est évaluée sur la machine référentiel MR_2. Chacun des éléments du résultat est envoyé sur la machine interface MI_1 pour être retourné à l'utilisateur.

Chapitre 3

Les index plein-texte dans Xyleme

Dans Xyleme, nous voulons être capable de récupérer, en interrogeant un index, les documents qui correspondent à une certaine structure, définie dans un arbre de motif. Notamment, il s'agit de pouvoir vérifier, en les interrogeant, si telle balise est une descendante (ou une fille) de telle autre dans un document.

Comment faire? N'oublions pas, nous avons une contrainte à respecter obligatoirement. Chaque index doit tenir en mémoire centrale tout en indexant le maximum de documents.

Nous avons choisi l'indexation plein-texte. Rappelons ce qu'est un index plein-texte. C'est une structure qui stocke, pour chaque mot indexé, l'ensemble des documents dans lesquels il apparaît. Cela est tout à fait adapté pour répondre à des recherches par mot clé, par contre, tout à fait insuffisant pour répondre à nos besoins.

Notre solution consiste à associer, à chaque mot indexé, un ensemble d'informations - nous dirons un ensemble de *codages* - chaque information indiquant non seulement le document dans lequel il se trouve mais aussi sa position, point de vue structure, dans le document. Pour être efficace, le codage doit satisfaire les contraintes suivantes :

1. **Les codages doivent être aussi denses que possible.** Nous voulons maintenir des index en mémoire centrale. De plus, nous ne voulons pas gérer plus de machines index que nécessaire. Par conséquent, la taille d'un codage doit être aussi petite que possible, tout en contenant toutes les informations nécessaires.

2. **La mise à jour d'un index doit avoir un coût raisonnable.** En effet, dans le futur, nous espérons que Xyleme va charger (ou recharger) des millions de documents par jour. Dans ces conditions, minimiser le coût des mises à jour est vital. Pour ce faire, quand un document indexé a changé, une manière de modifier l'index correspondant consisterait à effectuer un **diff** entre la version précédente et la version courante de ce document et d'ajouter/supprimer uniquement les éléments qui ont changé. Cela signifie que l'insertion/suppression d'un élément ne devrait pas invalider les codages de tous les autres.

3. **Après interrogation d'un index, l'accès aux données sur disque doit être rapide.** Dans la requête de la figure 2.1, une fois l'interrogation de l'index effectuée, il faut aller chercher, sur disque, la valeur du sous-élément *nom* et, s'ils existent, des sous-éléments *artiste* de chacun des éléments *peinture* sélectionnés. Avec l'index, nous récupérons des informations qui nous permettent d'accéder aux données sur disque et, notamment, à la valeur de balises. Les informations à stocker dans l'index sont donc très liées à la manière d'accéder aux données sur disque. Il doit y avoir correspondance.

4. **Pour favoriser l'interrogation des index, les codages associés à un mot doivent apparaître dans un certain ordre.** Nous expliquerons pourquoi dans la suite.

Dans ce chapitre, nous proposons deux codages : l'un est appelé codage préfixe-postfixe, l'autre, codage bit-structuré. Pour chacun, nous allons voir comment ils permettent d'évaluer un *PatternScan*, notamment, comment nous vérifions qu'une balise est une descendante (ou fille) d'une autre balise dans un document.

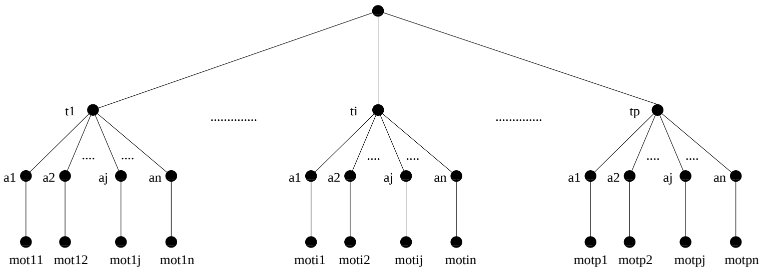


FIG. 3.1 – *Représentation, sous forme d'arbre, d'une relation contenant p n-uplets*

Nous allons également comparer ces deux codages et voir dans quelle mesure ils satisfont les contraintes citées ci-dessus.

Nous allons le voir, pour l'instant, aucun de ces deux codages ne les satisfait toutes.

3.1 Les codages

On peut dire qu'il existe deux catégories de documents XML : ceux générés à partir de données structurées, stockées, par exemple, dans un SGBD et les autres que nous appellerons documents XML types. Considérons la première catégorie. Représentons un document de cette catégorie sous forme d'arbre. Prenons comme exemple un document généré à partir d'une relation contenant p n-uplets. La figure 3.1 montre l'arbre associé à une telle relation. On s'aperçoit que cette représentation possède une caractéristique intéressante. En effet, il y a de grandes chances pour que, dans celle-ci, il y ait $p \times n$ (resp. p) noeuds internes représentant les attributs (resp. n-uplets) et $p \times n$ noeuds feuilles représentant les mots, valeurs de ces attributs. A l'opposé, dans les documents XML typiques, il y a de grandes chances pour que le nombre de noeuds internes soit insignifiant par rapport au nombre de noeuds feuilles. Autrement dit, la structure prend une place insignifiante par rapport au contenu textuel dans le document.

Chacun des deux codages que nous allons présenter ne convient pas simultanément pour les deux catégories : l'un - le codage préfixe-postfixe - est adapté aux documents XML typiques, l'autre - le codage bit-structuré - est adapté aux autres.

Pour expliquer ces codages, nous nous aidons de la figure 3.2. Elle comporte deux parties. La partie haute montre un document XML et sa représentation sous forme d'arbre. Comme vous pouvez le constater, dans l'arbre, les éléments et les attributs sont représentés de la même manière. Ils seront donc indexés - ou, encore, codés - de la même manière. La partie basse de cette figure représente, dans sa partie gauche (resp. droite), les entrées dans l'index quand le codage préfixe-postfixe (resp. bit-structuré) est utilisé.

Un index est représenté comme un tableau. Chaque entrée, dans celui-ci, correspond à un mot indexé. La structure d'une entrée est la suivante : [mot,{information}], càd. un couple constitué d'un mot et d'un ensemble d'informations. Nous allons maintenant voir ce en quoi consiste cette information dans les codages que nous proposons.

3.1.1 Codage préfixe-postfixe

Principe

Dans ce codage, chaque information dans l'ensemble des informations associé à un mot indexé est constituée :

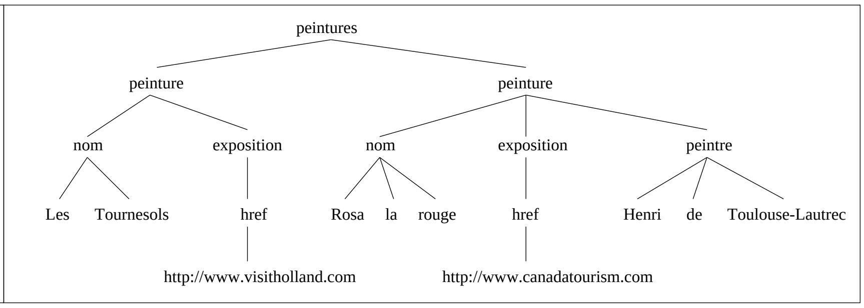
1. d'un identificateur de document. Comme son nom l'indique, il identifie le document dans lequel le mot apparaît.

2. d'un drapeau indiquant le “type” du mot. Il peut prendre trois valeurs : 'E', 'A' ou 'W'. Si le drapeau vaut 'E' (resp. 'A', 'W'), cela signifie que le mot indexé représente un élément (resp. attribut, mot).

3. d'un entier, appelé préfixe

4. d'un entier, appelé postfixe

```
<peintures>
  <peinture>
    <nom> Les tournesols </nom>
    <exposition href = "http://www.visitholland.com" />
  </peinture>
  <peinture peintre = "Henri de Toulouse-Lautrec">
    <nom> Rosa la rouge </nom>
    <exposition href = "http://www.canadatourism.com" />
  </peinture>
</peintures>
```



peintures	{[d1,'E',1,10,1]}
peinture	{[d1,'E',2,4,2] , [d1,'E',6,9,2]}
nom	{[d1,'E',3,1,3] , [d1,'E',7,5,3]}
exposition	{[d1,'E',4,3,3] , [d1,'E',8,7,3]}
href	{[d1,'A',5,2,4] , [d1,'A',9,6,4]}
http://www.visitholland.com	{[d1,'W',5,2,4]}
.....	
rouge	{[d1,'W',7,5,3]}

Codage prefixe-postfixe

peintures	{[d1,'E',1]}
peinture	{[d1,'E',1.1,2]}
nom	{[d1,'E',1.1.1]}
exposition	{[d1,'E',1.1.2]}
href	{[d1,'A',1.1.2.1]}
http://www.visitholland.com	{[d1,'W',1.1.2.1,1]}
.....	
rouge	{[d1,'W',1.1.1,2]}

Codage bit-structure

FIG. 3.2 – *Un document XML et les deux codages associés*

5. d'un entier, appelé niveau

Une information correspond à chaque occurrence d'un mot ou d'une balise dans un document. Considérons l'élément *peinture*. Au mot indexé *peinture* correspond deux informations, [d1,'E',2,4,2] et [d1,'E',6,9,2]. Chacune correspond à chaque occurrence de l'élément *peinture*.

Voyons comment sont calculés les préfixes, postfixes et niveaux. Ils sont obtenus en effectuant une traversée, en profondeur d'abord, de l'arbre représentant le document.

Dans un premier temps, cette traversée ignore les noeuds feuilles dans cet arbre. Elle n'attribue donc une information qu'aux balises du document. A chaque fois qu'on rentre dans un noeud, un préfixe lui est attribué. A chaque fois qu'on sort d'un noeud, un postfixe lui est attribué. La différence entre deux préfixes (resp. postfixes) est 1. L'entier appelé niveau représente le niveau dans lequel se trouve le noeud correspondant au mot indexé. Par exemple, à la balise *peintures* est associée l'information [d1,'E',1,10,1]. Ce niveau est nécessaire, nous allons le voir, pour vérifier qu'une balise est fille d'une autre balise.

Dans un second temps, les préfixes, postfixes et niveaux sont attribués à chaque noeud feuille. Les préfixes, postfixes et niveaux d'un noeud feuille sont les mêmes que ceux de son noeud parent. Cette solution a été adoptée dans le souci de minimiser la taille d'un codage et, donc, d'un index. En effet, avec celle-ci, pour tous les noeuds d'un document, l'entier maximal associé à un préfixe (resp. postfixe) est plus petit que si la numérotation des noeuds feuilles respectait celle des noeuds internes.

Remarque
Bien sûr, cette information est stockée dans le minimum de place. Chaque entier est stocké avec le minimum de bits dans l'index plein-texte.

Comment vérifier la relation de descendance ou de filiation?

Une fois les documents indexés, un exemple de question à laquelle on peut répondre est : Comment vérifier que, dans d1, *rouge* est sous un élément *peinture* qui contient *Lautrec*?

Avec le codage préfixe-postfixe, dans un document,

- Une balise **a** est une descendante d'une balise **b** si le préfixe associé à **a** est plus grand que le préfixe associé à **b** et le postfixe associé à **a** est plus petit que le postfixe associé à **b**.
- Une balise **a** est un enfant d'une balise **b** si elle est une descendante de **b** et si 1 sépare le niveau auquel elle appartient du niveau auquel **b** appartient.
- Un mot **a** est un descendant d'une balise **b** si le préfixe associé à **a** est plus grand que le préfixe associé à **b** et le postfixe associé à **a** est plus petit que le postfixe associé à **b** ou si le préfixe (resp. postfixe) associé à **a** est identique au préfixe (resp. postfixe) associé à **b**.
- Un mot **a** est un fils d'une balise **b** si il y a égalité entre les préfixes et les postfixes associés à chacun d'eux.

En effet, n'oublions pas : le codage associé à un noeud feuille est le même que celui de son noeud parent.
Dans le document d1, il existe bien un codage pour l'élément peinture ([d1,'E',6,9,2]) dans lequel le préfixe (resp. postfixe) est plus petit (resp. plus grand) que le préfixe (resp. postfixe) dans les codages relatifs à rouge ([d1,'W',7,5,3]) et à Lautrec ([d1,'W',10,8,3]).

3.1.2 Codage bit-structuré

Principe

Dans ce codage, chaque information dans l'ensemble des informations associé à un mot indexé est constituée :

1. d'un identificateur de document
2. d'un drapeau indiquant le “type” du mot (cf le codage préfixe-postfixe).
3. d'une information structurelle, qui est une séquence de bits
4. d'une information positionnelle, qui est, également, une séquence de bits

L'information structurelle, ainsi que l'information positionnelle, sont stockées, dans l'index, sous la forme d'une séquence de bits. Dans la figure 3.2, pour des raisons de clarté, elles sont représentées sous la forme de plusieurs entiers séparés par des points. Petite remarque : contrairement au codage préfixe-postfixe, un seul codage existe pour les deux occurrences de l'élément *peinture* dans le document d1. Dans ce codage, l'information structurelle est 1.1, l'information positionnelle est 2. Nous verrons ensuite comment chaque entier est codé en bits.
Voyons d'abord comment ces informations sont obtenues.
L'information structurelle est obtenue en analysant la DTD ou, si elle n'existe pas, un arbre représentant la structure du document indexé. Dans notre cas, cette structure est représentée dans la figure 3.3. Pour chaque noeud, l'information structurelle associée est indiquée entre parenthèses. Elle représente le chemin, d'un point de vue structure, qui mène de la racine à ce noeud. Par exemple, l'information structurelle associée à l'élément *peintre* est 1.1.3.
L'information positionnelle a une signification différente suivant qu'elle se rapporte à une balise ou à un mot. Prenons deux exemples. L'information positionnelle relative à l'élément *peinture* (l'entier 2) indique qu'il y a deux tels éléments dans le document d1. Celle relative au mot *rouge* (l'entier 2) indique que ce mot est sous la seconde occurrence de l'élément *peinture* dans le document d1.
Voyons maintenant comment ces deux informations sont codées en une séquence de bits. Supposons que 32 bits suffisent. Prenons plusieurs exemples.

- Pour l'élément *peinture*, la séquence est

00001 0000 100....00

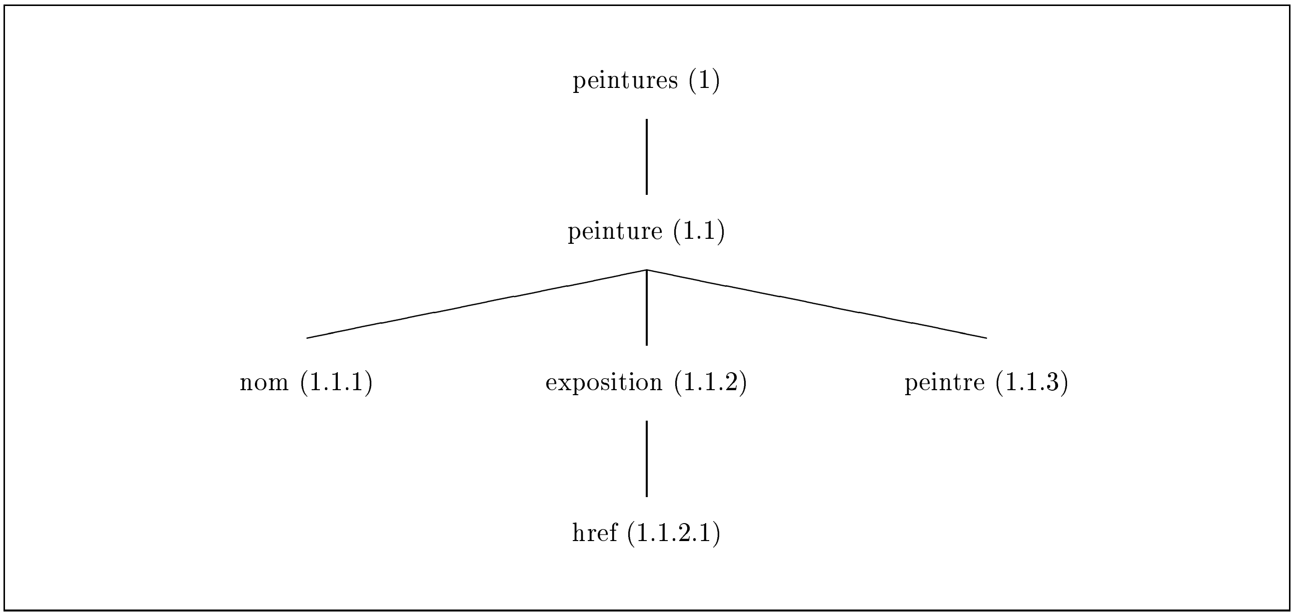


FIG. 3.3 – *Arbre représentant la structure du document indexé*

– Pour le mot *Lautrec*, la séquence est

00011 0100 100....00

– Pour le mot *http://www.visitholland.com*, la séquence est

00100 0010 000....00

Elle se décompose de la façon suivante :

1. Les cinq premiers bits sont réservés pour indiquer le nombre de bits qui représentent l'information structurelle dans les bits suivants. En effet, $2^5 = 32$. Prenons la séquence associée à *Lautrec*. Ces cinq bits (00011) indiquent que trois bits codent l'information structurelle.

2. Viennent ensuite les bits qui représentent l'information structurelle. Comme vous pouvez le constater, le nombre de bits (ici, quatre) pour coder cette information est le même pour tous les mots ou balises du document. Ce nombre est calculé de la façon suivante. Considérons la figure 3.3. En partant de la racine de cet arbre, on calcule, pour tous les noeuds d'un niveau, le nombre maximum d'enfants. Un nombre de bits est nécessaire pour coder ce maximum. Le nombre de bits pour coder l'information structurelle associée à chaque mot d'un même document est la somme de chacun de ces nombres. Pour être clair, reprenons notre exemple :

   – L'élément racine *peintures* ne peut avoir comme élément fils que l'élément *peinture*. Un seul bit est nécessaire.
   – L'élément *peinture* peut avoir, au maximum, trois éléments fils. Deux bits sont nécessaires.
   – Seul l'élément *exposition* peut avoir des éléments fils. Il n'en a qu'un : *href*. Un seul bit est nécessaire.

   Ce sont donc quatre bits qui permettent de coder l'information structurelle. Prenons la séquence associée à *Lautrec*. L'information structurelle est constituée des trois premiers bits (en effet, les cinq premiers bits indiquent 00011, càd. 3) des quatre bits codant l'information structurelle : 010. Elle représente 1.1.3. Le premier 1, indiquant la racine, n'est pas codé. Ensuite, chaque entier est codé en adoptant une numérotation à la C, càd. en partant de 0. L'entier 1 est codé par le bit 0. L'entier 3 est codé par 10.

3. L'information positionnelle a une signification différente suivant qu'elle se rapporte à une balise ou à un mot.

   - Prenons la balise *peinture*. L'information positionnelle est 100....00. Dans cette séquence, il y a autant de 1 que nécessaire. Le nombre de bits égaux à 1 représente le nombre de bits qu'il est nécessaire d'utiliser pour indiquer le nombre d'occurrences de la balise à ce niveau. Dans notre cas, il n'y a que deux balises *peinture*. Un seul bit suffit donc. S'il y avait sept balises *peinture*, l'information positionnelle associée à la balise *peinture* aurait été 11100....00. En effet, $111 \equiv 7$.
   - Prenons maintenant le mot *rouge*. L'information positionnelle indique que *rouge* est sous la seconde *peinture* du document. Autrement dit, elle est 100....00. Ici, le 1 veut dire “deuxième”. Prenons le mot *http://www.visitholland.com*. Il est sous la première *peinture*. L'information positionnelle est 000....00. Ici, le premier 0 signifie “première”.

Comment vérifier la relation de descendance ou de filiation?

Reprenons l'exemple que nous avons pris en décrivant le codage préfixe-postfixe.

L'unique occurrence de **rouge** est sous un élément **peinture** car la séquence de bits 1.1 qui représente l'information structurelle associée à l'élément **peinture** est préfixe de la séquence de bits 1.1.1 qui représente l'information structurelle associée à l'occurrence du mot **rouge**.

L'unique occurrence de **Lautrec** est également sous un élément **peinture** car la séquence de bits 1.1 qui représente l'information structurelle associée à l'élément **peinture** est préfixe de la séquence de bits 1.1.3 qui représente l'information structurelle associée à l'occurrence du mot **Lautrec**.

Enfin, l'information positionnelle relative à **rouge** et à **Lautrec**, l'entier 2, indique que ces deux occurrences sont toutes deux sous le second élément **peinture** du document.

3.1.3 Comparaison des codages

Densité. Le codage préfixe-postfixe est plus dense que le codage bit-structuré. Cependant, dans l'index, avec le codage préfixe-postfixe, il y a autant d'informations associées à un mot qu'il y a d'occurrences de celui-ci. Avec le codage bit-structuré, toutes les occurrences d'une même balise à un même niveau dans un document ne sont représentées qu'une seule fois dans l'index. Notre exemple montre cela avec l'élément *peinture*.

Mise à jour.

Si le codage préfixe-postfixe est utilisé, quand un document indexé est modifié, ce sont très souvent de nombreux codages dans l'index qui devront être modifiés. Prenons le document de la figure 3.2. Supposons que, dans la nouvelle version de celui-ci, le premier élément *peinture* est supprimé. Dans ce cas, ce sont tous les codages relatifs à ce document qui seront changés.

Par contre, si le codage bit-structuré est utilisé, dans la même situation, de nombreux codages resteront identiques à part celui relatif à l'élément peinture restant et à toutes les feuilles du sous-arbre correspondant à celui-ci. Une solution , avec ce codage, qui permettrait de minimiser les mises à jour dans l'index serait la suivante. Supposons que l'information positionnelle (l'entier 2) relative à l'élément *peinture* soit codée non pas sur un bit mais sur trois. Si, dans une nouvelle version du document, les deux premiers éléments peinture restaient identiques et que cinq autres étaient rajoutés, tous les codages existant déjà dans l'index resteraient inchangés.

Accès aux données sur disque. Les informations récoltées dans l'index doivent permettre d'aller chercher efficacement sur disque la valeur des balises retournées par une requête. Dans notre cas, la couche logicielle qui gère le repositoire est Natix [KM00]. Elle a été conçue et implantée à l'université de Mannheim. Celle-ci permet d'accéder rapidement à la valeur d'une balise d'un document étant donné l'identificateur de celui-ci et le préfixe associé à cette balise. Par conséquent, avec le codage préfixe-postfixe, aucun traitement préalable des informations récoltées dans l'index n'est nécessaire pour accéder aux données sur disque. Ce n'est pas le cas avec le codage bit-structuré. Les informations récoltées dans l'index, quand ce codage est utilisé, ne peuvent pas être exploitées telles quelles pour accéder à la valeur de balises sur disque.

Les informations associées à chaque mot devraient être ordonnées. Si, avec le codage préfixe-postfixe, les informations associées à un mot étaient triées, simultanément, sur les identificateurs de document et les préfixes, un algorithme efficace permettrait de vérifier que, dans un document, une balise est descendante (ou fille) d'une autre balise. Avec le codage bit-structuré, les informations associées à un mot indexé ne pourraient être triées que sur les identificateurs de documents. En effet, nous ne pouvons pas ordonner les séquences de bits.

3.2 Conclusion

Dès à présent, différents modèles ont été étudiés afin de stocker et de pouvoir interroger des documents XML : le relationnel [FK99, STZ+99, Ora], l'objet [ACC+97, CCM96, Exc] et le natif. Les approches proposées consistent à adapter des techniques d'optimisation existantes et à effectuer un couplage avec des index plein-texte [FKM00, CCMS98]. Au lieu de cela, nous avons adapté la technologie d'indexation plein-texte. Nous proposons deux méthodes d'indexation (nous appelons chaque méthode un codage) qui différent par l'information associée à chaque mot indexé : le codage préfixe-postfixe et le codage bit-structuré.

Dans la première implantation de Xyleme, nous avons retenu, comme codage, le codage préfixe-postfixe. La raison principale de ce choix est que les informations obtenues depuis l'index permettent d'accéder directement aux données sur disque (avec le préfixe).

De plus, vue la brève durée qui était impartie pour réaliser un prototype, nous avons choisi d'implanter les index dans Xyleme en utilisant les index d'Altavista [Alt].

Ceci nous a permis de nous concentrer sur les problèmes les plus importants. Malheureusement, avec cette technologie, nous ne maîtrisons pas l'ordre dans lequel sont placées les informations associées à chaque mot indexé. Nous n'avons donc pas pu implanter d'algorithme efficace pour vérifier les relations de descendance ou de filiation entre balises dans un document.

Chapitre 4

Evaluation d'un FtiScan

Au niveau logique, un **PatternScan** est une opération algébrique qui récupère, parmi tous les documents d'une collection, ceux qui correspondent à une structure spécifiée dans un arbre de motif.

Cette opération est implantée, physiquement, par un **FtiScan** ou une union de **FtiScans**. Chaque FtiScan récupère dans l'unique index (ou l'un des index) associé à la collection interrogée, les documents correspondant à l'arbre de motif.

Le résultat de cet opérateur physique est un ensemble de n-uplets. Chaque attribut d'un n-uplet de celui-ci correspond à chaque variable apparaîssant dans l'*arbre de motif.*

Dans ce chapitre, nous allons donner une définition précise de cet opérateur et, notamment, expliquer ce qu'est un arbre de motif.

Nous souhaitons évaluer, le plus rapidement possible, un FtiScan. Dans ce but, nous avons la même approche que pour optimiser une requête. Algébriquement, un FtiScan se représente par un arbre de jointures. Le problème est donc de réordonner ces jointures et de choisir un algorithme approprié pour évaluer chacune d'elles.

Les index interrogés tiennent en mémoire centrale. De ce fait, les jointures s'effectuent entre des ensembles qui tiennent en mémoire centrale. Nous avons donc choisi d'évaluer chacune d'elles par un algorithme par boucles imbriquées pour ne pas avoir à construire de structures intermédiaires telles que table de hachage ou ensemble trié. De plus, un arbre de jointures est effectué en pipeline. Comme nous le disions dans le chapitre précédent, notre implantation utilise les index d'Altavista [Alt]. Avec cette technologie, nous ne maîtrisons pas l'ordre dans lequel les informations associées à chaque mot indexé sont stockées. Nous n'avons donc pas pu implanter d'algorithme efficace qui aurait pu exploiter cet ordre.

Pour réordonner les jointures, nous utilisons une heuristique [Moe]. En un nombre restreint d'examens de l'arbre de motif, celle-ci choisit, parmi tous les arbres de jointure possibles, celui où la somme des cardinalités des résultats des jointures intermédiaires est proche du minimal ou, quand la chance est au rendez-vous, minimale. Cette somme est une bonne fonction de coût pour comparer deux arbres de jointures. En effet, comme les ensembles joints tiennent en mémoire centrale, c'est le nombre d'instructions élémentaires qu'il faut minimiser. Chaque jointure s'effectue par boucles imbriquées. Or, le nombre d'instructions élémentaires nécessaires pour effectuer la jointure entre deux ensembles de cardinalité n et p est $n \times p$. Par conséquent, plus les jointures se font entre des ensembles de petite taille, moins l'algorithme sera coûteux en termes d'instructions élémentaires.

4.1 Définition d'un FtiScan

Une opération **FtiScan** possède deux paramètres : l'index interrogé et un arbre de motif. Elle retourne un ensemble de n-uplets et s'exécute sur la machine index où l'index interrogé réside en mémoire centrale.

Nous allons reprendre l'exemple de FtiScan donné dans le chapitre 2. Celui-ci interroge l'index relatif à la collection **culture** et permet d'évaluer la requête **Q** de la figure 2.1.

Nous n'avions donné, précédemment, qu'une ébauche de l'arbre de motif, paramètre de ce FtiScan. La forme exacte de celui-ci est représentée dans la figure 4.1.

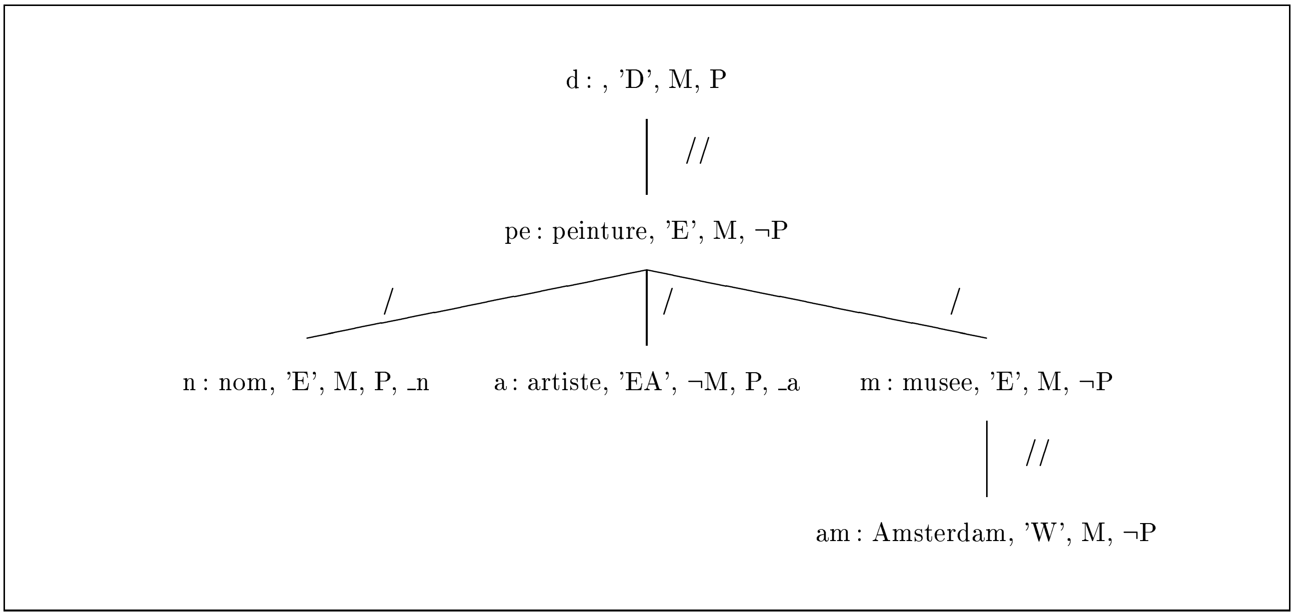


FIG. 4.1 – *Exemple d'arbre de motif*

Les informations contenues dans chaque noeud n d'un arbre de motif sont les suivantes :

- un mot, noté $mot(n)$. A part la racine de l'arbre, chaque noeud doit obligatoirement être caractérisé par un mot. Pour le noeud enfant de la racine, ce mot est *"peinture"*.
- un drapeau, noté $type(n)$, qui indique le type du mot. Ce drapeau peut prendre les valeurs suivantes : 'D' pour document, 'E' pour élément, 'A' pour attribut, 'EA' pour élément ou attribut, 'W' pour mot. Pour le noeud enfant de la racine, *peinture* représente un élément.
- un drapeau, noté $estProjeté(n)$, qui vaut vrai si la valeur de la balise caractérisée dans le noeud n est retournée par la requête évaluée. Dans la requête **Q**, *nom* est projeté (cela est symbolisé par P), *peinture* ne l'est pas (cela est symbolisé par ¬P).
- un drapeau, noté $estObligatoire(n)$, qui vaut vrai si la présence de la balise caractérisée par le noeud n est obligatoire dans les documents sélectionnés. *nom* est obligatoire (cela est symbolisé par M), *artiste* ne l'est pas (cela est symbolisé par ¬M). En effet, la requête **Q** retourne la valeur du sous-élément *nom* et, s'ils existent, la valeur des sous-éléments *artiste* des balises *peinture* sélectionnées.
- une étiquette, notée $étiquette(n)$. Elle annote l'arc qui relie le noeud courant avec son noeud parent. Cette étiquette prend deux valeurs : "//" ou "/". Par exemple, l'élément *nom* doit être un sous-élément de l'élément *peinture*. L'étiquette "/" signifie *estEnfant*. Le mot *"Amsterdam"* est contenu dans le sous-arbre, valeur de l'élément *musee*. L'étiquette "//" signifie *estDescendant*.
- deux variables. L'une existe toujours dans les noeuds, l'autre n'existe que si $estProjeté(n)$ vaut vrai. Considérons le noeud représentant l'élément *peinture*. La variable *pe* sert, nous allons le voir, lors de l'évaluation du FtiScan. Elle n'a que peu d'importance. La seconde variable est utilisée comme attribut des n-uplets résultats du FtiScan. Dans notre exemple, le résultat du FtiScan est un ensemble de n-uplets $[d : LocalDocId, \_a : entier, \_n : entier]$. d représente un identificateur de document local ($LocalDocId$ représente le type de ces identificateurs dans Xyleme), $\_a$ (resp. $\_n$) représente le préfixe associé à la balise *artiste* (resp. l'élément *nom*) dans les documents sélectionnés.

Quelques contraintes existent. Pour le noeud racine r d'un arbre de motif, $estProjeté(r)$ vaut toujours vrai. En effet, nous désirons toujours savoir les identificateurs de documents sélectionnés par un **FtiScan**.

Tous les noeuds enfants e d'un noeud n tel que $estObligatoire(n)$ vaut faux vérifient : $estObligatoire(e)$ vaut faux. Si $estObligatoire(n)$ est faux, alors $estProjeté(n)$ est vrai.

4.2 La problématique du FtiScan

Notre problème est de trouver l'algorithme le plus rapide pour exécuter un FtiScan. En effet, le coût de cet opérateur est critique quand on pense que ce sont des millions de documents qui sont indexés.

Pour résoudre ce problème, nous avons la même approche que lors de l'optimisation d'une requête. Un FtiScan, comme une requête, peut se représenter sous la forme d'une expression algébrique. Bien sûr, l'algèbre utilisée est très restreinte. Nous allons le voir dans ce chapitre, elle possède trois opérateurs de jointure (jointure normale, semi-jointure, jointure externe), un opérateur de projection (Map) et un opérateur de sélection. Leur définition est identique à celle que nous avons donnée dans la section 3.1 du chapitre 3 de la partie I où nous avons décrit une algèbre objet.

Plusieurs expressions algébriques peuvent représenter le même FtiScan. Le problème est donc de trouver celle qui s'exécute le plus rapidement.

Avant de montrer comment des exemples simples de FtiScan s'évaluent, faisons quelques remarques.

1. Ces opérateurs travaillent toujours sur des ensembles de nuplets de la forme

$$[localDocId : LocalDocId, prefixe : entier, postfixe : entier, niveau : entier]$$

   . En effet, c'est le codage préfixe-postfixe qui est implanté dans les index. Dans la suite, nous noterons chacun de ces ensembles :

$$\begin{aligned} IS_i &= IndexScan(w_i, k_i) \\ &= \{[localDocId : LocalDocId, prefixe : entier, postfixe : entier, niveau : entier]\} \end{aligned}$$

   Dans l'index interrogé, l'ensemble IS_i est l'ensemble des informations associées au mot w_i de type k_i. Quelquefois, nous utiliserons également la notation $IS_i[x_i]$ pour indiquer que x_i désigne un n-uplet de l'ensemble IS_i.

2. Le prédicat des sélections est toujours le même, quelque soit l'arbre de motif. Nous allons le voir ensuite, les sélections ne servent que pour les requêtes cherchant un élément donné comme racine.

3. L'opérateur de projection renvoie toujours un ensemble de n-uplets.

4. Les prédicats des jointures sont différents de ce que nous avions l'habitude de rencontrer dans OQL. Précisons leur forme. Si $x_1 \in IS_1$ et $x_2 \in IS_2$ et $p(x_1, x_2)$ désigne un prédicat, il est soit x_1/x_2, soit $x_1//x_2$.

   x_2 représente une balise. Deux possibilités existent pour x_1. Ces deux prédicats sont évalués de la façon suivante :

   (a) Si x_1 désigne une balise.

      $x_1//x_2$ est vrai si :

      i. $x_1.localDocId$ est égal à $x_2.localDocId$ et
      ii. $x_1.prefixe$ est plus grand que $x_2.prefixe$ et
      iii. $x_1.postfixe$ est plus petit que $x_2.postfixe$

      x_1/x_2 est vrai si :

      i. $x_1//x_2$ est vrai et
      ii. $x_1.niveau$ est égal à $x_2.niveau + 1$

   (b) Si x_1 désigne un mot.

      $x_1//x_2$ est vrai si :

      i. $x_1.localDocId$ est égal à $x_2.localDocId$ et

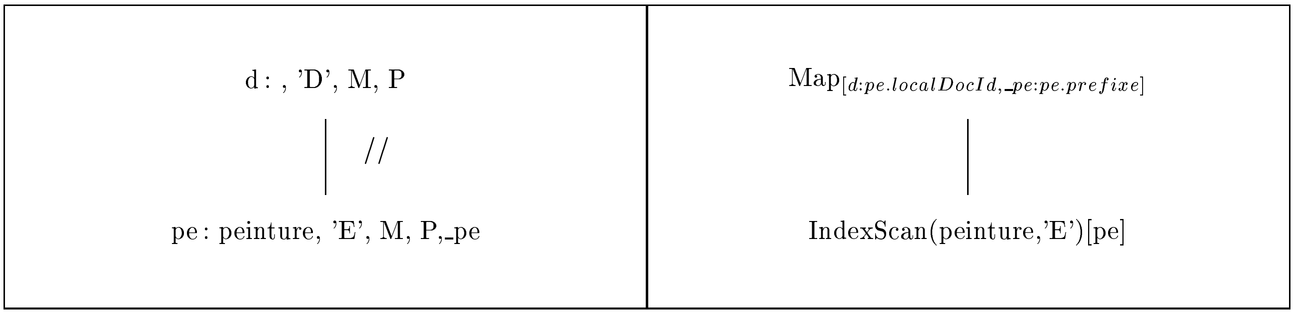


FIG. 4.2 – *Un FtiScan simple et sa traduction algébrique*

ii. soit, $x_1.prefixe$ est plus grand que $x_2.prefixe$ et $x_1.postfixe$ est plus petit que $x_2.postfixe$

iii. soit, $x_1.prefixe$ est égal à $x_2.prefixe$ et $x_1.postfixe$ est égal à $x_2.postfixe$

x_1/x_2 est vrai si :

i. $x_1.localDocId$ est égal à $x_2.localDocId$ et

ii. $x_1.prefixe$ est égal à $x_2.prefixe$ et

iii. $x_1.postfixe$ est égal à $x_2.postfixe$

5. Egalement, faisons une remarque à propos de la jointure externe. Supposons que nous effectuons $IS_1 \bowtie\!\sqsubset_p IS_2$.

   Quand aucun tuple de IS_1 ne joint avec un tuple x_2 de IS_2, alors, dans le résultat, apparaît l'élément suivant :

$$x_2 \circ [localDocId : NULL, prefixe : -1, postfixe : -1, niveau : -1]$$

4.2.1 Quelques exemples simples de FtiScan

Nous allons donner ici quelques exemples simples de FtiScan et le plan d'évaluation qui leur est associé.

L'opérateur Map

La figure 4.2 donne, dans sa partie gauche, un exemple d'arbre de motif et, dans sa partie droite, l'expression algébrique correspondante. Les documents recherchés sont ceux qui contiennent au moins un élément *peinture*. Ce FtiScan renvoie un ensemble de n-uplets [d:LocalDocId,_pe:entier].

Les différents opérateurs de jointure

La figure 4.3 représente trois exemples d'utilisation de jointure.

De haut en bas, le premier donne un exemple de jointure, le deuxième de semi-jointure et le troisième de jointure externe.

L'opérateur de sélection σ

Cet opérateur ne sert que dans un cas très précis : quand on cherche les documents qui possèdent un certain élément comme racine.

La figure 4.4 donne un exemple de tel FtiScan. Les documents recherchés possèdent l'élément peinture comme racine.

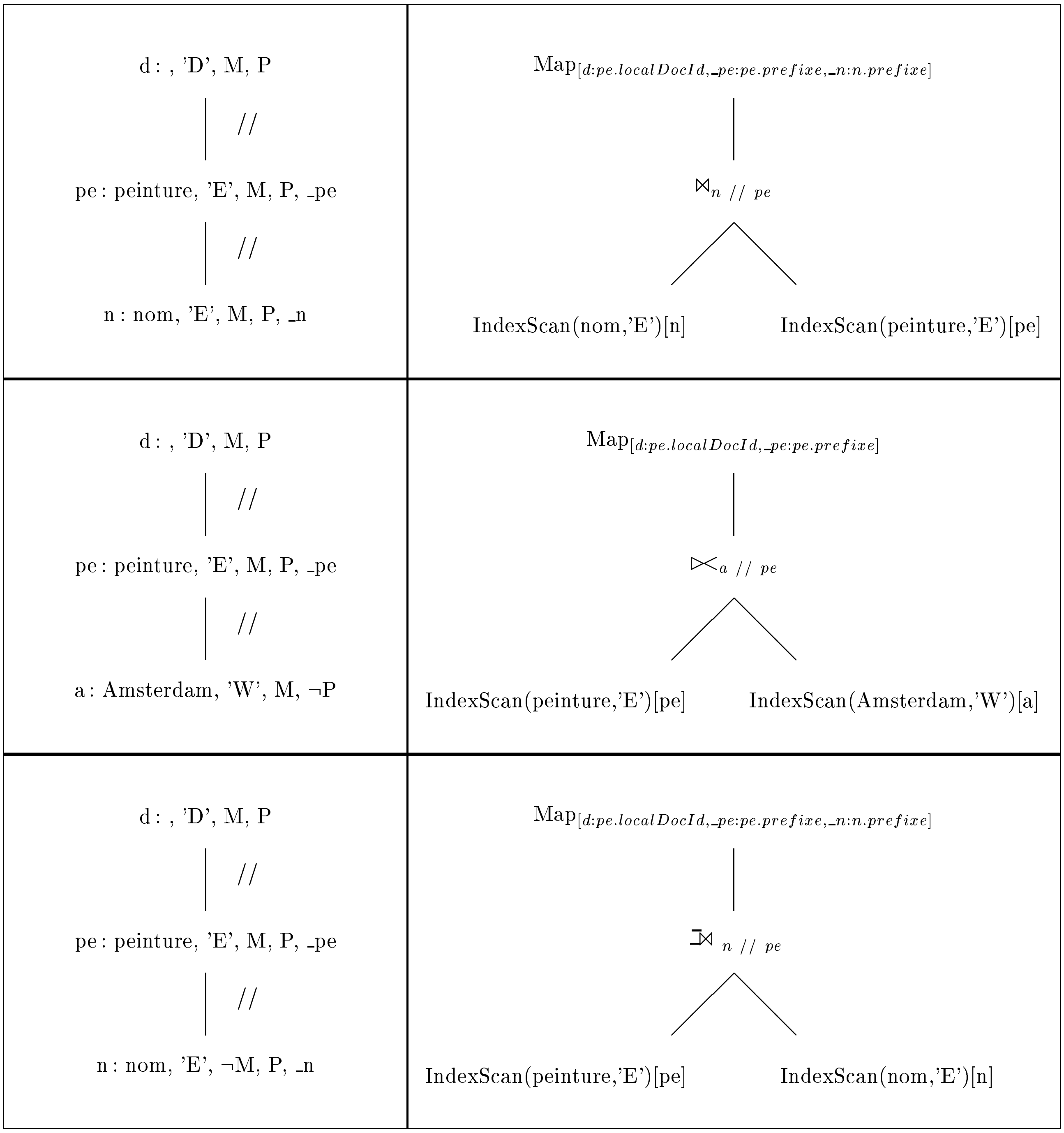


FIG. 4.3 – *Trois jointures différentes*

d : , 'D', M, P / pe : peinture, 'E', M, P,_pe	$\mathrm{Map}_{[d:pe.localDocId,\_pe:pe.prefixe]}$ $\sigma_{pe.niveau\,=\,1}$ IndexScan(peinture,'E')[pe]

FIG. 4.4 – *L'opérateur de sélection*

4.2.2 Problème posé

Etant donné ce que nous venons de voir, un FtiScan se représente algébriquement par un arbre de jointures.

Le problème est donc de trouver, parmi tous les arbres de jointures qui permettent de le représenter, celui qui va s'évaluer le plus rapidement possible.

Plus précisément, deux questions se posent : (1) Comment évaluer chaque jointure? (2) Comment réordonner les jointures?

C'est ce que nous allons voir tout de suite.

4.3 Solution proposée

4.3.1 Choix d'un algorithme de jointure

Les index interrogés tiennent en mémoire centrale. De ce fait, les jointures s'effectuent entre des ensembles qui tiennent en mémoire centrale. Nous avons donc choisi d'évaluer chacune d'elles par un algorithme par boucles imbriquées pour ne pas avoir à construire de structures intermédiaires telles que table de hachage ou ensemble trié. De plus, un arbre de jointures est évalué en pipeline. Comme nous le disions dans le chapitre précédent, notre implantation utilise les index d'Altavista [Alt]. Avec cette technologie, nous ne maîtrisons pas l'ordre dans lequel les informations associées à chaque mot indexé sont stockées. Nous n'avons donc pas pu implanter d'algorithme efficace qui aurait pu exploiter cet ordre.

4.3.2 Choix d'une heuristique

Pour réordonner les jointures, nous utilisons une heuristique [Moe]. En un nombre restreint d'examens de l'arbre de motif, celle-ci choisit, parmi tous les arbres de jointure possibles, celui où la somme des cardinalités des résultats des jointures intermédiaires est proche du minimal ou, quand la chance est au rendez-vous, minimale. Cette somme est une bonne fonction de coût pour comparer deux arbres de jointures. En effet, comme les ensembles joints tiennent en mémoire centrale, c'est le nombre d'instructions élémentaires qu'il faut minimiser. Chaque jointure s'effectue par boucles imbriquées. Or, le nombre d'instructions élémentaires nécessaires pour effectuer la jointure entre deux ensembles de cardinalité n et p est $n \times p$. Par conséquent, plus les jointures se font entre des ensembles de petite taille, moins l'algorithme sera coûteux en termes d'instructions élémentaires.

4.4 L'heuristique

Dans cette section, nous allons voir quelles sont les propriétés des jointures en termes de cardinalité. L'heuristique exploite ces propriétés.

4.4.1 Propriétés des jointures en termes de cardinalité

Hypothèse

Nous allons voir les propriétés des jointures en termes de cardinalités pour des documents dont la structure n'est pas récursive. Dans ceux-ci, une balise A ne peut donc pas avoir plusieurs balises ancêtres B.

Notation

Dans un document XML, chaque balise a un unique parent (un document se représente sous forme d'arbre). De plus, une balise peut avoir, comme descendants, plusieurs balises identiques. Par exemple, dans la figure 3.2, l'élément *peintures* a deux éléments fils *peinture*.

Les différentes jointures que nous avons évoquées ci-dessus s'effectuent donc entre des ensembles reliés par une association 1:1 (joindre les balises peinture avec les balises nom) ou 1:n (joindre les balises peintures avec les balises peinture).

Dans la suite, nous allons considérer les trois types de jointures (jointure normale, semi-jointure, jointure externe) en distinguant ces deux cardinalités d'association. Pour simplifier les notations, nous adopterons les notations suivantes :

- **IndexScan(ancêtre)** représente l'un des ensembles joints et correspond à l'ensemble des informations récupérées dans l'index pour l'ancêtre dans le prédicat de jointure.

- **IndexScan(descendant)** représente l'autre ensemble joint et correspond à l'ensemble des informations récupérées dans l'index pour le descendant dans le prédicat de jointure.

- **Join**$_{1:1}$ **(resp. Join**$_{1:n}$**)** représente $IndexScan(ancêtre) \bowtie IndexScan(descendant)$ quand l'ancêtre ne peut avoir qu'un seul descendant (resp. l'ancêtre peut avoir plusieurs descendants).

- **SemiJoinAncêtre**$_{1:1}$ **(resp. SemiJoinAncêtre**$_{1:n}$**)** représente

$$IndexScan(ancêtre) \rtimes IndexScan(descendant)$$

  quand l'ancêtre ne peut avoir qu'un seul descendant (resp. l'ancêtre peut avoir plusieurs descendants).

- **SemiJoinDescendant**$_{1:1}$ **(resp. SemiJoinDescendant**$_{1:n}$**)** représente

$$IndexScan(ancêtre) \ltimes IndexScan(descendant)$$

  quand l'ancêtre ne peut avoir qu'un seul descendant (resp. l'ancêtre peut avoir plusieurs descendants).

- **OuterJoinAncêtre**$_{1:1}$ **(resp. OuterJoinAncêtre**$_{1:n}$**)** représente

$$IndexScan(ancêtre) ⟕ IndexScan(descendant)$$

- L'autre type de jointure externe n'est pas considérée car elle ne peut pas exister.

Cardinalités des jointures

$\|E\|$ désigne la cardinalité de l'ensemble E. Les jointures ont les propriétés suivantes en termes de cardinalité.

$$\|Join_{1:1}\| \leq min(\|IndexScan(ancêtre)\|, \|IndexScan(descendant)\|) \quad (4.1)$$
$$\|Join_{1:n}\| \leq \|IndexScan(descendant)\| \quad (4.2)$$
$$\|SemiJoinAncêtre_{1:1}\| \leq min(\|IndexScan(ancêtre)\|, \|IndexScan(descendant)\|) \quad (4.3)$$
$$\|SemiJoinAncêtre_{1:n}\| \leq min(\|IndexScan(ancêtre)\|, \|IndexScan(descendant)\|) \quad (4.4)$$
$$\|SemiJoinDescendant_{1:1}\| \leq min(\|IndexScan(ancêtre)\|, \|IndexScan(descendant)\|) \quad (4.5)$$
$$\|SemiJoinDescendant_{1:n}\| \leq \|IndexScan(descendant)\| \quad (4.6)$$
$$\|OuterJoinAncêtre_{1:1}\| \leq \|IndexScan(ancêtre)\| \quad (4.7)$$
$$\|OuterJoinAncêtre_{1:n}\| \leq \|IndexScan(ancêtre)\| + \|IndexScan(descendant)\| \quad (4.8)$$

Expliquons chacune de ces inégalités.

- inégalité 4.1 : Prenons un exemple. Supposons que nous joignons des balises *peinture* avec des balises *nom*. L'association 1:1 indique qu'un nom ne peut caractériser qu'une peinture. Supposons qu'il y a 400 codages pour la balise *peinture* et 1000 codages pour la balise *nom* dans l'index interrogé. Au maximum, le nombre d'éléments dans le résultat de la jointure sera 400. En effet, une balise *nom* ne peut avoir qu'une seule balise parent *peinture*. S'il y a 1000 codages pour la balise *peinture* et 400 codages pour la balise *nom*, le nombre d'éléments dans le résultat de la jointure sera également 400.

- inégalité 4.2 : Supposons que nous joignons des balises *société* avec des balises *produit*. Une société peut être caractérisée par plusieurs produits. Supposons qu'il y a 400 codages pour la balise *société* et 1000 codages pour la balise *produit* dans l'index interrogé. Au maximum, le nombre d'éléments dans le résultat de le jointure sera 1000. En effet, chacun des produits peut joindre avec une société.

- inégalité 4.3 : L'explication est la même que pour l'inégalité 4.1.

- inégalité 4.4 : Même si plusieurs balises *produit* joignent avec une balise *société*, la balise *société* n'apparaîtra qu'une seule fois dans le résultat.

- inégalité 4.5 : C'est toujours la même explication que pour l'inégalité 4.1.

- inégalité 4.6 : Si plusieurs balises *produit* joignent avec une balise *société*, il y a un élément dans le résultat pour chacune de ces balises *produit*.

- inégalité 4.7 : Cf explication de l'inégalité 4.1.

- inégalité 4.8 : Joignons des balises *société* avec des balises *produit*. Supposons qu'il y a 1000 codages pour la balise *société* et 400 codages pour la balise *produit*. La cardinalité de la jointure est maximale quand une société joint avec tous les produits (il y a donc 400 couples (société,produit) dans le résultat) et que les 999 autres sociétés ne sont caractérisées par aucun produit. Il y a 999 couples (société,NULL) dans le résultat.

Conlusion

Dans Xyleme, quand un document est indexé, aucune analyse de la DTD correspondante, si elle existe, n'est effectuée. Il n'est donc pas possible de savoir la relation qui lie les ensembles joints.

Par conséquent, dans l'heuristique, chaque jointure n'est pas différenciée suivant l'association qui lie les deux ensembles joints.

De plus, les documents dont la structure est récursive sont rares.

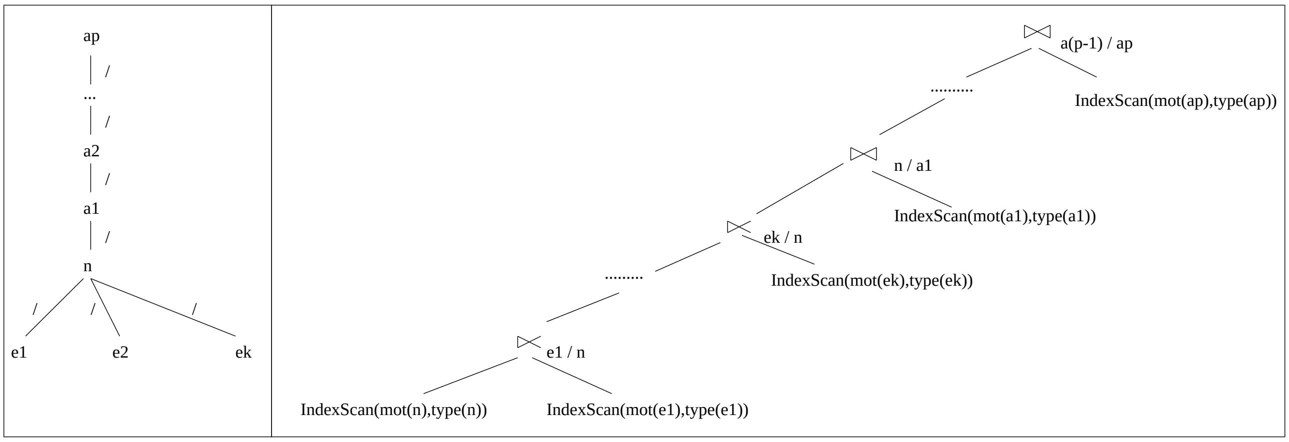


FIG. 4.5 – *Conséquence des propriétés des jointures*

Bien que les propriétés des jointures en termes de cardinalité ne soient pas vérifiées quand les documents ont une structure récursive, nous utilisons, dans l'heuristique, comme approximations des majorants de la cardinalité du résultat de chaque type de jointure (*Join*, *SemiJoinAncêtre*, *SemiJoinDescendant*, *OuterJoinAncêtre*), les suivantes :

1. **pour Join :** $\|IndexScan(descendant)\|$
2. **pour SemiJoinAncêtre :** $min(\|IndexScan(ancêtre)\|, \|IndexScan(descendant)\|)$
3. **pour SemiJoinDescendant :** $\|IndexScan(descendant)\|$
4. **pour OuterJoinAncêtre :** $\|IndexScan(ancêtre)\| + \|IndexScan(descendant)\|$

4.4.2 Description de l'heuristique

Conséquence des propriétés sur les jointures

Soit un arbre de motif A.
Soit n, un noeud de cet arbre.
Soit $cardinalitéIS(n) = \|IndexScan(mot(n), type(n))\|$.
Supposons que nous effectuons le FtiScan avec, comme arbre de motif, le sous-arbre de A, noté $ST(n, A)$, constitué des noeuds suivants :

– le noeud n

– chacun des noeuds enfants de n, notés $e_1, \ldots, e_k$, tels que, pour i de 1 à k, (1) $estProjeté(e_i)$ est faux, (2) e_i n'a pas d'enfants et (3) $cardinalitéIS(e_i) \geq cardinalitéIS(n)$

– chacun des noeuds ancêtres de n, notés $a_1, \ldots, a_p$ (plus i est petit, plus l'ancêtre a_i est jeune)

Supposons, de plus, que ce FtiScan est évalué comme il est indiqué dans la figure 4.5. A gauche est représenté l'arbre de motif. A droite est représenté l'arbre de jointures. Remarquons que nous avons, dans l'exemple, annoté chaque arc de l'arbre de motif par '/'. Un mélange de '/' et de '//' aurait pu, en fait, annoter les arcs. Sont d'abord effectuées les semi-jointures entre IndexScan(mot(n),type(n)) et chacun des IndexScan(mot(e_i),type(e_i)) pour i de 1 à k, puis, les jointures (nous allons le voir, certaines peuvent être des jointures externes) avec chacun des IndexScan(mot(a_i),type(a_i)) pour i de 1 à p.

Dans ces conditions, la cardinalité du résultat du FtiScan est majorée par :

1. si $estObligatoire(n)$ est vrai,
$$cardinalitéIS(n)$$

2. si $estObligatoire(n)$ est faux,

$$cardinalitéIS(n) + \sum_{p \text{ ancêtre de } n,\ \neg\ estObligatoire(p)} cardinalitéIS(p) + cardinalitéIS(q)$$

   avec q le plus jeune ancêtre de n tel que $estObligatoire(q)$ est vrai.

Expliquons pourquoi.
Quand $estObligatoire(n)$ est vrai,

1. les k premières jointures effectuées sont des semi-jointures. Comme nous venons de le voir, la cardinalité de telles semi-jointures (projection de l'information sur l'ancêtre dans le prédicat de jointure) est majorée par le minimum des cardinalités des deux ensembles joints. Dans notre cas, vu que $cardinalitéIS(e_i) \geq cardinalitéIS(n)$ pour i de 1 à k, la cardinalité du résultat de cette succession de semi-jointures est majorée par $cardinalitéIS(n)$.

2. les p autres jointures effectées sont des jointures "normales". Comme nous venons de le voir, la cardinalité d'une jointure est majorée par la cardinalité de l'ensemble joint qui représente le descendant dans le prédicat de jointure. Par conséquent, la cardinalité de la succession de jointures qui évalue le FtiScan est majorée par $cardinalitéIS(n)$.

Par contre, quand $estObligatoire(n)$ est faux,

1. aucune semi-jointure ne peut être effectuée. En effet, chaque enfant e du noeud n vérifie $estObligatoire(e)$ est faux, donc, $estProjeté(e)$ est vrai.

2. les premières jointures, dans la succession de jointures effectuée, peuvent être des jointures externes. En effet, prenons, par exemple, un noeud n qui possède quatre ancêtres : a_1, a_2, a_3 et a_4. Il se pourrait, par exemple, que (1) a_1 (resp. a_2) vérifie $estObligatoire(a_1)$ (resp. $estObligatoire(a_2)$) est faux et (2) a_3 (resp. a_4) vérifie $estObligatoire(a_3)$ (resp. $estObligatoire(a_4)$) est vrai. Dans la succession de jointures effectuée, les trois premières jointures seraient donc des jointures externes, la dernière, une jointure "normale". Nous l'avons vu, (i) la cardinalité du résultat d'une jointure externe est majorée par la somme des cardinalités des deux ensembles joints et (ii) la cardinalité d'une jointure est majorée par la cardinalité de l'ensemble joint qui représente le descendant dans le prédicat de jointure. Ceci explique la formule qui calcule le majorant dans cette situation.

Principe de l'heuristique

Pour évaluer un FtiScan avec un arbre de motif A, l'heuristique construit récursivement un arbre de jointures qui est une succession de jointures entre les FtiScans correspondant à des sous-arbres de A. Chaque FtiScan est choisi de façon à minimiser la somme des résultats des jointures intermédiaires.

- **Au premier appel :** elle choisit un noeud n_0 dans A tel que la cardinalité estimée du FtiScan dont l'arbre de motif est $ST(n_0, A)$ est minimale. Ce FtiScan est le premier dans la succession de jointures.

- **Aux appels suivants :** tant qu'il en existe, elle choisit un noeud n_0 dans A, qui n'a pas été choisi dans un sous-arbre précédent, tel que la cardinalité estimée de la jointure entre le FtiScan dont l'arbre de motif est $ST(n_0, A)$, sans les noeuds choisis aux appels précédents, et les jointures précédentes est minimale. Nous appelons $ST'(n_0, A)$, le sous-arbre $ST(n_0, A)$, sans les noeuds choisis aux appels précédents. L'arbre de jointures construit à la fin de chaque appel est la jointure entre le FtiScan dont l'arbre de motif est $ST'(n_0, A)$ et les jointures précédentes.

Par conséquent, à chaque appel, les jointures effectuées sont celles qui retournent le moins d'éléments.

Un exemple

Pour clarifier nos propos, reprenons l'arbre de motif, noté A, de la figure 4.1. Supposons que, dans l'index interrogé, le nombre de codages correspondant à chaque entité est :

$$\begin{aligned} \|IndexScan(Amsterdam,'W')\| &= 20 \\ \|IndexScan(artiste,'EA')\| &= 100 \\ \|IndexScan(musee,'E')\| &= 5 \\ \|IndexScan(nom,'E')\| &= 2000 \\ \|IndexScan(peinture,'E')\| &= 400 \end{aligned}$$

La figure 4.6 montre la succession de jointures entre FtiScans choisie par l'heuristique pour évaluer le FtiScan correspondant à A. Dans cette figure, nous indiquons également un majorant pour la cardinalité du premier FtiScan et des jointures suivantes.

Voyons comment l'heuristique se déroule.

1. Au premier appel : le noeud n_0 choisi est celui correspondant à l'entité *musee*. Le premier FtiScan a, comme arbre de motif, $ST(n_0, A)$, càd. le sous-arbre de A composé des noeuds correspondant aux entités *musee*, *Amsterdam* et *peinture*. La cardinalité du résultat de ce FtiScan est majorée par $\|IndexScan(musee,'E')\|$, càd. 5. n_0 a été choisi car la cardinalité estimée de ce FtiScan est minimale.

2. Au deuxième appel : le noeud n_0 choisi est celui correspondant à l'entité *artiste*. $ST'(n_0, A)$ est constitué d'une racine et du noeud correspondant à l'entité *artiste*. Le noeud parent de ce dernier a en effet déjà été choisi précédemment. C'est une jointure externe entre le FtiScan dont l'arbre de motif est $ST'(n_0, A)$ et le FtiScan précédent qui est effectuée. En effet, $estObligatoire(n_0)$ est faux. La cardinalité du résultat de cette jointure est majorée par la somme de la cardinalité estimée de la jointure précédente (càd. 5) et la cardinalité estimée du FtiScan dont l'arbre de motif est $ST'(n_0, A)$ (càd. 100).

3. Au troisième appel : le seul noeud restant est celui correspondant à l'entité *nom*. C'est une jointure qui est effectuée dont la cardinalité du résultat est majorée par $\|IndexScan(nom,'E')\|$, càd. 2000.

Ce FtiScan est optimal car les jointures s'effectuent entre des ensembles choisis pour leur petite taille. En effet, chaque jointure est évaluée avec un algorithme par boucles imbriquées. Comme les ensembles joints tiennent en mémoire centrale, c'est le nombre d'instructions élémentaires que nous voulons minimiser. Or, le nombre d'instructions élémentaires pour évaluer la jointure entre deux ensembles E et F de cardinalités respectives n et p avec un algorithme par boucles imbriquées est $n \times p$. La somme des cardinalités des jointures intermédiaires donne une bonne idée du nombre d'instructions élémentaires nécessaires.

Donnons un majorant de la somme des cardinalités des jointures intermédiaires pour la succession de jointures de la figure 4.6. Il vaut

$$5 + 5 + 105 + 2000 = 2115$$

Pourquoi?

1. Le premier FtiScan s'évalue grâce à deux jointures successives. La première est une semi-jointure entre IndexScan(Amsterdam,'W') et IndexScan(musee,'E'). La cardinalité du résultat de cette jointure est majorée par 5, le minimum des cardinalités des deux ensembles joints. La seconde joint le résultat de la jointure précédente avec IndexScan(peinture,'E'). La cardinalité du résultat de cette jointure est encore majorée par 5 (cardinalité de l'ensemble joint représentant le descendant dans le prédicat de jointure).

2. Ensuite, une jointure externe joint le résultat des jointures précédentes avec IndexScan(artiste,'EA'). La cardinalité de son résultat est majorée par la somme des cardinalités des deux ensembles joints, càd. $100 + 5 = 105$.

3. Enfin, une jointure joint le résultat des jointures précédentes avec IndexScan(nom,'E'). La cardinalité du résultat de cette jointure est majorée par la cardinalité de l'ensemble joint représentant le descendant dans le prédicat de jointure, càd. IndexScan(nom,'E'), càd. 2000.

Montrons que cette somme est plus petite que pour une autre succession de jointures entre FtiScans. La figure 4.7 représente une autre succession possible de FtiScans permettant d'évaluer le FtiScan correspondant à A.

Avec cette solution, un majorant de la somme des cardinalités des jointures intermédiaires est

$$2000 + 2100 + 5 + 5 = 4110$$

Cette solution est donc plus coûteuse en nombres d'instructions élémentaires. En effet,

1. Le premier FtiScan s'évalue grâce à une jointure entre IndexScan(peinture,'E') et IndexScan(nom,'E'). La cardinalité du résultat de cette jointure est majorée par 2000 (cardinalité de l'ensemble joint représentant le descendant dans le prédicat de jointure).

2. La deuxième jointure est une jointure externe entre IndexScan(artiste,'EA') et le résultat de la jointure précédente. La cardinalité de son résultat est majorée par la somme des cardinalités des deux ensembles joints, càd. $2000 + 100 = 2100$.

3. Le troisième FtiScan s'évalue grâce à une semi-jointure entre IndexScan(Amsterdam,'W') et IndexScan(musee,'E'). La cardinalité du résultat de cette semi-jointure est majorée par 5, le minimum des cardinalités des deux ensembles joints.

4. La cardinalité de la dernière jointure entre le résultat de ce FtiScan et le résultat des jointures précédentes est majorée par 5, la cardinalité de l'ensemble joint représentant le descendant dans le prédicat de jointure.

Maintenant, voyons plus précisément comment cette heuristique est implantée.

Algorithme

Nous évaluons le FtiScan correspondant à l'arbre de motif A. Nous allons le voir, afin de choisir, à chaque appel, le noeud n_0 qui convient, nous nous aidons d'un arbre de motif de structure identique à l'arbre de motif A. Nous appelons cet arbre, arbre d'heuristique, et nous le notons AH. Dans ce dernier, chaque noeud n est annoté, entre autres, par un entier que nous appelons $nombreDHeuristique(n)$.

A chaque appel, le noeud n_0 choisi par l'heuristique est un noeud qui n'apparaît pas dans un sous-arbre de A choisi précédemment et tel que $nombreDHeuristique(n_0)$ est minimal. En effet, cet entier représente :

- Au premier appel : un majorant de la cardinalité du FtiScan dont l'arbre de motif est $ST(n_0, A)$
- Aux appels suivants : un majorant de la cardinalité de la jointure entre le FtiScan dont l'arbre de motif est $ST'(n_0, A)$ et les jointures précédemment construites

Initialisation : construction d'un arbre d'heuristique

Comme nous le disions ci-dessus, la première étape consiste à construire un arbre d'heuristique, noté AH. Celui-ci est un arbre de motif de structure identique à l'arbre de motif A. Dans AH, chaque noeud n possède les informations suivantes :

1. un pointeur vers le noeud correspondant dans A. Celui-ci est désigné par $c(n)$ (c pour "correspondant").

2. un entier qui vaut $\|IndexScan(mot(c(n)), type(c(n)))[x]\|$. Nous le notons, comme précédemment, $cardinalitéIS(n)$. Il correspond au nombre de codages, dans l'index interrogé, correspondant à l'entité désignée par $mot(c(n))$.

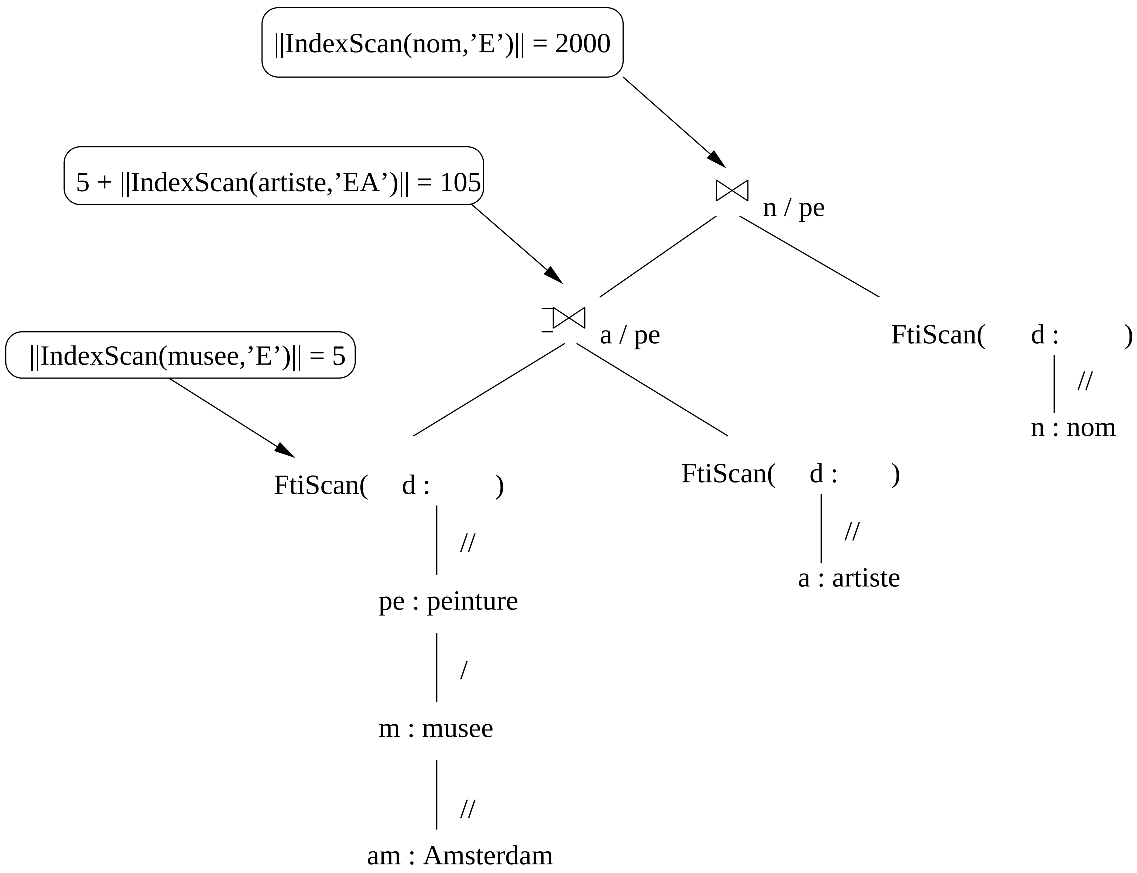


FIG. 4.6 – *Résultat de l'heuristique*

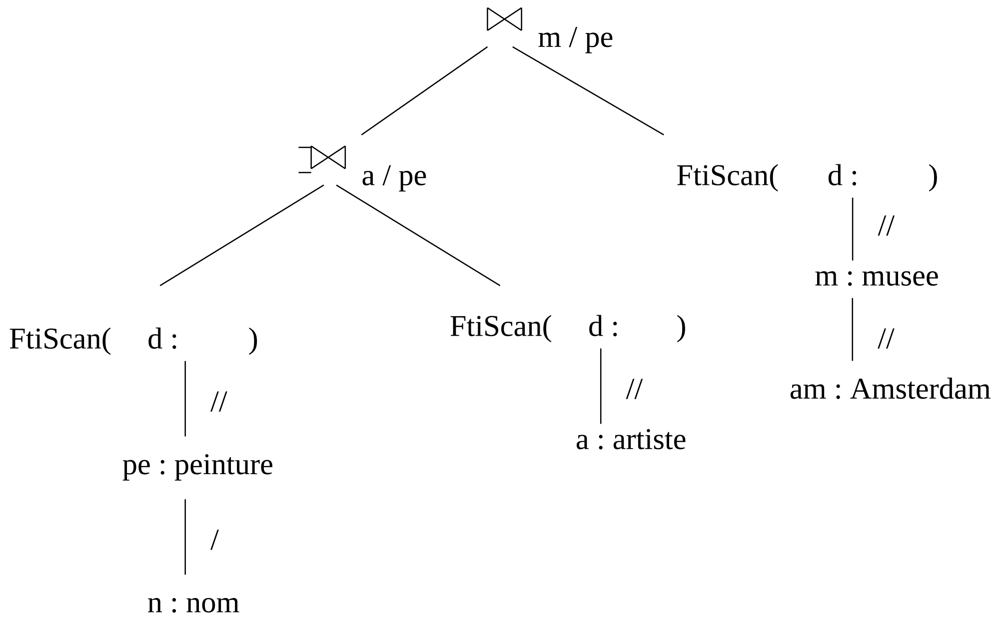


FIG. 4.7 – *Autre succession de jointures possible*

3. un autre entier, appelé $nombreDHeuristique(n)$. Pour un noeud n, alors que la valeur de $cardinalitéIS(n)$ reste constante au cours du temps, la valeur de $nombreDHeuristique(n)$ varie.
   - Pour les noeuds n tels que $estObligatoire(n)$ est vrai :
     - Tant qu'ils n'apparaîssent pas dans un sous-arbre de A choisi précédemment :
       $$nombreDHeuristique(n) = cardinalitéIS(n)$$
       Cela représente la cardinalité maximale du FtiScan dont l'arbre de motif est $ST(n, A)$ ou de la jointure entre le FtiScan dont l'arbre de motif est $ST'(n, A)$ et les jointures construites aux appels précédents.
     - Dès que le contraire est vrai : $nombreDHeuristique(n)$ est le nombre d'occurrences relatives à n dans le résultat
   - Pour les noeuds n tels que $estObligatoire(n)$ est faux : p désigne le noeud parent de n.
     - Tant qu'ils n'apparaîssent pas dans un sous-arbre de A choisi précédemment :
       $$nombreDHeuristique(n) = cardinalitéIS(n) + nombreDHeuristique(p)$$
       Cela représente la cardinalité maximale du FtiScan dont l'arbre de motif est $ST(n, A)$ ou de la jointure entre le FtiScan dont l'arbre de motif est $ST'(n, A)$ et les jointures construites aux appels précédents.
     - Dès que le contraire est vrai : $nombreDHeuristique(n)$ est le nombre d'occurrences relatives à n dans le résultat

La figure 4.8 montre l'arbre de motif A et l'arbre d'heuristique AH correspondant. Notez que, pour le noeud n correspondant à l'entité *artiste*, $nombreDHeuristique(n)$ vaut 500 (100 + 400).

Construction de l'arbre de jointures

La figure 4.9 montre l'arbre de jointures trouvé par l'heuristique.

A chaque appel, le noeud n_0 qui est choisi par l'heuristique est un noeud de A qui n'apparaît pas dans un sous-arbre de A choisi précédemment et tel que $nombreDHeuristique(n_0)$ est minimal.

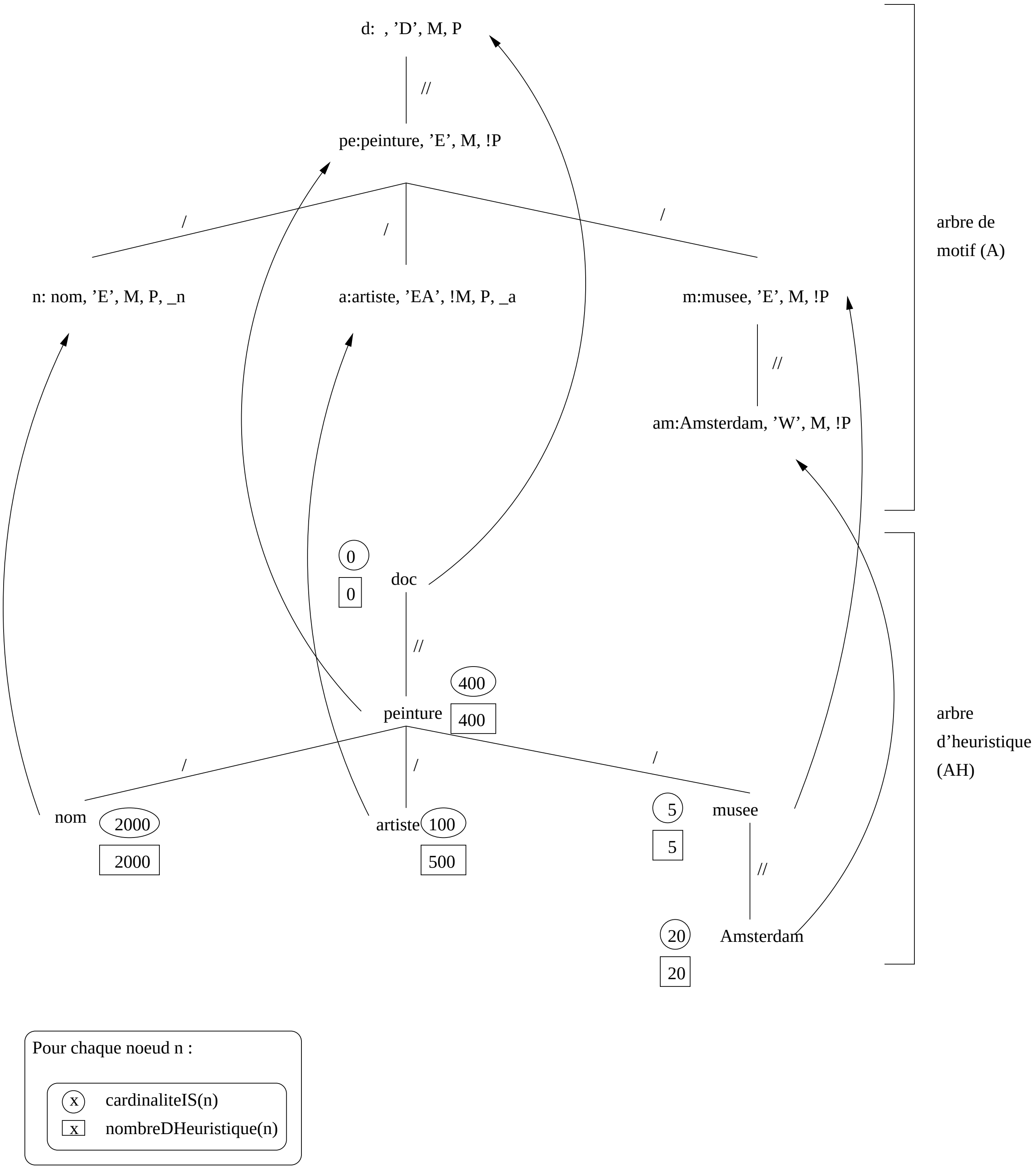


FIG. 4.8 – *L'arbre de motif A et l'arbre d'heuristique correspondant AH*

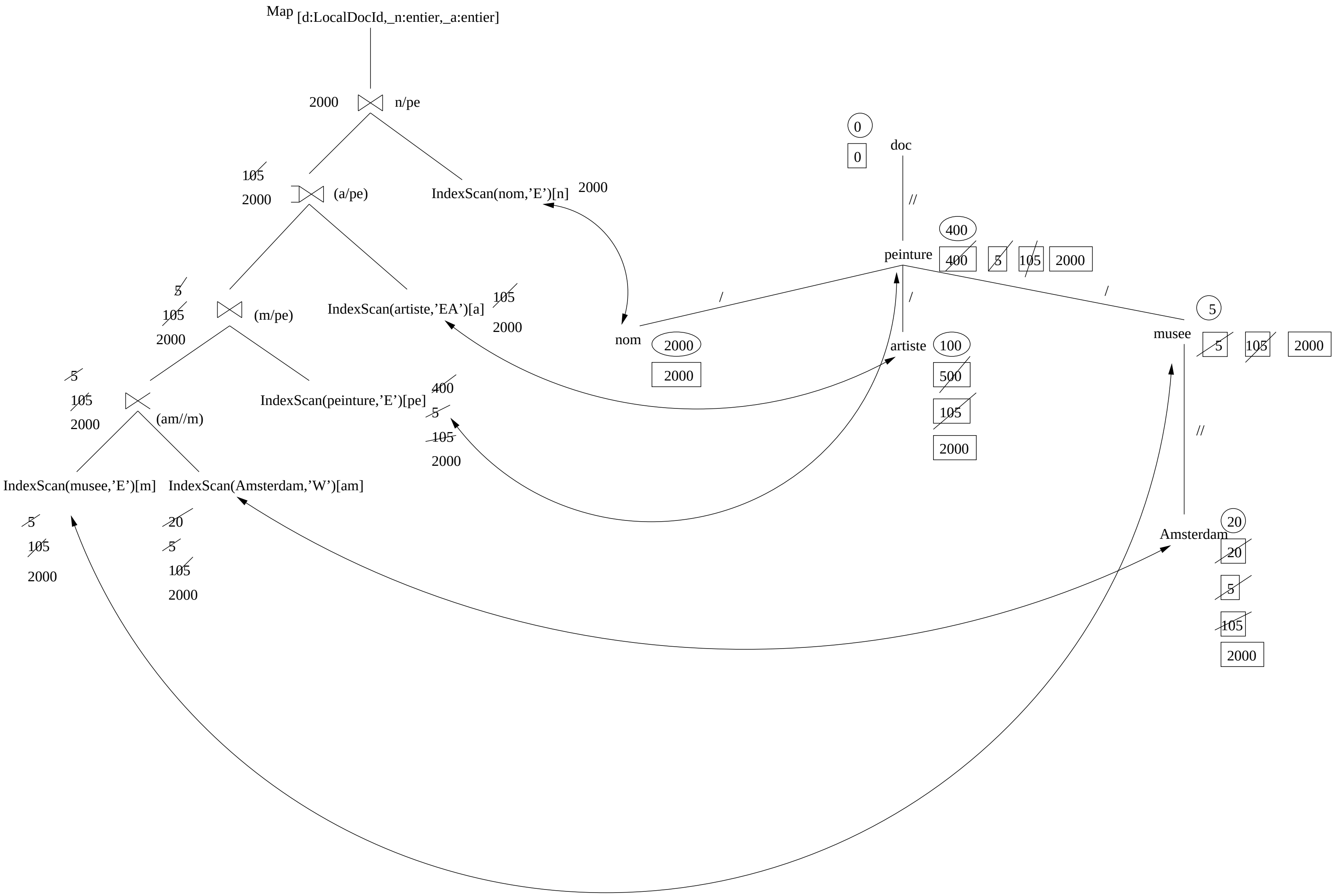


FIG. 4.9 – *L'arbre de jointures trouvé par l'heuristique*

A droite est représenté l'arbre d'heuristique AH. Dans celui-ci, chaque noeud n est annoté par des entiers :

- l'un deux, entouré par un cercle, représente $cardinalitéIS(n)$.
- les autres, entourés par des rectangles, représentent les valeurs successives de $nombreDHeuristique(n)$ lors de l'exécution de l'heuristique.

A gauche est représenté l'arbre de jointures trouvé par l'heuristique. Chaque opération, dans cet arbre, est annotée par un entier. Celui-ci représente le nombre maximal d'éléments lui correspondant dans le résultat. Dans la figure, sont indiquées les valeurs successives de cet entier lors de l'exécution de l'heuristique. Expliquons comment il est construit :

1. Premier appel :
   (a) Le noeud n_0 dans l'arbre de motif pour lequel $nombreDHeuristique(n_0)$ est minimal et qui n'apparaît pas dans un sous-arbre choisi précédemment est celui qui correspond à l'entité musee.
   (b) Comme le noeud enfant e de n_0 vérifie $estProjeté(e)$ est faux et qu'il n'a pas d'enfants, la première jointure effectuée est une semi-jointure entre $IndexScan(musee,'E')[m]$ et $IndexScan(Amsterdam,'W')[am]$.
   (c) Ensuite, cette semi-jointure est jointe avec $IndexScan(peinture,'E')[pe]$. En effet, *peinture* est l'entité correspondant au noeud parent du noeud correspondant à l'entité *musee*.
   (d) Le nombre maximum d'éléments dans le résultat de cette jointure est 5. Par conséquent, le nombre maximum d'éléments dans le résultat correspondant à toutes les autres opérations dans l'arbre de jointures construit est 5.
   (e) Dans l'arbre de motif, la valeur de $nombreDHeuristique(n)$ pour les noeuds n correspondant aux entités *peinture* et $Amsterdam$ devient 5. La valeur de $nombreDHeuristique(n)$ pour le noeud n correspondant à l'entité *artiste* est également modifiée. Elle vaut 105.
2. Deuxième appel :
   (a) Le noeud n_0 dans l'arbre de motif pour lequel $nombreDHeuristique(n_0)$ est minimal et qui n'apparaît pas dans un sous-arbre choisi précédemment est celui qui correspond à l'entité artiste. Il vaut 105. La nouvelle jointure qui est effectuée est une jointure externe entre $IndexScan(artiste,'EA')[a]$ et l'arbre de jointures construit précédemment.
   (b) Le nombre maximum d'éléments dans le résultat de cette jointure est 105. Par conséquent, le nombre maximum d'éléments dans le résultat correspondant à toutes les autres opérations dans l'arbre de jointures construit est 105.
   (c) Dans l'arbre de motif, la valeur de $nombreDHeuristique(n)$ pour les noeuds n correspondant aux entités peinture, musee et Amsterdam est modifiée en conséquence. Elle vaut 105.
3. Troisième appel :
   (a) Le noeud n_0 dans l'arbre de motif pour lequel $nombreDHeuristique(n_0)$ est minimal et qui n'apparaît pas dans un sous-arbre choisi précédemment est celui qui correspond à l'entité nom. Il vaut 2000. La nouvelle jointure qui est effectuée est une jointure entre $IndexScan(nom,'E')[a]$ et l'arbre de jointures construit précédemment.
   (b) Le nombre maximum d'éléments dans le résultat de cette jointure est 2000. Par conséquent, le nombre maximum d'éléments dans le résultat correspondant à toutes les autres opérations dans l'arbre de jointures construit est 2000.
   (c) Dans l'arbre de motif, la valeur de $nombreDHeuristique(n)$ pour chacun des noeuds n de cet arbre est modifiée en conséquence. Pour chacun des noeuds n de cet arbre, $nombreDHeuristique(n)$ vaut 2000.

Conclusion

Pour pouvoir répondre rapidement à des requêtes portant sur la structure d'un grand nombre de documents, nous avons proposé une méthode d'indexation plein-texte de documents XML. Traditionnellement, un index plein-texte permet de connaître la liste des documents dans lesquels un mot apparaît. Vues les requêtes que nous souhaitions évaluer, ceci était insuffisant. C'est pourquoi, à chaque mot indexé, nous avons fait correspondre une information - ou codage - qui indique non seulement le document dans lequel il se trouve mais aussi sa position dans ce document.

Celle-ci a été conçue pour répondre à trois critères. (1) D'abord, le codage doit être dense. Il doit permettre de stocker le maximum d'informations nécessaires dans le minimum de place. (2) Ensuite, le coût de la mise à jour des index ne doit pas être prohibitif. En effet, Xyleme recueille ses données du Web. Celles-ci sont constamment en changement. Par conséquent, les index risquent de subir très fréquemment des modifications. (3) Enfin, les index sont utilisés pour minimiser le nombre de lectures sur disque et pour récolter des informations afin d'accéder aux données sur disque. Les informations à stocker dans l'index sont donc très liées à la manière d'accéder aux données sur disque. Il doit y avoir correspondance.

Nous avons enfin montré comment, une fois les documents indexés, nous évaluons les requêtes. Chaque requête est, en fait, évaluée grâce à un arbre de jointures. Pour ne pas avoir à gérer de structures intermédiaires, nous avons choisi d'évaluer l'arbre en pipeline et chaque jointure par un algorithme par boucles imbriquées. Pour réordonner les jointures, nous avons choisi d'appliquer une heuristique qui, en un nombre restreint d'examens de l'arbre de motif, trouve, parmi tous les arbres de jointures possibles, celui où la somme des cardinalités des résultats des jointures intermédiaires est proche du minimal ou, quand la chance est au rendez-vous, minimale.

Afin d'implanter un prototype rapidement, nous avons utilisé les index d'Altavista. Ceci nous a permis de nous concentrer sur les problèmes réellement posés. Cependant, nous pensons qu'une implantation en partant de zéro nous permettrait d'obtenir de meilleures performances. Notamment, nous ne maîtrisons pas l'ordre dans lequel les codages associés à un mot apparaîssent dans l'index. Ceci est un inconvénient car, si nous avions le contrôle sur cet ordre, nous pourrions implanter un algorithme de jointures plus performant qui exploiterait celui-ci. Ceci accélérerait l'évaluation des requêtes.

Conclusion

Nous avons étudié comment optimiser des requêtes sur des données en forme d'arbres.

D'une part, nous avons considéré des arbres stockés dans le SGBD à objets O_2. Au sein de la division objet de la société Ardent Software (ancienne O_2 Technology), nous avons étudié les performances du système O_2 en réalisant un banc d'essais. Nous avons montré ses faiblesses concernant les gros accès associatifs et proposé des solutions pour y remédier. Nous avons montré que la taille des objets O_2 en mémoire est beaucoup trop importante pour permettre de bons accès associatifs. De plus, le système perd beaucoup trop de temps en allocations et désallocations. Nous avons aussi expliqué les avantages et les inconvénients de la navigation par rapport aux jointures pour évaluer des requêtes sur des bases stockant des structures arborescentes. Grâce à un banc d'essais, nous avons étudié l'influence de plusieurs paramètres : taille de la base, cardinalité de la relation 1:n, sélectivité des prédicats de sélection, organisation physique des données. Une des conclusions principales que nous avons tirée de nos expériences est que la navigation simple est sûrement profitable aux algorithmes sophistiqués de jointure lorsque les données conservent leur forme d'arbre sur le disque.

D'autre part, nous nous sommes intéressés à l'interrogation de documents XML. Les volumes de données, dans ce second cas, étaient plus importants et nous avons étudié comment utiliser des index pour permettre une interrogation efficace à grande échelle. Ce travail a été réalisé dans le cadre de l'action de recherche Xyleme, menée au projet Verso et en collaboration avec de nombreux groupes de recherche situés, entre autres, à l'Université de Mannheim, à l'Université d'Orsay et au CNAM. L'objectif était de concevoir et d'implanter un prototype d'entrepôt de données capable d'exploiter toutes les données XML qui peuvent être trouvées sur le Web. Le contexte s'est révélé être tout à fait différent de l'optimisation de requêtes dans le contexte objet. La taille des données interrogées est très importante, potentiellement, tout le Web. Celles-ci sont réparties. Les utilisateurs sont nombreux, exigeants et, en particulier, impatients.

Pour pouvoir répondre rapidement à des requêtes portant sur la structure d'un grand nombre de documents, nous avons proposé une méthode d'indexation plein-texte de documents XML. Traditionnellement, un index plein-texte permet de connaître la liste des documents dans lesquels un mot apparaît. Vues les requêtes que nous souhaitions évaluer, ceci était insuffisant. C'est pourquoi, à chaque mot indexé, nous avons fait correspondre une information - ou codage - qui indique non seulement le document dans lequel il se trouve mais aussi sa position dans ce document.

Celle-ci a été conçue pour répondre à trois critères. (1) D'abord, le codage doit être dense. Il doit permettre de stocker le maximum d'informations nécessaires dans le minimum de place. (2) Ensuite, le coût de la mise à jour des index ne doit pas être prohibitif. En effet, Xyleme recueille ses données du Web. Celles-ci sont constamment en changement. Par conséquent, les index risquent de subir très fréquemment des modifications. (3) Enfin, les index sont utilisés pour minimiser le nombre de lectures sur disque et pour récolter des informations afin d'accéder aux données sur disque. Les informations à stocker dans l'index sont donc très liées à la manière d'accéder aux données sur disque. Il doit y avoir correspondance.

Nous avons enfin montré comment, une fois les documents indexés, nous évaluons les requêtes. Chaque requête est, en fait, évaluée grâce à un arbre de jointures. Le problème était donc de savoir comment les réordonner et de choisir un algorithme pour les évaluer. Comme les ensembles joints tiennent en mémoire centrale, nous avons choisi d'évaluer chaque jointure avec un algorithme par boucles imbriquées pour ne pas avoir à construire de structure intermédiaire telle que table de hachage ou ensemble trié. Un arbre de jointures est évalué en pipeline. Les jointures sont réordonnées de façon à minimiser le nombre d'instructions élémentaires nécessaires pour les évaluer. Elle choisit, parmi tous les arbres de jointures possibles, celui où la somme des cardinalités des résultats des jointures intermédiaires est proche du minimal ou, quand la chance est au rendez-vous, minimale. Cette somme est une bonne fonction de coût pour comparer deux arbres de jointures. En effet, le nombre d'instructions élémentaires nécessaires pour effectuer la jointure entre deux ensembles de cardinalités n et p est $n \times p$. Par conséquent, plus les jointures se font entre des ensembles de petite taille, moins l'algorithme sera coûteux en termes d'instructions élémentaires.

Afin d'implanter un prototype rapidement, nous avons utilisé les index d'Altavista. Leurs performances n'ont pas encore pu être évaluées. Pour ce faire, nous pensons que des tests, grandeur nature, pourraient être réalisés. Ceux-ci permettraient (1) de voir le rapport de taille entre un index et les données auxquelles il correspond et (2) de se rendre compte de la rapidité de l'évaluation d'une interrogation.

L'utilisation de cette technologie nous a permis de nous concentrer sur les problèmes réellement posés. Cependant, nous pensons qu'il faudrait mieux réaliser notre propre implantation en partant de zéro. Notamment, nous pensons que, pour garantir de bonnes performances, nous devrions avoir le contrôle sur l'ordre dans lequel les codages associés à un mot apparaîssent dans un index.

Annexe A

Proposition d'algorithme

Nous proposons ici un algorithme de jointure auquel nous avions pensé. Malheureusement, les circonstances ont fait que nous n'avons pas pu l'étudier en détail. Dans ce chapitre, nous en décrirons donc uniquement les principes.

A.1 Hypothèses

Nous sommes dans les mêmes conditions que notre banc d'essais. Il y a deux collections Médecins et Patients reliées entre elles par une association 1:n. Nous évaluons la requête :

```
select f(m,pa)
from   m in Medecins, pa in m.clients
where pa.pat_id < k1 and m.med_id < k2
```

Pour être utilisé, cet algorithme suppose que le système indexe lui-même tous les objets de la collection Patients sur la page dans laquelle ils sont stockés. Dans la suite, cet index est noté **I**.

A.2 Principes de l'algorithme

Il utilise une table de hachage **TH**. Dans celle-ci, des objets de la collection Patients sont hachés sur leur propre Rid. Pour chacun de ces objets stockés dans la table de hachage, le ou les attributs projetés sont stockés.

Le principe de l'algorithme est le suivant :

```
TH := table vide

Pour chaque m dans Medecins tq. m.med_id < k2 Faire
    Pour chaque pa dans m.clients Faire
        Si pa appartient à TH Alors {
            /*la page dans laquelle pa se trouve a été déjà traitée*/
            Obtenir, dans TH, le ou les attributs de pa projetés
            Résultat + = {f(m,pa)}
            Enlever pa de TH }
        Sinon {
            Si pa.pat_id < k1 Alors
                /*nous lisons la page dans laquelle pa se trouve*/
                Résultat + = {f(m,pa)}

            numPage := numéro de la page dans laquelle pa se trouve
```

```
        Obtenir de l'index I tous les objets Patients se trouvant dans
        la page de numéro numPage

        Soient ensPatientsPageCourante l'ensemble de ces patients

        Pour chaque p dans ensPatientsPageCourante Faire
            Si p.pat_id < k1 Alors
                TH + = {p}
                /*ou, plus précisément, le ou les attributs concernant
                les patients qui sont projetés dans la requête */
                /*la jointure entre p et un médecin sera, éventuellement,
                exécutée dans une prochaine boucle*/
}
```

A.3 Ses avantages

Cet algorithme permet d'évaluer la jointure en minimisant le nombre d'E/S. En effet, il évite de lire de nombreuses fois une page contenant un ou plusieurs enregistrements relatifs à des patients. Chaque page contenant au moins un enregistrement relatif à un patient (resp. médecin) est lue une fois et une seule.

Par contre, l'interrogation de la table de hachage peut certainement se révéler très coûteuse si elle est de grande taille.

Malheureusement, nous n'avons pas eu le temps de réaliser des expériences pour se rendre compte des performances de cet algorithme.

Bibliographie

[ABD+89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented system manifesto. In *Proc. Int. Conf. on Deductive and Object-Oriented Databases (DOOD)*, Kyoto, Japan, December 1989.

[ABS99] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web : From Relations to Semi-structured Data and XML*. Morgan Kaufmann, 1999.

[ACC+97] S. Abiteboul, S. Cluet, V. Christophides, T. Milo, G. Moerkotte, and J. Siméon. Querying documents in object databases. *International Journal on Digital Libraries*, 1997.

[ACV+00] Vincent Aguilera, Sophie Cluet, Pierangelo Veltri, Dan Vodislav, and Fanny Wattez. Querying xml documents in xyleme. In *ACM SIGIR 2000 Workshop On XML and Information Retrieval*, 2000.

[AFS97] The Andrew File System, 1997.
`http://www.alw.nih.gov/Docs/AFS/AFS_toc.html/` .

[Alt] altavista.
`http://www.altavista.com`.

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

[Bab79] E. Babb. Implementing a relational database by means of specialized hardware. *ACM Trans. on Database Systems*, 1979.

[BFI91] K. Bennett, M.C. Ferris, and Y.E. Ioannidis. A genetic algorithm for database query optimization. In *Proceedings of the International Conference on Genetic Algorithms, San Diego*, pages 400–407, 1991.

[BJP98] Mokrane Bouzeghoub, Mireille Jouve, and Philippe Pucheral. *Modèles de stockage et d'accès : placement, indexation, navigation - collection : Les bases de données en question*. Hermès, 1998.

[CACS94] Vassilis Christophides, Serge Abiteboul, Sophie Cluet, and Michel Scholl. From structured documents to novel query facilities. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994*, pages 313–324. ACM Press, 1994.

[Cat97] R.G. Cattell. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann, 1997.

[CCM96] Vassilis Christophides, Sophie Cluet, and Guido Moerkotte. Evaluating queries with generalized path expressions. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 413–422. ACM Press, 1996.

[CCMS98] V. Christophides, S. Cluet, G. Moerkotte, and J. Siméon. Optimizing generalized path expressions using full text indexes. *Networking and Information Systems 1(2-3): 177-194*, 1998.

[CCS00] Vassilis Christophides, Sophie Cluet, and Jérôme Siméon. On wrapping query languages and efficient xml integration. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, volume 29, pages 141–152. ACM, 2000.

[CD92] Sophie Cluet and Claude Delobel. A general framework for the optimization of object-oriented queries. In Michael Stonebraker, editor, *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 383–392. ACM Press, 1992.

[CL90] M. J. Carey and G. Lapis. An incremental join attachment for starburst. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, Brisbane, Australia, 1990.

[Clu91] S. Cluet. *Langages et Optimisation de Requêtes pour SGBDOO.* PhD thesis, Université Paris XI, 1991.

[Clu98] S. Cluet. Designing oql: Allowing objects to be queried. *Information Systems, vol. 23, no. 5, pp. 279-305*, 1998.

[Clu99] S. Cluet. Bases de données : quelques clés pour l'ouverture vers le réseau, 1999. Habilitation à diriger des recherches.

[CM93] Sophie Cluet and Guido Moerkotte. Nested queries in object bases. In Catriel Beeri, Atsushi Ohori, and Dennis Shasha, editors, *Database Programming Languages (DBPL-4), Proceedings of the Fourth International Workshop on Database Programming Languages - Object Models and Languages, Manhattan, New York City, USA, 30 August - 1 September 1993*, Workshops in Computing, pages 226–242. Springer, 1993.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *CACM*, 13(6):377–387, 1970.

[Cod79] E. F. Codd. Extending the database relational model to capture more meaning. *TODS*, 4(4):397–434, 1979.

[CWN93] M. J. Carey, D. J. De Witt, and J. F. Naughton. The 007 benchmark. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, 1993.

[Der97] The Derby Schema, 1997.
`http://www.odbmsfacts.com/derby/1997.html/` .

[Exc] Excelon.
`http://www.odi.com/excelon`.

[FK99] Daniela Florescu and Donald Kossmann. Storing and querying xml data using an rdmbs. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[FKM00] Daniela Florescu, Donald Kossmann, and Ioana Manolescu. Integrating keyword search into xml query processing. In *WWW9 / Computer Networks 33(1-6): 119-135*, 2000.

[Gar99] Georges Gardarin. *Bases de données : objet et relationnel.* Eyrolles, 1999.

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

[GGT95] Georges Gardarin, Jean-Robert Gruser, and Zhao-Hui Tang. A cost model for clustered object-oriented databases. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 323–334. Morgan Kaufmann, 1995.

[GMUW00] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database System Implementation.* Prentice Hall, 2000.

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

[Goo] google.
`http://www.google.com`.

[Goo81] J.R. Goodman. An investigation of multiprocessor structures and algorithms for data base management. Electronics Research Laboratory Memorandum No. UCB/ERL M81/33, University of California, Berkeley, 1981.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[Gro98a] W3C DOM Working Group. Document object model (dom) level 1 specification, version 1.0, w3c recommendation. Technical report, World Wide Web Consortium, http://www.w3.org/TR/2000/WD-xmlquery-req-20000131, 1998.

[Gro98b] W3C XML Linking Working Group. W3c candidate recommendation, 3 july 2000. Technical report, World Wide Web Consortium, http://www.w3.org/TR/WD-xlink, 1998.

[Gro00a] W3C XML Linking Working Group. W3c candidate recommendation, 7 june 2000. Technical report, World Wide Web Consortium, http://www.w3.org/TR/WD-xptr, 2000.

[Gro00b] W3C XML Working Group. Extensible markup language (xml) 1.0. w3c recommendation 6 october 2000. Technical report, World Wide Web Consortium, http://www.w3.org/TR/REC-xml, 2000.

[HWM98] S. Helmer, T. Westmann, and G. Moerkotte. Diag-Join: An Opportunistic Join Algorithm for 1:N Relationship. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, New-York, 1998.

[IW87] Yannis E. Ioannidis and Eugene Wong. Query optimization by simulated annealing. In Umeshwar Dayal and Irving L. Traiger, editors, *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, California, May 27-29, 1987*, pages 9–22. ACM Press, 1987.

[KM94] A. Kemper and G. Moerkotte. Query optimization in object bases : Exploiting relational techniques. In G. Vossen, J.C. Freytag, and D. Maier, editors, *Proceedings of Dagstuhl Worshop on Query Processing in Object-Oriented, Complex-Object abd Nested Relation Databases.* Morgan Kaufmann, 1994.

[KM00] Carl-Christian Kanne and Guido Moerkotte. Efficient storage of xml data. In *International Conference on Data Engineering ICDE*, 2000.

[KTMO83] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of Hash to Data Base Machine and Its Architecture. *New Generation Computing, vol. 1, no. 1, p. 66-74*, 1983.

[LV91] Rosana S. G. Lanzelotte and Patrick Valduriez. Extending the search strategy in a query optimizer. In Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors, *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*, pages 363–373. Morgan Kaufmann, 1991.

[Mic99] Alain Michard. *XML : langage et applications.* Eyrolles, 1999.

[Moe] Guido Moerkotte. Building query compiler. http://pi3.informatik.uni-mannheim.de/staff/mitarbeiter/moer/myself.html.

[NSS86] S. Nahar, S. Sahni, and E. Shragowitz. Simulated annealing and combinatorial optimization. In *Proceedings of the 1986 ACM/IEEE Design Automation Conference, Las Vegas*, pages 293–299, 1986.

[Obja] Objectivity.
http://www.objectivity.com.

[Objb] Objectstore.
http://www.odi.com.

[Ont] Ontos.
http://www.ontos.com.

[Ora] Oracle XDK.
Oracle's XML Development Kit (XDK).

[Poe] Poet.
http://www.poet.com.

[RLS98] Jonathan Robie, Joe Lapp, and David Schach. Xml query language (xql). Technical report, World Wide Web Consortium, http://www.w3.org/TandS/QL/QL98/pp/xql.html, 1998.

[SAC+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In Philip A. Bernstein, editor, *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, May 30 - June 1*, pages 23–34. ACM, 1979.

[SC90] E. J. Shekita and M. J. Carey. A performance evaluation of pointer-based joins. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 300–311, Atlantic City, NJ, 1990.

[SG88] Arun N. Swami and Anoop Gupta. Optimization of large join queries. In Haran Boral and Per-Åke Larson, editors, *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, June 1-3, 1988*, pages 8–17. ACM Press, 1988.

[SO90] D. Straube and T. Ozsu. Queries and query processing in object-oriented database systems. *ACM Trans. on Information Systems*, 8(4), 1990.

[STZ+99] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 302–314. Morgan Kaufmann, 1999.

[SZ89] Gail M. Shaw and Stanley B. Zdonik. An object-oriented query algebra. In Richard Hull, Ronald Morrison, and David W. Stemple, editors, *Proceedings of the Second International Workshop on Database Programming Languages, 4-8 June, 1989, Salishan Lodge, Gleneden Beach, Oregon*, pages 103–112. Morgan Kaufmann, 1989.

[Tec] O_2 Technology. *Technical Overview of the O_2 System.*

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems*, volume I. Computer Science Press, 1988.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems: The New Technologies.* Computer Science Press, 1989.

[VD91] Scott L. Vandenberg and David J. DeWitt. Algebraic support for complex objects with arrays, identity, and inheritance. In James Clifford and Roger King, editors, *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, Denver, Colorado, May 29-31, 1991*, pages 158–167. ACM Press, 1991.

[Ver] Versant.
http://www.versant.com.

[WCB+00] Fanny Wattez, Sophie Cluet, Véronique Benzaken, Guy Ferran, and Christian Fiegel. Benchmarking queries over trees: Learning the hard truth the hard way. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, volume 29, pages 510–511. ACM, 2000.

[WKO+84] D. De Witt, R.H. Katz, F. Olken, L.D. Shapiro, M.R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Boston, Massachusetts, 1984.

[Xyl] xyleme.
`http://www.xyleme.com`.

[YO91] Li Yu and Sylvia L. Osborn. An evaluation framework for algebraic object-oriented query models. In *Proceedings of the Seventh International Conference on Data Engineering, April 8-12, 1991, Kobe, Japan*. IEEE Computer Society, 1991.

RESUME en français

Cette thèse étudie comment optimiser des requêtes sur des données dont le format de représentation est un arbre.

D'une part, les données interrogées sont stockées dans le SGBD objet O_2. En réalisant un banc d'essais de requêtes OQL, nous montrons les faiblesses du système concernant les gros accès associatifs et proposons quelques solutions pour les améliorer. Egalement, nous expliquons les avantages et les inconvénients de la navigation par rapport aux jointures sur des bases réelles.

D'autre part, les données interrogées sont des documents écrits en XML, càd., des données semi-structurées. Dans ce contexte, les techniques d'optimisation sont différentes. Pour pouvoir répondre rapidement à des requêtes portant sur la structure d'un grand nombre de documents, quelle technique adopter? Nous avons choisi d'indexer les documents dans des index plein-texte. Nous montrons comment nous utilisons cette technologie et comment, une fois les documents indexés, nous évaluons les requêtes.

TITRE en anglais

Optimization Of Queries Over Trees: Application To The Object Model And To XML.

RESUME en anglais

This thesis studies how to optimize queries over data whose structure is a tree.

On the one hand, data are stored in the OODBMS O_2. While running a benchmark of OQL queries, we emphasize that this system does not cope well with large associative accesses and we propose solutions to improve performances. Moreover, we show the pros and cons of navigation versus joins in a real system.

On the other hand, data are XML documents, i.e., semi-structured data. In this context, optimization techniques are different. In order to answer quickly structured queries over a huge number of documents, which technique can we use? We have chosen to index documents in full-text indexes. We explain how we extend this technology and how, once documents are indexed, queries are evaluated.

DISCIPLINE - SPECIALITE DOCTORALE

Informatique

MOTS CLES

Algorithmes optimaux, Bases de données orientées objets, Langage d'interrogation, XML (langage de balisage).

INTITULE ET ADRESSE DU LABORATOIRE

INRIA Rocquencourt, projet Verso
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex France