USTL
FLANDRES ARTOIS

LIFL
U.A. 369 du C.N.R.S.

LABORATOIRE D'INFORMATIQUE FONDAMENTALE DE LILLE

Numéro d'ordre : 684

THESE

Nouveau régime

présentée à
l'Université des Sciences et Techniques de Lille Flandres Artois

pour obtenir le titre de

DOCTEUR en INFORMATIQUE

par

Jean-Christophe NICOLAS

MACHINES BASES DE DONNEES PARALLELES : CONTRIBUTION AUX PROBLEMES DE LA FRAGMENTATION ET DE LA DISTRIBUTION

Soutenue le lundi 28 janvier 1991 devant la commission d'examen

Membres du jury

Président	M. MERIAUX
Rapporteurs	F. ANDRE
	M. MIRANDA
Directeur de thèse	B. TOURSEL
Examinateurs	E. DELATTRE
	M.P. LECOUFFE
	J.C. MARTI

UNIVERSITE DES SCIENCES
ET TECHNIQUES DE LILLE
FLANDRES ARTOIS

DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M.H. LEFEBVRE, M. PARREAU.

PROFESSEURS HONORAIRES DES ANCIENNES FACULTES DE DROIT ET SCIENCES ECONOMIQUES, DES SCIENCES ET DES LETTRES

MM. ARNOULT, BONTE, BROCHARD, CHAPPELON, CHAUDRON, CORDONNIER, DECUYPER, DEHEUVELS, DEHORS, DION, FAUVEL, FLEURY, GERMAIN, GLACET, GONTIER, KOURGANOFF, LAMOTTE, LASSERRE, LELONG, LHOMME, LIEBAERT, MARTINOT-LAGARDE, MAZET, MICHEL, PEREZ, ROIG, ROSEAU, ROUELLE, SCHILTZ, SAVARD, ZAMANSKI, Mes BEAUJEU, LELONG.

PROFESSEUR EMERITE

M. A. LEBRUN

ANCIENS PRESIDENTS DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

MM. M. PAREAU, J. LOMBARD, M. MIGEON, J. CORTOIS.

PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE FLANDRES ARTOIS

M. A. DUBRULLE.

PROFESSEURS - CLASSE EXCEPTIONNELLE

M. CONSTANT Eugène	Electronique
M. FOURET René	Physique du solide
M. GABILLARD Robert	Electronique
M. MONTREUIL Jean	Biochimie
M. PARREAU Michel	Analyse
M. TRIDOT Gabriel	Chimie Appliquée

PROFESSEURS - 1ère CLASSE

M. BACCHUS Pierre	Astronomie
M. BIAYS Pierre	Géographie
M. BILLARD Jean	Physique du Solide
M. BOILLY Bénoni	Biologie
M. BONNELLE Jean-Pierre	Chimie-Physique
M. BOSCQ Denis	Probabilitéss
M. BOUGHON Pierre	Algèbre
M. BOURIQUET Robert	Biologie Végétale
M. BREZINSKI Claude	Analyse Numérique

M. BRIDOUX Michel	Chimie-Physique
M. CELET Paul	Géologie Générale
M. CHAMLEY Hervé	Géotechnique
M. COEURE Gérard	Analyse
M. CORDONNIER Vincent	Informatique
M. DAUCHET Max	Informatique
M. DEBOURSE Jean-Pierre	Gestion des Entreprises
M. DHAINAUT André	Biologie Animale
M. DOUKHAN Jean-Claude	Physique du Solide
M. DYMENT Arthur	Mécanique
M. ESCAIG Bertrand	Physique du Solide
M. FAURE Robert	Mécanique
M. FOCT Jacques	Métallurgie
M. FRONTIER Serge	Ecologie Numérique
M. GRANELLE Jean-Jacques	Sciences Economiques
M. GRUSON Laurent	Algèbre
M. GUILLAUME Jean	Microbiologie
M. HECTOR Joseph	Géométrie
M. LABLACHE-COMBIER Alain	Chimie Organique
M. LACOSTE Louis	Biologie Végétale
M. LAVEINE Jean-Pierre	Paléontologie
M. LEHMANN Daniel	Géométrie
Mme LENOBLE Jacqueline	Physique Atomique et Moléculaire
M. LEROY Jean-Marie	Spectrochimie
M. LHOMME Jean	Chimie Organique Biologique
M. LOMBARD Jacques	Sociologie
M. LOUCHEUX Claude	Chimie Physique
M. LUCQUIN Michel	Chimie Physique
M. MACKE Bruno	Physique Moléculaire et Rayonnements Atmosph.
M. MIGEON Michel	E.U.D.I.L.
M. PAQUET Jacques	Géologie Générale
M. PETIT Francis	Chimie Organique
M. POUZET Pierre	Modélisation - calcul Scientifique
M. PROUVOST Jean	Minéralogie
M. RACZY Ladislas	Electronique
M. SALMER Georges	Electronique
M. SCHAMPS Joel	Spectroscopie Moléculaire
M. SEGUIER Guy	Electrotechnique
M. SIMON Michel	Sociologie
Melle SPIK Geneviève	Biochimie
M. STANKIEWICZ François	Sciences Economiques
M. TILLIEU Jacques	Physique Théorique
M. TOULOTTE Jean-Marc	Automatique
M. VIDAL Pierre	Automatique
M. ZEYTOUNIAN Radyadour	Mécanique

PROFESSEURS - 2ème CLASSE

M. ALLAMANDO Etienne	Composants Electroniques
M. ANDRIES Jean-Claude	Biologie des organismes
M. ANTOINE Philippe	Analyse
M. BART André	Biologie animale
M. BASSERY Louis	Génie des Procédés et Réactions Chimiques

Mme BATTIAU Yvonne	Géographie
M. BEGUIN Paul	Mécanique
M. BELLET Jean	Physique Atomique et Moléculaire
M. BERTRAND Hugues	Sciences Economiques et Sociales
M. BERZIN Robert	Analyse
M.BKOUCHE Rudolphe	Algèbre
M. BODARD Marcel	Biologie Végétale
M. BOIS Pierre	Mécanique
M. BOISSIER Daniel	Génie Civil
M. BOIVIN Jean-Claude	Spectroscopie
M. BOUQUELET Stéphane	Biologie Appliquée aux enzymes
M. BOUQUIN Henri	Gestion
M. BRASSELET Jean-Paul	Géométrie et Topologie
M. BRUYELLE Pierre	Géographie
M. CAPURON Alfred	Biologie Animale
M. CATTEAU Jean-Pierre	Chimie Organique
M. CAYATTE Jean-Louis	Sciences Economiques
M. CHAPOTON Alain	Electronique
M. CHARET Pierre	Biochimie Structurale
M. CHIVE Maurice	Composants Electroniques Optiques
M. COMYN Gérard	Informatique Théorique
M. COQUERY Jean-Marie	Psychophysiologie
M. CORIAT Benjamin	Sciences Economiques et Sociales
Mme CORSIN Paule	Paléontologie
M. CORTOIS Jean	Physique Nucléaire et Corpusculaire
M. COUTURIER Daniel	Chimie Organique
M. CRAMPON Norbert	Tectolique Géodynamique
M. CROSNIER Yves	Electronique
M. CURGY Jean-Jacques	Biologie
Melle DACHARRY Monique	Géographie
M. DEBRABANT Pierre	Géologie Appliquée
M. DEGAUQUE Pierre	Electronique
M. DEJAEGER Roger	Electrochimie et Cinétique
M. DELAHAYE Jean-Paul	Informatique
M. DELORME Pierre	Physioliogie Animale
M. DELORME Robert	Sciences Economiques
M. DEMUNTER Paul	Sociologie
M. DENEL Jacques	Informatique
M. DE PARIS Jean Claude	Analyse
M. DEPREZ Gilbert	Physique du Solide - Cristallographie
M. DERIEUX Jean-Claude	Microbiologie
Melle DESSAUX Odile	Spectroscopie de la réactivité Chimique
M. DEVRAINNE Pierre	Chimie Minérale
Mme DHAINAUT Nicole	Biologie Animale
M. DHAMELINCOURT Paul	Chimie Physique
M. DORMARD Serge	Sciences Economiques
M. DUBOIS Henri	Spectroscopie Hertzienne
M. DUBRULLE Alain	Spectroscopie Hertzienne
M. DUBUS Jean-Paul	Spectrométrie des Solides
M. DUPONT Christophe	Vie de la firme (I.A.E.)
Mme EVRARD Micheline	Génie des procédés et réactions chimiques
M. FAKIR Sabah	Algèbre
M. FAUQUAMBERGUE Renaud	Composants électroniques

M. FONTAINE Hubert	Dynamique des cristaux
M. FOUQUART Yves	Optique atmosphérique
M. FOURNET Bernard	Biochimie Sturcturale
M. GAMBLIN André	Géographie urbaine, industrielle et démog.
M. GLORIEUX Pierre	Physique moléculaire et rayonnements Atmos.
M. GOBLOT Rémi	Algèbre
M. GOSSELIN Gabriel	Sociologie
M. GOUDMAND Pierre	Chimie Physique
M. GOURIEROUX Christian	Probabilités et Statistiques
M. GREGORY Pierre	I.A.E.
M. GREMY Jean-Paul	Sociologie
M. GREVET Patrice	Sciences Economiques
M. GRIMBLOT Jean	Chimie Organique
M. GUILBAULT Pierre	Physiologie animale
M. HENRY Jean-Pierre	Génie Mécanique
M. HERMAN Maurice	Physique spatiale
M. HOUDART René	Physique atomique
M. JACOB Gérard	Informatique
M. JACOB Pierre	Probabilités et Statistiques
M. Jean Raymond	Biologie des populations végétales
M. JOFFRE Patrick	Vie de la firme (I.A.E.)
M. JOURNEL Gérard	Spectroscopie hertzienne
M. KREMBEL Jean	Biochimie
M. LANGRAND Claude	Probabilités et statistiques
M. LATTEUX Michel	Informatique
Mme LECLERCQ Ginette	Catalyse
M. LEFEBVRE Jacques	Physique
M. LEFEBVRE Christian	Pétrologie
Melle LEGRAND Denise	Algèbre
Melle LEGRAND Solange	Algèbre
M. LEGRAND Pierre	Chimie
Mme LEHMANN Josiane	Analyse
M. LEMAIRE Jean	Spectroscopie hertzienne
M. LE MAROIS Henri	Vie de la firme (I.A.E.)
M. LEROY Yves	Composants électroniques
M. LESENNE Jacques	Systèmes électroniques
M. LHENAFF René	Géographie
M. LOCQUENEUX Robert	Physique théorique
M. LOSFELD Joseph	Informatique
M. LOUAGE Francis	Electronique
M. MAHIEU Jean-Marie	Optique-Physique atomique
M. MAIZIERES Christian	Automatique
M. MAURISSON Patrick	Sciences Economiques et Sociales
M. MESMACQUE Gérard	Génie Mécanique
M. MESSELYN Jean	Physique atomique et moléculaire
M. MONTEL Marc	Physique du solide
M. MORCELLET Michel	Chimie Organique
M. MORTREUX André	Chimie Organique
Mme MOUNIER Yvonne	Physiologie des structures contractiles
Mme MOUYART-TASSIN Annie Françoise	Informatique
M. NICOLE Jacques	Spectrochimie
M. NOTELET Francis	Systèmes électroniques
M. PARSY Fernand	Mécanique

M. PECQUE Marcel	Chimie organique
M. PERROT Pierre	Chimie appliquée
M. STEEN Jean-Pierre	Informatique

Je tiens à remercier Monsieur Michel Mériaux d'avoir accepté de présider le jury de cette thèse,

Madame Françoise André et Monsieur Serge Miranda qui m'ont fait l'honneur de bien vouloir en être les rapporteurs. Je les remercie pour leur commentaires, critiques et observations qui m'ont permis de corriger ce document.

Je remercie également Madame Marie-Paule Lecouffe, Monsieur Eric Delattre et Monsieur Jean-Claude Marti, d'avoir accepté de participer au jury et de m'avoir fait part de leur remarques concernant ce travail.

Je remercie tout particulièrement Monsieur Bernard Toursel qui a dirigé ces travaux. Son expérience, ses remarques pertinentes et ses encouragements, m'ont permis de les mener à bien et ont indéniablement contribué à renforcer mon goût pour la recherche.

Mes remerciements s'adressent aussi aux membres de l'équipe N-ARCH,

aux collègues et amis du laboratoire grâce auquels j'ai pu évoluer dans un environnement plus que sympathique,

à tous les autres amis qui ne parlent jamais d'informatique.

Je tiens également à remercier Madame Annie Kaczmarek qui a réalisé une partie de la frappe, ainsi que Monsieur Henri Glanc qui assuré la reproduction du document.

Enfin, je ne saurais terminer sans avoir remercié ma famille,

mes parents pour les sacrifices qu'ils ont fait en privilégiant avant tout les études de leurs enfants,

et tout particulièrement Evelyne, ma femme, qui m'a toujours soutenu et encouragé dans cette entreprise, qui a eu la tâche ingrate de taper en partie et de relire le document, et à qui je dois beaucoup.

TABLE DES MATIERES

LISTE DES FIGURES

1. ETAT DE L'ART ET PRESENTATION DU PROBLEME 17

2. LA FRAGMENTATION VERTICALE 61

3. LA FRAGMENTATION HORIZONTALE 141

1. ETAT DE L'ART ET PRESENTATION DU PROBLEME

1.1. Un bref historique sur les bases de données.

L'histoire des Systèmes de Gestion de Bases de Données (SGBD) peut se décrire par rapport à l'évolution qualitative et quantitative que ces derniers ont observée. Les progrès théoriques et technologiques du domaine des bases de données ont contribué à leur évolution qualitative. L'accroissement du nombre et de la variété des applications des bases de données, conjugué à une plus grande diversité des matériels informatiques ont, quant à eux, entraîné leur évolution quantitative.

Ces SGBD n'échappent pas à une classification dans le temps. Ainsi, leur histoire est fréquemment résumée en distinguant trois générations. Notons que l'ancêtre du SGDB, le système classique de gestion de fichiers (SGF), est souvent vu comme la génération 0. Les SGBD étant principalement caractérisés par le modèle de données qu'ils supportent, cette classification est établie par rapport à ces modèles et leurs fonctionnalités.

La ***première génération*** de SGBD s'appuie sur les ***modèles de données hiérarchique et réseau*** [IBM69], [MRI72]. Les données sont représentées, au niveau des types d'article, par une hiérarchie ou un graphe. Ces modèles permettent, en fait, d'étendre les possibilités d'un système de gestion de fichiers en offrant des liaisons inter-fichiers matérialisées par des pointeurs. Les langages de manipulation de données associés à ce type de SGBD sont dits navigationnels puisqu'ils obligent l'utilisateur à spécifier les chemins d'accès aux données et donc, à naviguer dans le graphe de la base. L'***indépendance physique*** de tels systèmes est très faible puisque le niveau conceptuel et le niveau interne sont fortement liés [MIRA86a].

L'apparition du ***modèle relationnel*** vers 1970 [CODD 70] peut être vu comme le point de départ de la ***deuxième génération***. Les données y sont représentées, au niveau conceptuel, sous forme de ***relations normalisées*** (***tables***), indépendamment du niveau interne sous-jacent [MIRA86b].

Toute information de la base de données doit être explicitement représentée par des valeurs situées dans des tables. En particulier, il n'existe pas de pointeurs inter-relations. Un ensemble d'opérateurs appliqués aux relations, l'algèbre relationnelle, permet la recherche et la mise à jour des données. Outre les opérateurs classiques de la théorie des ensembles (produit cartésien, union, intersection, etc), l'algèbre relationnelle comporte des opérateurs qui permettent de composer des sous-ensembles d'une ou deux relations (projection, restriction, jointure, division). Enfin, un ensemble de contraintes d'intégrité sémantiques permet de définir les états cohérents de la base.

L'indépendance totale entre la description logique et la description physique des données constitue le principal avantage du modèle relationnel. Grâce à cette forte indépendance physique, des langages de définition et de manipulation de données de haut niveau ont été développés "au-dessus" de l'algèbre relationnelle. Ces langages de requêtes sont assertionnels (ie déclaratifs) puisque l'utilisateur n'a pas à spécifier les chemins d'accès aux données. L'optimisation des requêtes y est entièrement automatique. A ce titre, le langage SQL [CHAM76], [CHAM80]constitue une norme ANSI [ANSI86] pour ces langages, ce qui contribue bien sûr "à la promotion du modèle relationnel".

Enfin, sous le terme "troisième génération de SGBD", on désigne les systèmes qui supportent "bien" les nouveaux types d'application [1] (CAO, cartographie, génie logiciel, bureautique, etc) ou qui exploitent des environnements complexes (répartis, parallèles ou hétérogènes). Cette nouvelle génération, dont les premiers travaux datent de la fin des années 70, fait actuellement l'objet de nombreux travaux de recherche, dans des domaines très variés. Il n'y a pas, comme pour le modèle relationnel, une base précise et uniforme. Il est donc difficile d'en dégager de grandes lignes. Il faut néanmoins noter que certains produits spécifiques apparaissent aujourd'hui sur le marché.

Nous proposons de résumer ce bref historique par la figure suivante:

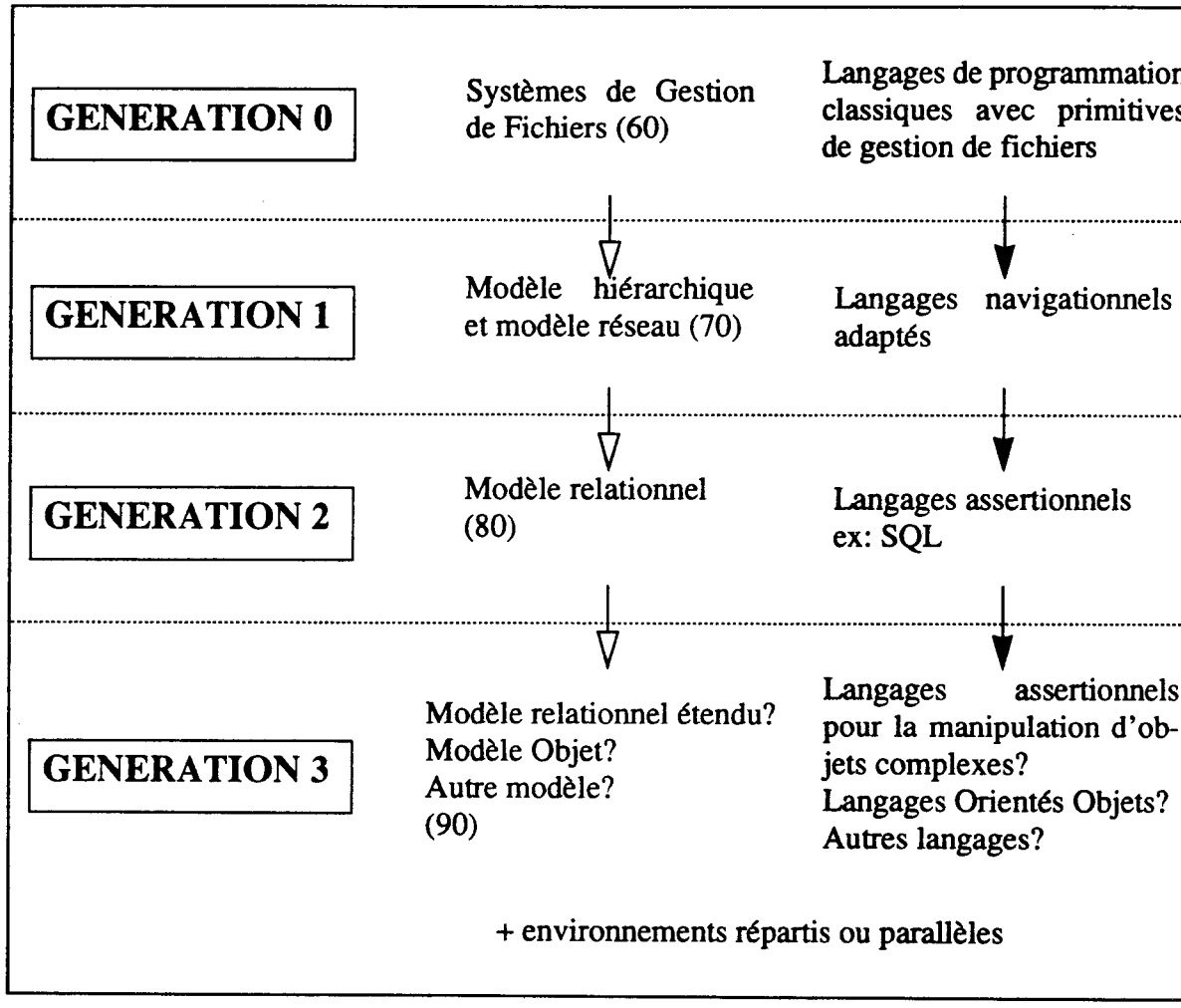


Fig. 1.1 : Evolution des Systèmes de Gestion de Bases de Données.

Nous reviendrons plus précisément, dans ce qui suit, sur les évolutions que la troisième génération se doit de prendre en compte. Auparavant, voyons comment se présente la situation actuelle.

1. Cette remarque est faite relativement aux SGBD relationnels qui sont mal adaptés à ces applications et qui ne les supportent donc pas de façon efficace. Nous reviendrons plus précisément sur cet aspect dans ce qui suit.

D'un point de vue commercial, le modèle relationnel est, à ce jour, majoritairement[1] utilisé. Il est important de noter que cette situation est récente (89) et qu'il s'est donc écoulé presque 20 ans entre la proposition de Codd [CODD70] et le moment où les systèmes relationnels sont devenus les Systèmes de Gestion de Bases de Données les plus utilisés. D'un point de vue langage de définition et de manipulation de données, SQL est unanimement reconnu.

A priori, nous pourrions donc considérer que tout va bien puisque nous disposons de SGBD efficaces, relativement simples d'emploi et totalement indépendants du matériel utilisé. En fait, différents "événements" obligent aujourd'hui les spécialistes à revoir leurs positions.

Considérons tout d'abord l'influence des nouveaux types d'application qu'il serait souhaitable que les SGBD traitent. Historiquement, la demande émanait essentiellement des applications de gestion. Les données étaient donc rarement complexes; ainsi les modèles proposés (hiérarchique, réseau, relationnel) n'avaient aucune difficulté à les supporter. Aujourd'hui, il existe un besoin important en gestion d'objets complexes (textes, graphiques, cartes, données multidimensionnelles). Entre autres, la conception assistée par ordinateur (CAO), les applications bureautiques, les ateliers de génie logiciel nécessitent de gérer de larges volumes d'objets aux structures complexes, objets partagés et persistants. Face à ces nouvelles applications, les systèmes relationnels s'avèrent être mal adaptés [GARD84], [ADIB86]. Une évolution au niveau du modèle et des langages associés semble donc nécessaire.

Autre problème important, celui de l'interface entre les langages de programmation classiques et les SGBD relationnels. En effet, les langages assertionnels (tels que SQL) attachés à ces systèmes ne permettent pas d'exprimer certaines manipulations sophistiquées de données. En d'autres termes, ces langages ne sont pas complets au sens de Turing. L'expression d'un traitement récursif est, par exemple, impossible avec la version actuelle de SQL. La solution consiste alors à intégrer le langage de requête dans un langage de programmation (exemple: C+SQL), ce qui, comme nous le verrons dans ce qui suit, pose de nombreux problèmes regroupés sous le terme de "impédance mismatch"[COPE84].

Enfin, l'augmentation incessante des volumes de données à traiter, et donc le besoin toujours croissant de meilleures performances, conduit naturellement à envisager l'utilisation de SGBD parallèles, systèmes sur lesquels beaucoup de travaux sont menés tant au niveau matériel que logiciel.

A la vue de cette première analyse globale, il semble donc que les SGBD de troisième génération soient difficilement définissables tant les objectifs qui s'y rapportent peuvent être différents.

Suite à ce tour d'horizon sur "l'univers" des bases de données, nous allons maintenant préciser les différentes évolutions qui semblent unanimement nécessaires. Nous consacrerons donc, dans ce qui suit, une partie sur l'évolution "attendue" des modèles de données et langages associés, et une seconde partie sur l'utilisation de systèmes parallèles.

Cette double présentation, relative à des préoccupations différentes, nous permettra alors de mettre en avant l'objet de notre travail ainsi que le contexte précis dans lequel il se situe.

---

1. Cette majorité est relative aux utilisations de SGBD et non pas aux applications qui gèrent, d'une façon ou d'une autre, des données. Beaucoup de ces applications utilisent encore, hélas, des systèmes de gestion de fichiers classiques.

1.2. Sur l'évolution des modèles de données et des langages associés.

Dans cette section, nous allons revenir en détail sur les problèmes relatifs aux modèles de données et aux langages de manipulation de données associés. Ceci nous amènera à dégager, sous forme d'un modèle et d'un langage idéals, les principales propriétés que nous jugeons nécessaire de leur octroyer.

Puisqu'il s'agit avant tout d'une remise en cause de la situation actuelle, notre analyse se basera sur le modèle relationnel et "ses" langages. Outre ces propriétés générales, nous présenterons les différentes solutions retenues dans les projets relatifs à ce type de travail.

1.2.1. Problèmes liés au modèle de données relationnel.

Nous aborderons ici, trois aspects du modèle relationnel qui le contraignent à certaines limitations pour le moins pénalisantes.

Nous discuterons des problèmes liés à la première forme normale, des contraintes attachées au modèle-valeur qu'est le modèle relationnel et, enfin, de la difficulté à pouvoir modèliser facilement des "applications complexes".

1.2.1.1. Influence de la première forme normale.

Rappelons que la contrainte de la première forme normale (notée 1NF) conduit à normaliser les objets hiérarchiques, c'est-à-dire à ne considérer comme valides que les attributs mono-valués. En d'autres termes, les relations décrites sont toujours "plates". Représentées sous forme d'une table, nous trouvons donc, au plus, une valeur "dans chaque case", une case représentant la valeur d'un attribut donné pour un nuplet donné.

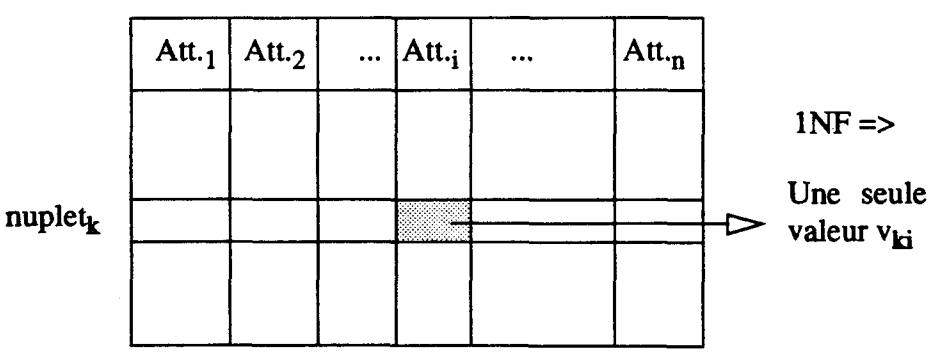


Fig. 1.2 : Principe de la première forme normale.

Par cette contrainte, tout objet initialement structuré (au niveau externe) se trouve donc éclaté, au niveau conceptuel, en un ensemble de nuplets "plats". En fait, ce que nous pouvons mettre en cause ici n'est pas la décomposition en elle-même, que de nombreuses méthodes

permettent d'obtenir de façon quasi-automatique[YAO85], mais plutôt les conséquences qu'elle peut avoir si elle est aveuglément appliquée à de "gros objets complexes" (exemple: objets hiérarchiques de CAO).

Tout d'abord, il faut bien voir que l'information sémantique attachée à la structure complexe d'un objet se trouve éclatée, sous forme de valeurs, dans différentes relations; ceci oblige alors l'utilisateur à adapter sa vision à celle du SGBD puisque la représentation conceptuelle obtenue n'est pas réaliste. En d'autres termes, cet utilisateur ne peut manipuler que des données "plates" en les reliant par des clés de jointure[1] souvent abstraites, ce qui le relègue à un niveau d'utilisation "sémantiquement" très pauvre. Ce point est, d'ailleurs, d'autant plus critique que les objets sont complexes. La modélisation d'une carte géographique précise demanderait, par exemple, beaucoup de relations qui écarteraient totalement l'utilisateur d'une manipulation "réaliste" des objets.

Outre cet aspect sémantique de la première forme normale, il faut aussi en analyser les conséquences au niveau des performances. En effet, la reconstruction de tout ou partie d'un objet complexe peut exiger un grand nombre de jointures, opérations coûteuses en temps et qu'il n'est pas souhaitable de multiplier à foison. Là encore, le problème est d'autant plus grave que les objets manipulés sont complexes.

En conclusion, la première forme normale semble donc être un double obstacle à la manipulation des objets complexes propres aux nouvelles applications. La relâche de cette contrainte s'avére donc être un premier pas nécessaire pour s'orienter vers une gestion directe et efficace de tels objets.

1.2.1.2. Contraintes attachées au modèle valeur qu'est le modèle relationnel.

Un autre problème important attaché au modèle relationnel est celui de l'identification des données. La solution, proposée par Codd [CODD70] afin d'identifier les nuplets d'une relation, consiste à employer un ou plusieurs attributs comme clé de la relation. Une clé de nuplet doit être alors unique pour tous les nuplets de la relation. Ce sont donc les valeurs de certains attributs qui dénotent l'identité d'un nuplet, ce qui peut nous amener à qualifier le modèle relationnel de modèle valeur puisqu'aucun autre "support" n'est utilisé.

Cette approche pose cependant plusieurs problèmes puisque les attributs clés jouent alors le rôle dual de données descriptives et d'identité. Il n'y a pas, dans ce cas, de distinction nette entre la valeur d'une donnée et son identité.

Le premier de ces problèmes concerne la manipulation des attributs clés qui ne peut être la même que celles des autres attributs. Les mises à jour de ces attributs clés sont, en effet, délicates puisqu'elles peuvent engendrer des mises à jour dans d'autres relations, dans lesquelles ils sont utilisés comme clé étrangère.

Supposons par exemple que la clé d'une relation ETUDIANTS soit constituée de l'attribut *nom* et de l'attribut *prénom*. Une relation EMPRUNTS modèlisant l'emprunt des livres de la bibliothèque universitaire, devra donc aussi "renfermer" ces attributs nom et prénom,

---

1. Rappelons que l'opération de jointure est l'opération de l'algèbre relationnelle qui permet, entre autres, de reconstruire une information initialement éclatée dans différentes relations et ce, par application d'un produit cartésien restreint par un prédicat de jointure.

afin de pouvoir "retrouver" les emprunteurs. De ce fait le changement de nom d'une étudiante entrainera aussi des mises à jour dans la relation EMPRUNTS.

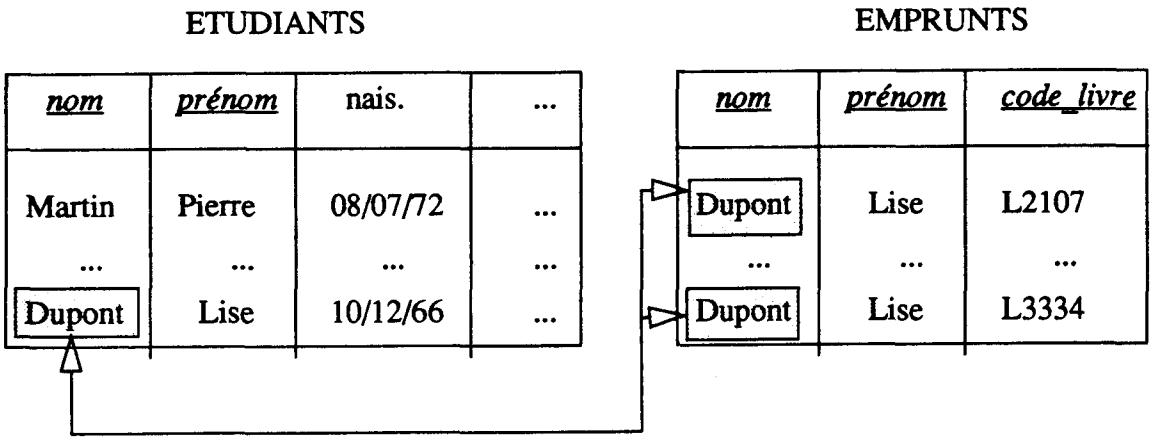


Fig. 1.3 : Exemple de mise à jour d'attributs clés à "propager"

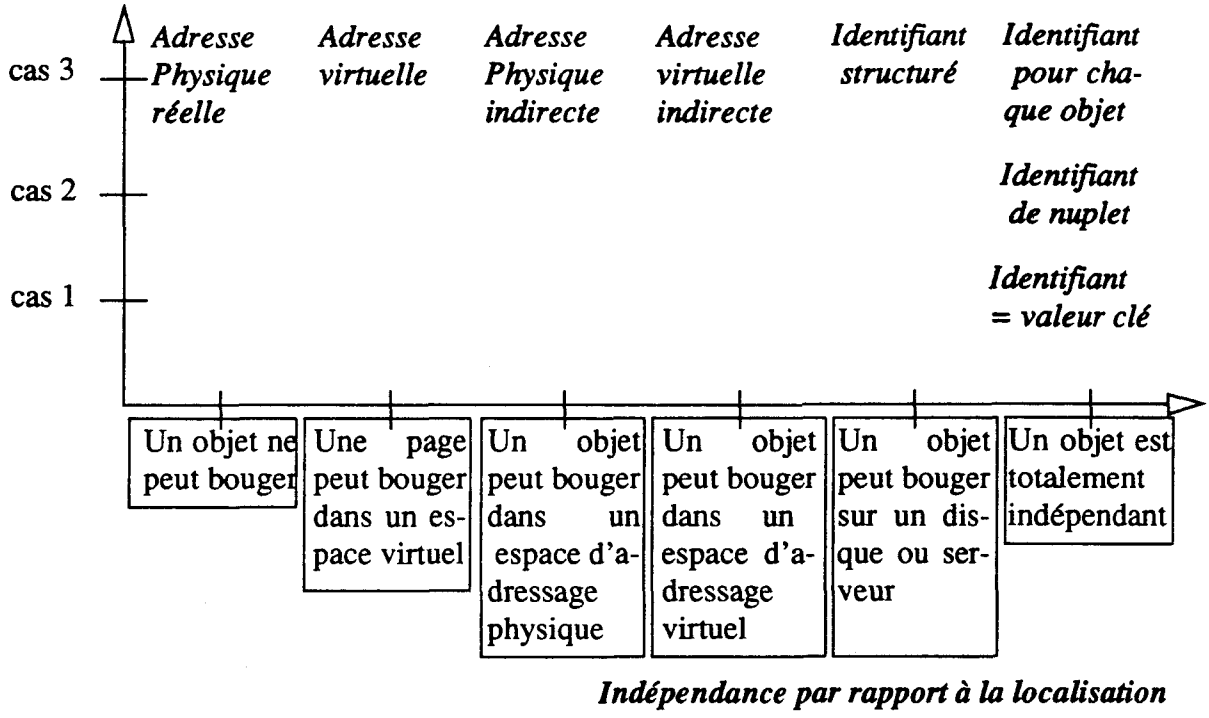
Le second désavantage que nous pouvons attribuer à cette identification par valeur, est relatif au choix des attributs clés. Différents besoins, au niveau conceptuel, peuvent conduire à des choix différents et incompatibles. A l'opposé, l'absence d'attributs suffisants pour identifier de façon unique chaque nuplet peut entraîner la création d'un attribut clé artificiel et donc abstrait.Le recours à de tels attributs, qui se transforment alors assurément en clés de jointure abstraites, ne va pas dans le sens d'une manipulation réaliste des données.

Afin de solutionner ces problèmes, il suffit de supporter directement la notion d'identité d'objets dans le modèle de données. Codd a préconisé cette solution dans le modèle RM/T où un identifiant unique et invariable est attribué à chaque nuplet lors de sa création [CODD79]. Dès lors, tous les attributs d'une relation jouent le même rôle descriptif et ne sont plus attachés à une quelconque notion d'identité.

Sur ce sujet, il est important de citer l'article de S.Khoshafian et G.Copeland, "Object Identity" [KHOS86], qui donne la définition suivante de l'identité d'objets "capacité de distinguer logiquement et physiquement un objet de tout autre". Les auteurs mettent en évidence l'intérêt du support de cette notion d'identité d'objets et donnent une classification des différents niveaux d'intégration qu'elle peut avoir (cf figure 1.4).

Replacée par rapport à cette classification, la proposition de Codd [CODD79] permet donc d'avancer d'un pas sur l'échelle de l'indépendance par rapport aux données. D'un identifiant valeur clé, elle nous amène à un identifiant de nuplet. Vis-à-vis du modèle relationnel, nous ne pouvons aller plus loin dans cette direction puisque les nuplets "plats" sont les seules données manipulées.

Nous verrons, lors de la définition d'un modèle "idéal" (cf 1.2.1.4), les conséquences de l'intégration de la notion d'identité d'objets dans un modèle de données qui supportent les objets complexes.



Avec:
cas1 <=> Dépendance par rapport à la valeur et la structure de la donnée
cas2 <=> Indépendance par rapport à la valeur, mais dépendance par rapport à la structure de la donnée
cas 3<=> Totale indépendance par rapport à la donnée

Fig. 1.4 : Différents niveaux d'intégration de la notion d'identité d'objet

1.2.1.3. Faible pouvoir descriptif du modèle relationnel.

Dernier point que nous aborderons sur le modèle relationnel, celui de la faiblesse de ses capacités de modélisation. En effet, les types de données supportés sont peu nombreux et limités à quelques domaines alphanumériques. D'autre part, ces types sont non extensibles, qu'il s'agisse d'une extension vers des types spécifiques ou vers des types génériques (liste, pile, etc).

Cet état de fait est historiquement lié aux besoins des applications de gestion pour lesquelles ces types courants étaient suffisants. Les problèmes posés sont donc relatifs aux nouvelles applications (CAO, cartographie, etc). La solution qui consiste à décrire ces types "riches" comme des chaînes de caractères, puis à les manipuler à l'extérieur du SGBD par un

langage de programmation, n'est bien sûr pas sérieuse. Outre les problèmes liés à ce langage de programmation extérieur (cf 1.2.2.), cette solution entraînera une forte dépendance entre programmes et données (notion habituellement désignée sous le terme de ***dépendance logique***) et une certaine inefficacité dans la manipulation d'objets atomiques de grande taille (exemple: représentation d'une image sous forme d'une chaîne de caractères!).

Il semble donc nécessaire de pouvoir étendre les types de données atomiques d'un modèle, tout en ayant la possibilité de décrire les opérations de manipulation de ces nouveaux types au niveau même du SGBD. Cela nous conduit bien évidemment à penser aux solutions "voisines" que constituent l'approche Orientée-Objets et l'approche par utilisation de Type Abstrait de Données (TADs).

Nous reviendrons dans ce qui suit sur l'approche Orientée -Objets. En ce qui concerne l'autre solution, celle du support des Types Abstraits de Données, qui consiste, rappelons-le, à "envelopper" un type d'objets et la collection d'opérations applicables à ces objets, nous nous contenterons de citer l'article de P.Valduriez "Objets complexes dans les systèmes de bases de données relationnels" [VALD87a]. Dans cet article, l'auteur étudie précisément, et suivant différents axes, les problèmes attachés au support des TADS.

1.2.1.4. Définition d'un modèle idéal.

Suite à cette analyse des problèmes attachés au modèle relationnel, nous sommes en mesure de donner les grandes lignes d'un modèle de données "idéal".

Un tel modèle se doit de pouvoir représenter "directement" des objets complexes ce qui implique, d'une part, qu'il n'y ait pas de contraintes de première forme normale, et d'autre part, qu'il y ait une possibilité d'étendre les types atomiques de base et de définir "localement"[1] les opérations associées à ces types.

Nous prendrons, à partir de maintenant, la définition suivante d'un ***objet complexe***: "objet dont les attributs ne sont pas nécessairement atomiques mais peuvent être eux-mêmes des objets". Un objet complexe aura donc généralement une structure hiérarchique et pourra donc être défini comme un objet-racine et une hiérarchie d'objets.

Enfin, pour un tel modèle idéal, les données manipulées doivent être identifiées à un niveau suffisamment fin et surtout indépendant de toute localisation. Il est important de signaler que l'intégration de la notion d'identité d'objets, dans un modèle d'objets complexes, va permettre le partage d'objets. En effet, puisqu'avec ce genre de modèle un objet peut être composant d'un autre, rien n'empêche un tel objet composant d'être partagé par plusieurs objets, c'est-à-dire d'être co-référencé par rapport à son identifiant. Dès lors, le support de graphes peut être directement modélisé. Cette fonctionnalité supplémentaire, amenée par l'identité d'objets, est très intéressante pour les nouvelles applications où le besoin de partage de données est souvent présent.

Nous donnons sur la figure suivante, à partir d'un exemple, une représentation schématique d'un tel partage.

---

1. Par "localement", nous faisons allusion à l'encapsulation propre aux modèles Orientés-Objets ou à "l'enveloppement" d'un type de données et des opérations qui s'y rapportent, propre aux TADs.

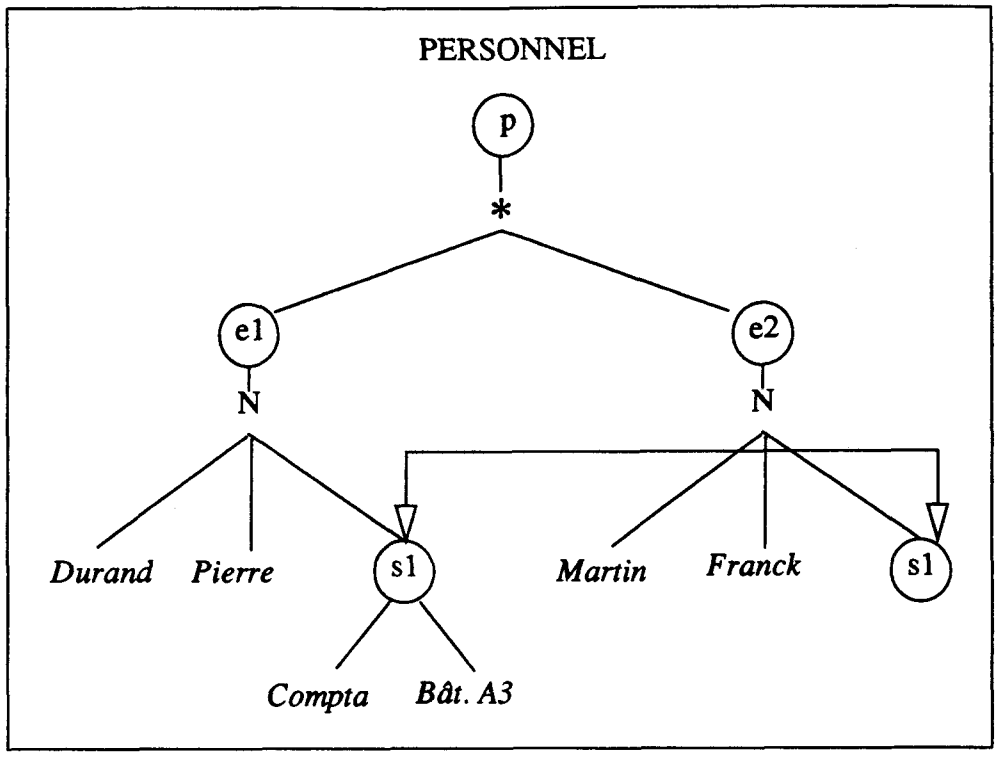


Fig. 1.5 : Exemple de partage d'objet composant.

Sur cette figure, il apparait donc un objet-racine de "type" PERSONNEL, constitué de deux objets de type EMPLOYE. Chacun de ces objets EMPLOYE a comme composant, un objet SERVICE. Sur l'exemple, les objets EMPLOYE partagent conceptuellement le même objet SERVICE.

Nous ne développerons pas plus ici, les aspects liés au modèle de données. Nous reviendrons là-dessus lors de la présentation des solutions proposées (cf 1.2.3.) mais aussi lors de la présentation du modèle de données qui servira de cadre à notre étude (cf 1.4.2. Présentation de FAD).

Nous allons maintenant nous intéresser aux problèmes liés aux langages de manipulation de données et tenter de dégager les grandes lignes d'un langage idéal.

1.2.2. Problèmes liés aux langages de manipulation de données.

Là encore, puisqu'il s'agit d'une remise en cause de la situation actuelle, nous baserons cette étude sur le langage de référence que constitue SQL (Structured Query Language) [DATE85]. Nous aborderons, dans ce qui suit, les insuffisances d'un tel langage, puis analyserons ce qui l'oppose aux langages de programmation classiques dans lesquels il est souvent intégré. Ce survol des principaux problèmes nous permettra alors de définir les propriétés d'un langage idéal.

1.2.2.1. Insuffisances des langages de requêtes actuels

Comme nous l'avons déjà souligné, un langage de requêtes tel que SQL ne permet pas certaines manipulations sophistiquées de données. Entre autres, les traitements récursifs ne sont pas directement exprimables.

Prenons, par exemple, le cas d'objets PIECES qui peuvent être récursivement composés d'autres objets PIECES. Modèlisé "de façon relationnelle", ce cas consiste à créer les relations suivantes:

PIECES

num	*désign*	*atelier_prod*
p1	clinche	at12
p12	porte	at3
p28	fenêtre	at1
p13	poignée	at1

COMPOSITION

Pcomposée	*Pcomposant*
p12	p1
p1	p13
p28	p13

Fig. 1.6 : Exemple de modèlisation relationnelle d'une situation récursive.

Avec la version actuelle de SQL[1], il n'est pas possible d'obtenir la liste de toutes les pièces qui composent une pièce donnée. Il faudrait, pour cela, connaître à l'avance le degré d'imbrication, i.e. la profondeur de l'arbre qui représente cette pièce.

Même en dehors des problèmes liés à la récursivité, il faut signaler qu'un tel langage de requête est toujours agrémenté d'un éditeur de rapport, pour la simple et bonne raison qu'il ne peut répondre de façon satisfaisante au problème de l'édition de données sous forme hiérarchique.

Nous sommes donc en présence d'une norme essentiellement relative à l'extraction de données. Dés qu'il s'agit de "présenter" ou d'exploiter ces données extraites, chaque système a ses petites recettes, plus ou moins satisfaisantes. Enfin, pour les traitements sophistiqués, la seule possibilité est le recours à un langage hôte dans lequel sont directement intégrées les requêtes SQL. Tous les produits proposent ce type d'interfaces (C+SQL, COBOL+SQL, etc), interfaces qui ne sont pas, comme nous allons le voir dans la section suivante, sans poser de problèmes.

---

1. Nous parlons de version actuelle, c'est-à-dire de la norme ANSI86, mais une autre version est actuellement proposée à la normalisation. Cette autre version prendrait en compte, entre autres, la récursivité.

1.2.2.2. Oppositions entre langages de programmation classiques et langages de requêtes.

L'intégration d'un langage de requêtes tel que SQL dans un langage de programmation pose de délicats problèmes relatifs aux oppositions "naturelles" qui existent entre de tels langages, problèmes que certains ont regroupé sous le terme "Impedance Mismatch" [COPE84].

Le premier de ces problèmes est relatif à l'aspect persistant des données. Le langage de requêtes manipule des données persistantes ou temporaires qu'il doit alors transmettre au langage de programmation, qui lui, ne manipule que des données temporaires qui lui sont propres; c'est-à-dire qui "appartiennent" à son espace d'adressage et surtout qui sont représentées en fonction des types qu'il supporte. En d'autres termes, ce langage de programmation n'a aucune influence directe sur les données persistantes que seul le langage de requête manipule et met à jour. Cela exige donc un va-et-vient peu élégant entre les deux mondes, ce que nous pouvons représenter par la figure suivante:

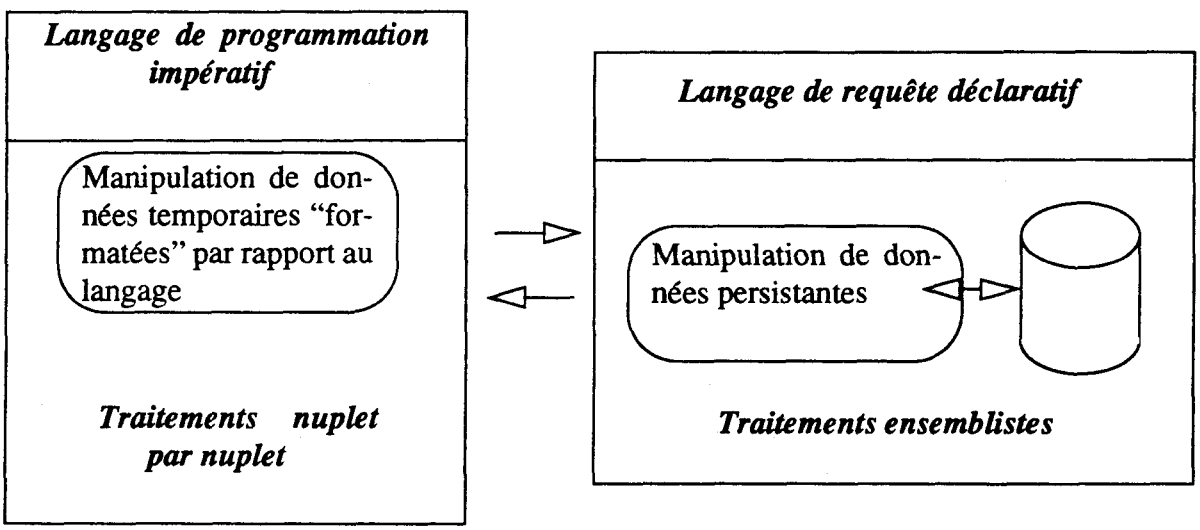


Fig. 1.7 : Oppositions entre langages de requêtes et langages de programmation

Deux autres problèmes sont directement liés à cette transmission de données entre deux univers différents. Tout d'abord, qui dit transmission de données, dit nécessité d'une certaine compatibilité entre les types manipulés de part et d'autre. Sur ce point, une donnée de type *date* manipulée par SQL a, par exemple, toutes les chances de se transformer en une donnée numérique ou chaîne de caractères au niveau du langage de programmation. La transmission nécessitera donc certaines conversions, conversions qui peuvent être bi-directionnelles si une donnée manipulée par le langage de programmation doit être réinscrite dans la base.

Autre aspect lié à cette transmission de données, celui du principe même de fonctionnement de chacun de ces langages. D'un côté nous avons un langage de requêtes qui fonctionne de façon ensembliste (puisque l'algèbre relationnelle est basée sur la théorie des ensembles), et de l'autre nous disposons d'un langage de programmation qui ne peut traiter les données reçues que nuplet par nuplet.

Cette opposition dans les modes de fonctionnement, oblige l'introduction, dans le monde SQL, d'une notion de curseur et des opérations de manipulation de curseurs associées. Un tel curseur sert alors à pointer l'enregistrement courant au niveau du langage de programmation. Il va sans dire que cette pratique est en totale opposition avec la théorie ensembliste simple sur laquelle repose le modèle relationnel.

Enfin nous ne pouvons ignorer l'opposition qui règne entre la nature des langages. Il est en effet important de souligner que l'interface se fait entre un langage de programmation classique, donc ***impératif***, et un langage de requêtes assertionnel que nous pouvons qualifier de ***déclaratif***. Là encore, l'union est pour le moins surprenante puisqu'il faut sans cesse changer de mode de travail en spécifiant suivant les cas uniquement ce que l'on veut ou, ce que l'on veut et comment on procède pour l'obtenir.

Le dernier point que nous aborderons sur cette opposition de phase entre langages, est celui de la traduction qu'elle a au niveau de l'identité des objets. Pour cela nous ferons une fois de plus référence à l'article de S. Khoshafian et G. Copeland sur l'identité d'objet [KHOS86]. Nous en donnons la figure qui représente les différents degrés de support de l'identité d'objet au niveau des langages.

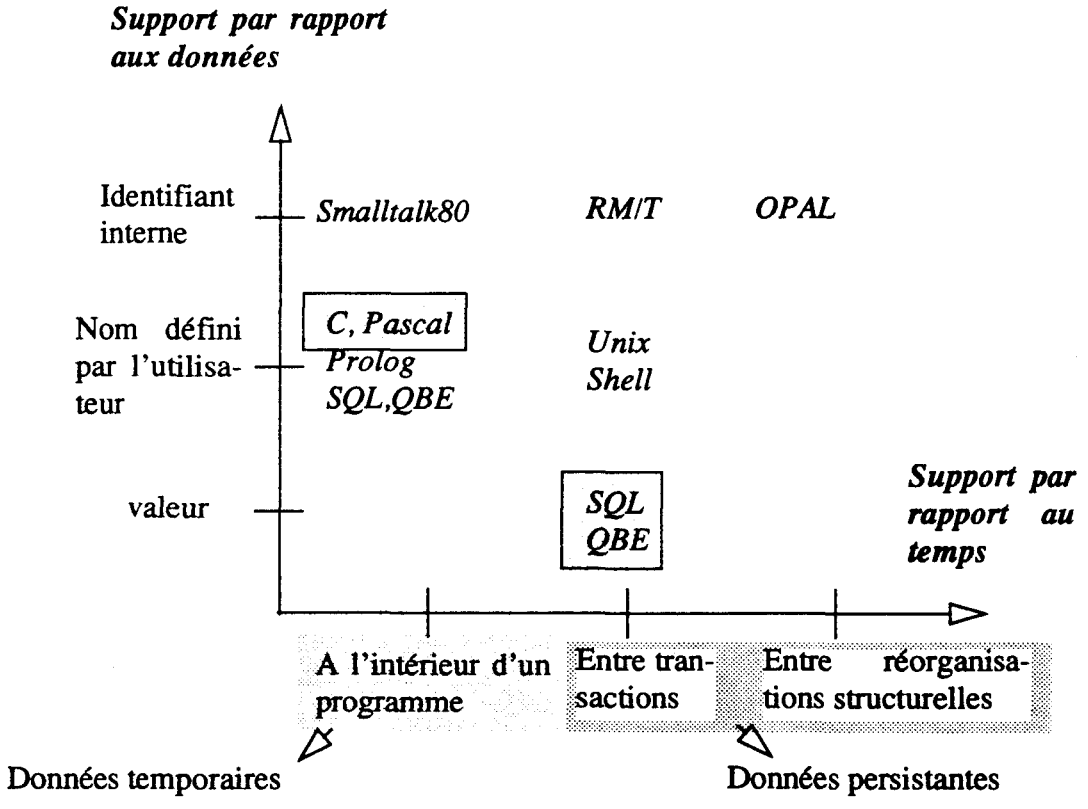


Fig. 1.8 : L'identité des objets au niveau des langages [KHOS86]

Cette figure met en valeur les différences qui existent entre les degrés de support de la notion d'identité d'objets qui se rapportent aux langages de programmation classiques et aux

langages de requêtes. Elle met surtout en évidence le chemin qu'a à faire chacun de ces langages pour supporter honorablement cette notion d'identité. En ce qui concerne les langages de requêtes, il s'agit avant tout de supporter l'identité d'objet à un niveau interne, point que nous avons déjà abordé. Pour les langages de programmation classiques, il s'agit d'obtenir une identité d'objet interne et surtout persistante.

Remarque: Le fait que SQL et QBE (Query By Example) apparaissent une seconde fois avec les langages tels que Pascal et C, est uniquement lié aux variables temporaires que ces langages de requêtes permettent d'utiliser. L'aspect principal de l'identité d'objet pour ces langages reste néanmoins celui de l'identification par rapport aux valeurs.

1.2.2.3. Définition d'un langage idéal

La définition d'un langage idéal de programmation de bases de données peut se résumer de la façon suivante. Un tel langage doit être puissant, doit traiter uniformément les types, l'identité des données et l'exécution des programmes, aussi bien pour des données temporaires que persistantes.

En d'autres termes, il doit permettre d'exprimer n'importe quel traitement sans avoir recours à un quelconque langage hôte. Les notions de type, d'identité et de persistance des données doivent y être orthogonales deux à deux. De plus il est souhaitable que cette identité soit supportée à un niveau interne, point qui dépend bien sûr du modèle de données. Enfin, l'exécution en elle même doit être orthogonale à tous les points précédemment cités, et ne doit pas dépendre du type ou de la nature (persistante ou non) des données traitées.

Pour terminer cette section relative aux évolutions attendues des modèles de données et des langages associés, nous allons présenter les travaux qui s'y rapportent.Il ne s'agit pas ici pour nous de faire une présentation précise et exhaustive de ces travaux. Nous nous contenterons de donner les grandes tendances et les références qui s'y rapportent.

1.2.3. Les solutions proposées

Nous présenterons ici deux familles de solutions, celle basée sur les modèles N1NF, c'est-à-dire les modèles où la contrainte de première forme normale n'existe pas (Non 1 Normal Form), et celle qui correspond à l'approche Orientée Objets. Nous donnerons pour chacune de ces familles, le principe général, les différentes écoles que l'on peut y distinguer, ainsi qu'un certain nombre de références. Nous terminerons cette présentation par la solution intermédiaire que constituent le langage FAD et son modèle de données.

Ce découpage que nous proposons, ne regroupe pas tous les travaux relatifs aux modèles de données et langages associés. Il est important de signaler les travaux sur le modèle Entités-Associations [CHEN76], sur les modèles sémantiques [KING85],[TSUR84] et sur le modèle logique [KUPE85]. Nous n'aborderons pas ici ces travaux.

1.2.3.1. Les modèles N1NF et les langages associés

Tout un ensemble de travaux porte donc sur la relâche de la première forme normale (1NF) attachée au modèle relationnel. Le but recherché est principalement de permettre la création et la manipulation d'objets structurés, encore appelés ***objets complexes***. La demande émane des nouvelles applications pour lesquelles, le support de tels objets, qui reflètent une connaissance hiérarchique des données, est nécessaire. Ces modèles N1NF (ou NF^2) se distinguent cependant du modèle hiérarchique dans le sens où ils sont proposés parallèlement à une algèbre ou à un calcul de prédicats qui permet de manipuler les données obtenues.

D'une façon générale, ces modèles combinent les avantages du modèle relationnel (entre autres l'indépendance données-programmes garantie par l'utilisation d'un langage algébrique ou d'un calcul de prédicats adapté) et ceux du modèle hiérarchique (possibilité d'organisation structurée des données).

Nous ne présenterons pas en détail les différentes solutions proposées qui d'une façon générale sont assez semblables. Seules quelques contraintes les différencient. Le principe est toujours le même, à savoir, que la valeur d'un attribut peut être un nuplet ou une relation. Dès lors des opérateurs algébriques spécifiques sont proposés pour pouvoir manipuler ces données structurées. Les références les plus significatives sur ces travaux sont [ABIT86],[ABIT87], [FISC83], [JAES82] et [VERS86].

Hormis le fait qu'ils permettent de manipuler des objets complexes, ces modèles héritent hélas des désavantages du modèle relationnel. Ainsi ils ne prennent pas du tout en compte la notion d'identité d'objets et ne permettent donc pas de modèlisation directe des données sous forme de graphe (ie partage des données). Les langages algébriques ou langages de prédicats qu'ils supportent ne sont toujours pas complets et souffrent donc des mêmes problèmes d'opposition de phase avec les langages classiques. Enfin les types de bases sont rarement extensibles, ce qui compromet leur utilisation dans le cadre des nouvelles applications.

Ces modèles qui ont fait couler beaucoup d'encre dans la première moitié des années 80, semblent maintenant "oubliés". Cela s'explique certainement par l'avènement du monde Orienté Objet dont les propositions, en ce qui concerne les bases de données, sont beaucoup plus ambitieuses.

1.2.3.2. Orienté Objet et Bases de Données

Pour commencer cette présentation sur l'utilisation de l'approche Orientée Objet pour les bases de données, nous emprunterons la définition suivante des objectifs recherchés, définition tirée de [GARD90]: "L'objectif des spécialistes des bases de données objets, est de développer une nouvelle génération de SGBD, basée sur un nouveau modèle de données et traitements intégrant les objets complexes, les interfaces de programmation et l'extensibilité: le modèle objet".

En ce qui concerne les langages de programmation pour bases de données, l'approche objet peut être vue comme une intégration des technologies des Langages Orientés Objets (LOO) afin de résoudre les problèmes d'interface entre ces langages et les SGBD.

Une fois passées ces définitions ambitieuses, qu'est-il vraiment des travaux ou pro-

duits proposés? A ce sujet il semble que l'on puisse distinguer deux écoles, une qui prend ces sources dans le monde des bases de données et qui tente d'y intégrer les concepts Orientés Objets, et l'autre qui part du monde Orienté Objet et qui essaie d'y glisser les concepts des SGBD. Ces deux écoles s'expliquent tout simplement par la présence "sur le terrain" de deux communautés scientifiques différentes, celle des SGBD et celle des LOO.

Cependant, les travaux qui s'y rapportent ne semblent pas converger, ce qui est dû, comme nous allons le voir aux préoccupation différentes qui existent.

Il est intéressant de remarquer, qu'une situation tout à fait analogue se reproduit autour des Bases de Données Déductives (BDD) [GARD90] qui se veulent être l'union entre le monde des SGBD et celui de la programmation logique.

. Des SGBD Relationnels vers les SGBD Orientés Objets (SGBDOO)

Nous allons donc nous intéresser ici aux travaux qui consistent à ajouter des fonctionnalités Orientées Objets à un SGBDR (Relationnel). Nous ne parlerons pas de la première solution "simpliste" qui consiste à ajouter une couche Orientée Objet au dessus du SGBDR. Cette solution développée pour les projets GEM [ZANO83], [TSUR84]et VOOD [GARD88b] pose bien évidemment d'énormes problèmes d'interface et aboutit logiquement à des produits peu efficaces.

Nous parlerons plutôt de la solution qui consiste à étendre la notion de domaine proposée par Codd [CODD70], et ainsi à intégrer au niveau du modèle relationnel certaines fonctionnalités Orientées Objets.

Codd a défini un ***domaine*** comme un ensemble de valeurs, mais il n'a fait aucune hypothèse sur le type de ces valeurs. Dès lors tout est possible puisqu'il suffit d'étendre le SGBDR pour qu'il puisse inclure de nouveaux domaines, correspondant à des valeurs atomiques de type quelconque et donc aussi complexes que l'on veut.

Cette approche nécessite bien sûr une extension des langages de requêtes habituels pour pouvoir manipuler ces données de tous types. Tous les niveaux du SGBDR, tels que les méthodes de placement, l'optimisation de requêtes, la concurrence, sont aussi à revoir. Suivant le principe de "l'enveloppement" des Types Abstraits de Données (TADs) ou de l'encapsulation des LOO, les opérations associées à un type de données peuvent alors être enveloppées ou encapsulées avec leurs données.

Partant de cette idée, un ensemble de travaux ont abouti à des produits dont les différences résident dans la possibilité de définition de nouveaux opérateurs, le choix du langage d'implantation des domaines, le support de la relation "est un" (sous-types des TADs ou sous-classes des LOO), le mode d'exécution (compilé ou interprété) et l'existence de méthode de placement spécifiques. Nous citerons parmi ces produits SABRINA [GARD88b], RAD [GARD88b] et "INGRES-OO" [STON86a], [STON86b].

Nous ne présenterons pas ces produits qui sont d'ailleurs plus à voir comme des prototypes. Nous nous contenterons de donner un exemple de définition de domaine et de fonctions associées en utilisant la syntaxe du produit SABRINA. Cet exemple n'a d'autre but que d'illustrer le principe général de l'approche SGBDR->SGBDOO. Le langage utilisé pour définir un domaine ou une fonction dans SABRINA est le LISP.

```
(dd#: Rectangle (x)
      (and (listp(x)
       (numberp(carx))
       (numberp(car(cdr x)))
       (null (cdr (cdr x))))
```

Fig. 1.9 : Exemple de définition d'un domaine Rectangle

```
de#: (#:Rectangle): Longueur(x) (car(x)))
de#: (#:Rectangle): Largeur(x) (car(cdr(x)))
de#: (#:Rectangle): Surface(x)  (* (:Longueur x)(:Largeur x)))
```

Fig. 1.10 : Exemple de définitions d'opérations associées au domaine Rectangle

Suite à la définition des domaines et fonctions associées, la description et la manipulation des données se fait grâce à une version étendue de SQL appelée E-SQL (Extended-SQL).

```
CREATE TABLE rectangle_colorés
(numrect int, couleur strings, côtes Rectangle);
```

Fig. 1.11 : Exemple de création de table complexe

```
SELECT couleur FROM rectangle_colorés
WHERE Surface(côtes) > 100;
```

Fig. 1.12 : Exemple de manipulation des données décrites

Ces produits souffrent malheureusement de nombreuses limitations et ne répondent que très partiellement aux problèmes soulevés précédemment. Entre autres, ils ne supportent pas l'identité d'objets, la surcharge d'opérateurs, l'héritage, et sont donc encore bien loin d'être "étiquetables" sous l'emblème "Orienté Objets". Reconnaissons par contre qu'ils possèdent toutes les propriétés qu'un SGBD se doit d'avoir, à savoir un langage de requêtes ad-hoc, la persistance, l'intégrité, la sécurité et le partage des données entre utilisateurs.

Pour en terminer avec cette première école, signalons le projet EDS [GARD88b], dont les objectifs sont d'étendre le modèle relationnel aux Types Abstraits de Données, de fournir une version de SQL étendue et compatible avec les précédentes, d'apporter à ce langage

E-SQL une sémantique fonctionnelle, le tout étant prévu, pour des raisons d'efficacité sur une architecture parallèle. Ce projet semble prendre en compte la composante commerciale (compatibilité de E-SQL) et s'oriente donc vers une solution qui ne remettrait pas tout en cause.

. Des LOO vers les SGBDOO

Nous allons emprunter ici le second chemin qui peut mener aux SGBD Orientés-Objets, chemin qui part des Langages Orientés-Objets. Il s'agit donc ici d'ajouter des fonctionnalités bases de données à un tel langage.

Nous pouvons attribuer aux LOO un certain nombre d'avantages, à savoir leur pouvoir de modèlisation, la notion de classe qui regroupe à la fois la structure et le comportement des objets et enfin leur aspect unifié (ie leur "modèle d'exécution" homogène). Partant de ce constat, nous pouvons donc dire qu'il suffit d'étendre ces LOO vers les accès associatifs, les structures de stockage de base et un environnement multi-utilisateurs pour les transformer en SGBDOO.

Cette seconde approche est typiquement celle qu'ont suivie les concepteurs de Gemstone, SGBDOO ayant eu comme point de départ le langage Orienté-Objet Smalltalk [COPE84]. Ainsi, par ce projet, Smalltalk s'est transformé en OPAL, langage de description et de manipulation de données, mais aussi langage de programmation.

Nous ne décrirons pas ici précisément ce produit. Par contre, nous allons, dans ce qui suit, mettre en avant certaines de ses insuffisances, ce qui nous permettra de faire quelques remarques générales sur la démarche utilisée.

Revenons tout d'abord sur les accès associatifs. Comme toutes les SGBDOO (produits commerciaux ou prototypes), Gemstone supporte des accès associatifs au travers d'un langage de requêtes. Ainsi les données peuvent être accédées de façon associative sans avoir recours aux inévitables passages de messages chers aux Langages Orientés Objets. Sur ce point, tout le monde semble d'accord pour dire qu'un SGBD sans accès associatifs n'est pas viable. En d'autres termes, aucun produit ne respecte totalement la règle de l'encapsulation. On en arrive alors souvent à deux types de fonctionnement, le fonctionnement "puriste" par passage de messages et celui par accès associatifs.

Cette situation peut se résumer par la figure ci-dessous, sur laquelle la frontière entre parties visible et invisible d'un objet "recule d'un cran".

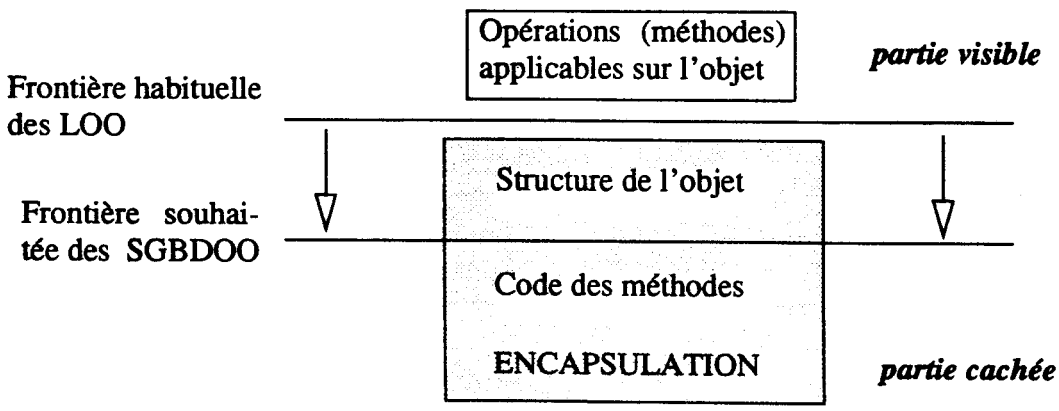


Fig. 1.13 : Visibilité souhaitée dans les SGBDOO

Ce support des accès associatifs est, bien sûr, pour des raisons de performances, tout à fait souhaitable. Pour l'instant, les produits ne proposent pas une optimisation automatique des requêtes ainsi exprimées. Par exemple, dans la version analysée de Gemstone [MAIE86], c'est "l'utilisateur qui décide de l'opportunité d'utiliser ou pas les index"! Sur ce point, il semble presque que nous ayons remonté le temps pour en revenir au moment où les requêtes basées sur l'algèbre relationnelle n'étaient pas optimisées.

Les autres points surprenants de ce produit sont, d'une part, la dépendance qui existe entre la notion de concurrence et d'autorisation et, d'autre part, la responsabilité de l'administrateur sur cette concurrence. Un segment se définit comme un groupe logique d'objets qui forment une unité de concurrence. Cependant, ces fragments sont contrôlables par l'administrateur et constituent aussi l'unité de propriétés et d'autorisations! Cette amalgame de notions habituellement (ie dans le contexte des SGBD) orthogonales est pour le moins surprenant mais est surtout extrêmement contraignant.

La mise en avant de ces quelques points précis n'avait d'autre but que de montrer, qu'avec cette seconde approche, les produits obtenus péchent par l'insuffisance de leur fonctionnalités bases de données. Nous n'aborderons pas ici d'autres produits issus de cette approche, mais quels qu'ils soient, il existe toujours un aspect "bases de données" négligé ou absent et, dans tous les cas, les performances obtenues laissent fortement à désirer.

A titre indicatif, les principaux produits sont:

- ORION réalisé au MCC d'Austin [KIM87],[WOEL87],

- GBASE développé par l'université de Compiegne et commercialisé par GRAPHAEL [GRAP87],

- VBASE de chez ONTOGIC [ONTO88] (produit déjà abandonné),

- ONTOS successeur de VBASE [ANDR88]

- IRIS de chez HEWLETT-PACKARD [FISH88],

- GEMSTONE commercialisé par SERVIO LOGIC [MAIE86] et

- O2 du groupement public ALTAIR [LECL88] [BANC88].

En ce qui concerne ce dernier produit O_2, la démarche utilisée semble plus réfléchie. Les concepteurs ne sont pas partis de l'un ou l'autre des bords (SGBDR ou LOO) pour, ensuite, ajouter tant bien que mal les fonctionnalités manquantes. Ils se sont basés sur une étude relative aux oppositions naturelles qui existent entre les LOO et les Bases de Données. De là, ils ont tiré un certain nombre de conclusions qui leur ont permis de prendre les orientations nécessaires pour leur produit. Cette approche "sage" nous semble être la seule capable d'aboutir un jour à un consensus pour les bases de données Orientés-Objets, terme sous lequel on trouve actuellement "un peu de tout".Sachant qu'il a fallu 20 ans au modèle relationnel pour "s'affirmer", alors que les principes étaient simples et stables, nous pouvons nous demander ce qu'il va advenir de la "vague" SGBDOO.

Nous terminerons cette aparté sur la solution Orientée-Objets par la figure suivante qui, nous semble-t-il, résume bien la situation actuelle.

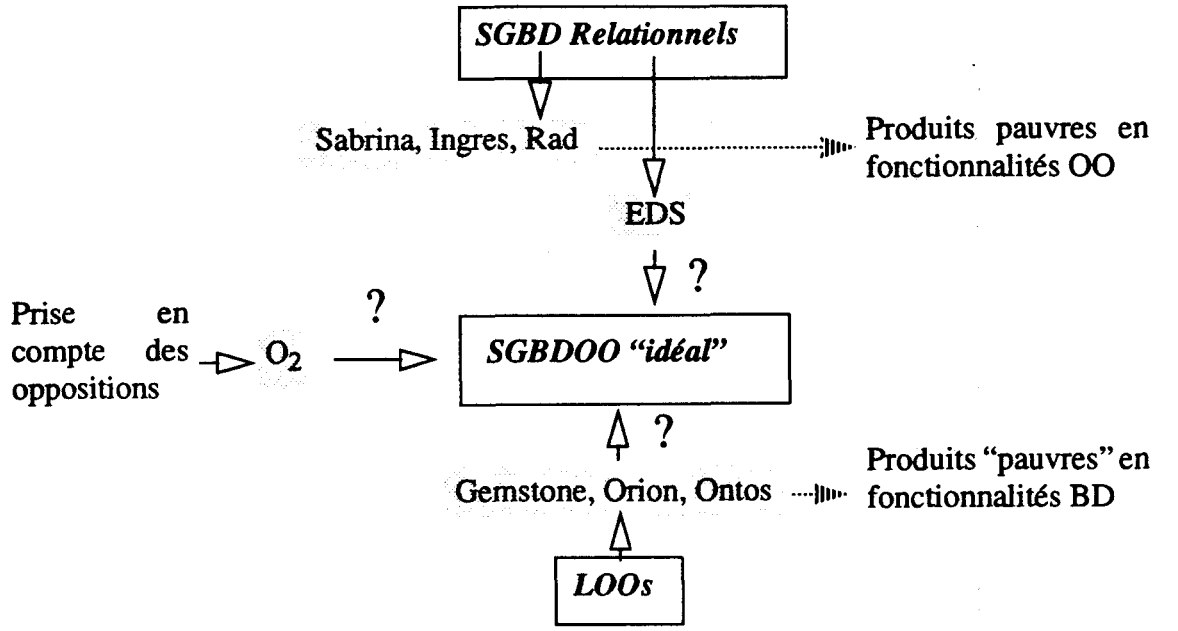


Fig. 1.14 : Situation des différentes approches SGBDOO

1.2.3.3. Une solution particulière: le langage FAD et son modèle de données.

Pour terminer cette section sur les solutions proposées, nous aborderons ici le cas du langage FAD et de son modèle de données.

FAD (Franco Armenian Data-language) est un langage de programmation fonctionnel qui a pour but la ***création et la manipulation d'objets complexes*** à l'intérieur d'un ***système bases de données hautement parallèle***. Il a été développé au MCC (Microelectronics and Computer technology Corporation) d'Austin (Texas) [BANC87], [DANF87], [DANF88a].

Il a, d'autre part, été implanté sur la machine parallèle BUBBA du MCC [BORA88], [HART88], [DANF88b], [VALD88].

Vu sous l'angle du langage de Description de Données (LDD), le langage FAD repose sur une couche de Types Abstraits de Données, ce qui lui confère des capacités de modélisation "infinies". L'identité d'objets est directement supportée par le modèle à un niveau interne (ie identifiants indépendants de toute valeur). Ainsi, les objets complexes, modélisés par combinaison arbitraire de nuplets, ensembles et disjonctions[1], sont partageables. Bien que les concepteurs ne lui attribuent pas cette étiquette, le modèle de données est très voisin des modèles de données Orientés-Objets [LECL88] [KIM87].

Vu maintenant sous l'angle du Langage de Manipulation de Données (LMD), FAD est de nature fonctionnelle, ce qui permet d'utiliser la composition fonctionnelle pour enchaîner les requêtes tout en conservant un formalisme déclaratif. Il a l'avantage d'être complet; il ne souffre pas en effet d'une quelconque limitation qui obligerait l'utilisateur à avoir recours à un

1. Une disjonction est une forme particulière de nuplet dans laquelle, pour un objet donné, seul un des attributs peut ne pas être nul.

langage hôte. Il est fortement typé et supporte évidemment la persistance des données. En résumé, les notions d'identité, de persistance, de type et d'exécution y sont orthogonales deux à deux.

Enfin, il est important de noter la composante parallèle du langage, qui permet à un utilisateur d'exprimer explicitement du parallélisme entre actions et d'utiliser des constructeurs d'actions dont l'exécution est implicitement parallèle. Notons bien qu'il ne s'agit là que d'une forme déclarative de parallélisme, l'utilisateur n'ayant à aucun moment une possibilité de contrôle effectif de ce parallélisme.

En conclusion, le modèle de données supporté a toutes les propriétés requises; le langage quant à lui, répond parfaitement aux attentes (cf 1.2.2), de plus il possède une composante parallèle qui n'est pas dénuée d'intérêt si l'on considère les nombreux travaux relatifs aux machines bases de données parallèles (cf 1.3).

Pour ces raisons, nous avons choisi ce langage et son modèle de données comme support de nos travaux sur le placement de données. Nous consacrerons donc dans ce qui suit, une section pour en faire une description plus précise (cf 1.4.2 Présentation de FAD).

Nous allons maintenant discuter de l'utilisation de machines parallèles pour les bases de données. Suite à cela nous présenterons le cadre et les objectifs de notre travail.

1.3. Sur l'utilisation de machines parallèles

L'utilisation des machines parallèles constitue l'autre aspect de l'évolution des Systèmes de Gestion de Bases de Données. Le besoin toujours plus grand de performances, explique cet engouement. Dans ce qui suit, nous consacrerons une section assez générale sur le parallélisme dans les bases de données, puis une section sur chacune des approches que nous distinguons, à savoir l'approche par spécialisation et l'approche sans partage.

1.3.1. Parallélisme et bases de données

1.3.1.1. Les différents types de parallélisme

Avant de présenter les travaux relatifs aux machines bases de données parallèles, il nous semble nécessaire de faire le point sur les différents types de parallélisme que l'on peut distinguer dans le contexte des bases de données.

. Le parallélisme inter-requêtes

Ce premier type de parallélisme correspond à l'exécution simultanée de différentes requêtes. Il faut pour cela que les données soient au moins partiellement différentes. Il est important de noter que cette forme de parallélisme existe déjà dans les systèmes classiques (ie non

parallèles) où, grâce à la ***sérialisation***, les requêtes de différents utilisateurs sont exécutées "simultanément". Il faut bien voir cependant, que dans ce cas, il ne s'agit pas d'une exécution parallèle réelle, mais plutôt d'une exécution séquentielle enchevêtrée des requêtes.

Grâce à une configuration parallèle[1], une véritable exécution parallèle est tout à fait envisageable et nous conduit alors à un fonctionnement de type MIMD[2], c'est-à-dire Multiple Instructions Multiple Data, ou en d'autres termes, exécution simultanée de différentes instructions sur différents flots de données.

Si nous nous référons à la classification proposée par J.P. Sansonnet [SANS90], nous pouvons dire que ce parallélisme inter-requêtes est une forme particulière de ***parallélisme de contrôle***, parallélisme de contrôle définit comme la possibilité "de faire différentes choses en même temps".

. Le parallélisme intra-requête

Ce second type de parallélisme correspond à une exécution parallèle d'une même requête. Il peut prendre deux formes suivant qu'il consiste à exécuter parallèlement différentes opérations de la requête ou, à appliquer la même opération à différentes données.

La première forme peut être assimilée à l'évaluation parallèle des différents arguments de la fonction que constitue, de façon imagée, la requête. Puisqu'il s'agit de faire différentes choses parallèlement, cette première forme de parallélisme intra-requête est donc une autre forme de parallélisme de contrôle et conduit donc aussi à un fonctionnement MIMD.

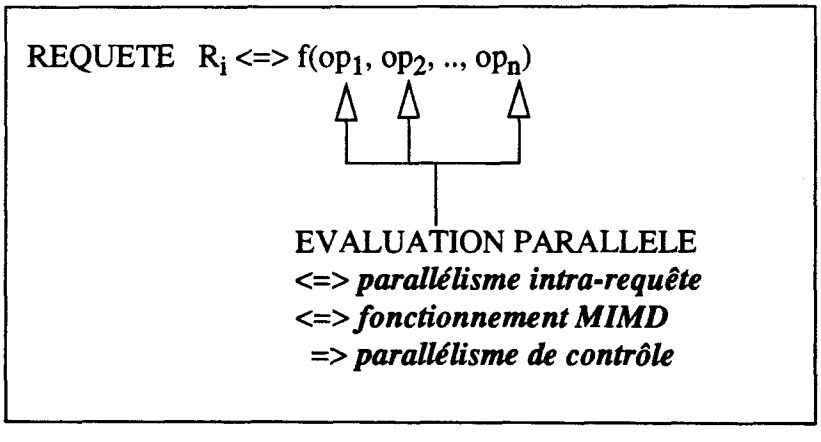


Fig. 1.15 : Première forme de parallélisme intra-requête

La seconde forme de parallélisme intra-requête, consiste à appliquer la même opération sur différents flots de données, et induit donc un fonctionnement SIMD (Single Instruction Multiple Data). Il correspond à la notion de ***parallélisme de données*** définie dans [SANS90].

---

1. Nous resterons pour l'instant volontairement vague sur la définition d'une telle configuration, cet aspect faisant l'objet d'une présentation ultérieure (cf 1.3.2 et 1.3.3).
2. Nous utilisons ici la classification proposée par Flynn [FLYNN66].

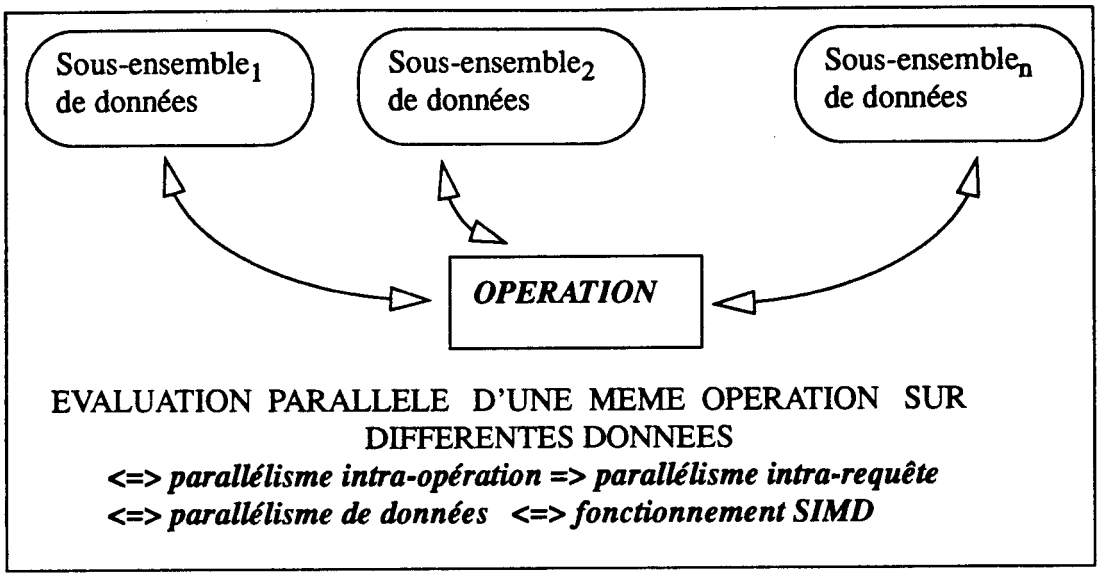


Fig. 1.16 : Seconde forme de parallélisme intra-requête

De par les volumes importants de données généralement manipulés dans les applications bases de données, cette dernière forme de parallélisme est sans nul doute "le gisement de parallélisme" le plus prometteur. Bien que les autres formes ne soient pas à négliger, c'est grâce à cette dernière que les gains obtenus seront les plus importants.

1.3.1.2. Parallélisme et bases de données: les différentes approches

Après avoir présenté les différents types de parallélisme, nous allons maintenant aborder les travaux qui se rapportent à l'utilisation de machines parallèles ou machines spécialisées. Nous pourrions établir pour ces travaux différentes classifications en fonction du matériel, du mode de fonctionnement de la machine ou encore du modèle de données supporté. Il en est une cependant, que nous jugeons suffisamment importante pour la placer au premier plan. Il s'agit de celle qui distingue les machines ou projets qui reposent sur l'***approche sans-partage*** (***shared nothing***), des autres que nous regrouperons sous l'emblème ***approche par spécialisation***. Notons que cette classification nous est tout à fait personnelle. Elle constitue pratiquement néanmoins, une classification dans le temps des différents travaux.

Dans le début des années 70, les systèmes classiques étant jugés insuffisants, la notion de ***Machines Bases de Données*** est apparue. L'idée était alors de décentraliser le traitement des données, et donc de reporter une partie plus ou moins importante du SGBD vers un calculateur spécialisé, périphérique du calculateur central et appelé ***dorsal*** (***Back-End***).

Ainsi la spécialisation matérielle de bas niveau en vue de la gestion de données et la duplication éventuelle des organes matériels spécialisés obtenus, ont abouti à la réalisation de différentes machines, plus ou moins parallèles, que nous considérerons donc issues de l'***approche par spécialisation***. Nous consacrerons à ces machines la section suivante (cf 1.3.2 Approche par spécialisation).

Là dessus, dans les années 83-84, une nouvelle façon de voir les machines bases de données apparait. Elle consiste à définir une telle machine comme un ensemble de noeuds qui disposent de capacités de stockage et de traitement, ces noeuds étant reliés par un réseau d'inter-connexions et communiquant entre eux par envoi de messages. De telles machines correspondent à l'approche ***sans-partage*** (shared nothing) définie par M. Stonebraker dans [STON86a].

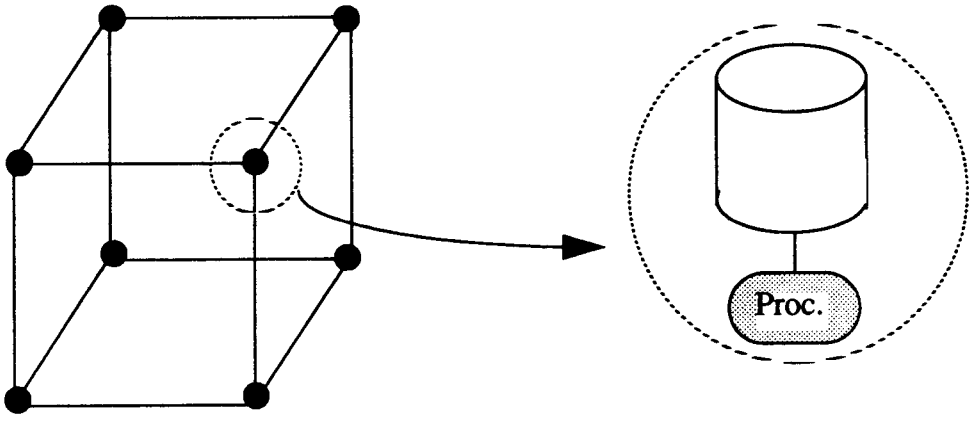


Fig. 1.17 : Principe de l'approche sans-partage

Ces machines peuvent donc être considérées comme des machines MIMD à mémoire distribuée. Mais puisque les données peuvent être "éclatées" sur les différents noeuds, un fonctionnement SIMD est aussi possible, ce qui conduit généralement à parler de machines MSIMD à mémoire distribuée.

Il est important de signaler que différentes interprétations peuvent être données à la notion de noeuds. Il peut s'agir des noeuds d'une machine parallèle classique (ex: IPSC/2 de chez Intel [PIER89]), ou de noeuds spécialisés "bases de données" (ex: DBC/1012 de chez Teradata [TERA84]), ou encore des noeuds d'un réseau local classique (ex: stations de travail).

Bien que l'échelle soit différente, cette approche sans-partage conduit à des solutions qui ne sont pas sans liaison avec le domaine des bases de données réparties [CERI85].

En fait, ces deux approches, par spécialisation et sans-partage, trouvent chacune leur justification dans le temps. Dans les années 70, l'informatique était une "science centralisatrice", non pas pour des raisons philosophiques mais plutôt pour des raisons matérielles liées aux capacités technologiques du moment et donc au coûts. La solution pour pallier les insuffisances des systèmes classiques ne pouvait être que le recours à des machines spécialisées de bas niveau.

Tout cela a fondamentalement changé en quelques années. D'énormes progrès technologiques dans tous les domaines (processeurs, disques, etc) sont intervenus. L'arrivée, grâce à ces progrès, de la micro informatique et des réseaux locaux a totalement bouleversé le paysage informatique. A part pour des applications très particulières telles que le calcul vectoriel ou la synthèse d'image, l'approche "machine spécialisée" perd peu à peu du terrain au profit de configurations plus générales, modulaires et surtout adaptables (machines à base de transputers, Connection Machine, machine hypercube, etc). Outre l'aspect adaptable et modulaire, les

constructeurs tirent aussi de ces machines des coûts de développement et de maintenance réduits. A ce titre l'article de D.A. Petterson est tout à fait significatif [PATT87].

Nous allons dans ce qui suit dresser une bibliographie des principaux travaux qui se rapportent à chacune des approches discernées. Nous passerons très vite sur l'approche par spécialisation qui, d'une part semble de plus en plus délaissée, et d'autre part sort du cadre de nos travaux. Par contre pour l'approche sans partage, nous consacrerons quelques lignes à chacun des projets ou réalisations actuels.

1.3.2. L'approche par spécialisation

Nous nous contenterons ici de donner les références des principaux travaux relatifs à cette approche, travaux qui, rappelons le, consistaient à construire une machine bases de données par spécialisation matérielle de bas niveau et duplication éventuelle de certains des organes matériels spécialisés obtenus.

Les quatre machines commercialisées sont DORSAL32 de Copernique [COPE82], IDM de Britton-Lee [BRIT82], IDBP de Intel[INTE81] et CAFS de ICL [BABB79], [ICL82].

En ce qui concerne l'ensemble des projets nous reprendrons la classification proposée dans [GARD88], classification qui est faite par rapport aux critères établis par Flynn [FLYNN66] à savoir simple ou multiple instructions (SI ou MI)et, simple ou multiple flot de données (SD ou MD).

	SI	*MI*
SD	IDM [BRIT82] IDBP [INTE81] DORSAL [COPE82] CAFS [BABB79] DB.1 [OKI84]	VERSO [BANC83]
MD	DIRECT [DEWI79] CASSM [SU75] SARI (Sintra France)	DBC [BANE78] DBMAC [MISS83] RDBM [SCHW83] SABRE [GARD81] DELTA [MURA83]

Machines commercialisées

Fig. 1.18 : Classification des machines issues de l'approche par spécialisation

Nous conclurons sur cette approche en faisant remarquer qu'elle ne semble plus

aujourd'hui "faire couler d'encre". La trop forte spécialisation matérielle et les coûts importants de développement rendaient fragiles de telles machines devant les progrès rapides dont bénéficiaient parallèlement les systèmes classiques. Replacées dans le contexte actuel, il y a fort à penser que la plupart de ces machines offriraient aujourd'hui des performances moins bonnes qu'un bon SGBD implanté sur un ordinateur performant. Hormis les aspects coûts et performances, la structure figée ou tout au moins limitée de ces machines constitue aussi un sérieux handicap à l'heure où la modularité, le partage et les communications sont plus que jamais attendus.

1.3.3. L'approche sans partage

1.3.3.1. Considérations générales

Comme nous l'avons signalé, dans cette approche sans-partage la définition d'une machine bases de données se limite à: "Ensemble de noeuds disposant de capacités privées de traitement et stockage, et communiquant par messages au travers d'un réseau d'inter-connexions" (cf figure 1.17).

Notons au passage, qu'avec cette définition générale, rien n'empêche d'avoir au niveau d'un noeud une quelconque forme de spécialisation matérielle plutôt qu'un processeur et un disque "classiques".Néanmoins, cela n'altère en rien la composante "sans-partage".

Pour ce type de machine, il est coutume de parler d'une ***exécution des requêtes "là où les données se trouvent"***. Une requête y est décomposée en sous-requêtes[1] en fonction de la localisation des données qu'elle accède. Chacune des sous-requêtes est alors envoyée vers le noeud qui lui correspond. Ce mouvement initial du code vers les données constituent une des principales caractéristiques de l'approche sans partage.

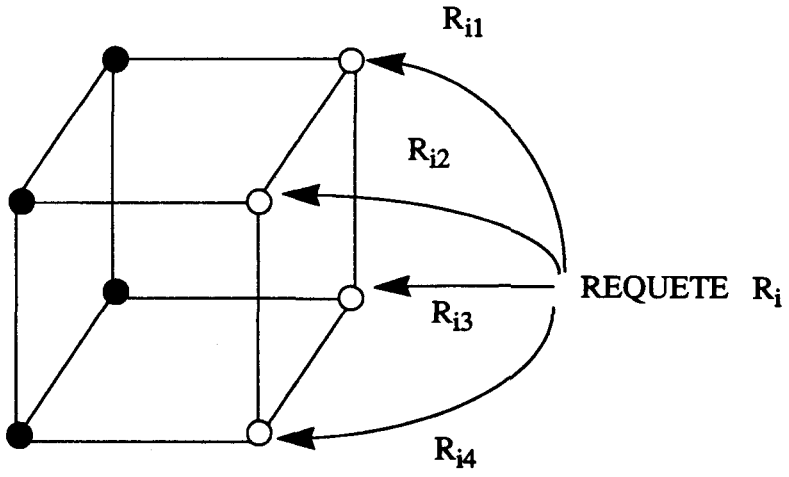
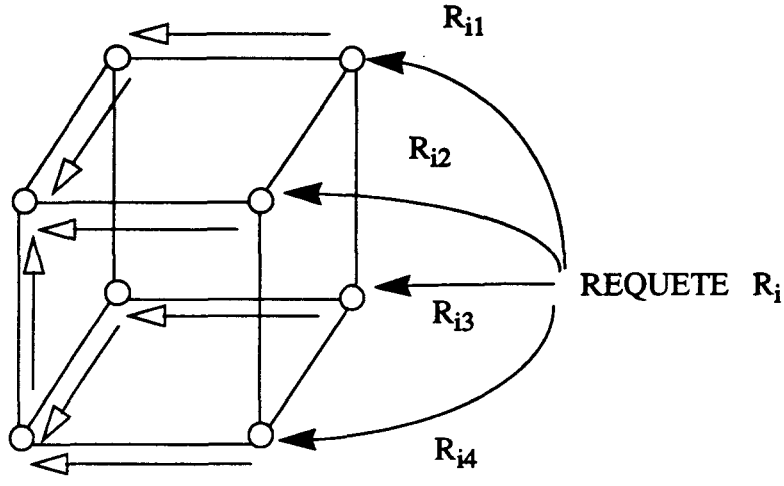


Fig. 1.19 : Principe de l'exécution "là où les données se trouvent"

1. Le terme sous-requêtes n'a ici rien à voir avec celui propre au langage SQL

Avec cette approche les transferts de données sont donc limités. Ce n'est pas pour autant qu'il n'en n'existe pas, puisque la récupération de résultats ou l'utilisation simultanée de données de différentes provenances, nécessite de tels transferts. Néanmoins, si de bonnes techniques d'optimisation sont supportées, ces transferts sont minimaux.

Le fonctionnement général d'une telle machine est le plus souvent de type ***dataflow*** [DEWI86], [DANF88b], [HART88], [LORI89]. En effet, une fois les données de base traitées, c'est l'arrivée d'autres données sous forme de messages qui permet de continuer, s'il y a lieu, l'exécution d'une requête.



⟶ <=> ***Phase 1: exécution là où les données se trouvent***

⟶▷ <=> ***Phase 2: Fin de la requête en mode dataflow***

Fig. 1.20 : Principe général d'une exécution dataflow

Nous en resterons là pour le moment sur le principe de fonctionnement d'une telle machine. Nous détaillerons précisément dans le dernier chapitre, à partir du langage FAD (cf 1.2.3.3) et de la machine BUBBA [COPE88], [BORA88], un exemple de ce type de fonctionnement. Cette présentation aura pour objet de donner un exemple d'intégration des solutions que nous préconisons en matière de placement de données sur ces machines sans partage.

Il va sans dire que ce placement des données sur les noeuds est d'une importance primordiale. Il constitue à ce titre le cadre de nos travaux. Avant d'entrer plus en détail sur ces travaux, nous allons consacrer quelques lignes à chacun des projets ou réalisations relatifs à l'approche sans partage.

1.3.3.2. La machine GAMMA

GAMMA [DEWI86] est une machine bases de données relationnelle issue de l'Université de Wisconsin. Elle est le symbole même de l'évolution entre l'approche par spécialisation et l'approche sans partage, puisque ses concepteurs ne sont autres que ceux de la machine DIRECT [DEWI79]. Ces derniers admettent avoir tiré une leçon des résultats, sur certains points désastreux, de leur première machine, pour prendre les orientations retenues dans GAMMA.

Ainsi ils sont conscients que la technologie qui se rapporte aux processeurs, évolue beaucoup plus rapidement que celle attachée aux disques. En conséquence, ils préconisent naturellement l'utilisation d'un système multi-disques. Ils se réfèrent ensuite à la règle des 90/10 qui dit que la plupart des données accédées par une requête, ne sont pas immédiatement nécessaires, ou en d'autres termes, qu'il n'est pas utile de transférer toutes les données concernées pour pouvoir commencer l'exécution d'une requête. Cette seconde remarque les conduit alors à s'orienter vers une architecture sans-partage, où chaque noeud est constitué d'un processeur et éventuellement d'un disque.

L'architecture du prototype cité dans [DEWI86] est constituée d'un ensemble de 20 VAX 11/750 reliés en anneau par un réseau de type "token ring" spécialisé. Huit de ces VAX sont pourvus d'un disque de 160M.

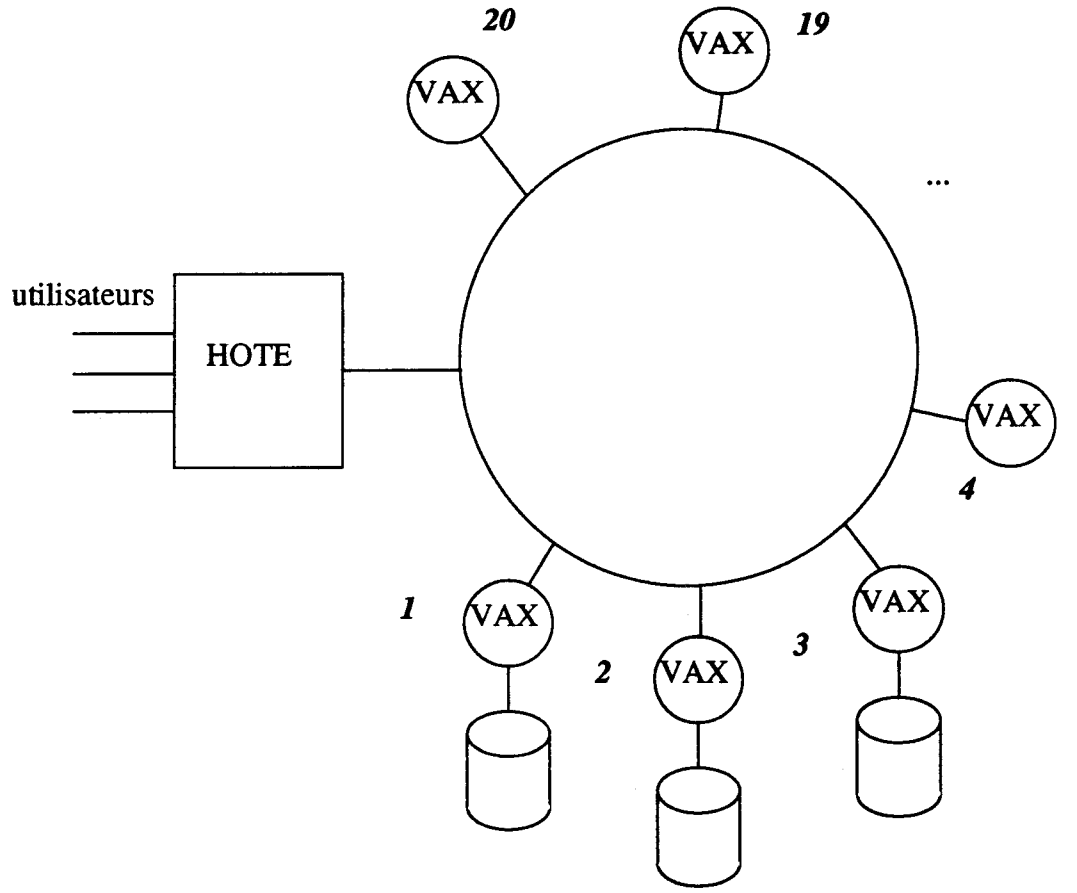


Fig. 1.21 : Configuration matérielle de la machine GAMMA

Globalement, GAMMA utilise les techniques relationnelles classiques pour "découper" et optimiser les requêtes, et pour générer le code. Les requêtes sont compilées en un arbre d'opérateurs. A l'exécution, chacun des opérateurs est exécuté par autant de processus qu'il existe de sites (noeuds) concernés. Nous n'entrerons pas plus ici dans le détail, qui est surtout relatif à l'exécution parallèle des opérateurs de l'algèbre relationnelle. En ce qui concerne le placement des données sur les sites, nous y ferons allusion lorsque nous aborderons, au sein de la présentation de notre travail, les problèmes relatifs à la distribution de données (cf chapitre 3).

1.3.3.3. La machine BUBBA

La machine BUBBA est développée au MCC d'Austin [COPE88], [DANF88b]. Elle a l'objectif ambitieux de supporter efficacement les applications bases de données et bases de connaissances des années 95.

Elle est composée d'Intelligent Repositories (IRS, ie "dépôts intelligents") et de processeurs d'interface (IPs). Chaque IR dispose d'un processeur, d'un contrôleur disque, d'un processeur de communication, d'une mémoire principale et d'un disque. L'approche étant sans partage, ni les disques, ni les mémoires ne sont partagés entre les IRs. Les IPs ont pour rôle de gérer les interfaces avec les utilisateurs ainsi que d'exécuter certaines fonctions centralisées. Les IRs et IPs sont connectés par un réseau d'inter-connexions de telle façon à ce que tout IR ou IP puisse envoyer un message à tout autre IR ou tout autre IP. Ce réseau est donc la seule ressource partagée.

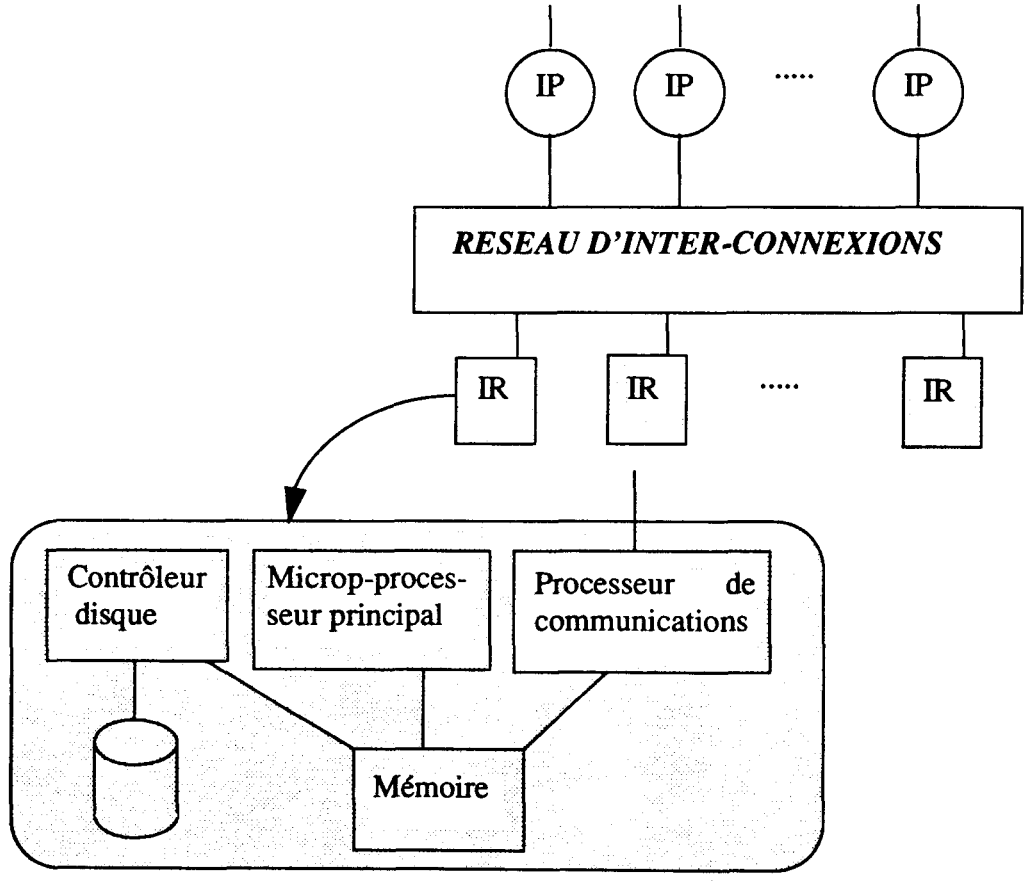


Fig. 1.22 : Configuration matérielle de la machine BUBBA

Nous signalerons pour finir, l'existence d'un mécanisme de chargement et d'activation dynamique, qui démarre une transaction sur un IR dès que le premier message est arrivé. Enfin, notons que plusieurs mécanismes de contrôle dataflow sont disponibles, ce qui permet au compilateur de choisir celui qui entraîne, pour un programme donné[1], le minimum de surcoût en communications [ALEX88].

Nous reviendrons à plusieurs reprises sur cette machine lors de la présentation de notre travail, notamment dans le chapitre 4 sur l'exemple d'intégration de nos solutions dans les phases de compilation du langage FAD.

1.3.3.4. La machine DBC/1012 de chez TERADATA

La première version de cette machine DBC/1012, développée par la société Teradata, date de fin 83. Elle est composée de 6 à 1024 "micro-ordinateurs" travaillant en parallèle, et ayant chacun une puissance de 0,4 MIPS[1]. Quelques uns de ces micros ordinateurs sont programmés pour gérer l'interface avec l'ordinateur hôte. Les autres gèrent des disques où les données sont stockées sous forme relationnelle.

En ce qui concerne les motivations pour l'utilisation d'une telle architecture, l'article "The genesis of a database computer" [TERA84] est tout à fait intéressant, puisqu'il analyse le problème par rapport à l'aspect technique et l'aspect commercial, mais aussi par rapport aux évolutions pressenties (en 1984) du paysage informatique.

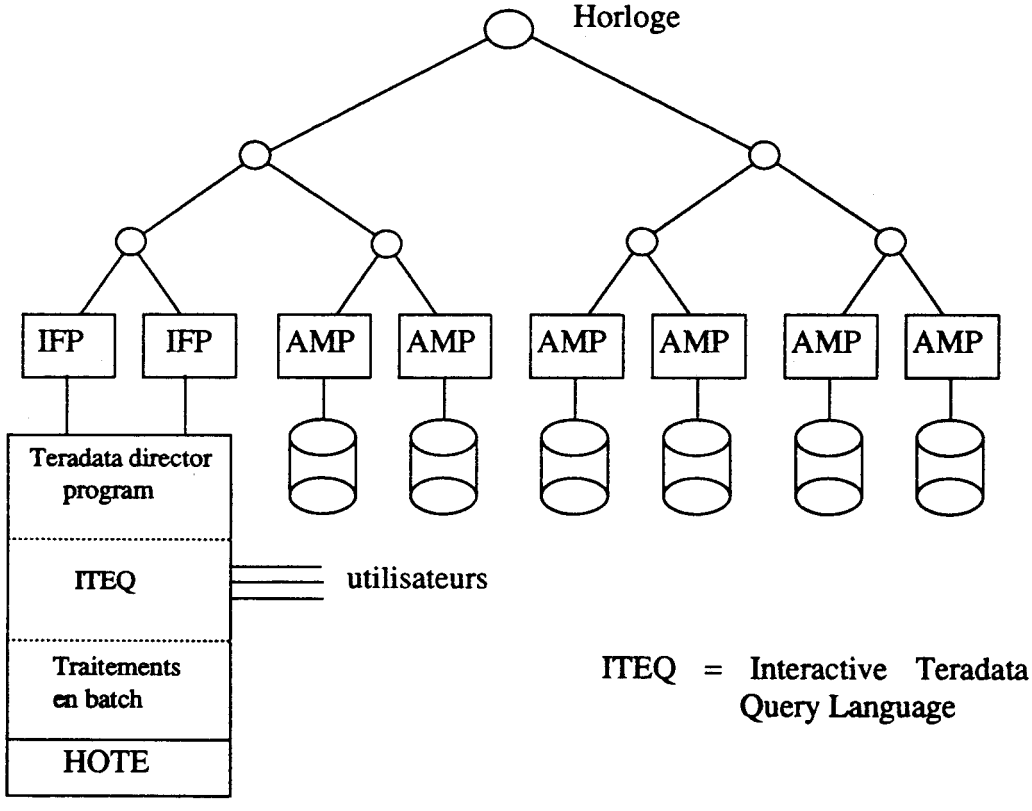


Fig. 1.23 : Architecture de la machine DBC/1012 (Teradata)

Revenons plus précisément sur l'architecture de la machine. Elle est donc composée de processeurs d'interface (notés IFP) et de processeurs d'accès (notés AMP, ie Acces Module Processor). L'unité principale de traitement de ces IFPs et AMPs est constituée d'un micro

---

1. Nous nous plaçons ici dans le contexte d'un programme écrit avec le langage FAD, langage dont une implantation a été réalisée sur la machine BUBBA.
1. Ces chiffres datent de 1984 et sont certainement différents maintenant. Nous ne disposons, à ce jour, d'aucune autre référence; il en existe cependant au moins une autre qui est : [NECH87].

processeur 8086 et d'un co-processeur arithmétique 8087 de chez Intel. Chaque AMP gère un disque de 474 M (noté DSU, ie Disk Storage Unit). L'ensemble de ces processeurs est relié par un réseau en Y, dont l'aspect général est donc un arbre binaire (cf figure 1.23).

Les processeurs d'interface ont pour rôle de traduire les requêtes en provenance de l'ordinateur hôte, en commandes internes qu'ils envoient, alors, sur le réseau vers les processeurs d'accès. Ces derniers recherchent, sur leur disque, les données demandées, et les placent alors sur le réseau qui les fusionne dans l'ordre désiré par l'utilisateur.

1.3.3.5. La "machine" TANDEM

Cette "machine" ou plutôt ce projet, est issu de la collaboration de plusieurs laboratoires et est coordonné par la société Tandem [TAND87].Le terme machine est à prendre ici entre guillemets, tant l'échelle considérée est importante.

Bien que le principe sans-partage reste de mise, ce projet prend une nouvelle dimension puisque le système peut s'étendre en ajoutant des processeurs ou des disques à une grappe locale, mais aussi en ajoutant des grappes à un site au travers d'un réseau de fibres optiques et enfin en ajoutant des sites sur un réseau de communication "public".

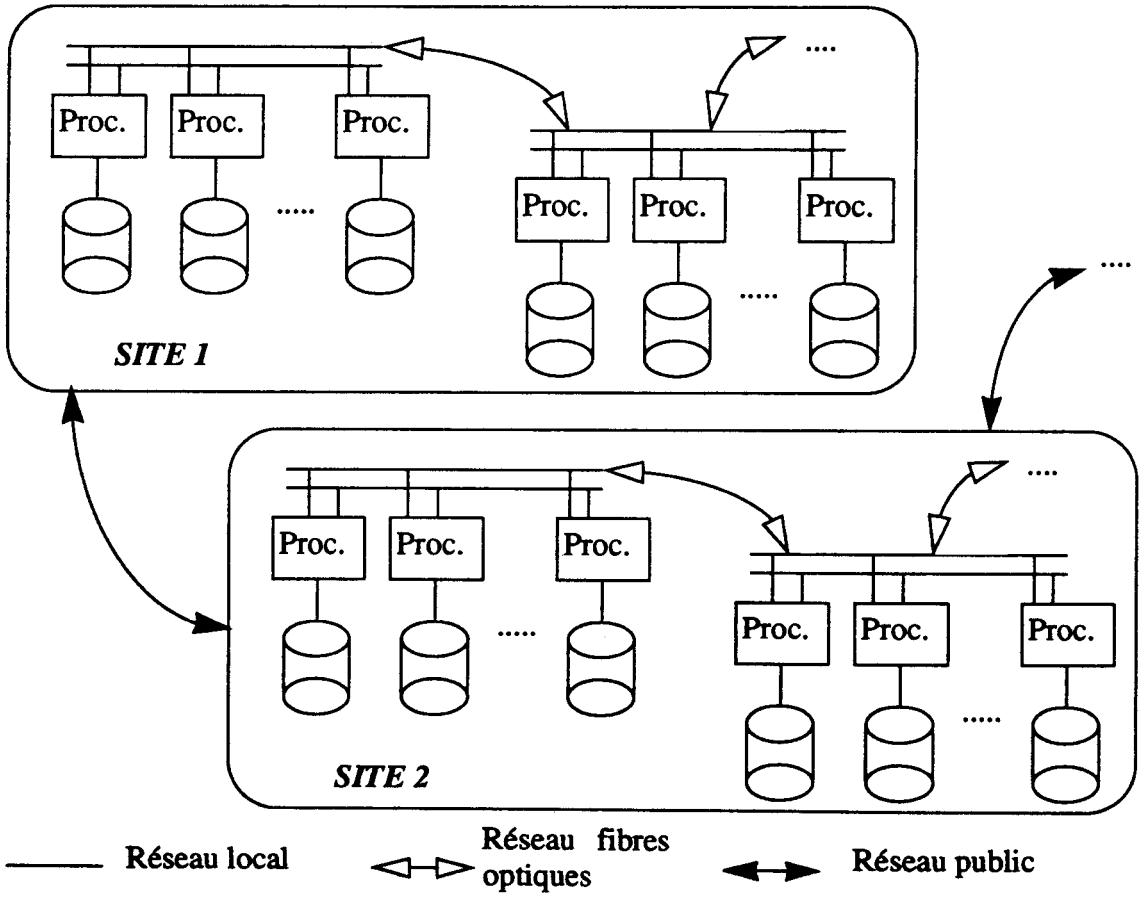


Fig. 1.24 : Configuration associée à l'approche TANDEM

Les chiffres maximaux annoncés sont les suivants:

- 16 processeurs par grappe, reliés par un réseau local dual dont le double rôle est de contrôler les communications et les disques,

- 224 processeurs par site, c'est-à-dire 14 grappes reliées par fibres optiques,

- jusque 4000 processeurs au total, c'est-à-dire environ 16 sites reliés par un réseau public quelconque.

Une version "NonStop" de SQL, a été développée pour cette "machine".

Cette architecture ambitieuse marie donc l'aspect Bases de Données Réparties et l'aspect Bases de Données Parallèles. Néanmoins, il est important de signaler que dans la première version, l'aspect parallèle a été partiellement repoussé. Nous ne disposons pas à ce jour, d'autres informations sur une éventuelle évolution dans ce sens.

1.3.3.6. La machine ARBRE

Pour ce projet récent, issu du centre de recherche Amalden de IBM à San-Jose (ARBRE = the Amalden Research Backend Relational Engine), hormis l'approche sans-partage, aucune hypothèse précise n'est faite relativement à la configuration matérielle [LORI89]. Les concepteurs supposent donc juste l'existence d'un ensemble de sites physiques (noeuds) qui disposent de capacités de traitement et de stockage, et qui communiquent par envoi de messages.

Leur intention est de réutiliser la majorité du code d'un SGBD Relationnel classique, ie non parallèle, et d'exploiter logiquement plusieurs instances d'un tel SGBD afin d'introduire du parallélisme intra-requête. Chaque instance sera alors responsable d'une partie de la base de données et sera associée à un processeur virtuel. Différents "plaquages" de ces processeurs virtuels sur les sites physiques seront ainsi réalisables. Leur approche étant avant tout caractérisée par cette couche virtuelle, les auteurs l'appellent ***virtualisation***.

En fait, ils reconnaissent être uniquement intéressés par le parallélisme, et en ce sens, ils n'essaient pas d'améliorer les performances des opérations locales exécutées sur un même processeur. Ils prennent donc les systèmes courants tels qu'ils sont et comptent tirer parti des améliorations que le matériel et le logiciel "local" enregistreront.

Enfin, le langage utilisé est SQL; les requêtes sont découpées en fragments de recherche qui sont, comme pour GAMMA (cf 1.3.3.2) exécutés autant de fois qu'il existe de sites concernés. Le modèle d'exécution est de type dataflow.

Ce projet se distingue donc des autres (GAMMA, DBC/1012, TANDEM) par le fait qu'il est le seul à permettre différents "plaquages" d'instances logiques sur les sites physiques. D'autre part, il laisse toute liberté matérielle et même, une certaine liberté logicielle, puisqu'il ne se préoccupe que des aspects parallèles. En mars 89, seul un prototype existait. Les auteurs comptaient s'en servir pour analyser certains points inhérents au parallélisme dans les bases de données.

Ainsi se termine la brève présentation de quelques machines qui répondent à l'approche sans-partage. Ce tour d'horizon n'est certainement pas complet. Son but était surtout de mettre en avant les grandes lignes de telles machines.

Nous allons maintenant passer à la présentation des objectifs de nos travaux et du contexte précis dans lequel ils se situent.

1.4. Présentation du problème traité

1.4.1. Cadre général

Somme toute, nous avons discuté dans ce qui précède, de l'évolution des Systèmes de Gestion de Bases de Données, suivant deux axes: l'un relatif à l'évolution des modèles de données et langages associés, et l'autre relatif à l'utilisation des machines parallèles.

Le premier aspect nous a conduit à définir un modèle et un langage idéals. Le second nous a orienté vers les machines bases de données issues de l'approche sans partage.

Dès lors, peu de chemin reste à parcourir pour admettre que le SGBD parfait peut se définir comme un système qui supporte ce modèle et ce langage idéal et qui est implanté sur une machine de type sans partage.

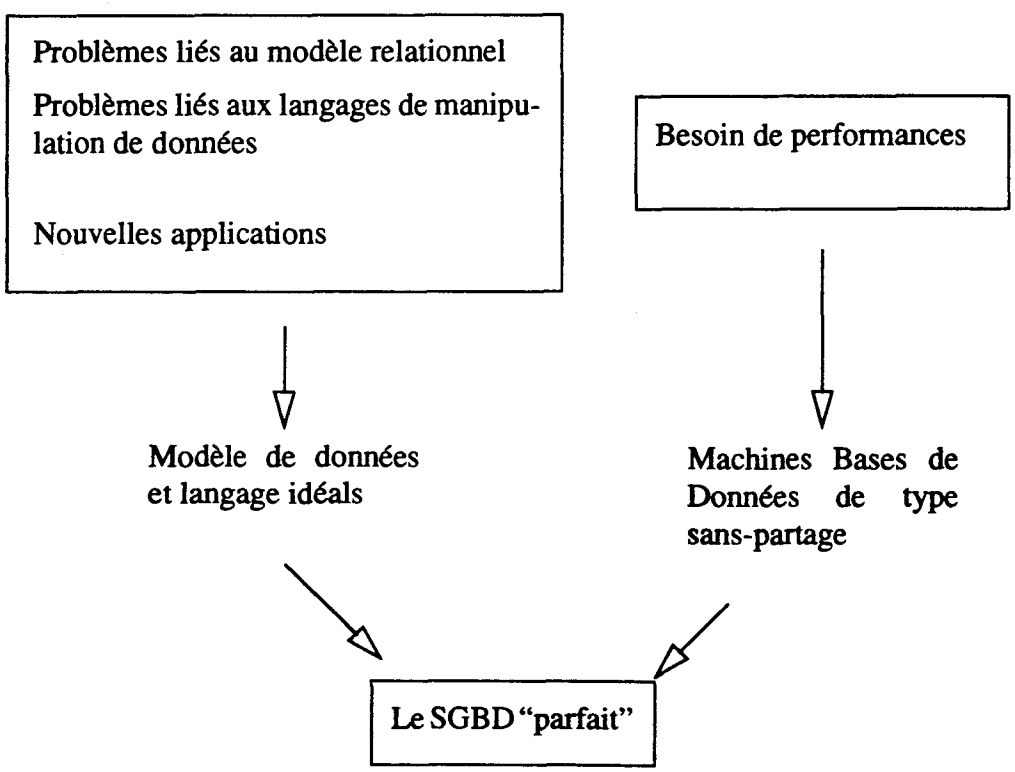


Fig. 1.25 : Caractérisation du SGBD "parfait"

Cependant, qui dit approche sans partage, dit importance primordiale du placement des données sur les noeuds.

Le but de notre travail est donc de proposer un ensemble de solutions pour répondre au problème du placement d'objets complexes (données supportées par le modèle idéal) sur une

machine "sans partage". En d'autres termes, nous allons nous intéresser au **problème de la transformation d'un schéma logique d'objets complexes en un schéma physique réparti.** Cette transformation du schéma logique se limitera au niveau macroscopique des noeuds de la machine. En conséquence, nous ne nous intéresserons pas à l'organisation physique des données sur chaque noeud.

En ce qui concerne le contexte précis de notre étude, nous ne nous fixons pas de machine précise. Notre but est plus de mettre en avant les différents problèmes et de proposer un certain nombre de solutions générales et adaptables. De ce fait, certaines des méthodes que nous préconisons seront incomplètes puisqu'elles nécessiteront la connaissance de certains paramètres qui dépendent de la machine cible.

Par contre, en ce qui concerne le modèle de données et le langage associé, nous utiliserons comme support la solution FAD [DANF87] qui, même si elle est un peu particulière, possède toutes les propriétés intéressantes, tant au niveau modèle de données, qu'au niveau langage de programmation (cf 1.2.3.3). De plus, la composante parallèle, présente dans le langage, le désigne tout particulièrement pour être utilisé dans un contexte parallèle.

Néanmoins, l'utilisation de la solution FAD comme support de notre étude, ne fait pas de cette dernière une étude dédiée et figée: l'ensemble des solutions proposées sera tout à fait utilisable pour d'autres modèles de données qui possèdent les mêmes propriétés.

Nous présentons dans la section suivante, de façon un peu plus précise le langage FAD et son modèle de données. Là encore la présentation se limite à dégager les grandes lignes utiles à la présentation de notre travail ; une description beaucoup plus détaillée peut être trouvé dans [BANC87], [DANF87], [DANF88a].

1.4.2. Présentation de FAD

FAD est un ***langage de programmation fonctionnel***, qui a pour but la ***création et la manipulation d'objets complexes*** à l'intérieur d'un ***système bases de données hautement parallèle***.

Il peut donc être vu sous trois angles différents: celui du langage de description de données (LDD), celui du langage de manipulation de données (LMD) et celui du langage source pour une exécution parallèle. L'aspect qui nous intéresse principalement ici, est le premier, c'est à dire FAD en tant que LDD. Néanmoins, nous aborderons tout de même les deux autres qu'il serait dommage de passer sous silence.

1.4.2.1. FAD en tant que langage de description de données

FAD manipule deux unités de ***données typées*** que sont les ***valeurs*** et les ***objets***. Une valeur peut être ***atomique*** (c'est à dire issue d'un type atomique ou d'un type abstrait de données) ou ***structurée*** (nuplet, ensemble homogène ou disjonction[1] de valeurs).

---

1. Rappelons qu'une disjonction est une forme particulière de nuplet pour laquelle un seul des attributs peut ne pas être nul. Elle servira donc lorsqu'un attribut peut répondre à différents "types" exclusifs entre eux. Nous donnons dans ce qui suit un exemple d'une telle disjonction.

Un objet se compose d'un ***identifiant*** et d'un ***état***. Cet état est soit une valeur atomique, soit une structure (nuplet, ensemble ou disjonction) contenant des valeurs et des objets: c'est donc au travers de cette notion d'état que la construction d'objets complexes est possible.

Une valeur n'est pas modifiable contrairement à l'état d'un objet. L'incorporation de la notion d'identité d'objets dans le modèle, permet donc de décrire les schémas conceptuels sous forme de graphes. Tout objet peut avoir des parents multiples qui le référencent par son identifiant.

Notons que cette distinction de deux types de données n'est pas spécifique à FAD. Certains modèles orientés-objets la préconisent aussi, le vocabulaire utilisé étant différent (O2 [LECL88],ORION [KIM87]).

Nous donnons ci-dessous, sous forme d'une pseudo-grammaire, les règles de "construction" des données FAD:

donnée -> valeur / objet

valeur -> type * valeur_atomique / type * valeur_structurée

valeur atomique -> nulls / bools / ints / float / nchars / strings / nom_d'attribut /
valeur_TAD1 / .../ valeur_TADn

valeur_structurée -> [valeurs] / {valeurs} / |valeurs|

objet -> identifiant * type * état

état -> valeur_atomique / objet_structuré

objet_structuré -> [données] / {données} / |données|

Fig. 1.26 : Règles de construction des données avec FAD

Remarque: Sur cette figure, [-] représente un nuplet, { - } un ensemble et | -| une disjonction.

Puisque FAD est construit au-dessus d'une couche de Types Abstraits de Données (TADs), un type définit un domaine de données, mais aussi des fonctions éventuellement héritées d'autres types, par l'intermédiaire d'une hiérarchie de types, pour manipuler les données de ce domaine.

Néanmoins, dans les nombreuses références dont nous disposons, aucune solution n'est proposée pour définir les fonctions associées à un type abstrait de données. Par contre, pour les types "classiques", cette notion de hiérarchie de type et d'enveloppement des fonctions associées est bien respectée.

Nous donnons dans la figure suivante, cette hiérarchie, ainsi que les différentes méthodes (i.e fonctions) associées.

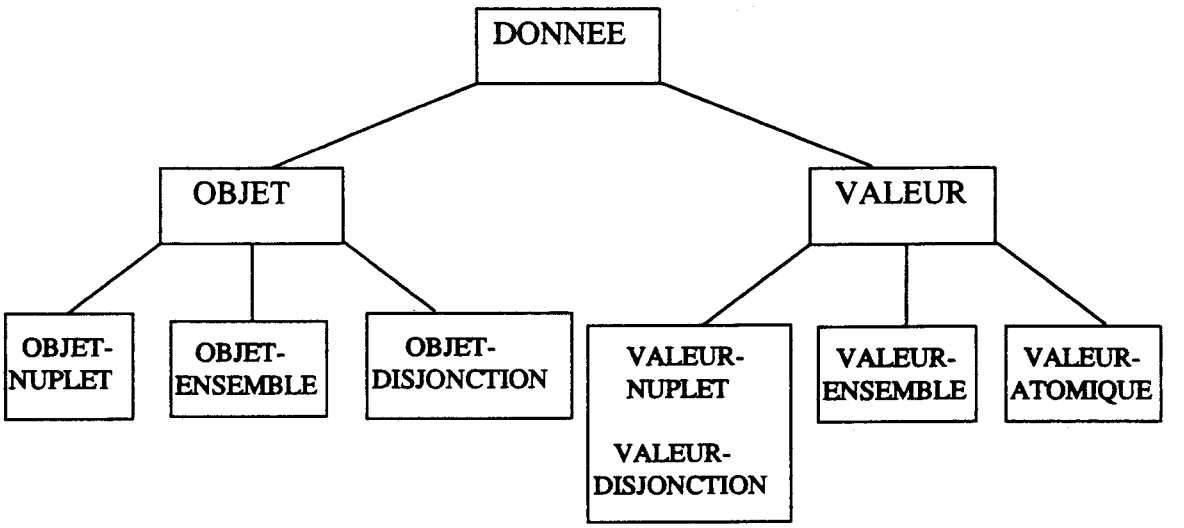


Fig. 1.27 : Hiérarchie de types dans FAD

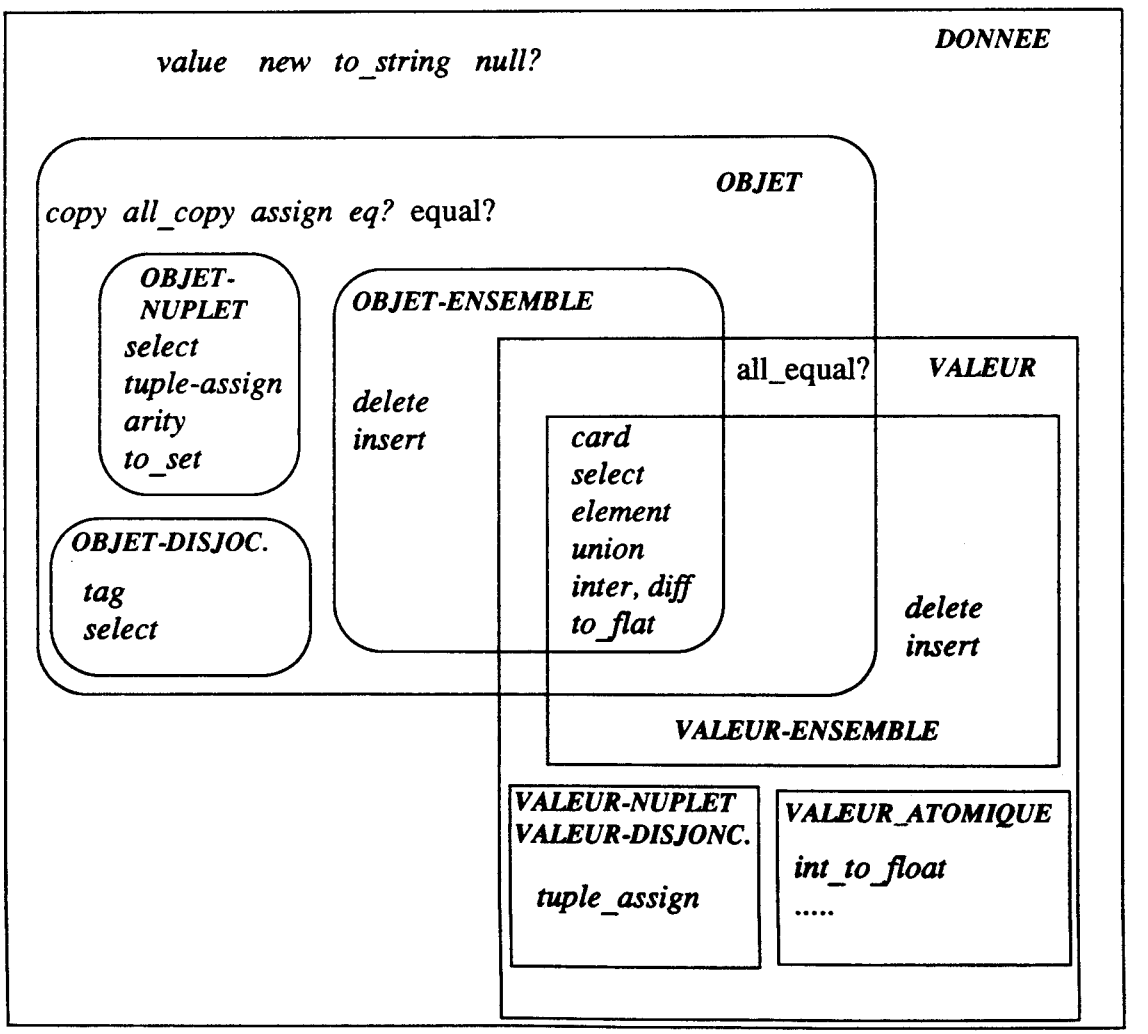


Fig. 1.28 : Fonctions associées aux types de base (illustration de l'héritage)

Une description détaillée des fonctions présentées peut être trouvée dans [DANF 87] ou [NICO 89c].

Dans tout ce qui suit nous regrouperons sous le terme de ***classe*** l'ensemble des données associées à un type. Cette appellation nous est tout à fait personnelle.

Afin d'illustrer nos exemples par une représentation conviviale, nous utiliserons la "syntaxe graphique" suivante:

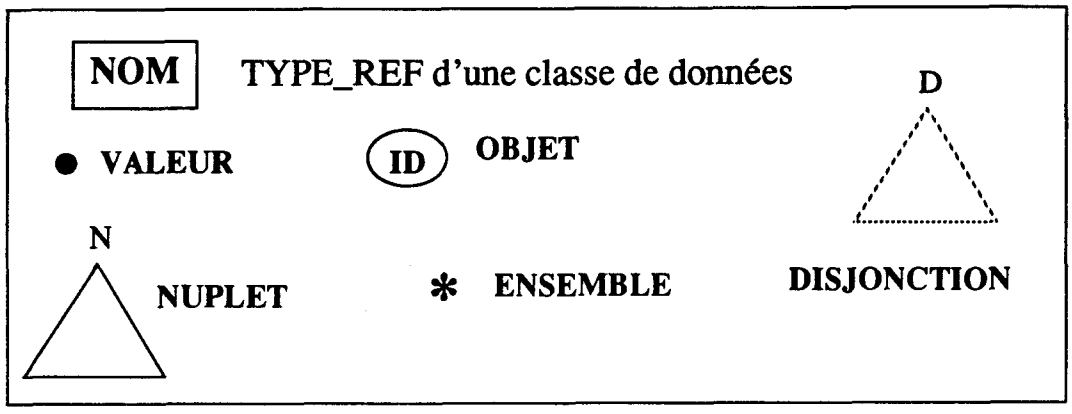


Fig. 1.29 : Eléments graphiques utilisés pour la représentation des données

Nous présentons ci-dessous un exemple de définition de données avec la syntaxe FAD (figure 1.30) et avec notre syntaxe graphique (figure 1.31).

```
EMPLOYE = obj [nom = strings, prénom = strings, salaire = strings,
                    ad = ADRESSE, famille = FAMILLE]
ADRESSE = obj [rue = strings, code_postal = int, ville = strings]
FAMILLE = obj {PERSONNE}
PERSONNE = obj [nom = strings, prénom = strings, nais. = date,
          situation =|sit_scolaire=strings, sit_militaire=strings, sit_travail=strings| ]
```

Fig. 1.30 : Exemple de description de données avec la syntaxe FAD

Nous terminerons cette présentation de FAD, en tant que langage de description de données, par une remarque sur les différents tests d'égalité nécessaires, suite au support de la notion d'identité d'objets.

Ainsi FAD distingue les trois tests d'égalité que sont l'***égalité forte*** (***eq ?***), la ***presque égalité*** (***equal ?***) et l'***égalité en profondeur*** (***all-equal ?***).

Deux objets sont fortement égaux s'ils ont le même identifiant et par conséquent le même état (conséquence de la contrainte de consistance des objets cf 1.4.2.2 [DANF 87]). Il est

important de signaler qu'aucune hypothèse n'est faite à propos de la nature temporaire ou persistante des objets et de la représentation physique (interne) du partage.

Deux objets sont presque égaux si leurs identifiants sont différents mais leurs états fortement égaux. Il s'agit donc d'un test d'égalité forte au premier niveau. La contrainte de consistance ne doit pas être vérifiée pour de tels objets, i.e l'état de chacun d'eux peut ensuite évoluer indépendamment de l'état de l'autre.

Enfin deux objets sont égaux en profondeur lorsque les valeurs atomiques des feuilles des arbres qui les représentent sont égales. Il s'agit donc d'un test récursif d'égalité qui ne s'intéresse qu'aux valeurs atomiques. Nous donnons dans [NICO 89c] un certain nombre d'exemples relatifs à ces différents types d'égalité. Nous terminerons sur ce point en donnant la relation d'implication qui lie les trois niveaux d'égalité, à savoir:

EQ ? => EQUAL ? => ALL-EQUAL ?

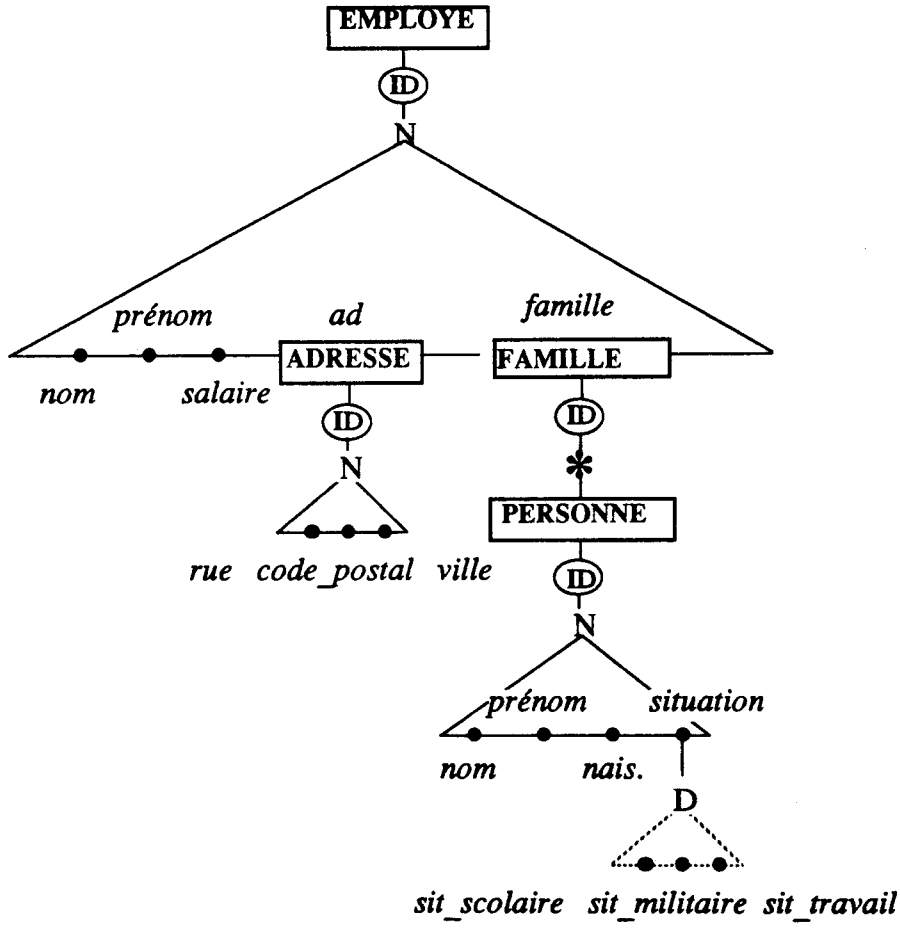


Fig. 1.31 : Représentation de l'exemple sous forme graphique

1.4.2.2. FAD en tant que langage de manipulation de données

FAD est de ***nature fonctionnelle***, ce qui permet d'utiliser la composition fonctionnelle pour enchaîner les requêtes, tout en conservant un formalisme déclaratif.

Le terme ***action*** est utilisé dans FAD pour indiquer un traitement qui retourne des données ou qui change des données existantes. L'application d'une fonction à ses arguments dénote une telle action.

L'***abstraction*** (notion familière aux langages fonctionnels) permet à l'utilisateur de créer des fonctions du premier ordre. Le langage fournit aussi un ensemble figé de fonctions d'ordre supérieur appelées les ***constructeurs d'actions***. Ils servent pour l'écriture des programmes et fournissent les structures de base (while ... do, if.. then .. else, etc). Nous retrouvons donc à ce niveau quelques similitudes avec le langage fonctionnel FP de Backus [BACK78].

Dans la terminologie FAD, un ***object-store*** désigne un ensemble ***consistant*** et ***fini*** d'objets temporaires ou persistants. La contrainte de consistance implique que deux objets fortement - égaux (même identifiant) ont toujours le même état. L'aspect fini de l'ensemble implique qu'un objet, composant de structure, ne peut ni se contenir, ni contenir une structure qui le contient.

Toute action a en entrée un object-store, et retourne en sortie, un code de statut entier, un nouvel object-store et éventuellement des données.

La forme générale d'une abstraction d'action est la suivante:

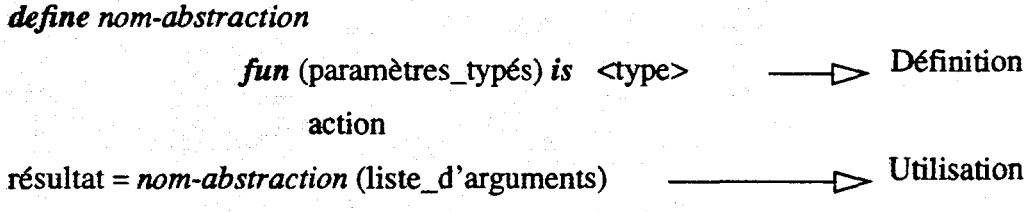


Fig. 1.32 : Définition et utilisation d'une abstraction d'action

```
define augmente_10%
    fun (e <employé> ) is <nulls>
        if  less_than (e.'salaire, 10000)
        then tupleassign (e.'salaire, * (1.1, e.'salaire))
```

Fig. 1.33 : Exemple de définition d'abstraction d'action

Les constructeurs d'actions permettent donc de combiner des actions atomiques afin d'en créer de plus importantes. Outre, les constructeurs classiques qui fournissent les structures de base (BEGIN-END pour l'exécution séquentielle, IF- THEN - ELSE pour le test et WHILE DO pour les boucles), FAD supportent un ensemble de constructeurs spécifiques, adaptés, pour certains, aux traitements ensemblistes requis par un SGBD, et prévus, pour d'autres, pour la spécification explicite de parallélisme.

Nous passons ci-dessous en revue ces constructeurs ; leur présentation détaillée et les exemples qui s'y rapportent peuvent être trouvés dans [NICO89c].

- Le constructeur DO - END

Il permet de spécifier explicitement le parallélisme désiré entre les différentes actions qu'il "encadre".

DO $action_1$, $action_2$,....$action_n$ ***END***

Il ne s'agit là que d'une forme déclarative de parallélisme, puisque à aucun moment l'utilisateur n'aura le contrôle effectif d'une telle exécution parallèle.

- Le constructeur LET

Le but de ce traitement est de pouvoir exprimer à la fois du parallélisme et de la séquentialité, pour la liaison des noms aux résultats d'actions, en permettant en plus, d'intercaler des actions sans effet de bord. Sa forme générale est:

LET {liaisons noms - résultats_d'actions}

{action / liaisons noms - résultats_d'actions}*

IN action

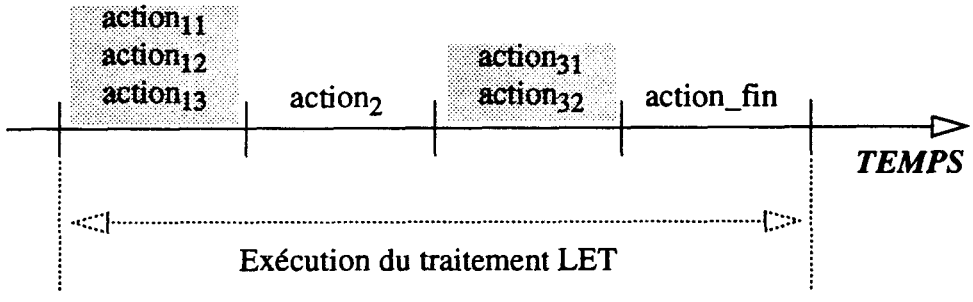
Exemple:

LET a = $action_{11}$, b = $action_{12}$, c = $action_{13}$

$action_2$

d = $action_{31}$, e = $action_{32}$

IN action_fin



<=> Exécution parallèle d'actions

Fig. 1.34 : Représentation dans le temps de l'exécution d'un traitement LET

- Le constructeur FILTER

Le rôle de ce traitement est d'appliquer une abstraction d'action à chaque élément du produit cartésien des ensembles passés en paramètres. C'est grâce à ce traitement que nous pourrons donc, entre autres, exprimer l'équivalent de l'opération de jointure du modèle relationnel.

Sa forme générale est:

FILTER (abstraction_d'action, ensemble$_1$, ensemble$_2$, ..., ensemble$_n$)

Exemple:

DATABASE = obj [personnel = {EMPLOYE}, structure = {SERVICE}]

EMPLOYE = obj [nom=strings, prénom=strings, salaire=ints, num_serv=strings]

SERVICE = obj [num_serv=strings, désignation=strings, bâtiment=strings]

FILTER (***fun*** (e <EMPLOYE>, s<SERVICE>) ***is*** <nulls>

if and? (eq? (e.'num_serv,s.'num_serv), eq?(s.'bâtiment, "M3"))

then *augmente_10%* [1](e),

DATABASE.'personnel, DATABASE.'structure)

<=> Augmentation de 10% des employés du bâtiment M3 qui ne gagnent pas 10000F.

Fig. 1.35 : Exemple de traitement FILTER

- Le constructeur PUMP

La forme générale de ce traitement est:

PUMP (*abstraction_unaire, abstraction_binaire, val, ensemble*).

Son but est d'appliquer l'abstraction unaire à chaque élément de l'ensemble argument, et de réduire la donnée résultat en utilisant l'abstraction binaire qui doit être associative et commutative. En fait c'est la ***stratégie diviser pour conquérir*** qui est ici appliquée. Si l'ensemble argument est vide l'action retourne *val*. S'il ne contient qu'une donnée x, elle retourne *abstraction_unaire* (x). Dans les autres cas elle retourne *abstraction_binaire* (x, y) avec:

ensemble = ensemble$_1$ U ensemble$_2$,

x = ***PUMP*** (*abstraction_unaire, abstraction _binaire, val, ensemble$_1$*),

y = ***PUMP*** (*abstraction_unaire, abstraction_bianire, val, ensemble$_2$*).

Exemple:

PUMP (fun (a <ints>) is <ints> *(a,a),

fun (x,y <ints>) is <ints> +(x,y),

0,

{1, 2, 12, 5, 6, 24})

Fig. 1.36 : Exemple de traitement PUMP (somme des carrés des éléments de l'ens.)

1. ***augmente_10%*** **est l'abstraction définie précédemment (cf figure 1.33).**

Enfin, nous signalerons juste l'existence des constructeurs MATCH et GROUP qui sont vraiment spécifiques (peut être trop) et dont la présentation d'un exemple intéressant nous emmènerait un peu loin.

Pour terminer cette revue de FAD en tant que langage de manipulation de données, nous discuterons de la nature d'un programme FAD. Un tel programme est aussi une abstraction d'action, introduite cette fois par le mot clé ***prog*** plutôt que ***fun***. Les définitions des abstractions d'actions utiles au programme peuvent être regroupées en ***modules*** dans lesquels doivent alors apparaître les définitions et déclarations nécessaires. Ces modules sont créés et maintenus dans un environnement hôte "classique". La commande ***execute*** est utilisée pour demander l'exécution d'une abstraction en tant que programme.

1.4.2.3. FAD en tant que langage source pour une exécution parallèle

Nous pouvons attribuer à FAD, certaines "propriétés parallèles" qu'il est important de noter. En fait, deux tendances parallèles peuvent être dégagées. Tout d'abord, il existe au sein du langage une forme de ***parallélisme implicite***. Ainsi les arguments d'une fonction sont implicitement évalués en parallèle, sauf dans le cas, où l'utilisateur exprime explicitement un certain degré de séquentialité (constructeur BEGIN - END, LET). D'autre part les constructeurs qui correspondent à des manipulations ensemblistes de données (FILTER, PUMP) reposent aussi sur une exécution parallèle implicite.

L'autre tendance parallèle propre au langage, concerne le ***parallélisme explicite*** entre actions (constructeurs DO - END, LET) que l'utilisateur peut énoncer de façon déclarative. Rappelons néanmoins à ce sujet, qu'il s'agit juste d'une expression déclarative du parallélisme et non pas d'un contrôle explicite de ce dernier.

Qu'il s'agisse du parallélisme explicite ou implicite, l'exécution parallèle effective des programmes dépend bien sûr de la machine sous-jacente, mais aussi des solutions retenues pour le placement de données. Nous aborderons dans le chapitre 4, les différentes phases de compilation du langage FAD, pour analyser la répercussion de nos solutions en matière de placement de données.

Ainsi se termine la présentation de FAD. Nous allons maintenant revenir un peu sur le principe de nos travaux avant d'entrer dans le "vif du sujet".

1.4.3. Principe de notre étude

Nos travaux concernent donc la transformation d'un schéma logique d'objets complexes, que nous supposerons (pour des raisons de présentation) décrit avec FAD, en un schéma physique réparti. Ils sont donc relatifs au problème crucial du placement de tels objets complexes sur une machine de type sans-partage et peuvent être symbolisés par la figure de la page suivante.

A l'heure actuelle, la plupart des projets ou réalisations qui utilisent ce type de machine, sont basés sur le modèle relationnel. Les techniques relatives au placement de données y sont fortement dépendantes de l'administrateur du système. De plus, ce placement y est sou-

vent rigide, et n'est en aucun cas dépendant de l'utilisation effective des données.

Face à cela, nous proposons dans ce qui suit un ensemble de techniques, qui d'une part, sont relatives à un modèle de données "plus riche", et d'autre part, permettent d'aboutir à un placement quasi-automatique guidé par l'utilisation des données.

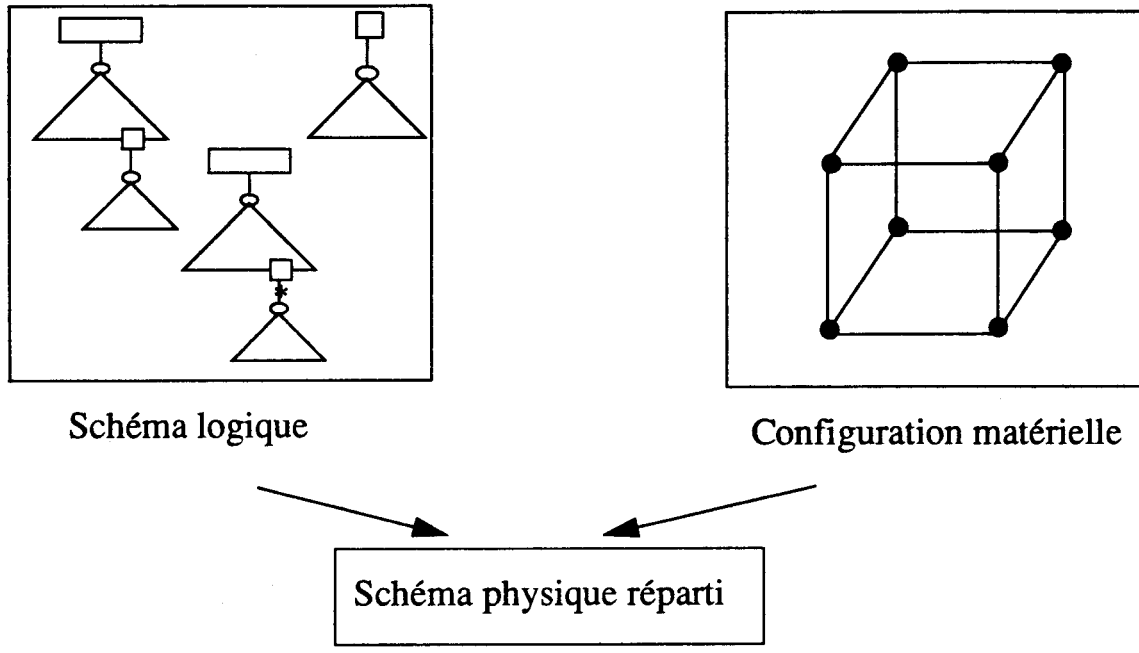


Fig. 1.37 : Principe général de nos travaux

Notre approche repose sur l'hypothèse de base qu'est la ***règle des 80/20*** [KNUT73]. Cette règle "affirme" que dans les applications bases de données, 80% des requêtes ne concernent en réalité que 20% des données. En conséquence, il semble nécessaire de prendre en compte l'utilisation effective des données pour en déterminer le placement.

Ce souci du placement des données en fonction de leur utilisation, n'est en effet pas nouveau et a suscité des travaux aussi bien dans le domaine des SGBD relationnels classiques (i.e centralisés) que dans celui des bases de données réparties[1]. Cependant en ce qui nous concerne, l'enjeu est important puisque nous nous plaçons par rapport à une configuration matérielle parallèle sur laquelle, les mouvements de données ont intérêt à être restreints.

Etant donné, que nos travaux conduisent à un placement des données dirigé par leur utilisation, cela laisse entrevoir un type de machine où les données pourraient être réorganisées afin d'obtenir un meilleur fonctionnement général.

La méthode[2] de placement que nous préconisons peut être décomposée en deux grandes étapes que sont le découpage vertical du schéma logique, étape que nous appellerons ***fragmentation verticale***, et la répartition des occurrences des fragments verticaux obtenus, seconde étape que nous qualifierons de ***fragmentation horizontale***.

---

1. Nous reviendrons d'ailleurs partiellement sur ces travaux dans le chapitre 2.
2. L'ensemble des solutions que nous préconsons peut être regroupé sous l'emblème: ***méthode de transformation d'un schéma logique en un schéma physique réparti***.

Les termes fragmentation verticale et horizontale ne nous sont pas personnels, ils sont relatifs à des notions connues que nous décrirons dans les chapitre 2 et 3 et pour lesquels nous apportons une interprétation dans notre contexte.

Nous consacrerons dans ce qui suit un chapitre par grande étape. Nous terminerons alors par un chapitre de conclusion dans lequel, d'une part, nous donnerons un exemple d'intégration de nos solutions et, d'autre part, nous discuterons des perspectives attachées aux travaux présentés.

Juste quelques mots pour terminer ce chapitre d'introduction. Dans ce qui suit nous adoptons une présentation un peu inhabituelle. Etant donné que nous abordons des points assez différents, qui sont même parfois relatifs à des domaines totalement disjoints, nous avons préféré situer "bibliographiquement"chacun de ces points au moment de son "entrée en jeu" dans notre méthode.

Nous aurions pu regrouper tout cet aspect bibliographique dans le chapitre d'introduction qui serait alors devenu très disparate du point de vue de son contenu. De plus cette solution nous aurait obligé à "maintenir un va et vient" pénible sous forme de références internes au document. Les chapitres que nous proposons forment donc un tout, ce qui explique certainement leur taille importante.

2. LA FRAGMENTATION VERTICALE

2.1. Présentation générale du problème

2.1.1. Définitions

D'une façon générale, la fragmentation verticale d'un ensemble homogène[1] de données, consiste à opérer diverses projections de cet ensemble par rapport aux propriétés (attributs) des données. Les ensembles ou encore fragments verticaux ainsi obtenus par projection, ont même cardinalité que l'ensemble de départ mais sont constitués de données partielles puisque uniquement relatives aux attributs de projection. Il est bien sûr nécessaire que ces ensembles permettent de reconstruire les données de départ dans leur intégralité.

Nous donnons ci-dessous un exemple de fragmentation verticale dans le contexte du modèle relationnel où, dans ce cas, les projections s'expriment bien sûr par des opérations de projection de l'algèbre relationnelle.

EMPLOYE

NUM	NOM	AGE	SERVICE	SALAIRE
01	DURAND	32	ATELIER1	7500
02	MARTIN	45	ATELIER2	8200
03	DUPONT	30	ATELIER3	6300
04	AIME	43	ATELIER1	5980
..	..	..	..	

NUM	NOM	AGE
01	DURAND	32
02	MARTIN	45
03	DUPONT	30
04	AIME	43
..	..	..

EMPLOYE.f1 = $\pi_{NUM,NOM,AGE}(EMPLOYE)$

NUM	SERVICE	SALAIRE
01	ATELIER1	7500
02	ATELIER2	8200
03	ATELIER3	6300
04	ATELIER1	5980
..	..	.. ..

EMPLOYE.f2 = $\pi_{NUM,SERVICE,SALAIRE}(EMPLOYE)$

Fig. 2.1 : Exemple de fragmentation verticale dans le contexte relationnel

1. Ensemble constitué de données de même nature, ie qui ont les mêmes propriétés.

L'idée sous-jacente à la fragmentation verticale est de pouvoir constituer des ensembles de données partielles dont les attributs ont des propriétés géographiques communes. Le mot géographique peut, comme nous allons le voir, être perçu à une échelle différente suivant que l'on se place dans un contexte centralisé, réparti ou parallèle. Néanmoins, la conséquence d'une opération de fragmentation verticale est toujours la même, à savoir l'assignation de chacun des fragments verticaux à une entité physique différente. Cette opération s'inscrit donc dans la phase de conception physique d'une base de données.

Hormis l'approche classique qui consiste à regrouper les attributs en fonction de leur utilisation, c'est-à-dire à exploiter les propriétés géographiques communes, on peut mettre en avant une autre forme de fragmentation verticale: celle qui fournit autant de fragments que les données ont d'attributs. Nous reviendrons dans ce qui suit sur ce modèle de stockage totalement décomposé [COPE85] et présenterons certains travaux qui s'y rapportent. Pour l'heure nous allons préciser le concept de fragmentation verticale par rapport aux différents contextes dans lesquels elle peut être utilisée.

2.1.1.1. Contexte relationnel centralisé

Comme nous l'avons vu sur notre exemple, la fragmentation verticale se caractérise pour le modèle relationnel par une "fonction qui partitionne une relation en ensembles de sous-nuplets, chacun étant défini par une opération de projection appliquée à la relation" [GARD90]. Il est important de signaler qu'une telle opération peut violer la règle de non duplication chère au modèle, règle que les phases de normalisation garantissent. Dans notre exemple, l'attribut NUM est en effet dupliqué dans chacun des fragments. Rappelons néanmoins que la fragmentation verticale n'est en rien une étape de la phase de conception logique des données, phase pour laquelle la normalisation offre les gardes-fous nécessaires et suffisants pour maintenir l'intégrité des données au niveau utilisateur.

En tout état de cause, la fragmentation verticale doit être transparente pour cet utilisateur. Dès lors, l'utilisation de fragments verticaux qui se recouvrent ou pas (ie qui ont ou pas des attributs communs) est uniquement liée à l'existence des mécanismes internes adéquats pour maintenir l'intégrité des données au niveau interne.

Dans un contexte centralisé, la fragmentation verticale favorise naturellement le traitement des requêtes de projection qui portent sur les attributs utilisés dans la définition des fragments, en limitant le nombre de fragments à accéder.

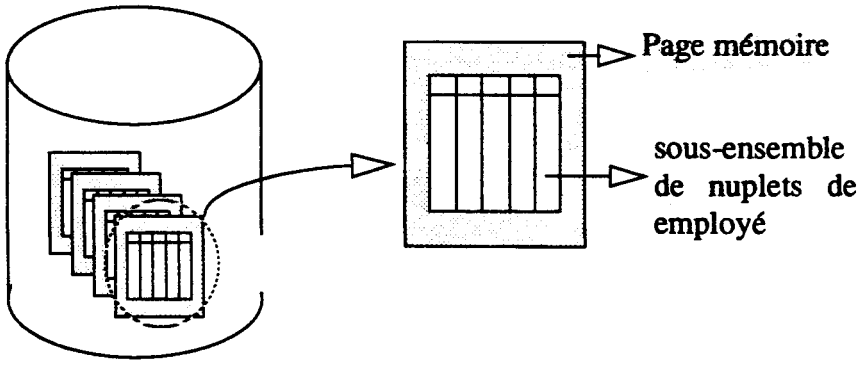


Fig. 2.2 : Stockage usuel d'une relation

En effet, dans la plupart des systèmes de gestion de bases de données relationnels le stockage d'un nuplet se fait d'une façon globale, c'est-à-dire que toutes les valeurs d'attributs de ce nuplet sont sur une même page mémoire (cf figure 2.2). En conséquence, tout accès à l'une de ces valeurs entraîne l'accès systématique à toutes les autres, même si elles sont inutiles à la requête. Il en découle donc, pour chaque requête, une certaine quantité d'informations inutilement accédées.

Par contre, l'utilisation d'une fragmentation verticale qui regroupe les attributs fréquemment utilisés ensemble, se traduit par une diminution de ces accès inutiles qui est, bien sûr, synonyme ici d'une diminution du nombre de pages physiques "visitées" par la requête. Cette diminution des accès inutiles constitue le principal objectif de la fragmentation verticale dans un contexte centralisé.

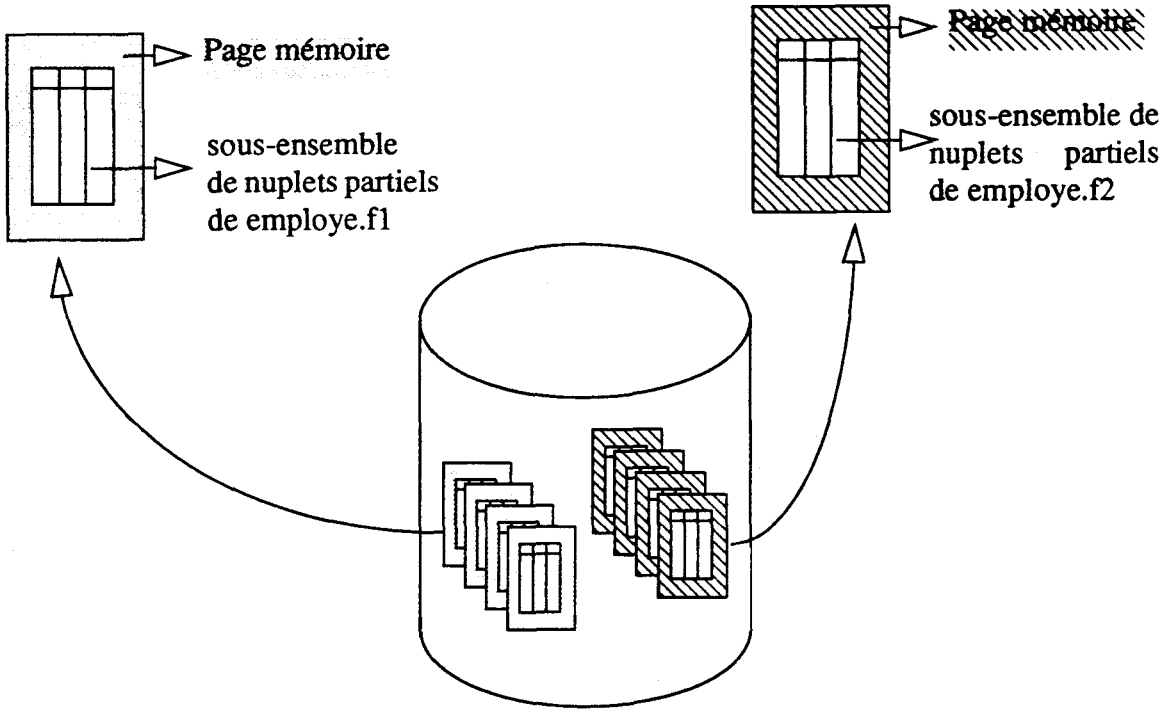


Fig. 2.3 : Fragmentation verticale dans un contexte centralisé

Un des problèmes attachés à l'utilisation de la fragmentation verticale dans le contexte relationnel est celui de la reconstruction des nuplets à partir des sous-nuplets. Cette opération nécessite en effet le recours à une information de liaison entre les différents sous-nuplets d'un même nuplet. En fait, cette information de liaison constitue l'information minimale nécessaire à la jointure interne des fragments concernés, pour reconstruire tout ou partie de la relation initiale. Pour solutionner ce problème, deux techniques sont envisageables.

La première consiste à répéter dans chaque sous-nuplet la clé du nuplet, ce que nous avons fait dans notre exemple en répétant l'attribut NUM dans chacun des fragments. Cela conduit à l'obtention de fragments qui se recouvrent et ce, non pas pour une raison d'utilisation simultanée des attributs, mais uniquement pour pouvoir garantir la reconstruction des nuplets. Avec cette technique, des mécanismes pour le maintien de l'intégrité des données au niveau physique sont à mettre en oeuvre. Nous nous trouvons ici confrontés à un problème induit par le fait que le modèle relationnel ne distingue pas les notions d'identification et de valeur (contenu) (cf 1.2.1 Problèmes liés au modèle de données relationnel).

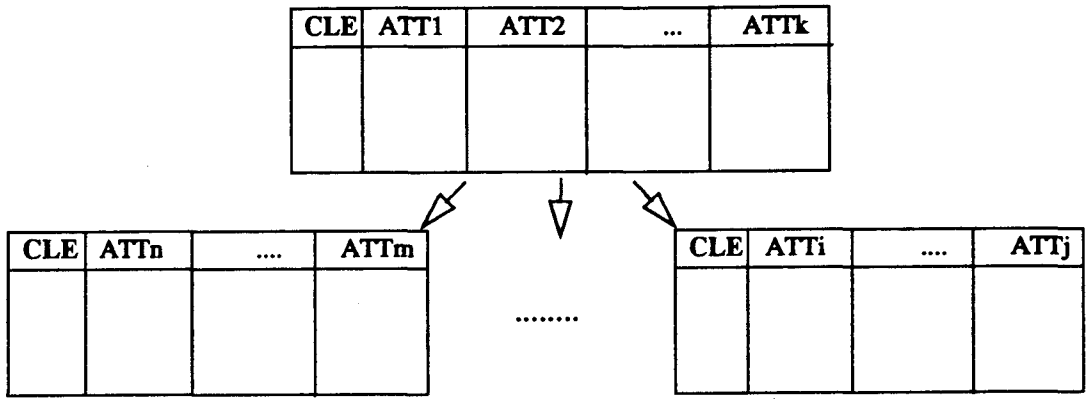


Fig. 2.4 : Fragmentation avec répétition de la clé du nuplet

La seconde solution, pour faire face à ce problème de reconstruction des données, consiste à attribuer à chaque nuplet un identifiant interne qui est alors répété dans chacun des fragments pour le sous-nuplet concerné. Elle n'a d'autre but que de distinguer clairement identification et contenu.

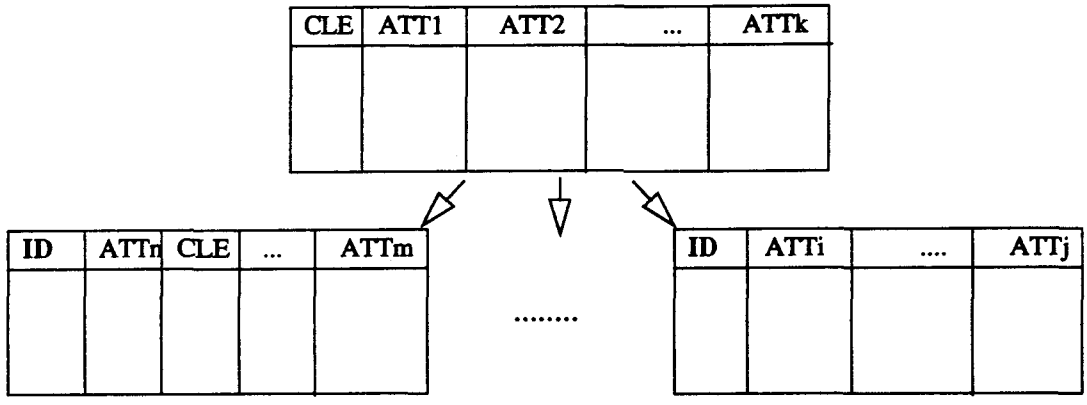


Fig. 2.5 : Fragmentation verticale avec identifiant interne

2.1.1.2. Contexte relationnel réparti

Le principe général reste bien sûr le même. Seule, l'interprétation de la notion de propriétés géographiques communes entre attributs, change. Il ne s'agit plus ici de limiter un nombre de pages à accéder, mais plutôt de découper verticalement une relation par rapport à un ensemble de sites. Il faut donc pour cela qu'il existe une certaine adéquation entre les sites et l'utilisation des données qui y est faite.

En fait, dans un tel contexte, deux phases se distinguent lorsque l'on parle de fragmentation[1]: d'une part la fragmentation en elle même, attachée à des critères logiques tels que

1. Cette remarque d'ordre général s'applique aussi à la fragmentation horizontale, concept qui fait l'objet du chapitre 3.

l'utilisation des données et, d'autre part, la phase d' allocation des fragments qui est relative à des critères physiques.

Ces deux phases ne sont pas totalement indépendantes. Les problèmes rencontrés pour l'allocation de fragments ne sont pas simples puisqu'il s'agit d'aboutir à une localité satisfaisante de l'information tout en évitant les duplications pénalisantes et qui, avouons-le constituent la bête noire des bases de données réparties.

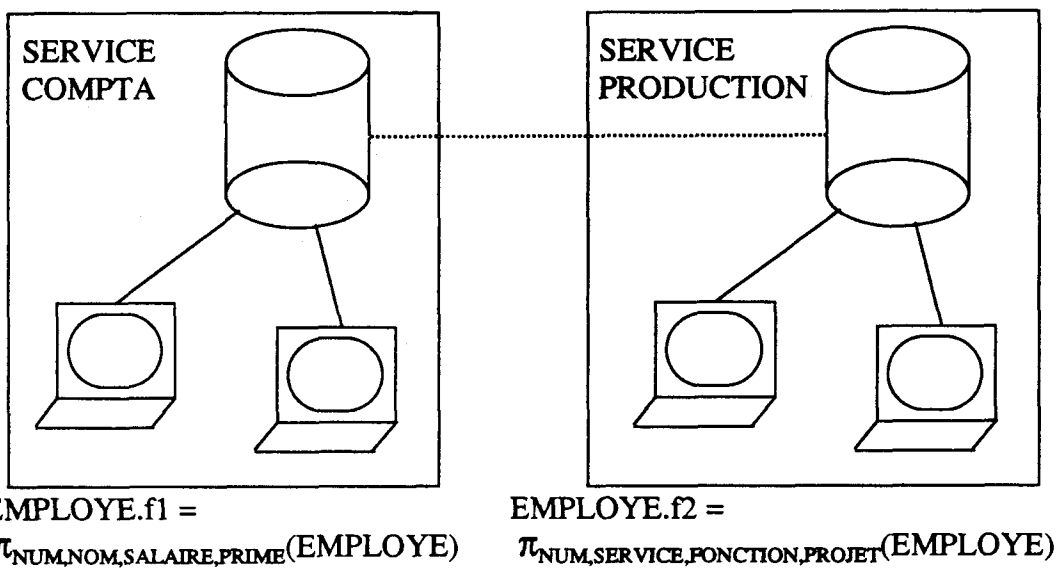


Fig. 2.6 : Exemple de fragmentation verticale dans un contexte réparti

L'objectif principal de la fragmentation est donc ici l'obtention d'une certaine localité pour les traitements, ie d'une certaine disponibilité locale de l'information, sous réserve de garantir la fiabilité de celle-ci.

2.1.1.3. Contexte parallèle

Plaçons nous maintenant dans notre contexte, celui d'une machine bases de données à mémoire distribuée, et analysons intuitivement ce que peut nous apporter la fragmentation verticale.

Comme pour le contexte réparti, nous disposons d'un ensemble d'unités de traitement et de stockage. Ces noeuds sont cependant à distinguer des sites d'une base de données réparties. Ils appartiennent en effet à une même "machine" ce qui leur confère des propriétés de proximité physique d'un tout autre niveau. De plus, il n'est pas question ici de considérer chaque noeud comme un système base de données à part entière, c'est-à-dire de voir la machine comme un ensemble coopérant de bases de données (définition que nous pouvons attribuer plutôt aux bases de données réparties). De même, la notion de site utilisateur n'a plus de sens.

Une fois de plus, un des objectifs visé par la fragmentation verticale est la recherche d'une certaine localité entre les données. Située entre la localité fine (au niveau des pages mémoire) du contexte centralisé et la localité "globale" (au niveau des sites) du contexte réparti,

cette localité est ici considérée par rapport aux noeuds de la machine.

En fait, nous en revenons toujours, au niveau près, au même problème. Etant donné un ensemble de données, certaines règles d'utilisation de ces données, et un environnement physique, il s'agit toujours, grâce à la fragmentation verticale, d'aboutir à un placement qui tire profit au mieux de l'environnement physique.

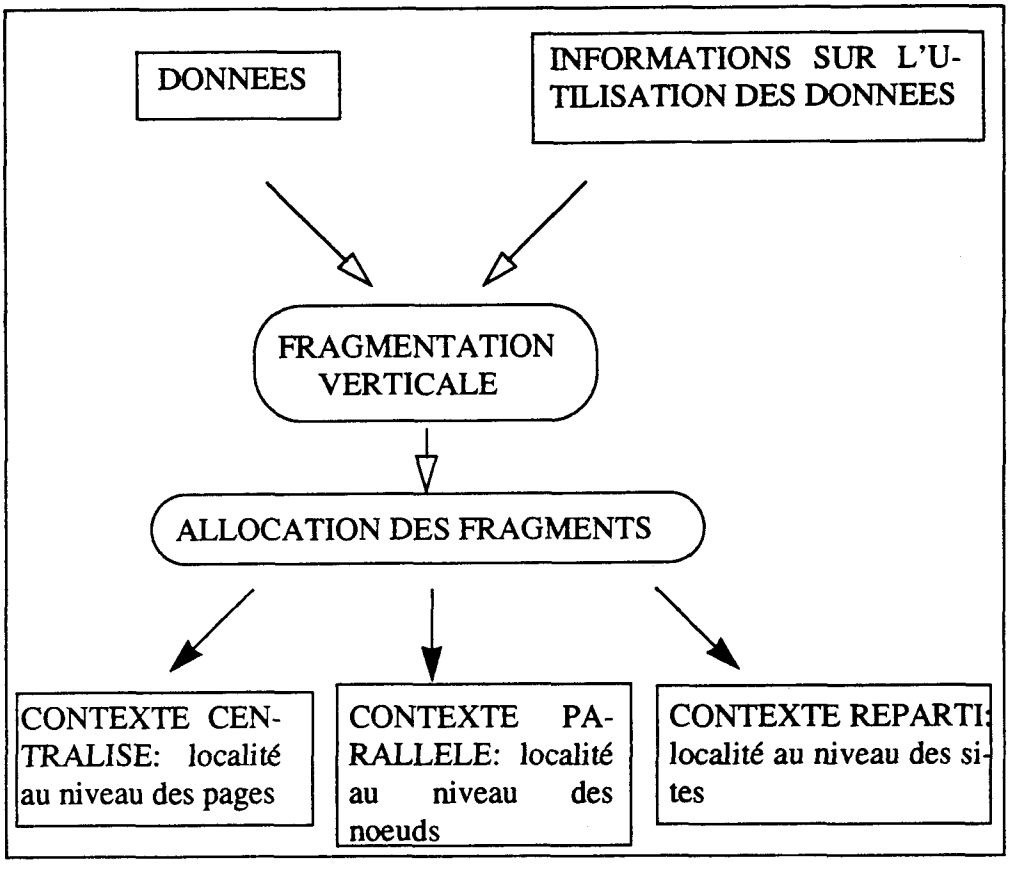


Fig. 2.7 : La fragmentation verticale dans les différents contextes

Il est important de signaler que plus l'échelle augmente, plus les problèmes se compliquent. Dans le contexte centralisé, il s'agit juste d'obtenir différents fichiers physiques pour lesquels il n'y a aucun problème d'allocation puisque l'on ne dispose que d'une seule unité de stockage. Avec le contexte parallèle commence à se poser le problème de l'allocation des fragments, mais chaque noeud est néanmoins perçu de la même façon devant ce problème. Enfin, pour le contexte réparti, la contrainte forte d'adéquation entre fragments et sites vient encore compliquer les choses.

Autre point important face à cette échelle de grandeur: celui de la reconstruction parfois nécessaire de tout ou partie de l'information initiale. Là encore, plus on s'éloigne du modèle centralisé, plus les problèmes soulevés sont importants et plus ils vont à l'encontre de l'utilisation de la fragmentation verticale.

Outre la recherche d'une certaine localité entre les données, nous pouvons attribuer à la fragmentation verticale dans le contexte parallèle l'autre objectif qu'est celui de l'augmentation du parallélisme inter-requêtes.

En effet, puisque le résultat de la fragmentation verticale consiste à assigner chacun des fragments à une entité physique différente, l'utilisation parallèle de ces fragments devient tout à fait souhaitable dans un contexte parallèle.

En fait, si la fragmentation verticale d'un ensemble de données se fait par rapport à l'utilisation qui en est faite (ie regroupement des attributs fréquemment utilisés ensemble), cela implique que les fragments obtenus sont relativement indépendants vis-à-vis de cette utilisation. En conséquence, leur assignation à des entités physiques différentes permet d'envisager leur "traitement" en parallèle et donc une augmentation potentielle du taux de parallélisme inter-requêtes par un fonctionnement MIMD. Nous pouvons résumer cet état de fait par la figure suivante:

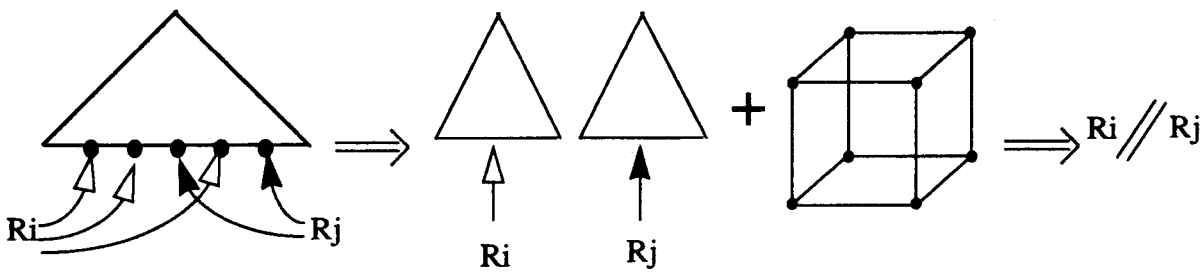


Fig. 2.8 : Fragmentation verticale et parallélisme inter-requêtes

Avec la terminologie utilisée dans le chapitre 1 (cf 1.3.1 Parallélisme et bases de données) nous pouvons donc conclure que, dans un contexte bases de données parallèles, une fragmentation verticale de données permet d'augmenter le parallélisme de contrôle potentiel d'une application. En effet, si nous considèrons que le parallélisme de contrôle représente l'ensemble des opérations différentes simultanément exécutables, il va de soi qu'un découpage des données en fonction de ces opérations va dans le sens d'une augmentation de parallélisme inter-requêtes. Or, ce parallélisme inter-requêtes est une des deux formes que prend le parallélisme de contrôle pour les applications bases de données.

Enfin, il est important de signaler que l'avantage précédemment cité pour le contexte centralisé et qui était celui de la limitation des accès inutiles se retrouve ici. Si l'on reprend l'exemple de la figure précédente, les requêtes Ri et Rj peuvent s'exécuter parallèlement et de plus, elles n'accèdent que les données qui leur sont utiles.

Avant d'en terminer avec cette approche intuitive de la fragmentation verticale dans le contexte parallèle, nous aimerions préciser ce que nous entendons par allocation des fragments verticaux dans ce contexte. Dans ce qui précède, nous avons toujours parlé de l'assignation de chaque fragment vertical à une entité physique différente. Différentes interprétations peuvent être données à l'expression "entité physique" suivant que l'on s'autorise ou pas à éclater sur l'ensemble des noeuds les données partielles d'un fragment vertical. Cela n'altère en rien l'augmentation du parallélisme inter-requêtes induit par la fragmentation verticale. L'éclatement ou pas des fragments verticaux relève de l'utilisation ou non de techniques de ***fragmentation horizontale*** qui, dans un contexte parallèle, visent l'augmentation du parallélisme de données, c'est-à-dire l'exécution de certaines opérations en mode SIMD. Une présentation détaillée de cet autre concept qu'est la fragmentation horizontale peut être trouvée dans l'introduction du chapitre 3.

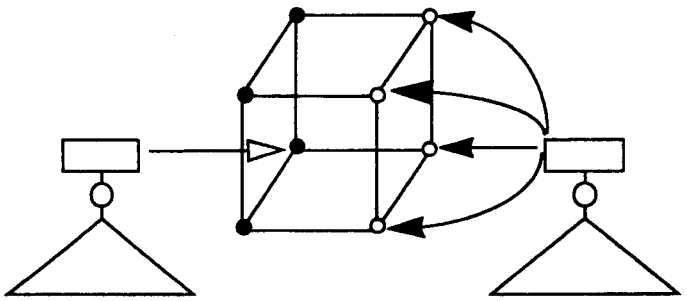


Fig. 2.9 : Assignation des fragments verticaux avec ou sans "éclatement"

2.1.2. Travaux sur la fragmentation verticale

Nous allons, dans cette section, faire un tour d'horizon des différents travaux qui concernent la fragmentation verticale. Ces travaux plus ou moins récents ont comme cadre le modèle relationnel. Ils n'en sont pas moins importants dans notre contexte puisqu'ils nous permettront de situer notre approche de la fragmentation verticale d'objets complexes dans un environnement parallèle. Nous distinguerons deux types de travaux: d'une part, ceux qui se réfèrent à l'utilisation des données et, d'autre part, ceux qui s'orientent vers un modèle de stockage systématique.

2.1.2.1. Travaux basés sur l'utilisation des données[1]

Nous considérons ici que le principe de base de la fragmentation verticale est d'éclater les relations sur différents objets physiques en regroupant les attributs fréquemment utilisés ensemble. Pour effectuer une telle fragmentation, les problèmes rencontrés sont de deux ordres. Tout d'abord, nous nous trouvons confrontés à un phénomène d'explosion combinatoire. En effet, pour une relation de n attributs, il existe environ n * n/2 fragmentations possibles. Ensuite, les coûts de traitement de certaines requêtes utilisant des attributs qui appartiennent à des fragments différents, peuvent se révéler supérieurs à ce qu'ils seraient dans une configuration non partitionnée. En conséquence, la fragmentation doit tenir compte de ces requêtes.

Parmi les travaux de ce contexte, nous pouvons distinguer ceux qui consistent en la recherche d'une solution optimale par la minimisation des coûts induits et ceux basés sur l'utilisation d'heuristiques.

. Recherche d'une solution optimale

Les travaux de Eisner et Severance [EISN76] visent à répartir les attributs en deux sous-fichiers suivant leur fréquence d'utilisation et leur longueur. La formulation du problème

1. Cette synthèse est en partie tirée de [BOUZ87] et [CERI85].

est un graphe sur lequel est appliqué l'algorithme de partitionnement dont le coût s'avère être exponentiel. En outre, les segments sont définis sans tenir compte de la taille des blocs (unité physique de transfert).

March et Severance [MARC77] ont généralisé et amélioré cette approche en prenant notamment en compte cette dernière contrainte. En effet, leur algorithme sélectionne en fonction de la taille des blocs, la fragmentation qui optimise le coût de transfert.

Face à ces travaux, l'étude faite dans [NAVA84] montre qu'aucun des travaux de cette première catégorie n'apporte une solution optimale réalisable. En effet, cette approche présente deux inconvénients principaux. D'une part, pour que l'expression de coût soit aisément manipulable, il est nécessaire d'introduire des hypothèses simplificatrices mais pénalisantes sur la structure des transactions, la complexité des expressions d'accès et l'ensemble des partitions envisageables. D'autre part, même dans ce contexte restreint, les équations de coûts obtenues nécessitent des hypothèses de linéarité qui ne reflètent pas réellement le comportement de la base de données.

Le problème étant avant tout de nature combinatoire, seule une approche qui s'appuie sur des heuristiques permet de réduire le nombre de solutions possibles.

. Utilisation d'heuristiques

- Travaux de Hoffer et Severance

Les premiers travaux avec une approche heuristique sont ceux de Hoffer et Severance [HOFF75]. Ils utilisent l'algorithme BEA (Bond Energy Algorithm) développé par Mac Cormick [MCOR72]. Le BEA est une méthode d'analyse développée pour identifier les regroupements qui se produisent dans des tableaux à deux dimensions. Il permute les lignes et les colonnes de telle sorte que les plus grands éléments soient regroupés pour une identification plus facile. Hoffer et Severance l'appliquent à une matrice composée de coefficients d'affinité entre attributs. Un ensemble de paramètres attachés aux principales requêtes soumises au système permet de calculer ces coefficients d'affinité. Le BEA permet donc de regrouper les attributs qui sont utilisés simultanément en fournissant une matrice sous forme semi-bloc diagonale.

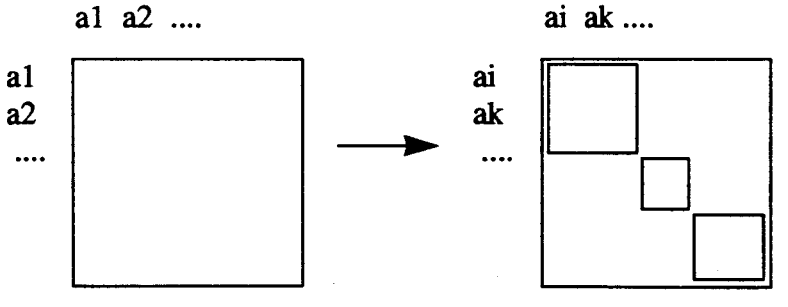


Fig. 2.10 : Principe général de l'utilisation du BEA pour la fragmentation verticale

Puisque les premières étapes de notre méthode reposent sur ce principe général, nous le détaillerons plus précisément au moment de la présentation de ces étapes (cf 2.2 Phase1 de la fragmentation verticale). Ce qu'il faut retenir à ce niveau, c'est que la méthode de Hoffer

et Severance ne garantit pas, de par sa nature heuristique, une solution optimale. De plus, Hammer et Niamir [HAMM79] ont soulevé deux problèmes non résolus dans cette approche. Premièrement, le BEA détermine un ordre parmi les attributs mais c'est encore l'administrateur qui décide comment regrouper physiquement les attributs. Deuxièmement, la prise en compte de l'affinité uniquement entre paires d'attributs peut se révéler insuffisante et devrait être appliquée à de plus grands groupes d'attributs.

- Travaux de Hammer et Niamir

Ils ont donc proposé une alternative basée aussi sur des heuristiques mais qui utilise en plus un modèle de simulation pour l'évaluation des performances présentées par le schéma après fragmentation verticale. Les deux principales composantes de cette méthode sont, d'une part, un générateur de partitions et, d'autre part, un évaluateur de performances qui attribue une valeur de mérite à chaque partition.

L'évaluation des performances est faite par un estimateur de coûts de transactions. Celui-ci utilise le modèle relationnel pour simuler le nombre de pages de mémoire secondaire transférées pendant l'exécution des transactions. Afin de simplifier la définition de la charge de travail, les auteurs regroupent ces transactions par classe et la simulation s'effectue au niveau de chaque classe. Ils introduisent, en outre, une restriction sur la nature des transactions puisqu'elles ne comportent pas de jointure. En effet, les jointures n'ont pas d'influence sur le choix des partitions puisqu'au cours de cette étape, la possibilité de stocker ensemble les attributs de différentes relations n'est pas envisagée.

Avec le modèle utilisé, les sous-relations obtenues après fragmentation sont stockées dans des fichiers relatifs, chaque nuplet étant repéré par un identifiant (TID) correspondant à sa position dans le fichier. Les autres structures d'accès introduites sont les index pour les attributs qui apparaissent dans les prédicats de sélection et les liens qui sont des pointeurs logiques qui connectent entre eux les sous-nuplets d'un même nuplet.

Le principe général de l'évaluateur est alors d'utiliser les index pour créer une liste des TIDs, puis de chaîner les autres sous-relations contenant des attributs non indexés. Deux heuristiques sont utilisées pour éviter d'analyser tous les chaînages possibles: les plus petites sous-relations sont d'abord liées entre elles et parmi les sous-relations restantes, celles contenant les attributs les plus sélectifs sont liées en priorité.

Le générateur de fragmentations fonctionne aussi selon un principe heuristique. Cette heuristique comprend deux étapes itératives qui sont appliquées alternativement jusqu'à ce qu'aucune amélioration ne puisse être obtenue. Puisque chaque étape s'arrête sur une solution, on est assuré de la convergence de l'ensemble du processus de fragmentation. La fragmentation trouvée n'est pas forcément optimale, cependant c'est une solution localement optimale.

La structure générale des deux étapes est identique: à chaque itération, le générateur construit un ensemble de variations autour de la fragmentation courante. Les variations sont ensuite soumises à l'évaluateur et la fragmentation qui représente le plus faible coût est sélectionnée comme fragmentation courante de l'itération suivante.

Les différences entre les deux étapes concernent le choix de la fragmentation initiale (blocs d'un seul attribut pour la première étape et résultat de la première étape pour la seconde) et la construction des variations autour de la fragmentation courante. Pour la première étape, les variations sont générées par exécution de toutes les fusions (deux à deux) possibles

des blocs de la fragmentation courante, alors que dans la seconde, le générateur construit ces variations par mouvement d'un seul attribut à la fois de son bloc vers tous les autres blocs.

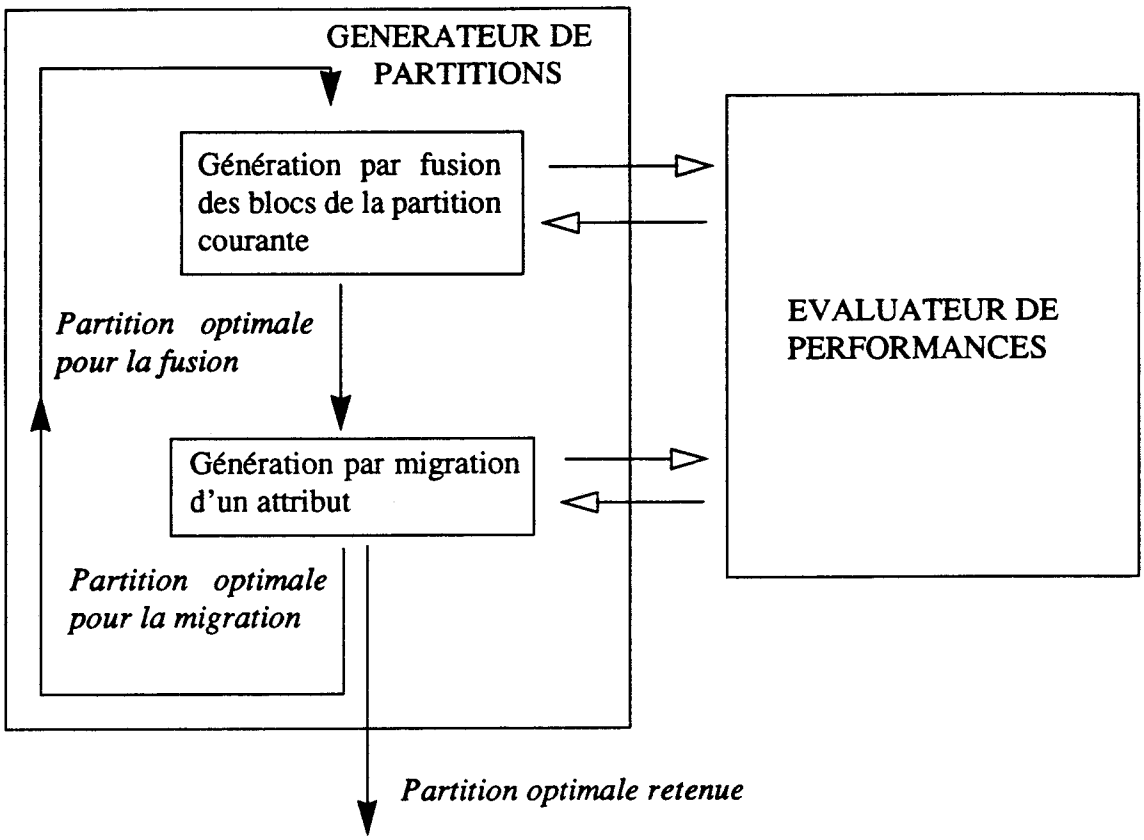


Fig. 2.11 : Principe général de la méthode de Hammer et Niamir

Pour diminuer le nombre de partitions à évaluer au cours de la première étape (qui est en $O(n^3)$), les auteurs utilisent une notion d'attirance entre blocs d'attributs. Cette attirance mesure l'amélioration de performance présentée par la fusion de deux blocs. Le nombre d'évaluations est ainsi ramené à un ordre de $O(n^2)$.

Cette méthode montre l'intérêt d'une approche heuristique face à la complexité et au phénomène d'explosion combinatoire rencontrés au cours de l'étape de fragmentation. Néanmoins, elle soulève un problème au niveau de l'enchaînement des étapes de conception physique. En effet, elle suppose que le choix d'index sur les attributs figurant dans les expressions de sélection ait été effectué. Or ces choix ne sont normalement faits qu'après l'étape de définition du format de stockage.

- Travaux de Navathe

A partir des travaux de Hoffer et Severance, puis de Hammer et Niamir, Navathe [NAVA84] a proposé un outil général de fragmentation verticale composé d'une série d'algorithmes.

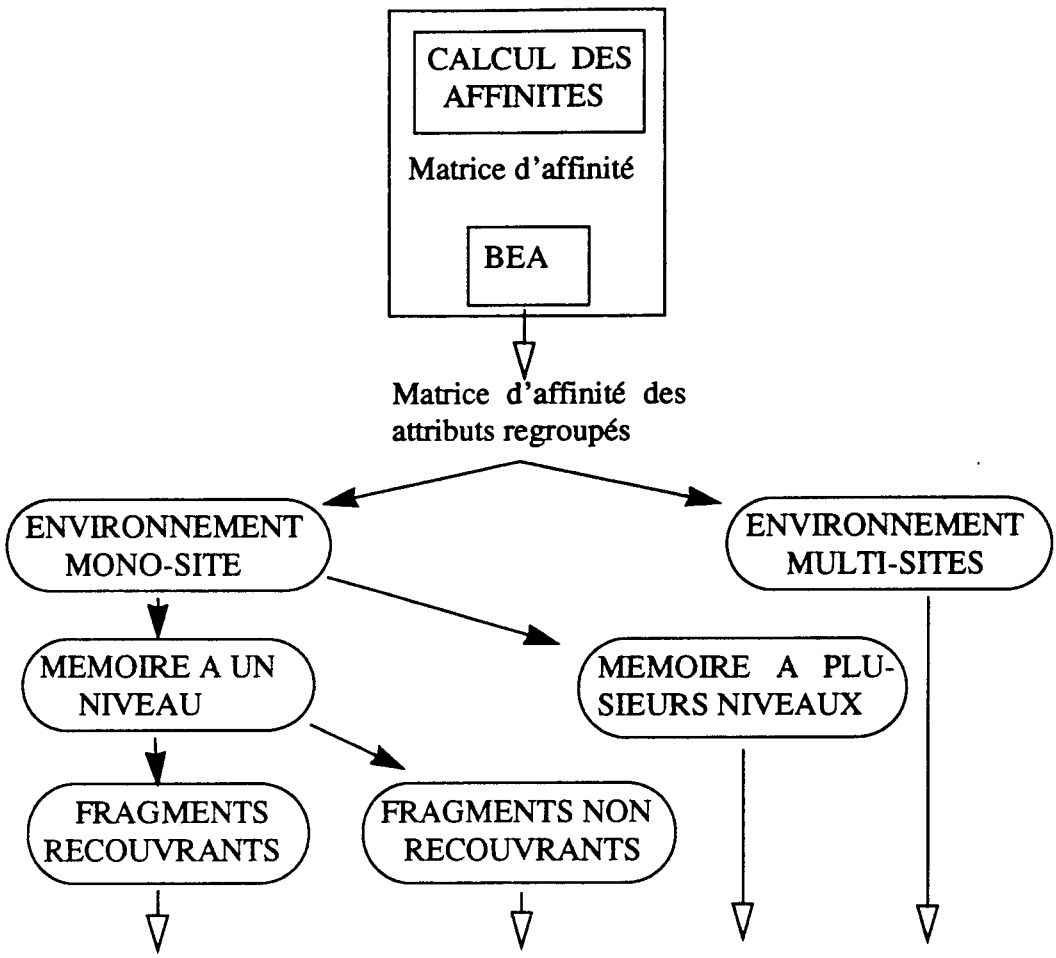


Fig. 2.12 : Caractérisation des travaux de Navathe

Leur démarche comporte deux parties. D'une part, la conception empirique de la fragmentation basée sur la fonction d'affinité entre paires d'attributs de Hoffer et Severance avec quelques modifications. D'autre part, l'implantation physique basée sur des facteurs de coûts relatifs à un environnement physique, tels que le coût de stockage, d'accès, etc. Cette étape optimise la fragmentation obtenue à l'étape précédente et, si possible, effectue l'allocation indépendamment d'un SGBD particulier. Elle lève donc la critique de Hammer et Niamir puisque l'administrateur n'a plus à décider seul le regroupement des attributs.

Ces travaux sont très généraux puisqu'ils s'adaptent à plusieurs types de problèmes comme le montre la figure 12.Etant donné que notre approche est au départ la même, nous reviendrons dans ce qui suit sur certains points précis de cette méthode.

. Travaux orientés modèle de stockage systématique.

Nous présentons ici les travaux de G. Copeland et S.N. Khoshafian sur un modèle de stockage totalement décomposé appelé DSM [COPE85]. Les auteurs proposent d'éclater tout ensemble de données en autant de fragments verticaux que ces données ont d'attributs. En

d'autres termes, ils préconisent une fragmentation verticale extrême. Il ne s'agit donc plus ici de regrouper les attributs fréquemment utilisés ensemble, mais tout simplement d'aboutir à une entité physique par attribut appelée relation d'attribut. Les données de départ sont supposées posséder un identifiant interne. Dès lors, toute relation d'attribut est constituée de deux champs qui sont l'identifiant et l'attribut en question.

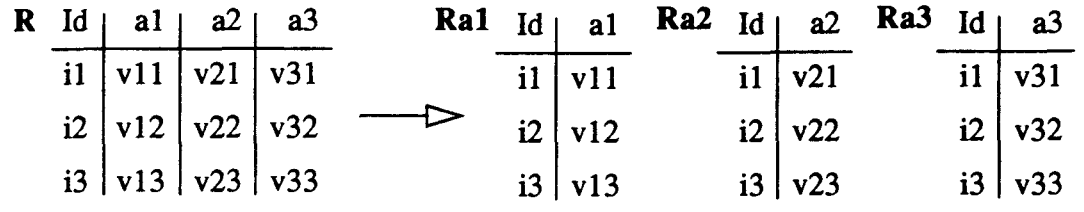


Fig. 2.13 : Principe général du modèle de stockage décomposé (DSM)

Les auteurs supposent même l'utilisation de deux copies pour chaque relation d'attribut; l'une regroupée sur les valeurs de l'attribut et l'autre sur les identifiants. Ils attribuent à leur modèle un ensemble de propriétés qu'il nous semble intéressant de citer.

Tout d'abord, le DSM permet de manipuler facilement les attributs multi-valués avec plus d'indépendance par rapport aux données et sans augmentation de la complexité. Changer, par exemple,un attribut de mono à multi-valué ne change rien pour les structures de stockage.

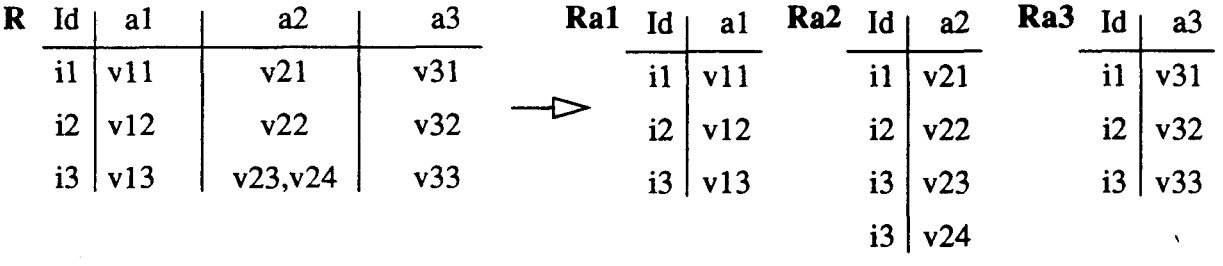


Fig. 2.14 : Implantation des attributs multi-valués avec le DSM

Un autre point important en faveur du DSM est qu'il supporte la notion "d'entité" (ainsi nommée par les auteurs) où l'identité individuelle d'un objet est préservée en la représentant explicitement et indépendamment de sa valeur. Pour cela, il suffit de créer une relation d'entité qui ne comprend que les identifiants. Cela permet alors de gérer des objets qui ont des attributs inconnus sans avoir à stocker explicitement les valeurs nulles. Dans ce cas, en effet, il n'existe pas d'entrée dans la ou les relations d'attribut correspondantes.

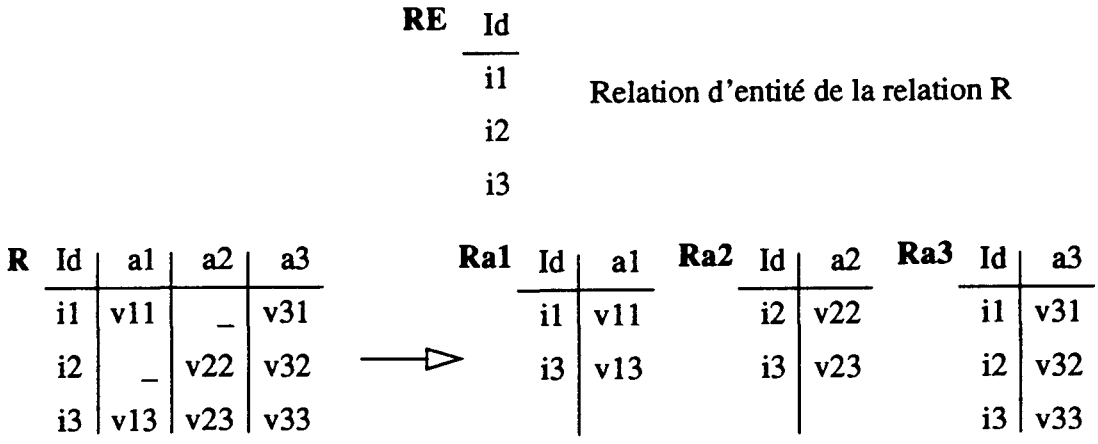


Fig. 2.15 : Implantation des valeurs nulles avec le DSM

Le DSM supporte aussi les relations multi-parents sans stockage redondant d'information comme l'induirait un stockage normalisé classique.

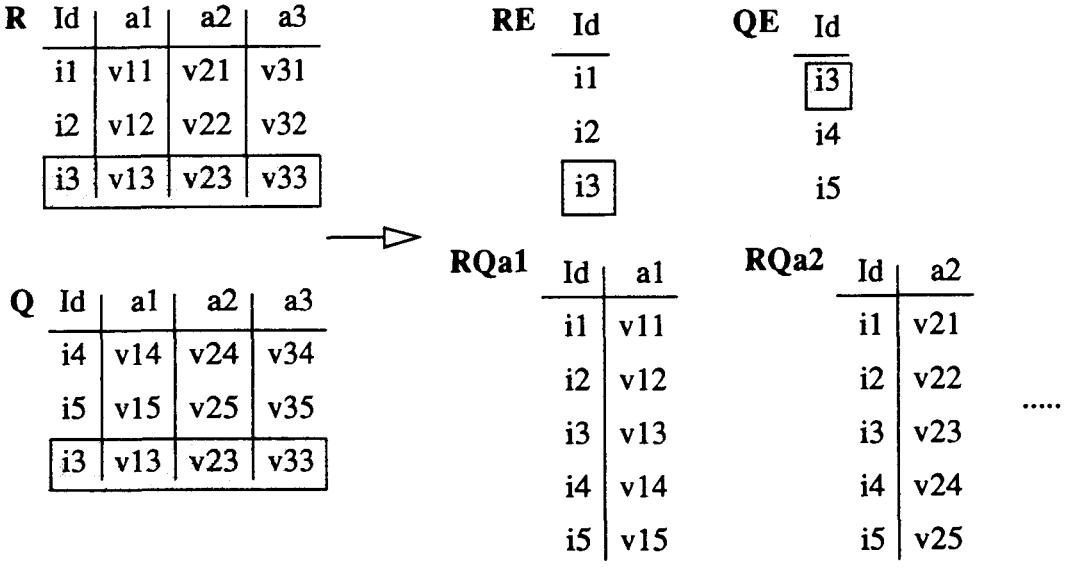


Fig. 2.16 : Implantation d'une relation multi-parents avec le DSM

Grâce à cette propriété on peut envisager une implantation facile de la notion de partage d'objets qui existe dans le modèle que nous considérons. On peut y voir aussi une solution pour la transposition à un niveau interne du concept d'héritage propre aux modèles orientés objets.

Rapidement on peut encore signaler que le DSM supporte les enregistrements hétérogènes (cas des disjonctions du modèle FAD par exemple), les graphes directs (aspect langages orientés objets), les versions et un nombre quelconque d'attributs par relation.

Outre ces propriétés intéressantes, les auteurs proposent une étude quantitative sur l'influence du mode de stockage qu'ils préconisent. Ils évaluent ainsi les besoins de stockage

comparativement à l'approche normalisée habituelle, les performances de mises à jour, de recherche et d'insertion, effacement. Nous ne présenterons pas en détail ici ces résultats qui ne sont, d'ailleurs, pas toujours en faveur du DSM. Etant donné l'éclatement extrême des données il était prévisible que certains traitements "n'y trouvent pas leur compte", et les résultats ainsi obtenus n'ont rien de surprenants. En fait, cette technique est, à notre avis, trop "totalitaire" pour pouvoir répondre efficacement à tous les cas de figure. Elle constitue une recette parmi d'autres, dont l'utilisation systématique ne peut être miraculeuse.

Les auteurs en sont conscients et répondent à cela en mettant en avant d'autres caractéristiques de leur modèle telles que la petite taille des résultats intermédiaires, les accès uniquement sur des données utiles, l'augmentation du potentiel de parallélisme inter-requêtes et surtout la simplicité d'implantation.

Avant de situer le concept de fragmentation verticale dans notre contexte, il est important de signaler qu'à notre connaissance, aucun des projets de machine bases de données à mémoire distribuée n'envisage d'utiliser la fragmentation verticale, ce qui explique l'absence de références dans ce cadre précis.

2.1.3. La fragmentation verticale dans notre contexte

Rappelons avant tout que notre objectif est le placement de données complexes sur une machine bases de données à mémoire distribuée. Nous sortons donc complètement du cadre relationnel centralisé, habituel pour la fragmentation verticale. En ce sens, nous allons devoir adapter ce concept à nos besoins. Nous proposons ici une analyse de l'influence sur la notion de fragmentation verticale, des différents points qui caractérisent notre contexte.

2.1.3.1. Influence du modèle de données

Le modèle de données considéré nous permet de construire des données structurées, ce qui constitue d'entrée un premier problème pour la fragmentation verticale, habituée aux nuplets "plats" du monde relationnel. De plus, ces données peuvent être de deux types: valeurs ou objets. Enfin, ces objets peuvent être partagés.

Sur l'exemple de la figure 2.17, il apparait qu'un objet de la classe CLIENT est composé d'attributs valeurs simples[1] mais aussi de deux attributs objets qui ne sont autres que des objets d'autres classes (ADRESSE et ENS_CDES). L'un de ces attributs objets peut indirectement être vu comme un attribut multi-objets puisqu'il est lui même composé d'objets nuplets de la classe COMMANDE.

L'exemple laisse aussi entrevoir différents types de partage que l'on peut interpréter de façon imagée en énonçant que deux clients peuvent partager une même adresse, un même ensemble de commandes ou qu'une commande peut être partagée par des clients différents au travers d'ensembles de commandes différents.

---

1. Dans ce qui suit nous appelerons ***valeur simple*** une valeur non structurée.

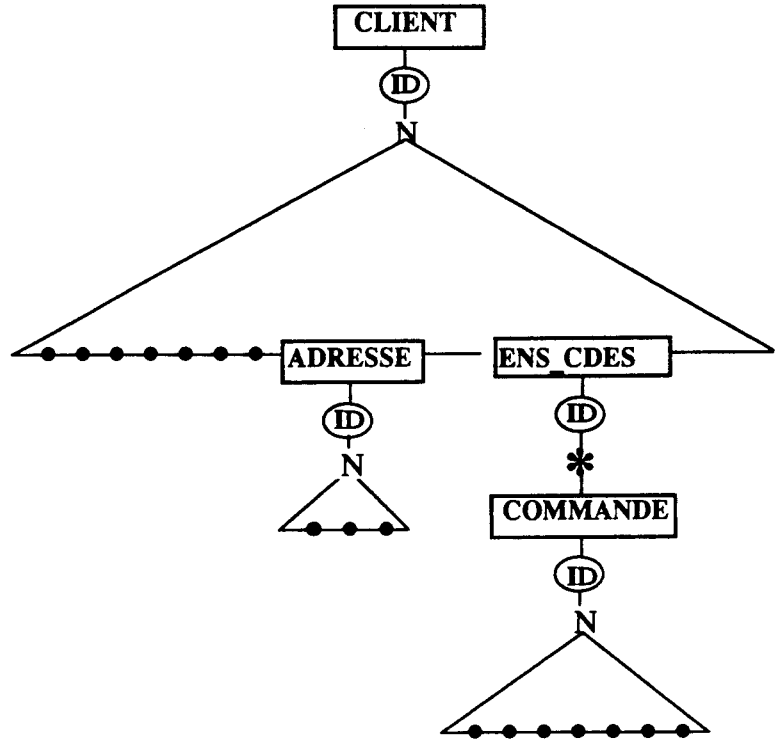


Fig. 2.17 : Exemple de description de classes d'objets complexes

En fait, si nous nous restreignons les classes d'objets nuplets aux attributs valeurs simples, nous nous ramenons à un cas de figure "connu", celui d'ensembles de données normalisées pour lesquels il existe des techniques de fragmentation verticale.

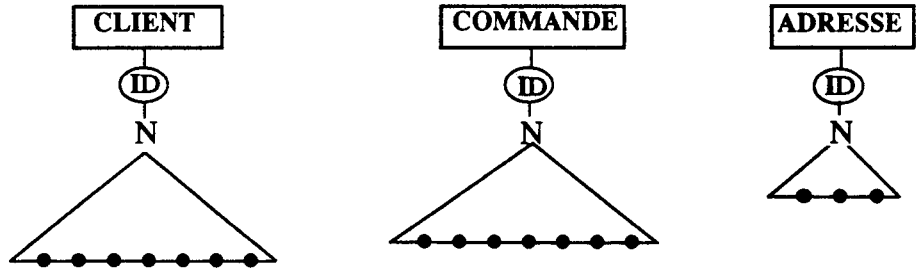


Fig. 2.18 : Classes d'objets nuplets restreintes à leurs attributs valeurs simples

Le problème vient donc essentiellement des attributs objets ou des attributs valeurs structurées que l'on peut voir comme un cas dérivé des attributs objets. En tout état de cause, la notion de fragmentation verticale devra être adaptée pour prendre en compte les nouveaux concepts manipulés par le modèle tels que la structuration ou le partage des données.

Avant d'en terminer avec cette première confrontation entre le modèle et la notion de fragmentation verticale, il est important de faire une remarque sur la notion d'identité d'objet

et sur son utilisation dans le cadre d'une telle fragmentation.

Rappelons en effet que, dans le modèle, tout objet est identifié de façon interne et indépendamment de sa valeur (cf 1.4.2 Présentation de FAD). Dans le cas de la fragmentation verticale d'un ensemble d'objets nuplets, l'information de liaison (permettant de reconstruire à partir des fragments tout ou partie des objets) est toute trouvée puisqu'il suffit de répéter l'identifiant de l'objet au sein de chacun des objets partiels correspondants.

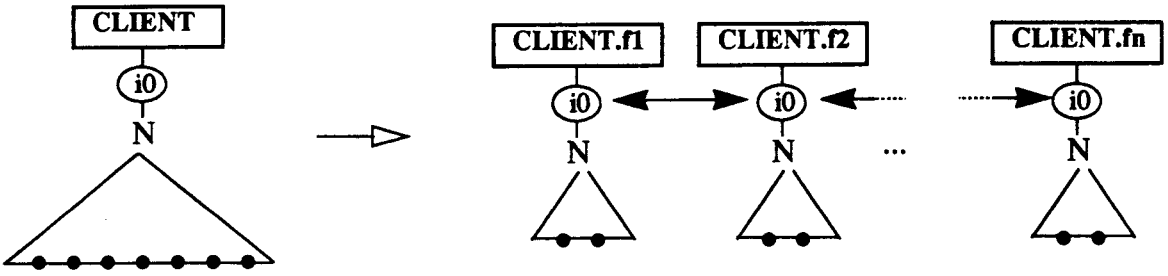


Fig. 2.19 : Utilisation de l'identifiant pour la fragmentation verticale

En fait, la fragmentation verticale d'un objet nuplet produira un certain nombre d'objets partiels qui auront tous le même identifiant (celui de l'objet de départ) mais qui seront différents par leurs attributs.

Par contre dans le cas de la fragmentation d'un ensemble de valeurs nuplets, qui ne sont donc pas identifiées (puisque valeurs), nous retrouverons le problème "du relationnel" puisqu'il faudra construire de toute pièce une information interne de liaison entre valeurs partielles si nous ne voulons pas dupliquer dans chaque fragment la ou les valeurs clé. L'utilisation de la fragmentation verticale sur de telles valeurs nuplets nécessitera donc, soit un mécanisme interne supplémentaire d'identification, soit une refonte partielle du modèle par laquelle les données structurées seraient identifiées mais resteraient non partageables.

Dans ce qui suit, nous considérerons ce cas des attributs valeurs structurées comme un cas dérivé des attributs objets et le traiterons donc de la même façon, sauf pour le cas attribut valeur nuplet, pour lequel nous proposerons une solution particulière. Nous supposerons donc l'existence des mécanismes internes nécessaires.

2.1.3.2. Influence du contexte architectural

Après avoir vu l'influence du modèle de données lui même, nous allons maintenant situer la fragmentation verticale par rapport au contexte architectural. Nous nous plaçons donc dans l'environnement parallèle avec mémoire distribuée retenu.

La fragmentation verticale est donc à voir comme une étape de la conception physique d'une base de données mais au niveau macroscopique des noeuds de la machine cible. Comme nous l'avons signalé auparavant (2.1.1.3 contexte parallèle), les objectifs de la fragmentation verticale dans un tel contexte sont de deux ordres. D'une part, obtenir une certaine localité de l'information au niveau des noeuds tout en évitant les accès inutiles, et d'autre part, augmenter le parallélisme inter-requêtes potentiel de l'application.

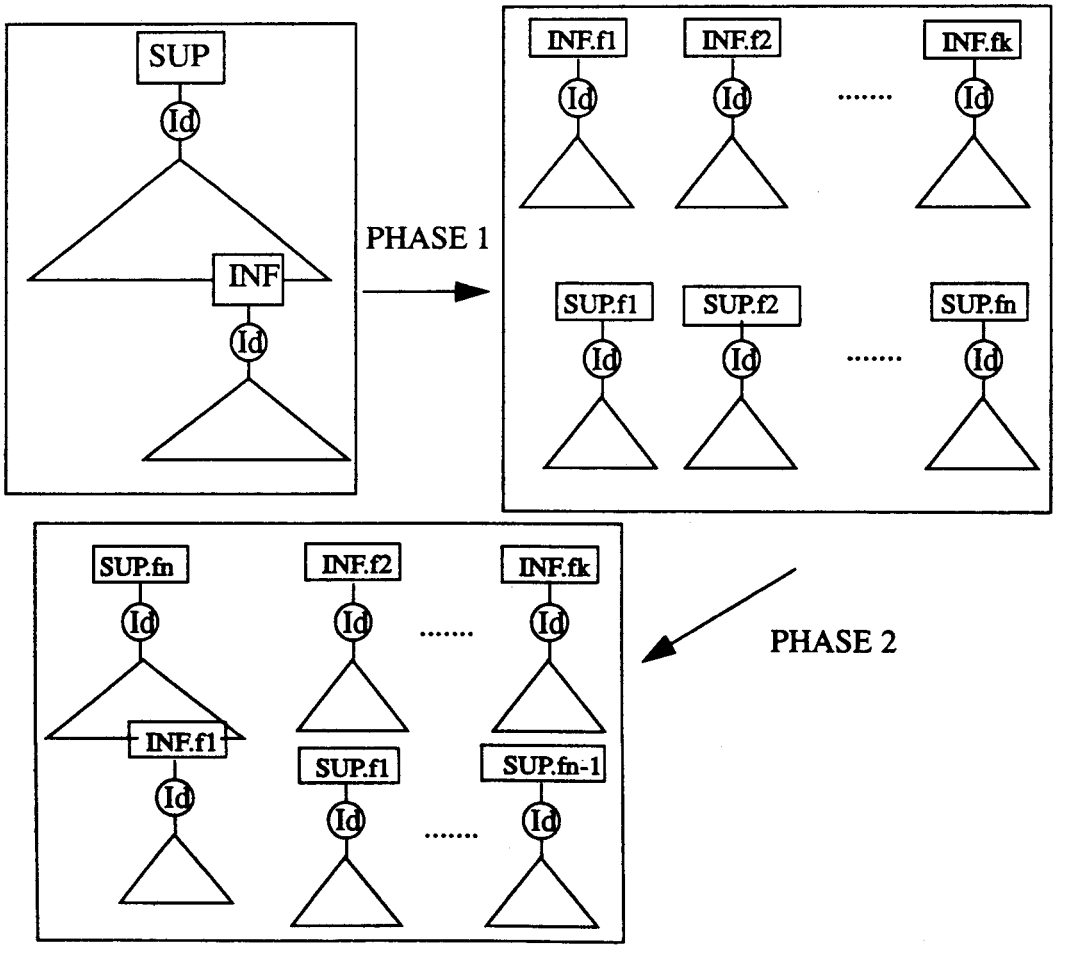
Dans notre cas, la fragmentation verticale des objets en objets partiels nous permettra d'introduire un parallélisme intra-objet (ou inter objets partiels) puisque chacun des objets pourra dès lors être traité en parallèle par des requêtes différentes. Ce nouveau type de parallélisme augmentera donc le parallélisme inter-requêtes, et donc le parallélisme de contrôle, potentiel de l'application.

Si l'on se réfère à l'étude de J. P. Sansonnet [SANS90] (cf 1.3.1 Parallélisme et bases de données), le parallélisme de contrôle constitue la moins importante des trois composantes que sont le parallélisme de contrôle, de flux et de données. De plus, il se trouve vite borné par le type même de l'application. En reprenant l'image de l'auteur qui consiste à considérer ce parallélisme de contrôle comme la potentialité d'évaluation parallèle des arguments d'une fonction, il semble en effet difficile de faire plus d'évaluations parallèles qu'il n'y a d'arguments. Ce que nous faisons ici avec la fragmentation verticale, c'est en fait de scinder ces paramètres en parties indépendantes vis-à-vis de l'évaluation, ce qui permet d'être moins limité en parallélisme de contrôle. Bien sûr la limite existe toujours mais la barre est tout de même relevée.

2.1.4. Présentation de la suite de l'exposé

La suite de ce chapitre concerne les solutions que nous proposons pour la fragmentation verticale d'un schéma logique d'objets complexes, en vue de sa transformation en un schéma physique réparti. La présentation suit l'ordre chronologique de la méthode préconisée et est accompagnée d'un exemple simple afin d'illustrer les différents points.

Fig. 2.20 : Principe général de notre approche de la fragmentation verticale



Notre approche consiste à décomposer la fragmentation verticale en deux phases : la première que l'on peut qualifier de "classique" est relative à la fragmentation verticale des classes d'objets nuplets restreintes à leurs attributs valeurs simples, c'est-à-dire pour lesquelles on ignore, dans un premier temps, la composante structurée. La seconde étape consiste en la prise en compte des attributs objets, ce qui peut être vu comme un prolongement du concept de fragmentation verticale au modèle de données considéré.

Chacune de ces étapes fera l'objet d'une section. Nous terminerons alors ce chapitre par une synthèse sur la méthode proposée et par une présentation sur les différentes extensions que l'on peut envisager. C'est au sein de cette synthèse (cf 2.4) que nous discuterons du "pourquoi" de cette approche en deux étapes. Faute d'une connaissance précise des problèmes, une telle discussion serait, pour l'heure, prématurée, ce qui justifie sa place dans le document.

2.2. Phase 1: Fragmentation verticale des classes d'objets nuplets restreintes à leur attributs valeurs simples

2.2.1. Présentation du problème et des différentes étapes

Nous allons donc nous intéresser ici à la première phase de la fragmentation verticale qui consiste donc à ne prendre en compte que les attributs valeurs simples des objets nuplets. Cette restriction nous permettra de nous placer dans un contexte proche de ceux dans lesquels la fragmentation verticale est habituellement proposée. Ainsi nous pourrons situer notre approche pour cette première phase par rapport aux différents travaux qui existent (cf 2.1.2).

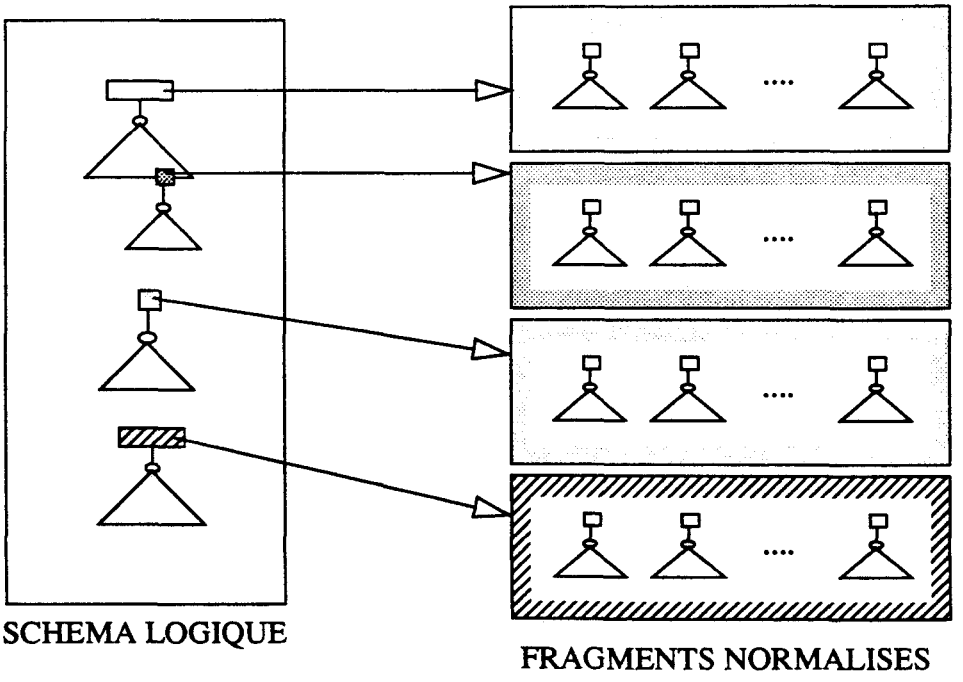


Fig. 2.21 : Principe général de la première phase de la fragmentation verticale

Bien sûr, le but recherché par la décomposition en deux phases n'est pas d'obtenir une première phase "comparable" à l'existant. Il s'agit avant tout de répondre à deux types différents de problèmes qui sont, d'une part, la fragmentation par rapport aux propriétés "simples", c'est-à-dire les attributs non structurés et propres[1] de l'objet, et, d'autre part, par rapport aux attributs "complexes", ie structurés et (ou) objet eux-mêmes.

Nous présentons donc dans ce qui suit une démarche pour fragmenter verticalement toutes les classes d'objets nuplets d'un schéma logique en ne tenant compte que des attributs valeurs simples. Le résultat de cette première phase sera un ensemble de fragment verticaux normalisés (puisque composés de valeurs simples) sur lesquels sera appliquée la seconde phase.

La méthode que nous proposons est à classer parmi celles qui utilisent des heuristiques. Elle est à voir comme une adaptation mais aussi comme une remise en cause de l'outil général de fragmentation verticale proposé par Navathe [NAVA84]. La fragmentation est donc basée sur la fonction d'affinité entre paires d'attributs de Hoffer et Severance [HOFF75] avec néanmoins certaines révisions. Les fragments verticaux normalisés sont produits de façon automatique et ce par l'utilisation d'une fonction paramètrable. Sur ce point, nous nous écartons des propositions faites par Navathe en donnant un sens à la fonction utilisée et en la rendant adaptable.

L'approche utilisée est descendante. En effet, à chaque itération, chacun des fragments verticaux précédemment obtenus est soumis à une éventuelle fragmentation binaire. Contrairement à l'approche ascendante de Hammer et Niamir [HAMM79] qui procèdent à des regroupements (fusions) progressifs, nous partons ici d'un ensemble composé de tous les attributs valeurs atomiques pour arriver, par éclatements progressifs, aux fragments normalisés.

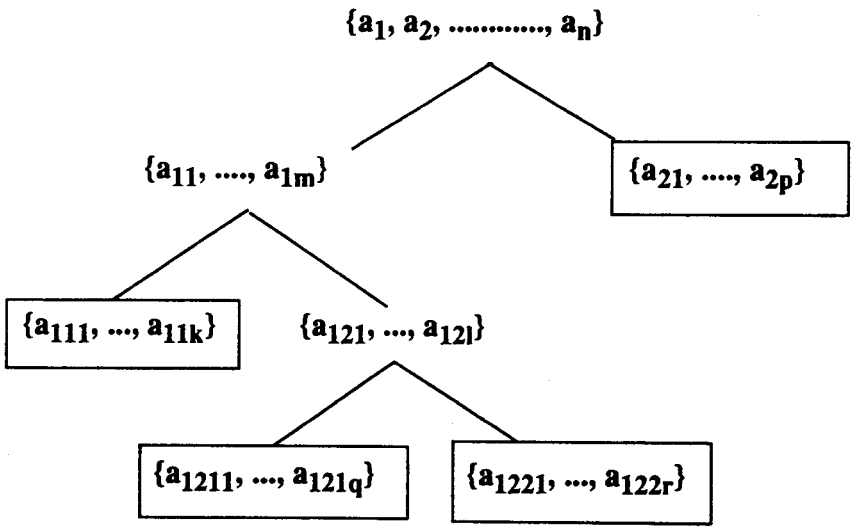


Fig. 2.22 : Principe de l'approche descendante utilisée

Enfin, les fragments élémentaires produits sont disjoints. Néanmoins, nous aborderons les problèmes attachés à l'obtention de fragments non disjoints et discuterons des modifi-

1. Par "propres" nous entendons les attributs valeurs de cet objet qui correspondent à des propriiétés uniquement attachées à l'objet et non relatives à une quelconque autre classe d'objets.

cations à apporter à notre méthode.

La fragmentation verticale de toute classe d'objets nuplets restreinte à ses attributs valeurs, se décompose en quatre étapes. La première étape concerne la construction de la matrice d'affinité entre attributs-valeurs. Pour cela, un coefficient d'affinité est calculé pour tout couple d'attributs et ce, à partir des paramètres qui caractérisent les principales requêtes soumises au système. La seconde étape consiste à transformer la matrice d'affinité pour la mettre sous une forme "exploitable", ce qui se fait grâce au BEA (Bond Energy Algorithm), algorithme de recherche opérationnelle. La troisième étape consiste à produire récursivement les fragments à partir d'une analyse de la matrice d'affinité transformée. Enfin, la quatrième et dernière étape consiste en une éventuelle amélioration que nous qualifirons d'optimisation locale de la fragmentation obtenue. Chacune de ces quatre étapes fera dans ce qui suit l'objet d'une sous-section.

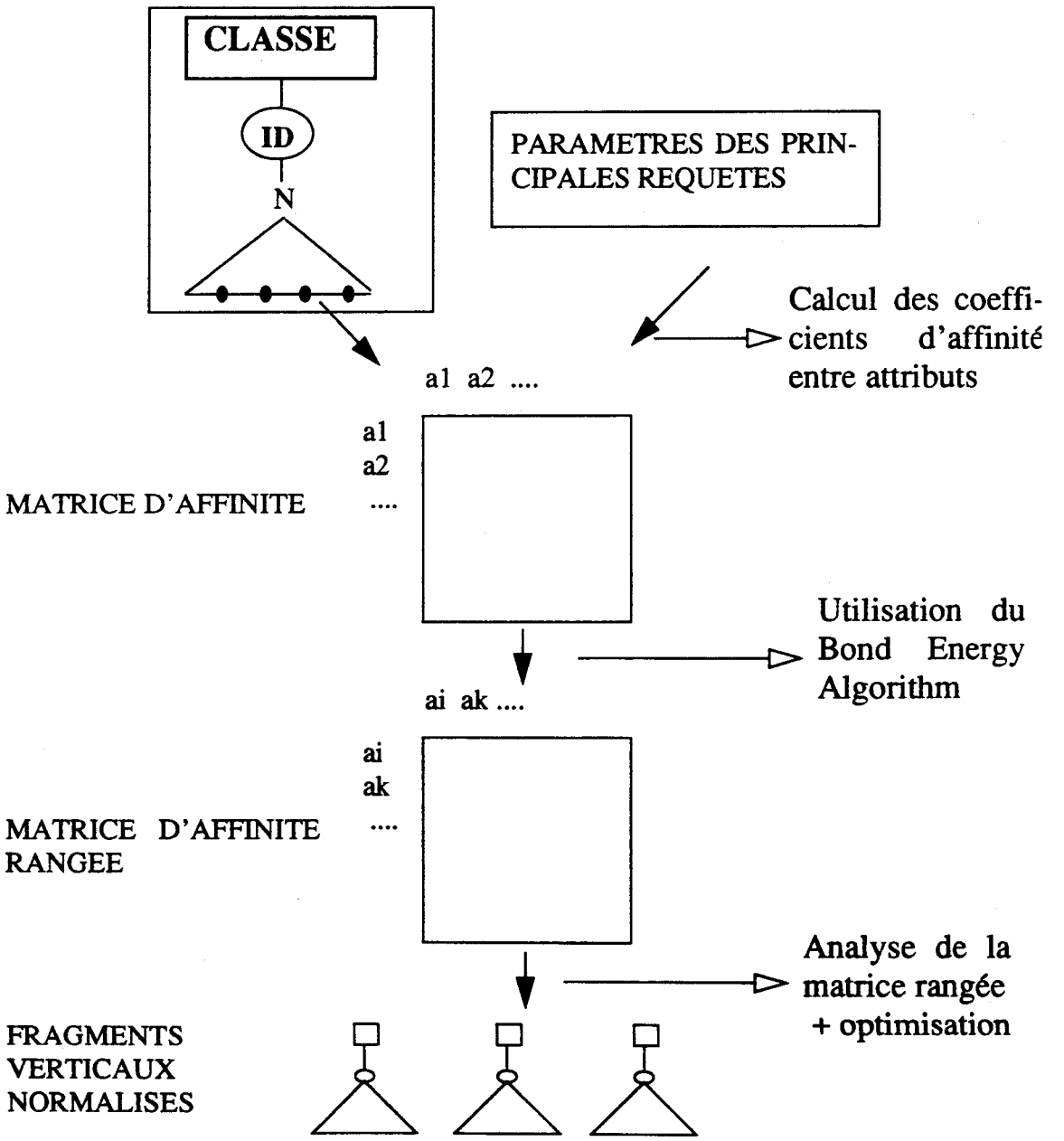


Fig. 2.23 : Les différentes étapes de la phase1 de fragmentation verticale

Afin d'illustrer ces étapes, nous utiliserons l'exemple simple exprimé ci-dessous en langage FAD et sous forme graphique.

```
CLIENT=obj [numero=strings, prem_cde=date, texte=strings, remise=int, nom= strings,
total_cdes=float, societe=strings, ad=ADDRESSE, commandes=ENS_CDES ]
ENS_CDES=obj { COMMANDE }
COMMANDE= obj [ numero=strings, vendeur=strings, date=date, date_livraison= date,
lieu_livraison=strings, montant=money, solde=money ]
ADRESSE= obj [ rue=strings, code_postal=strings, ville=strings ]
```

Fig. 2.24 : Expression de l'exemple utilisé en langage FAD

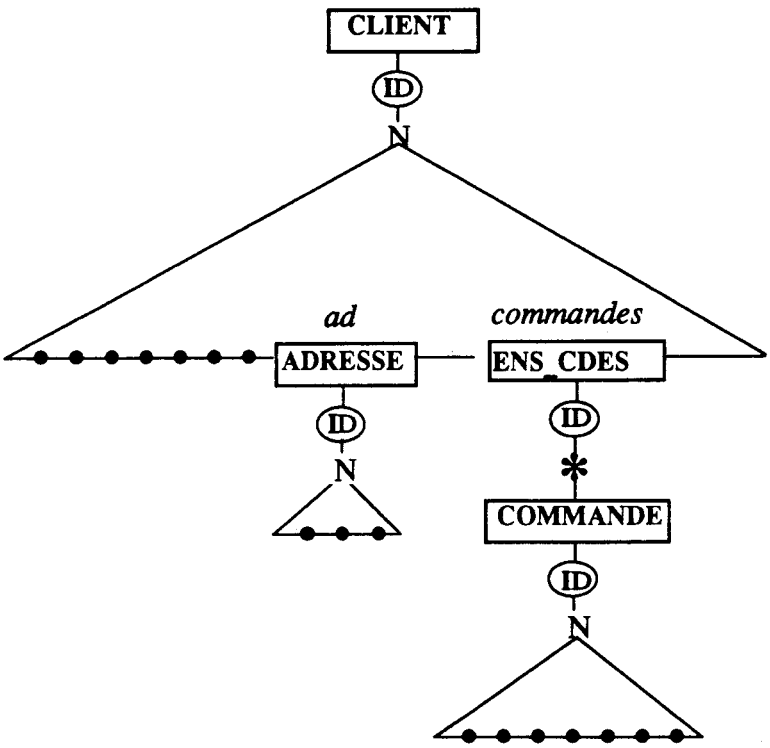


Fig. 2.25 : Expression de l'exemple utilisé sous forme graphique

2.2.2. Construction de la matrice d'affinité

A partir d'un certain nombre de paramètres propres à la classe C d'objets nuplets à fragmenter et aux requêtes qui y accèdent, nous pouvons calculer un coefficient d'affinité pour tout couple d'attributs valeurs de cette classe.

Un tel coefficient d'affinité entre deux attributs a_i et a_j va mesurer la "force" d'un lien imaginaire entre les deux attributs en se basant sur leur utilisation simultanée par les requêtes.

Pour chaque requête R_k, les paramètres utiles sont:

- ***S_k(classe)*** la sélectivité absolue de la requête sur la classe d'objets considérée, c'est-à-dire le pourcentage d'objets qui seront atteints par la requête,

- ***u_{ki}*** : les coefficients d'utilisation des attributs, c'est-à-dire l'information qui permet de savoir si une requête R_k utilise ou pas un attribut a_i,

u_{ki}= 1 <->R_k utilise a_i u_{ki}=0 <-> R_k n'utilise pas a_i

- ***$FREQ_k$*** la fréquence de la requête.

En ce qui concerne la classe d'objets en elle-même, il faut bien sûr connaître sa cardinalité que nous noterons ***CARD(C).***

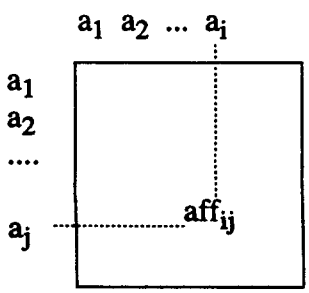
A partir de ces paramètres, nous pouvons calculer pour toute requête R_k le nombre d'objets auquels elle accéde dans la classe C:

$$N_k(C)=S_k(C) * CARD(C)$$

Dès lors, nous pouvons exprimer la définition d'un coefficient d'affinité entre deux attributs a_i et a_j par la formule suivante:

$$aff_{i,j} = \sum_{k \text{ tq } u_{ki}=1 \text{ et } u_{kj}=1} N_k(C) \times FREQ_k$$

Un coefficient d'affinité entre deux attributs correspond donc au cumul des accès faits par unité de temps par les requêtes qui utilisent simultanément ces deux attributs.On peut donc le voir comme un nombre d'accès logiques simultanés par unité de temps à ces attributs. Nous employons ici le terme d'accès logiques puisque ce nombre est calculé uniquement à partir des paramètres liés aux requêtes, éléments qui reflètent l'utilisation logique des données au niveau logique utilisateur. Aucun aspect physique n'est ici pris en compte.



Une fois calculés, ces coefficients d'affinité seront regroupés dans une matrice carrée de dimension le nombre d'attributs valeurs de la classe.

Fig. 2.26 : Matrice d'affinité produite pour chaque classe

La propriété importante d'une telle matrice d'affinité est que toute valeur diagonale est maximale sur sa ligne et sur sa colonne, ce qui peut se traduire par:

$$(\forall i)\ (\forall j)\ (aff_{ij} \leq aff_{ii} \wedge aff_{ij} \leq aff_{jj})$$

Cela s'explique logiquement par le fait qu'il est normal que le nombre d'accès simultanés à deux attributs différents soit inférieur ou égal aux nombres d'accès à chacun de ces attributs pris séparément, nombres que mesurent les valeurs diagonales. Cette propriété nous sera d'une grande utilité dans l'étape de construction récursive de fragments.

A titre d'exemple, nous avons défini un ensemble de requêtes simples sur les classes d'objet CLIENT et COMMANDE dont nous donnons ci-dessous les paramètres. Les attributs sont, dans ces tableaux, numérotés dans leur ordre de description.

Rk. \ Att.	1	2	3	4	5	6	7	S_k	$FREQ_k$
0	1	0	0	1	1	0	1	0.4	1
1	1	0	1	0	1	0	1	0.5	0.14
2	0	0	0	0	0	0	0	0	0.03
3	0	0	0	1	0	1	0	0.05	0.03
4	1	0	0	0	1	0	0	0.001	100
5	1	1	0	0	1	0	1	0.001	0.03
6	1	1	1	1	1	1	1	0.2	0.003
7	1	0	0	1	1	0	1	1	0.03

Classe CLIENT
(1000 objets)

Rk \ Att.	1	2	3	4	5	6	7	S_k	$FREQ_k$
0	1	0	0	1	1	0	0	0.004	1
1	1	0	1	1	0	1	1	0.025	0.14
2	0	1	1	0	0	1	1	0.1	0.03
3	0	0	0	0	0	0	0	0	0.03
4	1	0	1	1	0	1	0	0.0005	100
5	0	0	0	0	0	0	0	0	0.03
6	0	0	1	0	0	0	0	0.1	0.003
7	0	0	0	0	0	0	0	0	0.03

Classe COMMANDE
(100000 objets)

Fig. 2.27 : Paramètres relatifs aux requêtes

Les matrices obtenues pour chacune des classes sont les suivantes:

	1	2	3	4	5	6	7
1	601	1	71	431	601	1	501
2	1	1	1	1	1	1	1
3	71	1	71	1	71	1	71
4	431	1	1	432	431	2	431
5	601	1	71	431	601	1	501
6	1	1	1	2	1	2	1
7	501	1	71	431	501	1	501

Matrice d'affinité de la classe CLIENT

	1	2	3	4	5	6	7
1	5750	0	5350	5750	400	5350	350
2	0	300	300	0	0	300	300
3	5350	300	5680	5350	0	5650	650
4	5750	0	5350	5750	400	5350	350
5	400	0	0	400	400	0	0
6	5350	300	5650	5350	0	5650	650
7	350	300	650	350	0	650	650

Matrice d'affinité de la classe COMMANDE

Fig. 2.28 : Matrices d'affinité obtenues sur notre exemple

En y regardant de plus près, nous pouvons discerner dans ces matrices des similitudes entre certains coefficients d'affinité. Prenons, par exemple, le cas des attributs 1 et 5 de la classe CLIENT.

Les coefficients d'affinité qui s'y rapportent sont égaux, ce qui signifie qu'ils sont toujours accédés ensemble, puisque tout accès à l'un d'entre eux est forcément simultané aux deux ($aff_{1,1}=aff_{5,5}=aff_{1,5}$). Dans une moindre mesure, l'attribut 7 est sollicité par les requêtes presque de la même façon que le 1 et le 5. Il apparait donc intéressant de former un fragment vertical avec ces trois attributs qui se distinguent des autres par leur utilisation.

Le problème qui se pose cependant est celui de l'analyse d'une telle matrice "désordonnée". Il apparait que la tache de détermination des fragments verticaux serait beaucoup plus aisée si l'on disposait d'une forme arrangée de cette matrice, où les coefficients d'affinité semblables seraient regroupés. Ce problème de regroupement de valeurs dans un tableau à deux dimensions peut être solutionné grâce au BEA (Bond Energy Algorithm) [MCOR72] que nous présentons en détail dans la section suivante.

Avant d'en terminer avec cette partie sur la construction des matrices d'affinité, il est important de signaler, que dans certains travaux, la taille des attributs est prise en compte pour le calcul des coefficients d'affinité [HOFF75]. Nous avons pris partie ici de ne pas en tenir compte pour diverses raisons. Ce qui nous intéresse avant tout, c'est de déterminer les éventuelles similitudes entre les accès logiques aux attributs et non pas celles entre les quantités d'informations accédées. Il faut bien voir qu'avec une pondération des accès logiques par la taille des attributs concernés, un accès à un gros attribut peut devenir équivalent à un grand nombre d'accès sur de petits attributs.On risque alors de regrouper, à cause de coefficients d'affinité semblables, des attributs qui ne sont pas du tout accédés à la même fréquence!

Une telle façon de procéder nous écarterait des objectifs recherchés par la fragmentation verticale.Vouloir prendre en compte la taille des attributs au moment de la construction de la matrice d'affinité n'est pas adapté et, notons-le, fait perdre les propriétés intéressantes d'une telle matrice. Par contre, nous verrons dans ce qui suit que cette prise en compte peut tout à fait s'intégrer dans l'étape de production récursive des fragments.

2.2.3. Transformation de la matrice d'affinité

Nous utilisons le BEA (Bond Energy Algorithm), proposé par Mac Cormick [MCOR72], afin de transformer la matrice d'affinité pour la mettre sous une forme plus exploitable et suivons donc, à ce titre, la démarche de Hoffer et Severance [HOFF75].

Cet algorithme de recherche opérationnelle est un outil général qui, par des permutations de lignes et de colonnes d'un tableau de données, le transforme sous une forme "semi-bloc diagonale". Son but est de mettre en évidence des regroupements dans des tableaux de données. Il est basé sur l'utilisation d'une mesure d'efficacité appliquée à toute permutation de ligne ou colonne du tableau initial.

La fonction objective[1] qu'il cherche à maximiser, tend à regrouper les valeurs semblables entre elles. En d'autres termes, la fonction est telle qu'une configuration comportant des blocs denses de données fournit une meilleure mesure qu'une autre où les données sont "éparpillées".

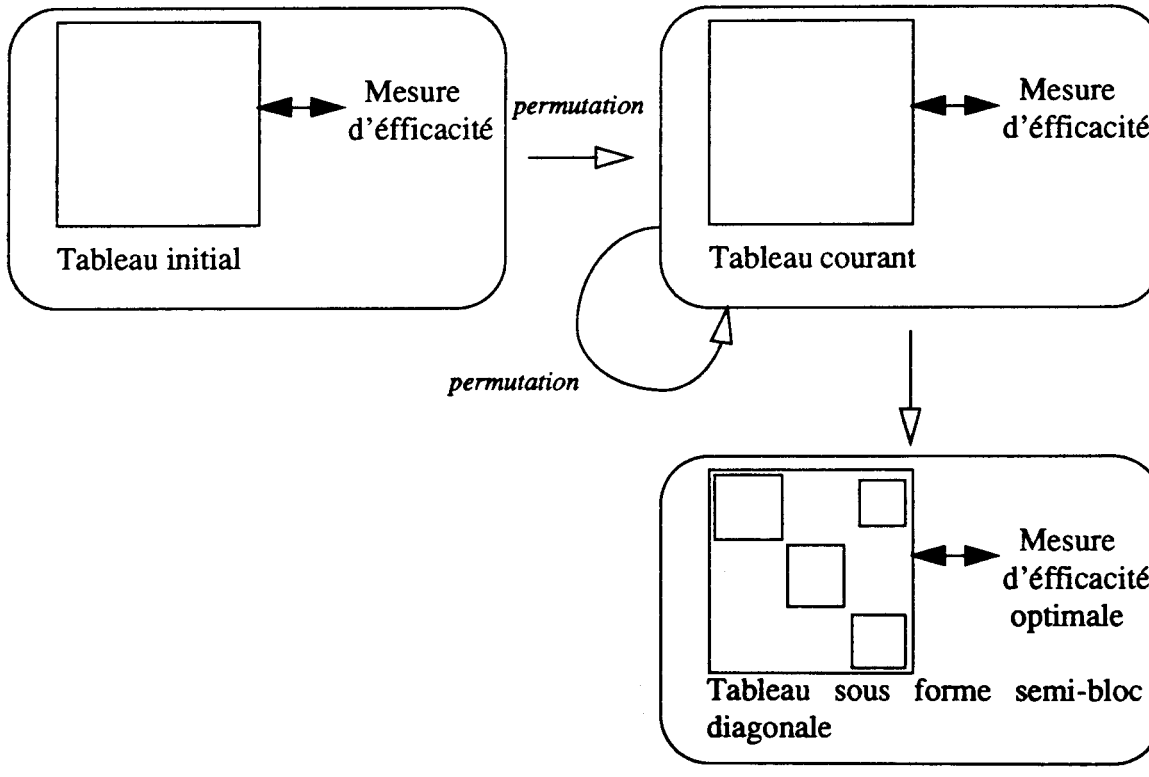


Fig. 2.29 : Principe général du BEA

1. Il est important de noter que Mac Cormick propose dans son article une famille de fonctions plutôt qu'une seule fonction. Nous n'entrerons pas ici dans l'étude des subtilités qui différencient ces fonctions, et nous nous contenterons d'utiliser la fonction de référence.

La fonction habituellement calculée pour chaque configuration est la suivante:

$$f(config) = \sum_{i=1}^{n}\sum_{j=1}^{n} aff_{i,j} \times [aff_{i-1,j} + aff_{i+1,j} + aff_{i,j-1} + aff_{i,j+1}]$$

Les hypothèses ci-dessous sont retenues:

$aff_{i,0} = aff_{0,j} = aff_{i,n+1} = aff_{n+1,j} = 0$

Dans notre cas, le tableau de données est donc une matrice d'affinité carrée . De plus, cette matrice est symétrique ce qui entraîne des permutations liées des lignes et colonnes et donc une diminution de la complexité de l'algorithme puisque moins de configurations sont à tester.

L'application de l'algorithme fournit une nouvelle matrice d'affinité dans laquelle les blocs sur la principale diagonale correspondent à des groupes d'attributs-valeurs accédés à la même fréquence. Ces groupes sont donc candidats pour devenir fragments verticaux.

Une remarque avant de présenter l'application de l'algorithme à notre exemple. Nous avons apporté une modification mineure en considérant que la matrice d'affinité est bordée d'une valeur non nulle "assez grande" plutôt que 0. La raison en est assez simple.

Par l'utilisation de 0, les coefficients d'affinité les plus importants se retrouvent au centre de la matrice rangée, ce qui s'explique tout à fait si on se penche un peu sur la nature même de la fonction utilisée. Le problème qui se pose alors est que l'on risque de trouver des coefficients d'affinité semblables à chaque extrémité diagonale (principale) de la matrice. Il faudrait donc alors considérer cette matrice rangée comme sphérique pour pouvoir y déceler l'existence de ce groupe de coefficients d'affinité semblables.

	a1	a2	a3	a4
a1	40	10	10	40
a2	10	100	100	10
a3	10	100	100	10
a4	40	10	10	40

Fig. 2.30 : Exemple de résultat obtenu en considérant la matrice bordée de 0

Pour pallier cet inconvénient, qui risquerait d'alourdir inutilement l'étape ultérieure de production de fragments, nous proposons donc "d'entourer" la matrice d'une valeur importante, ce qui aura pour effet d'envoyer sur une des extrémités diagonales le bloc formé des plus grandes valeurs et ainsi de fournir une matrice où les blocs seront ordonnés[1] sur la principale diagonale. Ainsi nous n'aurons plus de problèmes pour détecter ces blocs de coefficients d'affinité semblables qui ne seront plus sujets aux effets de bords.

1. Cet ordre n'est pas un ordre par rapport aux valeurs diagonales elles-mêmes, mais par rapport aux blocs de valeurs semblables. En ce sens, la diagonale n'est pas forcément ordonnée.

	a4	a1	a2	a3
a4	40	40	10	10
a1	40	40	10	10
a2	10	10	100	100
a3	10	10	100	100

Fig. 2.31 : Résultat en considérant la matrice bordée de sa valeur moyenne

Dans les nombreux essais réalisés à partir de matrices aléatoires, nous avons utilisé la valeur moyenne comme "bordure". Les résultats avaient toujours la propriété précédemment citée, propriété qui nous sera de toute utilité dans l'étape suivante.

A partir de notre exemple, nous obtenons les matrices rangées ci-dessous:

	1	2	3	4	5	6	7
1	601	1	71	431	601	1	501
2	1	1	1	1	1	1	1
3	71	1	71	1	71	1	71
4	431	1	1	432	431	2	431
5	601	1	71	431	601	1	501
6	1	1	1	2	1	2	1
7	501	1	71	431	501	1	501

	6	2	3	4	1	5	7
6	2	1	1	2	1	1	1
2	1	1	1	1	1	1	1
3	1	1	71	1	71	71	71
4	2	1	1	432	431	431	431
1	1	1	71	431	601	601	501
5	1	1	71	431	601	601	501
7	1	1	71	431	501	501	501

Fig. 2.32 : Transformation de la matrice d'affinité de la classe CLIENT

Il y apparait des blocs d'affinité qui semblent être d'excellents candidats pour devenir fragments verticaux. Nous allons donc passer maintenant à l'étape de production de ces fragments.

	1	2	3	4	5	6	7
1	5750	0	5350	5750	400	5350	350
2	0	300	300	0	0	300	300
3	5350	300	5680	5350	0	5650	650
4	5750	0	5350	5750	400	5350	350
5	400	0	0	400	400	0	0
6	5350	300	5650	5350	0	5650	650
7	350	300	650	350	0	650	650

	1	4	3	6	7	2	5
1	5750	5750	5350	5350	350	0	400
4	5750	5750	5350	5350	350	0	400
3	5350	5350	5680	5650	650	300	0
6	5350	5350	5650	5650	650	300	0
7	350	350	650	650	650	300	0
2	0	0	300	300	300	300	0
5	400	400	0	0	0	0	400

Fig. 2.33 : Transformation de la matrice d'affinité de la classe COMMANDE

2.2.4. Production récursive de fragments

Nous proposons donc dans cette section une méthode pour produire de façon automatique les fragments verticaux à partir de la matrice d'affinité rangée.

Contrairement à Hoffer et Severance [HOFF75], nous ne laissons donc pas à l'administrateur le rôle délicat de la détermination des fragments à partir de cette matrice rangée. Bien que Navathe [NAVA84] ait proposé une solution au problème, nous n'utiliserons pas sa méthode qui se sert, à notre avis, de fonctions discutables et qui apparait comme peu adaptable. Nous reviendrons lors de la présentation de notre approche, sur leur méthode afin d'établir différentes comparaisons avec la solution que nous préconisons.

2.2.4.1. Principe général de la méthode

La fragmentation se fait d'une façon récursive descendante puisqu'elle s'applique aux fragments obtenus. A chaque itération, chacun des fragments précédemment déterminés est soumis à une éventuelle fragmentation binaire et peut ainsi être scindé en deux nouveaux fragments. Le fragment initial est constitué de l'ensemble des attributs valeurs de la classe considérée. Le processus s'arrête quand aucun des fragments obtenus n'est fragmentable.

A une itération donnée, l'éventuel découpage binaire de l'un des fragments obtenus à l'étape précédente, se fait par une analyse du bloc d'affinité correspondant qui n'est autre que

l'ensemble des coefficients d'affinité entre les attributs du fragment considéré. Nous donnons ci-dessous une représentation imagée d'un tel processus de fragmentation verticale. Les feuilles de l'arbre binaire ainsi obtenu correspondent aux blocs d'affinité relatifs aux fragments retenus.

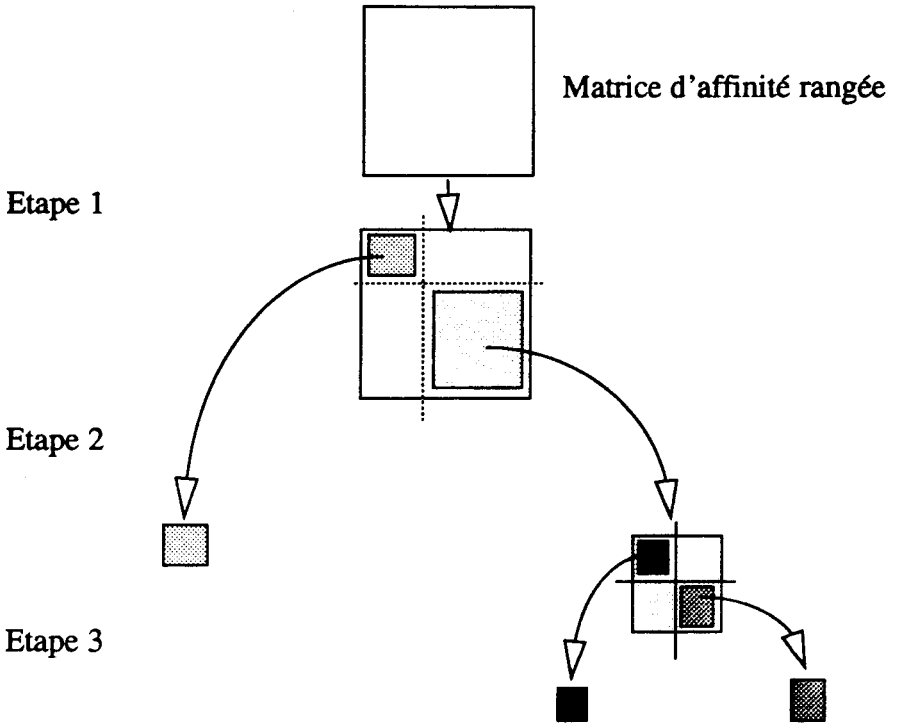


Fig. 2.34 : Processus de détermination des fragments verticaux

Nous pouvons exprimer la procédure générale de fragmentation dans un pseudo-langage de programmation par:

```
FRAGMENTER(BLOC)
    découpage_optimal=recherche_découpage_optimal(BLOC);
    SI (découpage_optimal <> AUCUN_DECOUPAGE)
    ALORS FRAGMENTER (BLOC_SUP);
            FRAGMENTER(BLOC_INF);
    FSI
FIN_FRAGMENTER
```

2.2.4.2. Recherche du découpage optimal d'un bloc d'affinité

Soit B un bloc d'affinité qui correspond à un fragment fp obtenu à l'étape i du processus de fragmentation et que l'on soumet à un éventuel découpage à l'étape i+1.

Le principe général de la recherche du découpage binaire optimal du bloc B consis-

te à mesurer les inconvénients induits par chacun des découpages candidats, y compris le non découpage, et à retenir celui qui minimise ces inconvénients. En d'autres termes, nous allons chercher à minimiser une fonction de coûts qui sera calculée pour chacun des k découpages possibles. Il est important de noter que la fonction sera aussi évaluée pour un bloc non découpé ce qui permettra par comparaison avec les différents découpages, d'arrêter ou pas le processus de fragmentation.

Un découpage binaire quelconque du bloc B fournit une configuration en quatre sous-blocs dont certains peuvent être nuls (cas du non découpage) et dont les deux qui se situent sur la seconde diagonale sont égaux par rapport aux valeurs qu'ils contiennent.

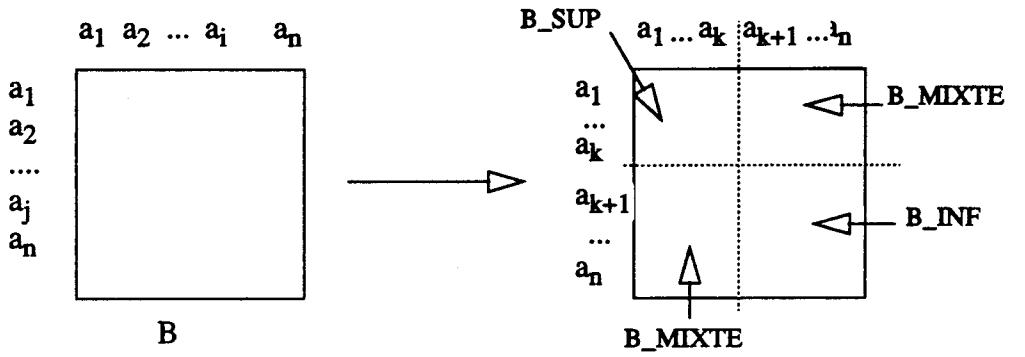


Fig. 2.35 : Configuration du bloc B découpé

La mesure des inconvénients induits par un tel découpage, ie l'évaluation de la fonction de coûts, consiste à analyser chacun des différents sous-blocs et à lui attribuer suivant sa nature un certain type d'inconvénients. Les deux types de mesure réalisés sont:

- les accès inutiles dans les sous-blocs B_SUP et B_INF
- les accès supplémentaires dans le sous-bloc B_MIXTE

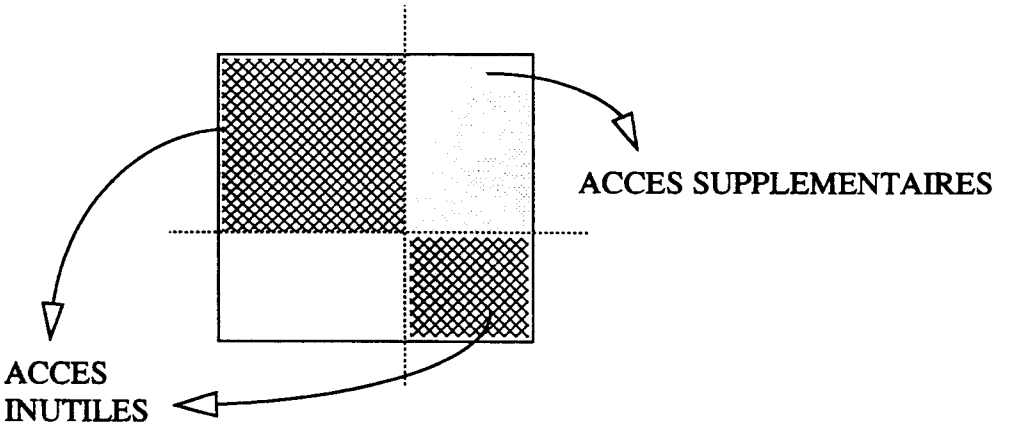


Fig. 2.36 : Les différentes mesures effectuées pour un découpage donné

. Mesure des accès inutiles dans B_SUP et B_INF

Considérons de près les valeurs contenues dans un tel sous-bloc. Ce sont des coefficients d'affinité entre attributs qui représentent donc un nombre d'accès simultanés logiques à deux attributs.

Il est cependant important de remarquer que puisqu'elles appartiennent à un bloc de la diagonale principale, elles sont relatives à des attributs qui appartiendront, si le découpage est retenu, à un même fragment. La conséquence finale de la fragmentation verticale étant l'assignation de chaque fragment à une entité physique différente, tout coefficient d'affinité dans B_SUP (ou B_INF) représente donc un nombre d'accès logiques simultanés à deux attributs qui sont physiquement regroupés.

Ce regroupement physique des attributs d'un même fragment a néanmoins un inconvénient. En effet, de ce fait, tout accès à l'un d'entre eux entraîne un accès à tous les autres. Nous retrouvons là l'inévitable problème des accès inutiles, non plus au niveau de l'objet dans son intégralité, mais au niveau de chaque objet partiel issu de la fragmentation verticale.

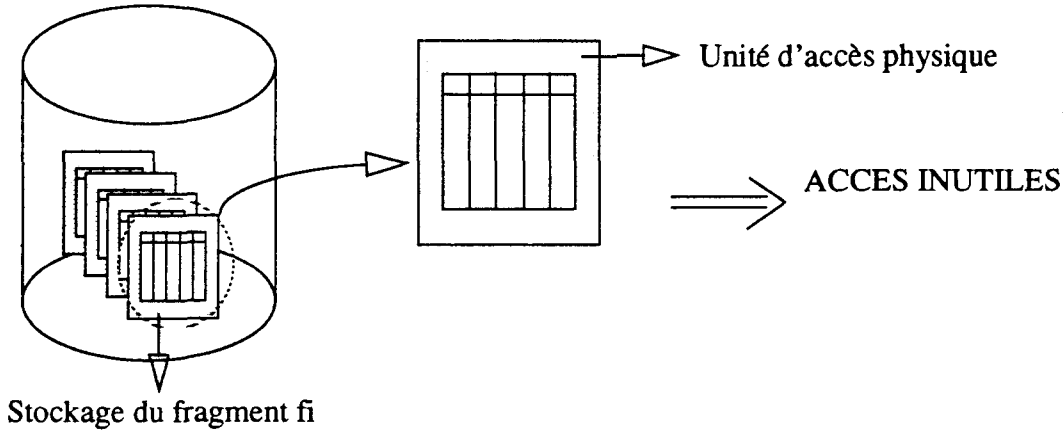


Fig. 2.37 : Accès inutiles au niveau de chaque fragment

Nous allons donc mesurer à partir des sous-blocs B_SUP et B_INF les accès inutiles induits par le découpage considéré. Pour cela nous partons des constatations suivantes:

$(\forall i)\ (\forall j)$ tq $(1 \leq i \leq k \wedge 1 \leq j \leq k) \wedge (k+1 \leq i \leq n \wedge k+1 \leq j \leq n)$

aff_{ii}-aff_{ij} <=> nombre d'accès inutiles à a_j lors des accès à a_i

aff_{jj}-aff_{ij} <=> nombre d'accès inutiles à a_i lors des accès à aj

Remarquons que les valeurs diagonales, qui a priori n'avaient pas trop de sens (coefficient d'affinité entre un attribut et lui même), sont ici de toute importance et que l'on trouve une juste interprétation de leur condition de maximalité.

Nous donnons ci-dessous l'expression des fonctions utilisées pour mesurer les accès inutiles dans B_SUP et B_INF.

$$ACCES\_INUT(B\_SUP) = \sum_{i=1}^{k}\sum_{j=i}^{k} [\,(aff_{i,i} - aff_{i,j}) + (aff_{j,j} - aff_{i,j})\,]$$

$$ACCES\_INUT(B\_INF) = \sum_{i=k+1}^{n}\sum_{j=i}^{n} [\,(aff_{i,i} - aff_{i,j}) + (aff_{j,j} - aff_{i,j})\,]$$

Etant donnée la nature symétrique des sous blocs (par rapport à leur diagonale principale), nous ne faisons les calculs que sur des demi-blocs.

Exemple: Supposons que nous étudions le découpage ci-dessous de la matrice d'affinité relative à la classe CLIENT.

	6	**2**	**3**	**4**	**1**	**5**	**7**
6	2	1	1	2	1	1	1
2	1	1	1	1	1	1	1
3	1	1	71	1	71	71	71
4	2	1	1	432	431	431	431
1	1	1	71	431	601	601	501
5	1	1	71	431	601	601	501
7	1	1	71	431	501	501	501

Fig. 2.38 : Exemple étudié de découpage

La mesure des accès inutiles dans le bloc B_INF s'interprète comme suit:

.tout accès à 1 (resp5) est aussi un accès à 5 (resp1) puisque $aff_{1,5}=aff_{1,1}=aff_{5,5}$. Par conséquent il n'y aura pas d'accès inutiles "entre" ces attributs.

.tout accès à 7 est aussi un accès à 1 puisque $aff_{7,7}=aff_{1,7}$, mais par contre un accès à 1 n'est pas forcément un accès à 7 car $aff_{1,1}>aff_{1,7}$. Donc $aff_{1,1}-aff_{1,7}$ représente le nombre d'accès inutiles à 7 durant les accès à 1.

. Mesure des accès supplémentaires dans B_MIXTE

Nous allons maintenant nous intéresser à l'analyse du sous-bloc B_MIXTE induit par le découpage considéré. Les coefficients d'affinité de ce sous-bloc correspondent toujours à des accès logiques simultanés à deux attributs. Néanmoins, les attributs concernés appartiendront cette fois, si le découpage est retenu, à des entités physiques différentes.

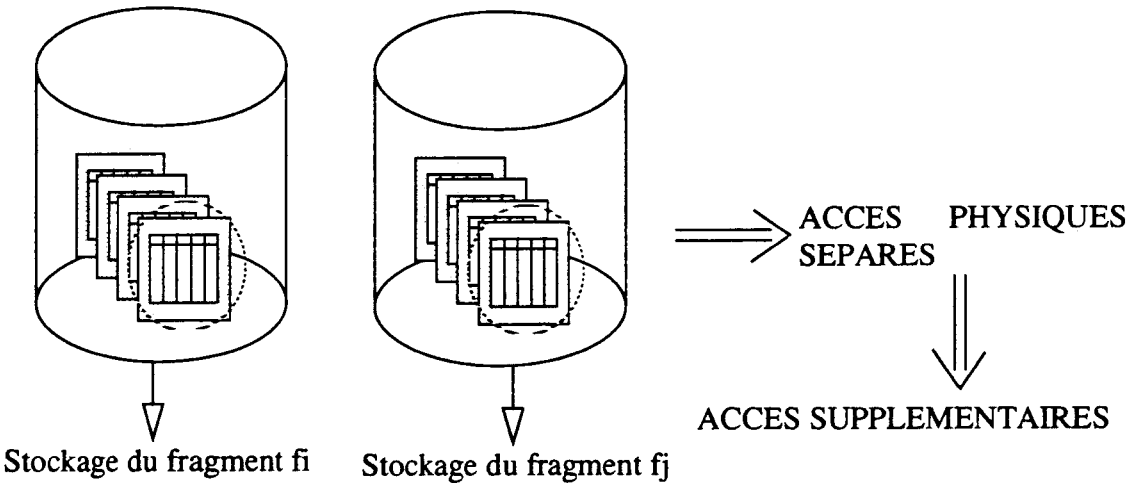
A un accès logique simultané correspond donc deux accès physiques. Nous pou-

vons donc énoncer que le découpage d'un bloc entraîne des accès physiques supplémentaires. Ces derniers sont comptabilisés par les valeurs du bloc B_MIXTE puisque tout accès logique simultané qui y figure implique un accès physique supplémentaire.

Grâce à cette notion d'accès supplémentaires nous prenons donc en compte l'influence néfaste de la fragmentation verticale pour les requêtes qui utilisent simultanément deux fragments.

La fonction utilisée pour l'évaluation, pour un découpage donné, de ces accès supplémentaires est donnée ci-dessous. L'évaluation se fait bien sûr uniquement pour un seul des blocs B_MIXTE.

$$ACCES\_SUP(B\_MIXTE) = \sum_{i=1}^{k} \sum_{j=k+1}^{n} aff_{i,j}$$



2.2.4.3. Utilisation des mesures effectuées

En fait, les deux notions d'accès inutiles et d'accès supplémentaires nous permettent de quantifier les deux types d'inconvénients attachés à chaque découpage candidat. Les accès inutiles reflètent le comportement interne de chacun des fragments issus du découpage considéré. Les accès supplémentaires sont eux l'image du comportement relatif de ces fragments.

Rappelons que, pour la détermination du découpage optimal d'un bloc B, le non découpage est aussi évalué. Il n'y a pas, dans ce cas, d'accès supplémentaires puisque le maintien du fragment (relatif au bloc d'affinité étudié) est considéré. Seuls les accès inutiles relatifs à ce fragment unique sont pris en compte.

Le découpage binaire optimal du bloc B est celui qui minimise une fonction qui prend comme paramètres accès inutiles et accès supplémentaires. A ce niveau, nous sommes confrontés au problème de l'importance relative de chacun de ces types d'inconvénients. Quels

poids donner en effet à un accès supplémentaire vis-à-vis d'un accès inutile?

Il faudrait, pour pouvoir déterminer ce type de dépendance, avoir une connaissance précise de la machine cible et des mécanismes d'accès utilisés. Nos travaux ne s'appuient pas sur une machine particulière. Ils sont plutôt à considérer comme une approche globale des différents problèmes. En ce sens, nous ne chercherons pas ici à mettre en évidence une quelconque fonction de liaison entre accès inutiles et accès supplémentaires. Ce genre de travail relève de la mise en oeuvre des méthodes que nous préconisons, sur une machine donnée, élément dont d'ailleurs nous ne disposons pas.

Pour l'heure nous nous contenterons de supposer qu'il existe une dépendance linéaire entre accès inutiles et accès supplémentaires (1 ACCES_SUP = C ACCES_INUT) et considérerons donc une fonction du type:

F(découpage(B)) = ACCES_INUT(B_SUP) + ACCES_INUT(B_INF) +
C * ACCES_SUP(B_MIXTE)

2.2.4.4. Résultats obtenus sur notre exemple

L'application de la méthode proposée aux matrices d'affinité rangées de notre exemple, fournit les résultats suivants:

	6	**2**	**3**	**4**	**1**	**5**	**7**
6	2	1	1	2	1	1	1
2	1	1	1	1	1	1	1
3	1	1	71	1	71	71	71
4	2	1	1	432	431	431	431
1	1	1	71	431	601	601	501
5	1	1	71	431	601	601	501
7	1	1	71	431	501	501	501

CLIENT

	1	**4**	**3**	**6**	**7**	**2**	**5**
1	5750	5750	5350	5350	350	0	400
4	5750	5750	5350	5350	350	0	400
3	5350	5350	5680	5650	650	300	0
6	5350	5350	5650	5650	650	300	0
7	350	350	650	650	650	300	0
2	0	0	300	300	300	300	0
5	400	400	0	0	0	0	400

COMMANDE

Fig. 2.39 : Matrices d'affinité découpées

Pour l'exemple, accès inutiles et accès supplémentaires sont considérés comme de même poids (C=1). Les classes d'objets nuplets CLIENT et COMMANDE se trouvent donc ainsi, chacune verticalement découpée en trois fragments.

2.2.4.5. Prise en compte éventuelle de la taille des attributs

Nous pouvons tout à fait analyser le problème de la fragmentation verticale d'une façon parallèle en "l'auscultant" cette fois par rapport aux quantités d'informations accédées plutôt que par rapport au nombres d'accès effectués.

Puisque cela ne change pas le principe général de notre approche, nous le proposons plutôt comme une extension de la solution précédemment proposée.

La prise en compte de la taille des attributs peut se faire de façon simple lors du calcul des accès inutiles. Si nous pondérons ces accès par la taille des attributs, nous obtenons en effet une quantité de données inutilement accédées.

En ce qui concerne les accès supplémentaires, leur pondération par la taille des attributs correspondants, reflète la quantité de données qu'il faudra "recoller" par unité de temps, et ce, à cause de la fragmentation verticale.Nous pouvons donc exprimer les nouvelles fonctions élémentaires par:

$$QUTE\_INUT(B\_SUP) = \sum_{i=1}^{k}\sum_{j=i}^{k} [\,(aff_{i,i} - aff_{i,j}) \times t(a_j) + (aff_{j,j} - aff_{i,j}) \times t(a_j)\,]$$

$$QUTE\_INUT(B\_INF = \sum_{i=k+1}^{n}\sum_{j=i}^{n} [\,(aff_{i,i} - aff_{i,j}) \times t(a_j) + (aff_{j,j} - aff_{i,j}) \times t(a_j)\,]$$

$$QUTE\_RECOL(B\_MIXTE) = \sum_{i=1}^{k}\sum_{j=k+1}^{n} aff_{i,j} \times (t(a_i) + t(a_j))$$

Remarque: $t(a_i)$ dénote la taille de l'attribut a_i.

Là encore, nous utiliserons une fonction qui prend en compte ces mesures afin de déterminer le découpage optimal d'un bloc B.

Nous n'irons pas plus loin dans la présentation de cette alternative qui n'a de réel intérêt que si l'on manipule des attributs de taille vraiment différente. A ce titre, la prise en compte de gros attributs (textes, images,...), tels que ceux requis par les nouvelles applications (cf chapitre 1), constitue une des utilisations potentielles de cette autre approche.

2.2.4.6. Quelques comparaisons avec la solution de Navathe [NAVA84]

Pour leur part, S. Navathe et S. Ceri proposent, afin de fragmenter binairement un bloc d'affinité, l'évaluation d'une fonction uniquement basée sur des considérations logiques liées à l'utilisation des données.

Ainsi pour tout découpage binaire d'un bloc d'affinité, ils calculent[1]:

- CU : nombre d'accès logiques réalisés par les requêtes dont tous les attributs cibles appartiennent au premier fragment induit, c'est-à-dire par les requêtes qui utilisent uniquement ce fragment,

- CL : idem mais par rapport au second fragment induit,

- CI : nombre d'accès logiques réalisés par les requêtes dont les attributs cibles sont répartis dans les deux fragments.

Ils cherchent alors à maximiser la fonction suivante:

$$z = CU * CL - CI^2$$

Nous allons dans ce qui suit faire un certain nombre de remarques sur cette méthode en tentant de la comparer avec celle que nous préconisons.

Tout d'abord, il est important de signaler que l'analyse du problème est menée de façon différente dans chacune des méthodes. La solution proposée dans [NAVA84] est basée sur une décomposition (à partir des deux fragments exhibés) de l'ensemble des requêtes en trois sous-ensembles et, sur l'***évaluation d'un nombre d'accès logiques*** pour chacun de ces sous-ensembles.

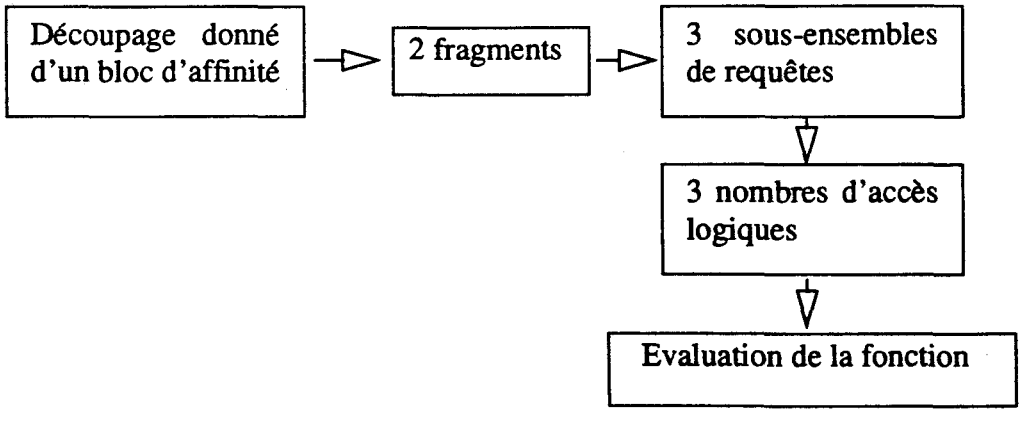


Fig. 2.40 : Principe de la fonction utilisée dans [NAVA84]

1. Nous utilisons ici les mêmes notations que dans [NAVA84].

Notre méthode repose sur l'exploitation directe du bloc d'affinité découpé. Elle ne se préoccupe donc pas de classer les requêtes mais s'intéresse à ***interpréter physiquement les coefficients d'affinité, i.e. les accès logiques simultanés***, par rapport aux fragments en question.

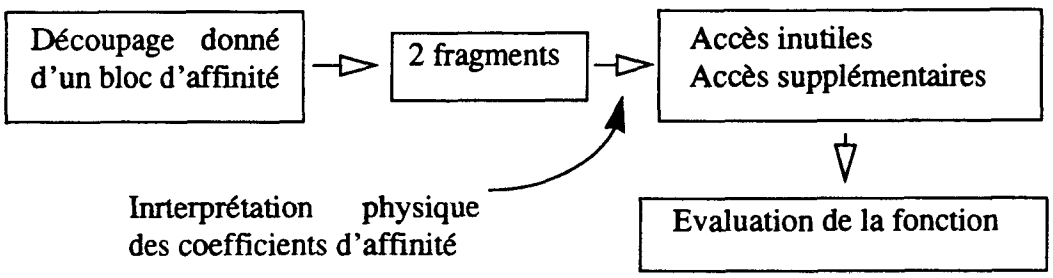


Fig. 2.41 : Principe de la fonction utilisée dans notre méthode

En conséquence, dans la méthode de [NAVA84], la matrice d'affinité ne sert que de support à l'établissement d'un certain ordre parmi les attributs (grâce à l'application du Bond Energy Algorithm). Cet ordre permet, pour chaque couple de fragments associé à un découpage binaire, de déterminer les sous-ensembles de requêtes induits.

En ce qui concerne l'évaluation des "sous-fonctions" CU, CL et CI, tout se fait "globalement" par rapport aux requêtes. L'information "riche", contenue dans les différents sous-blocs d'affinité, n'est en aucune façon prise en compte.

Le fait de considérer le problème de façon globale par rapport aux requêtes entraîne, il nous semble, une perte d'informations notable. En effet nous pouvons nous demander pourquoi les auteurs en reviennent à l'exploitation d'une information globale alors qu'ils disposent, au travers des coefficients d'affinité, d'une information plus précise?

La réponse se trouve peut-être dans le rôle relativement général qu'ils attribuent à leur opération de fragmentation verticale : "notion empirique de fragmentation où l'on suppose que la nécessité pour une transaction de visiter plusieurs fragments, devrait être minimisée". Dans le sens où ils ne s'attachent pas à l'interprétation physique des accès logiques considérés, leur solution peut être défendue, à condition toute fois de faire abstraction des quelques problèmes engendrés par la forme même de la fonction z, problèmes sur lesquels nous reviendrons dans ce qui suit.

Pour notre part, nous considérons la fragmentation verticale avant tout comme une phase de conception physique d'une base de données. En ce sens, nous jugeons qu'il est nécessaire de donner une certaine interprétation physique aux données logiques dont nous disposons (accès logiques ou accès logiques simultanés). C'est cette interprétation physique qui distingue clairement notre approche de celle de [NAVA84] en la faisant descendre "d'un cran de plus" vers le niveau physique.

Nous terminerons ces remarques par quelques considérations sur la nature même de la fonction utilisée dans [NAVA84] qui est: $z = CU * CL - CI^2$.

Dans le cas où l'une des deux valeurs CU ou CL est nulle, c'est-à-dire lorsqu'il existe aucune transaction qui accède uniquement à l'un des deux fragments, l'autre valeur n'a plus d'effet sur la fonction qui se réduit alors à $z = - CI^2$. En d'autres termes, tout une partie de l'analyse se trouve "effacée" à cause de la forme de la fonction utilisée. A ce sujet, la seule solution proposée par les auteurs est d'ignorer les découpages qui conduisent à des valeurs négatives de z.

Un autre reproche que nous adressons à cette fonction est son caractère figé. Même si elle répond d'une certaine façon à l'idée empirique que se font les auteurs de la fragmentation verticale, il nous semble dommage qu'elle ne comporte pas un certain aspect paramétrable. En fait, on peut distinguer dans cette fonction deux types d'accès logiques, ceux comptabilisés par CU et CL (qui sont de même nature) et ceux comptabilisés par CI. Une certaine latitude pour la prise en compte relative de ces accès différents serait certainement intéressante et permettrait d'adapter la fonction à différentes situations.

Enfin, nous conclurons sur cette fonction en discutant de la propriété d'équilibrage des fragments obtenus, que lui attribuent les auteurs. Ces derniers soulignent que les valeurs maximales de z seront obtenues (pour un CI constant) pour des valeurs semblables de CU et CL[1], ce qui leur fait conclure que les fragments produits seront équilibrés devant la charge, i.e. devant les accès qui s'y rapportent. Ils voient dans cette recherche d'équilibre, un des objectifs de la fragmentation verticale, point de vue que nous ne partageons pas vraiment.

A titre de remarque, ce sont les résultats surprenants obtenus sur notre exemple, qui nous ont amené à reconsidérer la méthode proposée par Navathe. Notre cas d'école entrait visiblement dans des cas non prévus et aboutissait à des phénomènes difficilement interprétables tels que l'obtention de valeurs négatives pour la fonction z, la production de fragments autres que ceux qui apparaissaient de toute évidence dans le bloc traité, l'absence de prise en compte de CU (respectivement CL) à cause d'une valeur nulle pour CL (resp. CU), etc.

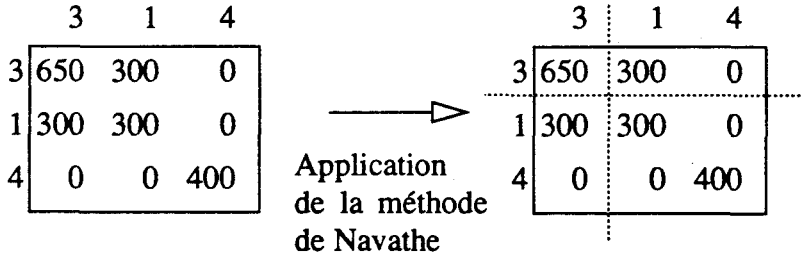


Fig. 2.42 : Exemple de résultat anormal obtenu par la méthode de Navathe

Sur le petit exemple présenté ci dessus, les fragments évidents que devraient fournir une méthode automatique sont f1={3,1} et f2={4}, puisque ces attributs ne sont, visiblement, jamais utilisés ensemble. Ce n'est pourtant pas ce qui ce passe avec la méthode de Navathe qui à cause de valeurs nulles pour CU ou CL "désintègre" tout une partie de l'analyse et fournit alors l'autre découpage comme solution.

---

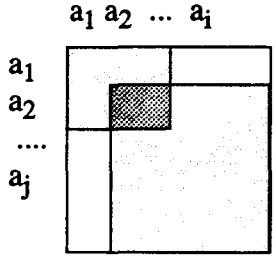
1. Il nous semble justifier, à ce niveau, d'émettre une petite "réserve mathématique" sur le bien fondé d'une telle affirmation.

Face à ces problèmes, notre approche a tout d'abord consisté à tenter de rectifier la méthode proposée en ajoutant certains garde-fous. Les résultats obtenus n'en furent pas meilleurs ce qui nous amena à remettre totalement en cause cette méthode et à essayer, dans un premier temps, de produire les fragments uniquement par une analyse mathématique de la matrice d'affinité rangée. Là encore, les nombreuses fonctions testées qui manipulaient moyennes, écarts type ou autre indices de dispersion des coefficients d'affinité, ne furent pas concluantes. De plus, il eut été présomptueux de leur attribuer une quelconque signification vis-à-vis du problème de fragmentation verticale qui nous intéressait.

Nous en sommes donc revenus à une analyse posée du problème qui, par une prise en compte des phénomènes induits par la fragmentation verticale, nous a permis de discerner les différents paramètres (accès inutiles, accès supplémentaires) qu'il est utile d'évaluer. Reste, comme nous l'avons souligné, le problème de la coordination de ces paramètres, problème qu'il est impossible de résoudre d'une façon générale. Ce problème est, avant tout, lié aux caractéristiques de la machine utilisée et il n'est pas question pour nous d'exhiber une quelconque règle de liaison entre accès inutiles et accès supplémentaires.

2.2.4.7. Extension vers des fragments recouvrants

Avec la méthode précédemment présentée, les fragments obtenus sont deux à deux disjoints. On peut envisager une extension qui permettrait le recouvrement de fragments. Dans ce cas, certains attributs-valeurs pourraient appartenir à deux fragments et seraient alors dupliqués pour être assignés aux deux objets physiques différents.



Les inconvénients liés à une telle fragmentation sont de trois types:
- accès inutiles dans B_SUP et B_INF
- accès supplémentaires dans B_MIXTE
- propagation des mises à jour des attributs dupliqués

Fig. 2.43 : Exemple de recouvrement de fragments

La fonction de détermination du découpage optimal devrait pour cela prendre en compte les trois mesures correspondantes.

Nous allons maintenant nous intéresser à la dernière étape de la première phase de fragmentation verticale, qui concerne l'optimisation éventuelle de la fragmentation obtenue.

2.2.5. Optimisation de la fragmentation obtenue

La méthode précédemment présentée est critiquable dans le sens où elle produit d'une façon "dichotomique" les différents fragments et, par conséquent, n'étudie pas toutes les

possibilités de découpage. Elle a cependant l'avantage d'être polynomiale contrairement à celle qui analyse toutes les solutions. Pour une matrice construite à partir de n attributs, n*(n+1)/2 fragmentations[1] au plus sont testées alors qu'il en existe 2^{n-1}.

Pour une classe d'objets nuplets qui comporte 25 attributs-valeurs, ce qui, en soi, n'est en rien irréalisable, nous étudions au plus 325 configurations alors qu'il en existe 17 millions. A raison d'un centième de seconde par configuration, il faudrait quelques 4h40 pour tout analyser.

Remarquons tout de même, que si le temps apparait comme peu important, les fonctions que nous proposons sont tout à fait utilisables pour une analyse de toutes les configurations possibles. Leur application se ferait, dans ce cas, non plus récursivement mais d'un seul jet sur chaque découpage potentiel.

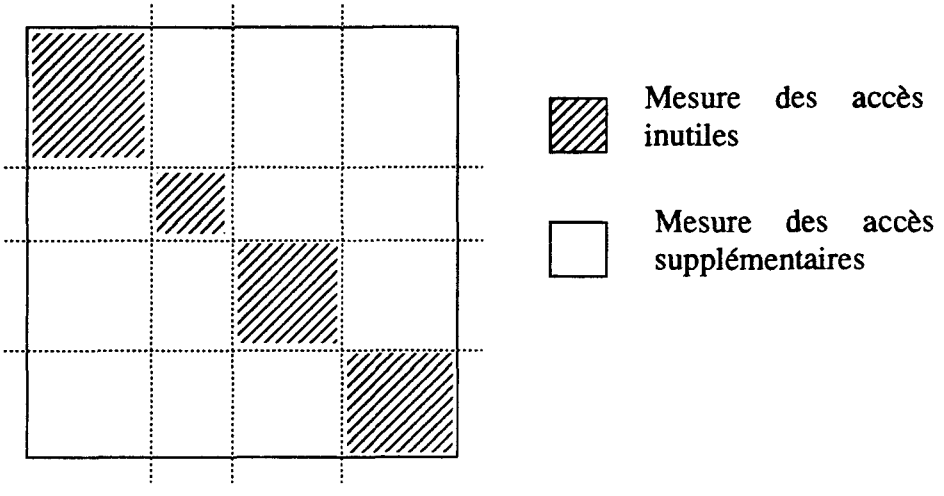


Fig. 2.44 : Applications des fonctions sur un découpage global

Nous proposons ci-dessous une optimisation de la fragmentation binaire récursive que nous avons proposée. Il n'est pas question dans cette étape de remettre tout en cause. Nous procédons ici à une optimisation que l'on peut qualifier de locale, par fusion éventuelle de fragments précédemment obtenus. Pour cela, nous partons de la constatation suivante.

A chaque itération, nous avons procédé à un éventuel découpage binaire des fragments obtenus à l'étape précédente, et ce, en tenant compte uniquement des coefficients d'affinité du bloc qui correspond à chaque fragment. De ce fait, nous avons obtenu des fragments adjacents (voisins dans la matrice d'affinité rangée) qui sont issus de calculs indépendants.

Sur l'exemple de la figure suivante, il apparait que certains fragments adjacents ont été produits indépendamment, c'est-à-dire que ce n'est pas l'étude du bloc d'affinité qui les caractérise, qui les a engendré. Bien sûr les coefficients d'affinité qui s'y rapportent ont, dans une itération antérieure du processus de fragmentation, été pris en compte. Il se peut cependant, que ces fragments fusionnés constituent une solution plus intéressante.

---

1. Cette borne supérieure est obtenue en retenant le cas le plus défavorable qui consiste à considérer qu'à chaque itération tout fragment de taille k est découpé en deux fragments, l'un de taille 1 et l'autre de taille k-1.

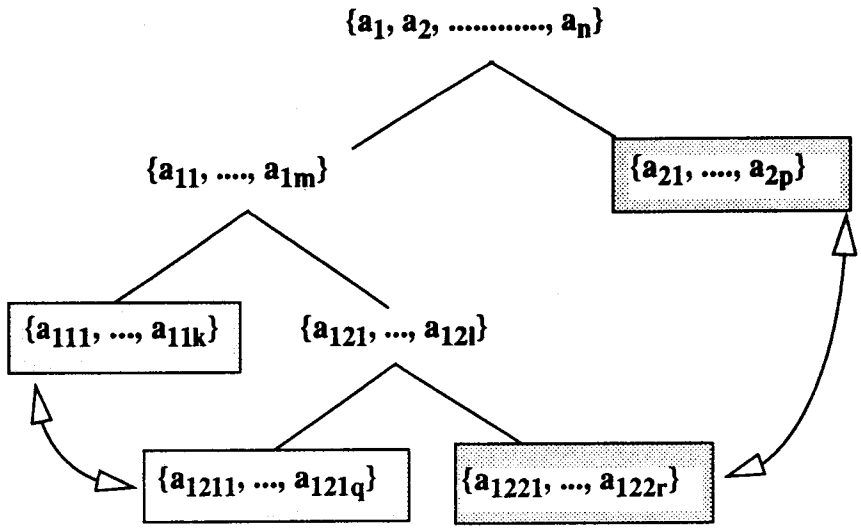


Fig. 2.45 : Exemple de fragments calculés indépendamment

L'optimisation locale que nous proposons consiste juste à tester si la fusion de tels fragments n'entraîne pas une meilleure solution. Nous nous limitons à cette rectification simple afin de ne pas retomber sur l'analyse d'un grand nombre de cas. Le fragmentation rectifiée n'est bien sûr pas forcément la solution globalement optimale.

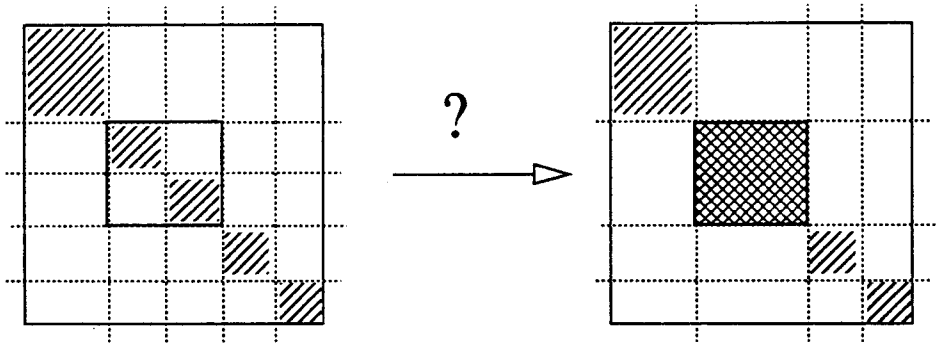


Fig. 2.46 : Principe de l'optimisation locale proposée

Cette étape d'optimisation locale ne modifie pas la fragmentation obtenue sur notre petit exemple. D'ailleurs il faut signaler, que d'une manière générale, le nombre de fragments obtenus dans les cas réel ne sera jamais très élevé, ce qui limitera la profondeur de l'arbre ainsi que le nombre de tels fragments adjacents. L'étape d'optimisation risque donc d'être pratiquement toujours sans effet.

A ce titre, il serait certainement intéressant de tenter d'établir certaines propriétés mathématiques entre la fragmentation initiale et l'éventuelle fragmentation localement optimisée. Cela nous permettrait de mettre en évidence certaines conditions limite relatives à la forme

de la matrice d'affinité. Nous ne développerons pas ici une telle analyse qui risquerait fort de nous emmener trop loin.

2.2.6. Conclusion sur la première phase de fragmentation verticale

Dans cette première phase nous avons donc proposé une solution pour fragmenter verticalement toute classe d'objets-nuplets restreinte à ces attributs valeurs atomiques. Nous obtenons, suite à cette phase, un ensemble de fragments normalisés pour chacune de ces classes. Il est important de signaler au passage que notre approche constitue une nouvelle solution tout à fait utilisable dans le contexte habituel du relationnel.

Revenons-en, néanmoins, à notre objectif qu'est la fragmentation verticale d'objets complexes dans un contexte parallèle. Nous allons donc maintenant nous intéresser à l'extension de la notion de fragmentation verticale afin qu'elle puisse s'adapter aux caractéristiques particulières du modèle de données que nous considérons.

Nous proposons donc, dans ce qui suit, une seconde phase afin de prendre en compte cette fois les attributs objets[1] propres à chaque classe d'objets nuplets.

2.3. Phase 2: Prise en compte des attributs objets

2.3.1. Présentation du problème

Avant d'entrer dans le vif du sujet, nous allons analyser de plus près les problèmes liés à la manipulation des objets complexes et voir à quel niveau la fragmentation verticale peut entrer en jeu.

2.3.1.1. Problèmes liés à la manipulation d'objets complexes

Le principal problème posé par l'utilisation d'un modèle de données qui supporte la notion d'objets complexes, est celui du passage de l'objet conceptuel à l'objet ou aux objets internes, qui seront les informations effectivement manipulées par le système. Rappelons, en effet, qu'avec un tel modèle un objet peut lui-même être composé d'autres objets que l'on appelle objets composants. Certaines applications (bureautique, CAO, etc) ont, rappelons-le, le réel besoin de pouvoir supporter efficacement ces objets complexes.

En fait, ce problème n'est autre que celui du stockage des objets complexes. Sa résolution n'est pas simple puisque nous sommes en présence de deux objectifs qui entrent en conflit. D'une part, nous devons assurer des accès performants à l'objet conceptuel dans son intégralité (composé + composants) et, d'autre part, nous devons aussi garantir de bonnes performances pour les accès "individuels" aux objets composants.

Face à ce problème, différents modèles de stockage des objets complexes ont été

---

1. Rappelons que nous aborderons aussi ici le cas des valeurs structurées que nous considérons comme un cas dérivé.

proposés [VALD86], [VALD87a]. Nous en donnons ci-dessous une représentation graphique.

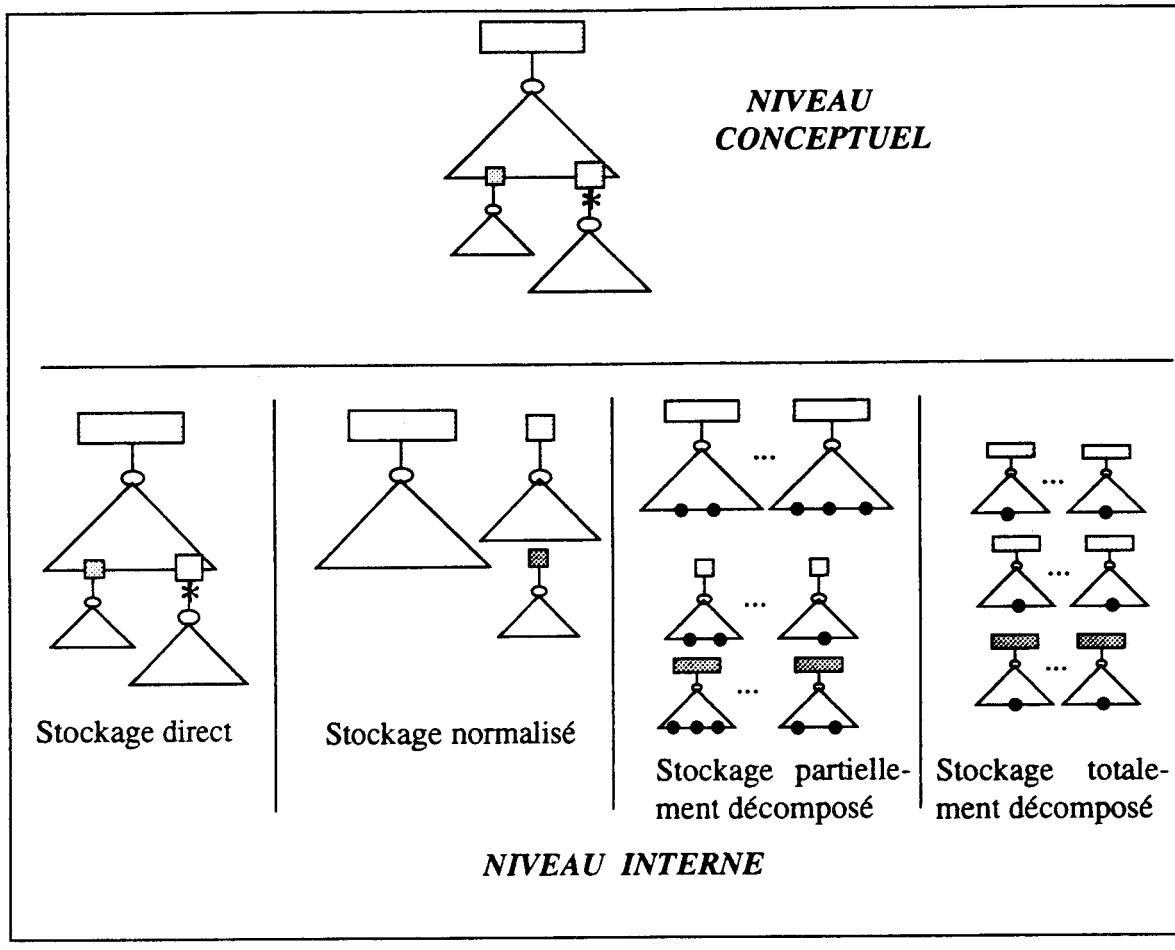


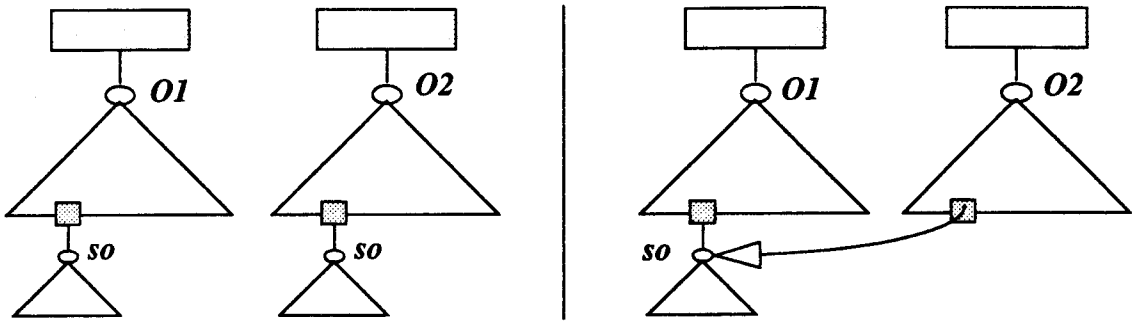
Fig. 2.47 : Différents modes de stockage des objets complexes

Considérons tout d'abord le stockage direct. Il consiste donc à stocker l'objet complexe tel qu'il est décrit au niveau conceptuel. On peut attribuer à ce mode de stockage deux avantages qui sont, d'une part, l'efficacité des accès à l'objet complet et, d'autre part, la simplicité induite au niveau de la compilation des requêtes ou programmes, puisqu'il existe alors une correspondance 1-1 entre l'objet conceptuel et l'objet interne. Au titre des désavantages, il faut bien sûr citer l'inaptitude du modèle à répondre efficacement aux accès "individuels" aux objets composants, ainsi que les problèmes physiques liés à la manipulation de gros objets (exemple: manipulation d'objets de taille supérieure à une page mémoire).

Si nous ajoutons à cela la composante parallèle et la notion de partage d'objets, nous pouvons surenchérir que le stockage direct inhibe toute possibilité de traitement parallèle des différentes parties d'un objet et que, de plus, il nécessite la duplication des objets composants partagés, ou le recours à une quelconque technique de pointeurs (cf figure 2.48).

Voyons maintenant le cas du stockage normalisé. Il consiste donc à éclater un objet complexe par rapport à ses objets composants. En d'autres termes, cela revient à obtenir au ni-

veau interne une relation normalisée pour chaque classe d'objets.



Duplication de l'objet composant partagé

Utilisation de pionteurs physiques pour le partage d'objets composants

Fig. 2.48 : Stockage direct et partage d'objets

Ce modèle de stockage normalisé favorise bien évidemment les accès individuels aux objets composants, mais nécessite par contre plusieurs opérations de jointure pour l'accès à un objet dans son intégralité et complique ainsi la compilation des requêtes. Le problème du partage d'objets est résolu puisqu'il suffit de maintenir l'information de liaison entre composés et composants, information qui ne nécessite pas de duplication de données. Enfin, dans un contexte parallèle, l'utilisation parallèle de différents composants est tout à fait envisageable.

Suivant cette progression qui consiste à affiner de plus en plus le modèle de stockage, nous trouvons ensuite le stockage partiellement décomposé qui consiste à découper chaque classe d'objets "normalisée" en fonction de l'utilisation des attributs et à stocker séparément chacun des fragments verticaux normalisés obtenus. Cela reviendrait donc, pour nous, à stocker séparément chacun des fragments obtenus lors de notre première phase de fragmentation verticale. Pour différentes raisons exposées dans ce qui suit, nous proposerons néanmoins une autre solution.

Enfin, dernier modèle de stockage recensé, le stockage entièrement décomposé (DSM) [COPE85] que nous avons présenté dans l'introduction de ce chapitre puisqu'il constitue une forme particulière de fragmentation verticale (cf travaux orientés modèle de stockage systématique). Nous ne reviendrons donc pas sur les tenants et les aboutissants d'une telle solution qui s'écarte d'ailleurs, de par son principe, de nos préoccupations.

En fait, ces différents modes de stockage n'ont rien de bien surprenant. Ce qui nous parait par contre étonnant, c'est que leur utilisation soit toujours préconisée en "tout ou rien". En d'autres termes, chaque modèle a ses adeptes qui en font alors une utilisation exclusive. Ainsi W. KIM [KIM88] préconise l'utilisation systématique du stockage direct pour lequel il donne une implantation particulière. Il considère que les applications CAO qu'il traite nécessitent "plus souvent" l'accès aux objets dans leur intégralité. A l'opposé Valduriez, Khoshafian et Copeland [COPE85] [VALD86] croient eux "dur comme fer" au stockage entièrement décomposé pour lequel ils proposent même des structures de données spécifiques (les index de jointures simples et hiérarchiques [VALD87b]) afin de limiter les inconvénients dus à l'éclatement des

données.

Force est de constater qu'il existe différentes "recettes" pour le stockage d'objets complexes, que chacune d'elles a un certain nombre d'avantages, mais que personne ne songe à les employer simultanément pour répondre au problème.

La solution que nous préconisons dans ce qui suit est une solution hybride qui cherche à tirer partie de chacune des solutions "classiques". Nous voulons ainsi étendre, au niveau des objets complexes, le principe général de la fragmentation verticale qui est de stocker les données au niveau interne, en fonction de leur utilisation.

2.3.1.2. Fragmentation verticale et objets complexes

Revenons-en à notre contexte. Suite à la phase 1 de la fragmentation verticale, nous disposons d'un ensemble de fragments normalisés. Comme nous l'avons signalé précédemment, nous pourrions nous contenter d'en rester là en préconisant l'utilisation du modèle de stockage partiellement décomposé.

Cependant, le recours à une telle méthode laisse entrevoir deux types de problèmes. Tout d'abord, qu'il s'agisse de cette méthode ou d'une autre, l'utilisation exclusive d'une solution particulière ne nous semble pas recommandable puisque l'on hérite alors au même titre des avantages et des inconvénients. De ce fait, il subsiste toujours des cas de figure pour lesquels l'unique solution retenue n'est pas adaptée. Deuxièmement, il faut bien voir qu'au travers des fragments normalisés de la phase 1, toute composante complexe des objets initiaux a disparu. Ne pas aller plus loin dans l'analyse, c'est-à-dire ne pas tenir compte de cette composante complexe pour le placement définitif des données, reviendrait donc à la considérer uniquement comme un artifice conceptuel qui n'aurait aucune répercussion au niveau interne.

Nous proposons donc, dans ce qui suit, une seconde phase de fragmentation verticale qui va consister à prendre en compte cette composante complexe ou, en d'autres termes, les attributs objets des classes d'objets nuplets. Cette prise en compte se fera par rapport aux fragments normalisés obtenus lors de la première phase. Nous pouvons dire de façon imagée que cette seconde phase va tenter de "recoller", en fonction de leur utilisation, les fragments normalisés issus de classes d'objets nuplets initialement liées dans le schéma logique.

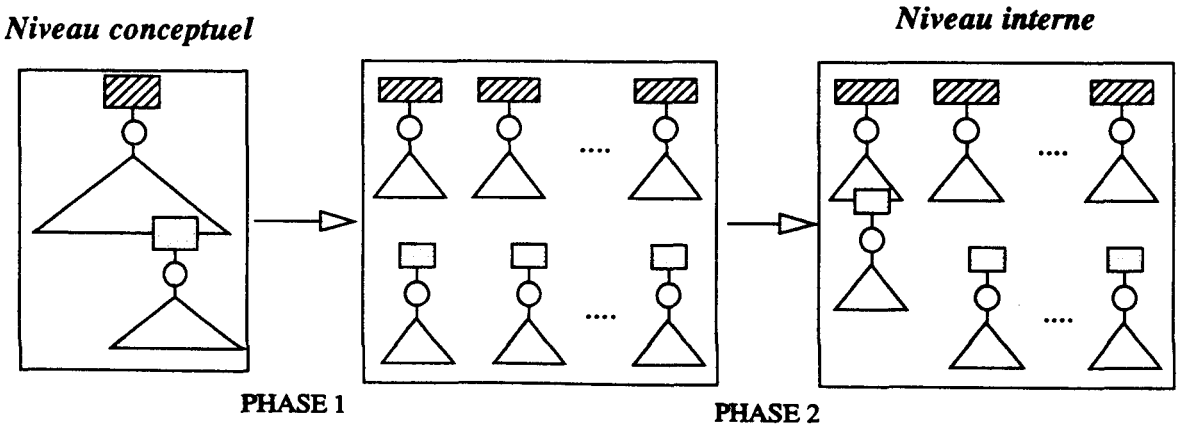


Fig. 2.49 : Principe de la seconde phase sur un exemple

Alors que la première phase se référait à l'utilisation simultanée des attributs valeurs atomiques, la seconde phase va, elle, prendre en compte l'utilisation simultanée de fragments normalisés issus de classes initialement liées. Ainsi nous proposons un prolongement de la notion de fragmentation verticale par rapport à la composante complexe du modèle de données.

Cette seconde phase nous fournira des fragments normalisés inchangés et des fragments complexes qui seront tous à l'image de l'utilisation qui en est faite. Nous obtiendrons donc, au niveau interne, des objets partiels normalisés et des objets partiels complexes; nous permettrons ainsi l'utilisation adaptée des deux types de stockage que sont:

-le stockage normalisé partiel

-le stockage direct partiel

2.3.1.3. Présentation de la suite de l'exposé

Dans ce qui suit, nous analyserons les deux principaux cas de figure que nous présentons ci-dessous à partir de notre exemple.

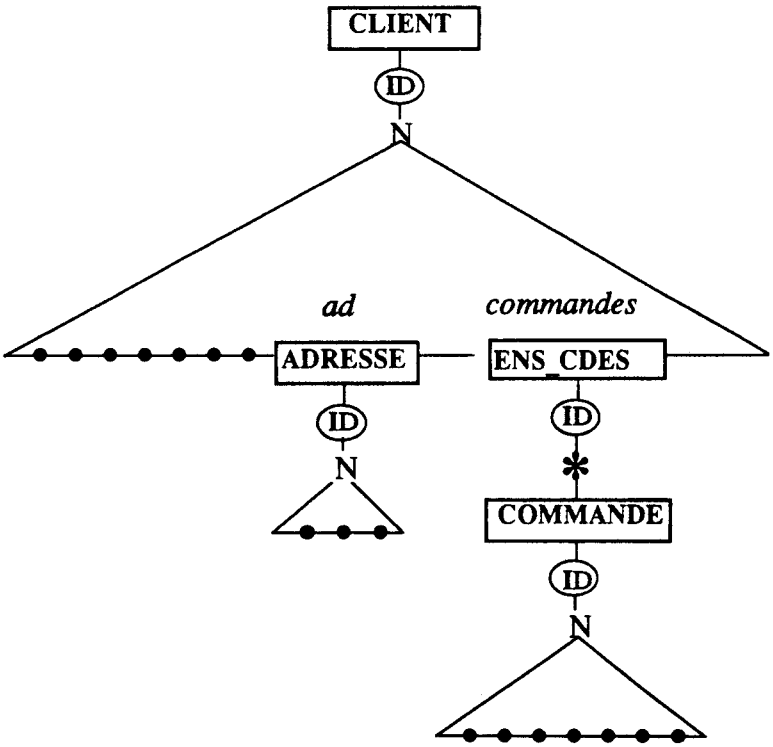


Fig. 2.50 : Représentation des cas étudiés

Dans le premier cas, nous considérerons que l'attribut objet étudié, est lui-même un objet nuplet, ce que nous pouvons désigner par attribut mono-objet. Dans l'exemple, l'attribut *ad* de la classe CLIENT correspond à ce cas de figure. Dans le second cas, nous nous intéresserons à un attribut qui est un objet ensemble d'objets nuplets, et donc, qui est indirectement un attribut multi-objet (cas de l'attribut *commandes* de notre exemple).

Nous consacrerons une section à chacun des cas et terminerons par une synthèse de cette phase et par quelques propositions d'extension que l'on peut envisager.

2.3.2. Premier cas : l'attribut objet est un objet nuplet

2.3.2.1. Point de vue général

Dans cette section, nous allons donc nous intéresser au cas d'un attribut objet qui est lui-même un objet nuplet[1]. Signalons au passage, que certains des modèles de données qui manipulent des données structurées, interdisent ce genre de construction (nuplet attribut d'un nuplet) et obligent l'alternance au niveau hiérarchique entre attributs-ensembles et attributs-nuplets [VERS86].

Le cas que nous étudions ici répond donc à la représentation graphique ci-dessous.:

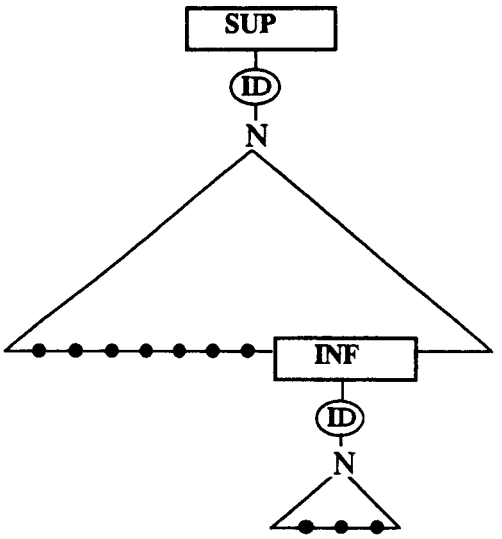


Fig. 2.51 : Représentation graphique du premier cas

Dans ce qui suit, nous appellerons donc ***SUP*** la classe des objets composés et ***INF*** celle des objets composants. Nous noterons ***SUP.fi*** et ***INF.fj*** les fragments normalisés des classes SUP et INF, obtenus par la phase 1 de fragmentation verticale. Enfin, les occurrences d'un fragment SUP.fi seront appelées ***objet partiels composés*** et celle d'un fragment INF.fj ***objets partiels composants***.

Il est important de voir, qu'au delà de ce lien hiérarchique qui existe entre les classes SUP et INF, bien d'autres phénomènes peuvent intervenir. La classe INF peut, par exemple, être partagée par d'autres classes; rien n'empêche en effet ses objets d'être composants (directement ou indirectement) d'objets d'autres classes. Elle peut elle-même utiliser une autre classe, ie ses objets peuvent avoir comme composants d'autres objets. Nous pouvons bien sûr en dire autant de la classe SUP.

En fait, de par les possibilités offertes par le modèle, ces classes peuvent être utilisées de différentes façons par les requêtes. En conséquence, l'étude relative de leurs fragments doit prendre en compte toutes ces possibilités.

Nous donnons ci-dessous une représentation graphique de toutes les situations que nous pouvons rencontrer autour de deux fragments normalisés donnés de SUP et INF. Sur cette

---

1. Nous nous permettons ici un "raccourci" abusif puisqu'il faudrait plutôt dire: "attribut objet dont l'état est lui même un objet nuplet".

figure un cadre représente une classe d'objets. Une ellipse représente un fragment normalisé d'une telle classe (lorsqu'il s'agit d'une classe d'objets nuplets). Un seul fragment est représenté pour les classes autres que SUP et INF afin de ne pas surcharger la figure. Un segment entre deux fragments représente les éventuelles requêtes qui utilisent ces deux fragments. A un tel segment,nous pouvons donc associer un nombre d'accès logiques réalisés par ces requêtes sur les objets partiels correspondants.

Nous utilisons aussi dans cette figure la notion de liaison entre classes. Nous dirons en fait qu'une classe C1 est ***liée*** à une classe C2 s'il existe un lien hiérarchique entre ces classes dans le schéma logique. Nous aurons alors C1 soit "supérieure", soit "inférieure" à C2. Enfin, rappelons que deux classes peuvent être utilisées simultanément sans pour autant être liées (cas des jointures explicites), ce qui nous amène aussi à faire figurer les classes non liées.

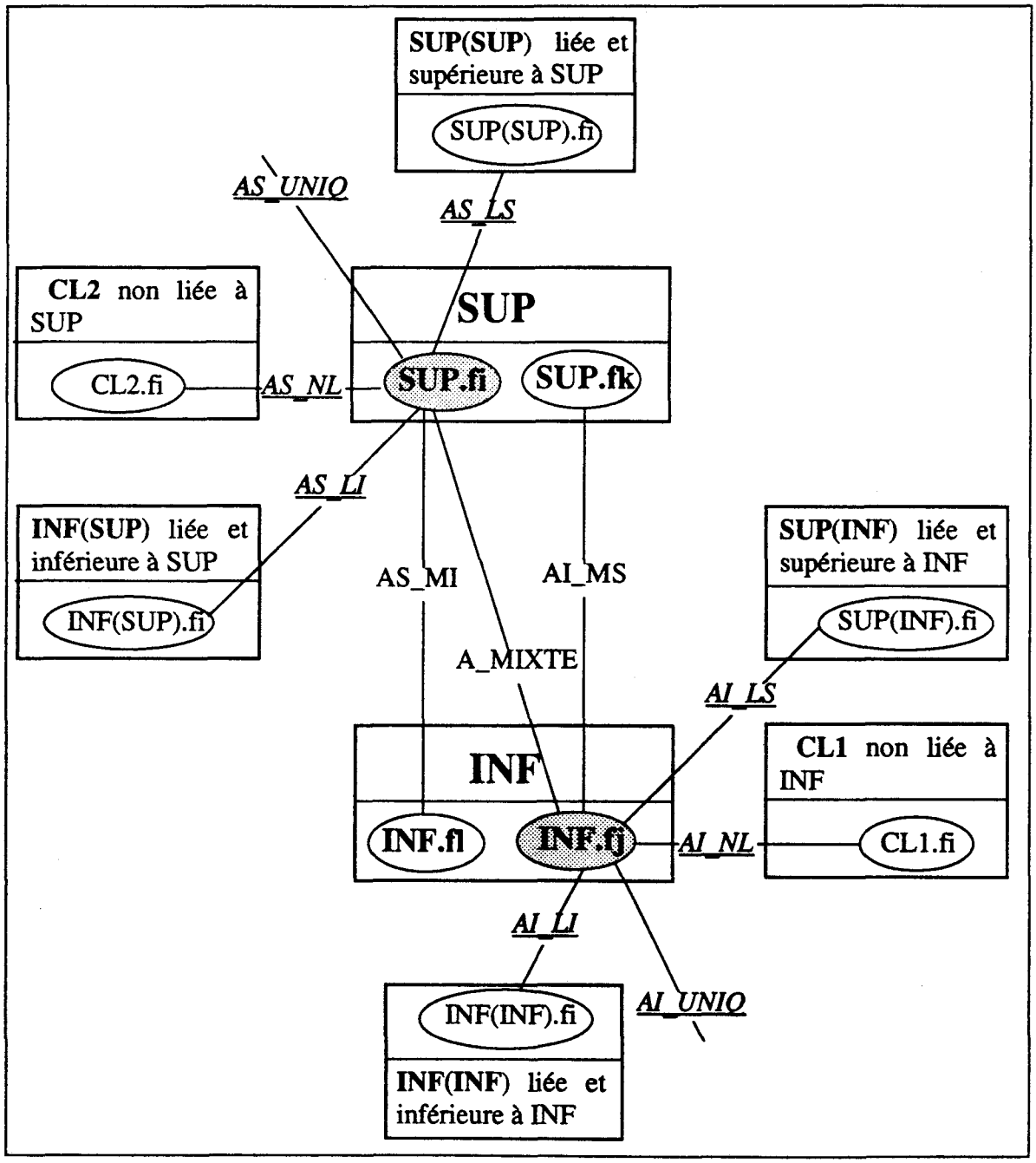


Fig. 2.52 : Différents cas "autour" de deux fragments donnés de SUP et INF

Remarque: Un lien hiérarchique entre deux classes peut être indirect par transitivité ou par utilisation du constructeur d'ensembles (cas du lien entre CLIENT et COMMANDE sur notre exemple).

Les différents nombres d'accès représentés sur la figure sont nommés à partir des règles suivantes:

-***A_MIXTE*** pour les requêtes qui utilisent simultanément SUP.fi et INF.fj

- pour les requêtes qui accèdent à SUP.fi (respectivement INF.fj) sans accéder à INF.fj (resp. SUP.fi) le nom commence par ***AS*** (resp. ***AI***). Il comprend ensuite:

. ***UNIQ*** si SUP.fi (resp. INF.fj) est le seul fragment accédé,

. ***NL*** pour les requêtes qui utilisent aussi un fragment d'une autre classe non liée à SUP (resp. INF),

. ***LI*** pour les requêtes qui utilisent aussi un fragment d'une autre classe liée inférieurement à SUP (resp. INF),

. ***LS*** pour les requêtes qui utilisent aussi un fragment d'une autre classe liée supérieurement à SUP (resp. INF),

. ***MI*** (resp. MS) pour les requêtes qui utilisent aussi un autre fragment de INF (resp. de SUP).

Quelques exemples de noms:

- AI_LI accès à INF.fj (AI) et à un fragment d'une classe liée inférieurement à INF (LI),
- AS_MI accès à SUP.fi (AS)et à un autre fragment de INF (MI = Même INF),
- AS_NL accès à SUP.fi (AS)et à un autre fragment d'une classe non liée à SUP (NL).

Nous allons donc étudier ici, pour tout couple de fragments (SUP.fi, INF.fj), leur éventuel rattachement. Cette étude va se faire par rapport à l'utilisation logique de ces fragments. Son but est le choix, pour ces fragments, du mode de stockage le plus adapté, à savoir le stockage normalisé partiel ou le stockage direct partiel.

En ce qui concerne le stockage direct partiel, nous ne considérerons que sa version redondante, c'est-à-dire avec duplication des objets partiels composants partagés. Le recours à un mode de stockage direct partiel non redondant nécessiterait au niveau interne l'utilisation d'une "technique de pointeurs". Un des objets partiels composés qui partagent l'objet partiel composant en question, serait alors favorisé, puisqu'il en serait "propriétaire" physiquement. Il en découlerait une non-homogénéité des traitements et surtout de nouveaux problèmes à résoudre tels que la suppression de l'objet partiel composé privilégié (propriétaire), le choix de cet objet père à privilégier, etc.

Bien sûr,l'utilisation du stockage direct partiel redondant suppose l'existence au niveau interne d'un mécanisme pour le maintien de l'intégrité des données dupliquées.

2.3.2.2. Cas particulier où l'attribut objet n'est pas partageable

Avant de passer au principe général de notre étude, nous allons aborder un cas particulier qui peut être traité de façon différente.

Considérons, en effet, le cas où un objet de INF ne peut être composant que d'un seul objet de SUP, ce qui revient à supposer, au niveau conceptuel, que le partage des objets de INF est impossible ou interdit (cas d'un lien 1:1 entre les classes SUP et INF). Notons au passage qu'un tel cas n'est pas conforme aux principes du modèle de données. En effet, dès que l'on utilise la notion d'objets pour décrire des données, ces dernières sont potentiellement partageables. Décrit dans les règles de l'art, ce cas ne devrait être autre que celui où les données de INF sont des valeurs nuplets, donc des données non identifiées et non partageables.

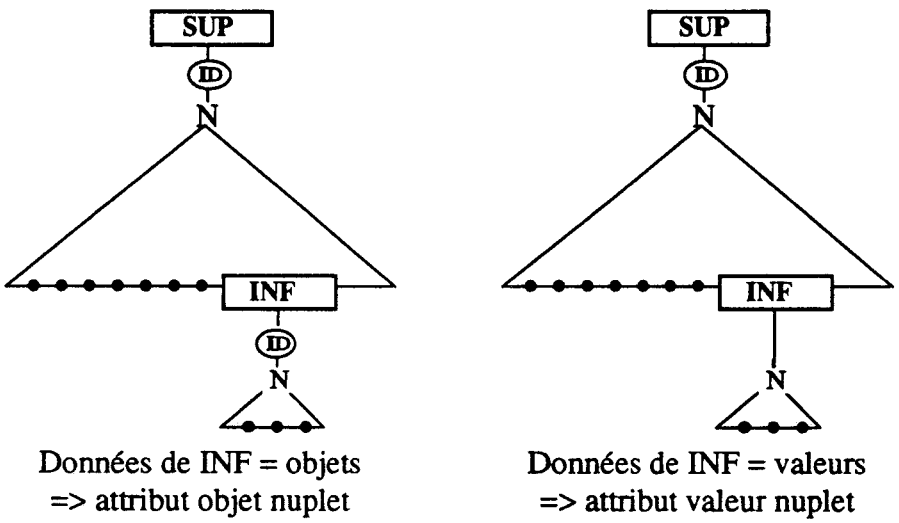


Fig. 2.53 : Les deux descriptions possibles des données de INF

Quoiqu'il en soit, ie que nous considérions des objets nuplets non partageables (ou tout simplement non partagés) ou des valeurs nuplets, la résolution de ce cas est la même. Dans ce cas, en effet, les attributs valeurs atomiques de INF jouent le même rôle que ceux de SUP, à l'indirection près induite par leur structuration en nuplet. Un tel cas peut donc être traité dans la phase 1 de la fragmentation verticale en calculant une matrice d'affinité commune aux deux classes d'objets. Les fragments "normalisés"[1] obtenus sont alors constitués d'attributs-valeurs en provenance des deux classes.

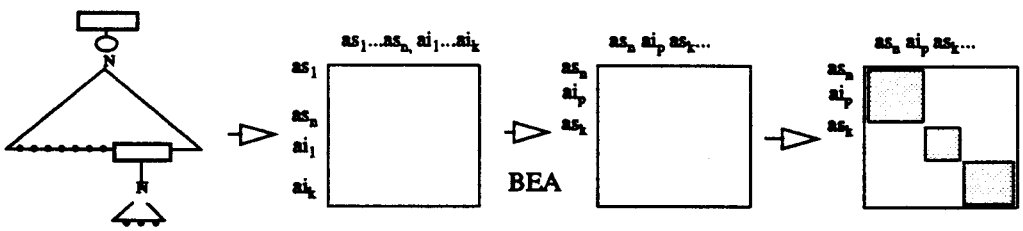


Fig. 2.54 : Cas particulier de fragmentation verticale

1. Le terme "normalisé" est bien sûr dans ce cas à interpréter différemment puisque les données obtenues peuvent avoir plusieurs "sources conceptuelles".

Nous n'en dirons pas plus sur ce cas et retiendrons juste que la phase 1 permet de répondre simplement au cas des attributs qui sont des valeurs nuplets.

Revenons-en maintenant au cas général, c'est-à-dire celui d'un attribut objet-nuplet partageable.

2.3.2.3. Principe de l'étude et évaluation des fonctions utilisées

Nous étudions donc ici le rattachement éventuel entre deux fragments SUP.fi et INF.fj issus de la première phase de fragmentation verticale. Pour cela, nous allons prendre en compte deux aspects. D'une part, nous évaluerons les inconvénients attachés à chacune des solutions que sont le stockage normalisé partiel et le stockage direct partiel des objets partiels concernés. D'autre part, au-delà de ces inconvénients, nous déterminerons un coefficient d'utilisation favorable du stockage direct partiel afin de prendre la décision finale.

. Evaluation des inconvénients attachés à chaque mode de stockage

Comme nous l'avons souligné précédemment, le stockage direct partiel redondant du fragment INF.fj "avec" le fragment SUP.fi nous conduit à devoir gérer les redondances des objets partiels composants partagés. L'efficacité d'une telle technique de stockage dépend bien évidemment du nombre de mises à jour à propager. Ces mises à jour constituent donc un des éléments clés à évaluer pour la prise de décision finale.

Notons ***PSOP(INF)*** le pourcentage d'objets de INF partagés par des objets de SUP (PSOP <=> Pourcentage de Sous-Objets Partagés). Pour le calcul de ce taux, un objet de INF est compté autant de fois qu'il est partagé.

Exemple: Supposons que nous ayons 10 objets conceptuels dans INF dont un est partagé par 3 objets conceptuels de SUP, un autre par deux. Nous aurons alors:

PSOP(INF)= (2+3) / (8+2+3) = 5 / 13 = 0.38

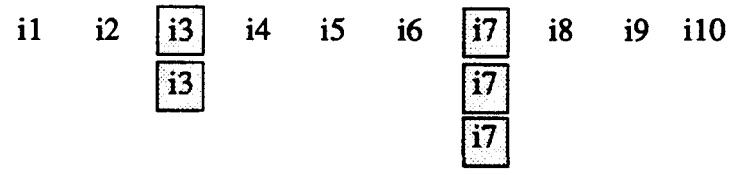


Fig. 2.55 : Principe du calcul d'un coefficient PSOP

Il est important de remarquer que cette valeur PSOP propre à la classe d'objets INF, se propage de la même façon au sein de chaque fragment normalisé de cette classe. En effet, elle constitue pour chacun de ces fragments, aussi le pourcentage d'objets partiels partagés.

En conséquence, nous pouvons évaluer pour chacun des fragments de INF ces mi-

ses à jour à propager par la fonction suivante:

$$MAJ(SUP.fi,INF.fj) = \sum_{k \text{ tq } R_k \text{ modifie INF.fj}} FREQ_k \times N_k(INF) \times PSOP(INF)$$

Puisque N_k(INF) représente le nombre d'accès logiques réalisés par une requête R_k sur la classe INF, l'expression $FREQ_k * N_k(INF) * PSOP(INF)$ dénote le nombre de mises à jour par unités de temps effectuées par cette requête sur des objets partiels de INF.fj partagés, si elle modifie l'un des attributs de INF.fj.

Si nous nous référons aux paramètres qui étaient utilisés pour la phase 1 (cf 2.2.2 Construction de la matrice d'affinité), ils nécessitent d'être précisés puisqu'il faut, en fait, connaître pour chaque requête le type d'accès qu'elle réalise, à savoir accès en écriture ou en lecture.

Relativement au schéma général donné en introduction (cf figure 2.50), les requêtes prises en compte pour cette fonction sont du type A_MIXTE ou AI_* (toutes les autres requêtes qui accèdent à INF.fj).

Passons maintenant aux inconvénients attachés au stockage normalisé partiel de SUP.fi et INF.fj. Un tel mode de stockage entraîne forcément des opérations de jointures internes[1] pour les requêtes qui accèdent simultanément à ces fragments. Il faut en effet, dans ce cas, "recoller les morceaux". Ces opérations de jointure se feront par rapport aux identifiants des objets partiels composants.

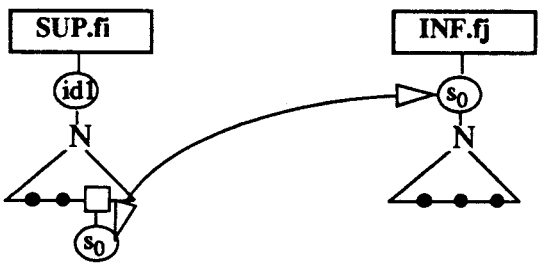


Fig. 2.56 : Jointure élémentaire interne en cas de stockage normalisé partiel

Il est donc important, si on envisage l'utilisation du stockage normalisé partiel, d'évaluer le nombre de ces "recollements" élémentaires qui, dans un contexte parallèle, influenceront les performances globales. Pour cela, nous utilisons la fonction suivante:

$$RECOL(SUP.fi,INF.fj) = \sum_{k \text{ tq } R_k \in A\_MIXTE} FREQ_k \times N_k(SUP)$$

Notons que A_MIXTE représente bien l'ensemble des requêtes qui accèdent simulta-

1. Nous parlons ici de jointures internes puisqu'il ne s'agit pas d'opérations explicitement exprimées par l'utilisateur, mais induites par le mode de stockage considéré.

nément les deux fragments. Si nous considérons de plus que A_MIXTE(SUP.fi,INF.fj) comptabilise aussi ces accès simultanés aux fragments, nous obtenons la relation suivante:

$$RECOL(SUP.fi,INF.fj) = A\_MIXTE(SUP.fi,INF.fj)$$

Nous pouvons aussi exprimer cette fonction sous une autre forme, relative cette fois à la classe INF, en pondérant avec le taux de partage des objets de cette classe.

$$RECOL\ (SUP.fi,INF.fj) = \sum_{k\ tq\ R_k \in A\_MIXTE} FREQ_k \times N_k\ (INF) \times (1 + PSOP\ (INF))$$

Au-delà de cette première catégorie d'inconvénients attachés à chaque mode de stockage, nous allons maintenant nous attacher à déterminer un coefficient d'utilisation favorable du stockage direct partiel.

. Evaluation d'un coefficient d'utilisation favorable du stockage direct partiel

Comme nous l'avons signalé précédemment, le stockage direct est favorable aux accès aux objets dans leur intégralité (composé + composants) alors que le stockage normalisé avantage, lui, les accès "isolés" aux parties d'objets. Ramenées à notre niveau, ces remarques sont toujours valables. En conséquence, afin de choisir le mode de stockage adapté, il faut disposer aussi d'une information quantitative relative à l'utilisation logique qui est faite des fragments. Nous définissons une telle information par la fonction suivante qui évalue un Coefficient de Liaison Simple[1] entre les fragments:

$$CLS(SUP.fi,INF.fj) = \frac{A\_MIXTE(SUP.fi,INF.fj)}{A\_MIXTE(SUP.fi,INF.fj) + A\_AUTRES(SUP.fi,INF.fj)}$$

A_MIXTE(SUP.fi,INF.fj) dénote bien sûr les accès simultanés aux fragments, A_AUTRES(SUP.fi,INF.fj) représente tous les autres accès aux fragments, ce que nous pouvons énoncer par la formule générique suivante:

A_AUTRES(SUP.fi,INF.fj)=AI_* (SUP.fi,INF.fj) + AS_*(SUP.fi,INF.Fj), l'étoile étant à prendre pour tous les cas possibles (cf figure 52).

Il est important de remarquer qu'en fonction de l'aspect général du schéma logique et des requêtes prises en compte, seuls certains de ces différents types d'accès entreront en jeu. Au plus le schéma sera complexe, au plus nous rencontrerons différents types d'accès supérieurs

1. Nous employons pour cette mesure le nom de Coefficient de Liaison Simple puisqu'elle est relative à un attribut "mono-objet" et donc à la liaison entre un objet partiel composé et un simple (≠ multiple) objet partiel composant.

(sur SUP.fi) et inférieurs (sur INF.fj) et au plus nous aurons de chances d'obtenir un faible coefficient de liaison simple.

Cette décomposition des accès aux fragments SUP.fi et INF.fj n'a d'autre but que de mettre en évidence les différentes influences envisageables. Pour le calcul effectif de A_AUTRES(SUP.fi, INF.fj), nous utiliserons les fonctions suivantes:

A_AUTRES(SUP.fi,iNF.fj) = A_INF(SUP.fi,INF.fj) + A_SUP(SUP.fi, INF.fj) avec:

$$A\_INF(SUP.fi,INF.fj) = \sum_{k \text{ tq } R_k \text{ accède uniquement à INF.fj}} FREQ_k \times N_k(INF)$$

$$A\_SUP(SUP.fi,INF.fj) = \sum_{k \text{ tq } R_k \text{ accède uniquement à SUP.fi}} FREQ_k \times N_k(SUP)$$

Etant donné que le stockage direct partiel favorise les accès mixtes (les accès simultanés à l'objet partiel composé et à l'objet partiel composant), nous pouvons conclure que ce coefficient de liaison simple exprime un taux d'utilisation favorable de ce type de stockage puisqu'il mesure la part relative des accès simultanés aux fragments.

Nous allons maintenant nous intéresser à la mise en oeuvre des fonctions proposées, c'est-à-dire à leur insertion dans un processus général de décision.

2.3.2.4. Mise en oeuvre des fonctions proposées

A ce niveau, nous retombons sur le problème de l'importance relative des différentes mesures effectuées. Bien que nous situions notre étude à un niveau purement logique, relatif à l'utilisation des données, nous ne pouvons nous permettre de rester à ce niveau pour déterminer certains paramètres manquants, attachés à ces mesures.

La détermination du poids à attribuer aux propagations de mises à jour face aux "recollements" élémentaires, ainsi que l'obtention du seuil souhaitable pour le coefficient de liaison simple, ie de la valeur au-delà de laquelle le stockage direct partiel est plus intéressant, nécessitent une connaissance de la machine cible, du modèle d'exécution retenu et des mécanismes internes utilisés (tels que celui pour le maintien de l'intégrité des données). Comme nous le verrons dans le dernier chapitre, relatif aux travaux futurs, nous pensons que de telles informations peuvent être issues d'un outil de simulation prenant en entrée un benchmark adapté.

Nous nous contenterons pour l'heure, comme pour la phase 1, de donner un algorithme général qui suppose la connaissance de ces informations manquantes, et qui n'a d'autre but que de donner une interprétation générale du rattachement éventuel des fragments de SUP et INF et ainsi de mettre en évidence les problèmes qui peuvent alors survenir.

Nous proposons donc de "traduire" le processus d'analyse du lien entre les classes SUP et INF de la façon suivante:

```
ANAL(SUP,INF)
    POUR TOUT fragment SUP.fi de SUP FAIRE
      POUR TOUT fragment INF.fj de INF FAIRE
        SI (POIDS_MAJ * MAJ(SUP.fi,INF.fj) < RECOL(SUP.fi,iNF.fj))
        ALORS /* les redondances ne constituent pas un sérieux handicap;
                  on peut envisager le stockage direct partiel de INF.fj et SUP.fi*/
          SI CLS(SUP.fi,INF.fj) > SEUIL_CLS
          ALORS /*forte utilisation simultanée des fragments
                    stockage direct partiel préconisé*/
            stockage(SUP.fi,INF.fj)=DIRECT_PARTIEL
          SINON /*faible utilisation simultanée des fragments,
                    stockage normalisé partiel préconisé*/
            stockage(SUP.fi,INF.fj) =NORMALISE_PARTIEL
          FSI
        SINON /* trop de mises a jour à propager*/
            stockage(SUP.fi, INF.fj)=NORMALISE_PARTIEL
        FSI
      FAIT
    FAIT
  FIN_ANAL
```

Nous pouvons donner une interprétation géométrique de ce processus appliqué à deux fragments donnés SUP.fi et INF.fj.

Pour cela, nous utiliserons une représentation en trois dimensions que sont les mises à jour à propager (MAJ(SUP.fi,INF.fj)), les accès mixtes (qui mesurent aussi, rappelons-le, les "recollements" élémentaires) et enfin les autres accès(A_AUTRES(SUP.fi,INF.fj)).

Chacune des deux conditions testées dans l'algorithme permet de déduire un plan dans cet espace.

Ainsi la condition POIDS_MAJ * MAJ(SUP.fi,INF.fj) < RECOL(SUP.fi,INF.fj) nous permet de représenter le plan P1 d'équation:

(P1): POIDS_MAJ * MAJ(SUP.fi,INF.fj) = RECOL(SUP.fi,INF.fj)

La seconde condition, CLS(SUP.fi,INF.fj) > SEUIL_CLS nous fournit un second plan P2 dont l'équation donnée ci-dessous est directement déduite de l'égalité : CLS(SUP.fi,INF.fj) = SEUIL_CLS

(P2):

(1-SEUIL_CLS) * A_MIXTE(SUP.fi,INF.fj) = SEUIL_CLS * A_AUTRES(SUP.fi,INF.fj)

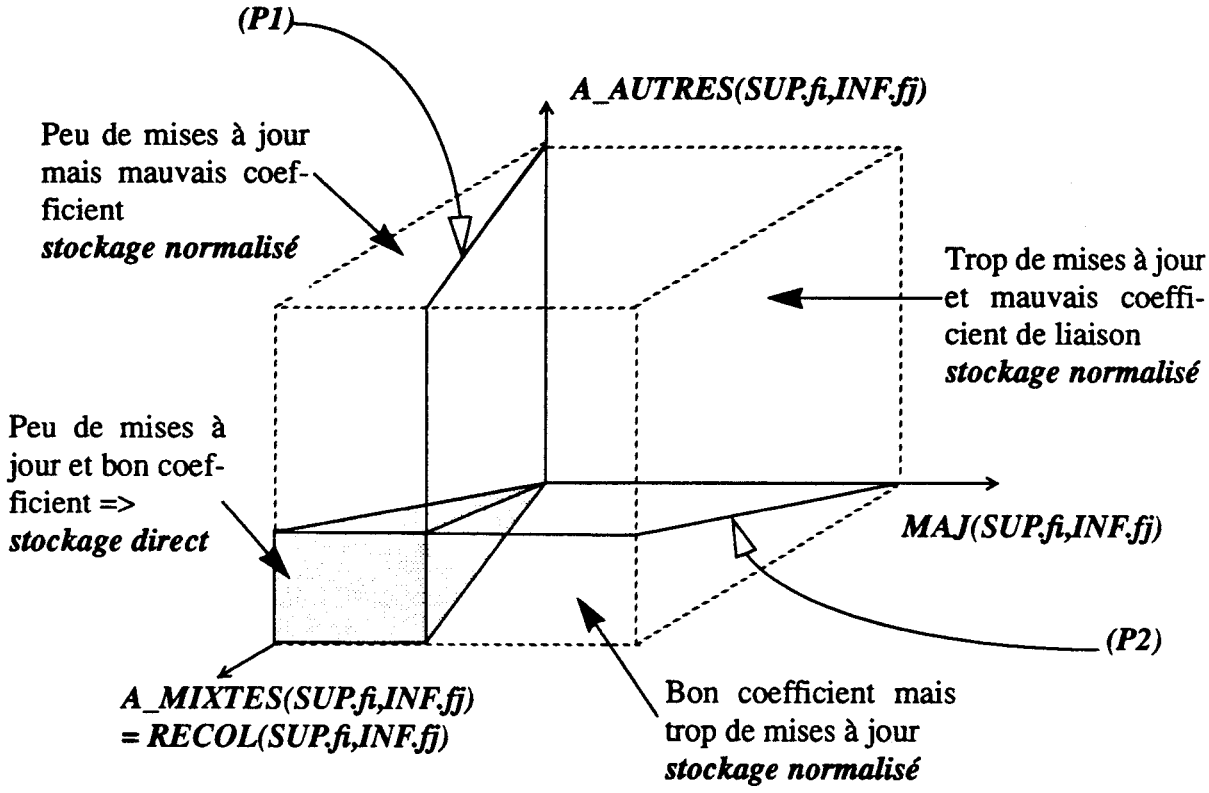


Fig. 2.57 : Interprétation géométrique de la méthode proposée

Ces deux plans délimitent quatre régions à chacune desquelles nous pouvons attribuer une interprétation différente (cf figure 2.57).

Nous avons volontairement attribué sur la figure une faible proportion au stockage direct. D'une part, on peut estimer sans prendre trop de risques que le poids des mises à jour sera plus important que celui des "recollements" élémentaires. En effet, les mécanismes à mettre en oeuvre pour la gestion des redondances et le maintien de l'intégrité des données, rendront la propagation d'une mise à jour plus coûteuse (surtout dans un contexte parallèle) qu'un "recollement" élémentaire. D'autre part, le SEUIL_CLS sera plus proche de 1 que de 0. Le stockage direct partiel nécessitera, en effet, des mécanismes d'accès secondaires pour éviter le passage obligé par les occurrences de SUP.fi lors des requêtes de type AI_*. De plus, il provoquera un certain nombre d'accès inutiles aux occurrences de INF.fj lors des requêtes AS_*. Hors, avec un SEUIL_CLS < 0,5 on pourrait choisir le stockage direct malgré l'influence majoritaire de requêtes de type A_AUTRES (AI_* ou AS_*).

2.3.2.5. Quelques résultats obtenus sur notre exemple

A partir de l'exemple utilisé dans la phase 1 et avec les paramètres suivants pour la classe ADRESSE nous obtenons les résultats présentés dans le tableau de la figure 2.59.

Att. / R_k	1	2	3	S_k	$FREQ_k$
0	1	1	1	0.4	1
1	1	1	1	0.5	0.14
2	0	0	0	0	0.03
3	0	0	0	0	0.03
4	1	1	1	0.001	100
5	0	0	0	0	0.03
6	1	1	1	0.02	0.03
7	1	1	1	1	0.03

Fig. 2.58 : Paramètres relatifs aux requêtes pour la classe ADRESSE

Ces paramètres conduisent à la construction d'un seul fragment (noté AD.f) lors de la première phase puisque les attributs sont toujours utilisés ensemble. Nous allons donc analyser les dépendances logiques entre chacun des trois fragments de CLIENT (cf figure 2.39) et le fragment de ADRESSE.

Si nous supposons que R_6 est la seule requête qui modifie le fragment ADRESSE.f, nous obtenons comme nombre de mises à jour à propager MAJ(AD.f)=0,003.

Pour chacun des couples de fragments de la forme (CLIENT.fi, AD.f), puisque ce nombre de mises à jour n'est pas prédominant, nous calculons le coefficient de liaison simple. Les résultats montrent que le fragment ADRESSE.f sera stocké directement avec CLIENT.f3 puisque l'on obtient un coefficient de liaison voisin de 1.

Couple / Fonction	CLIENT.f1-AD.f	CLIENT.f2-AD.f	CLINT.f3-AD.f
RECOL()	0.6	70.6	600.6
A_INF()	600	530	0
A_SUP()	1.53	0	1.53
CLS()	10^{-3}	0.12	0.997

Fig. 2.59 : Résultats quantitatifs liés à l'analyse de CLIENT et ADRESSE

Ainsi les objets de la classe ADRESSE (qui n'ont pas été fragmentés) seront stockés directement avec les objets partiels occurrence du troisième fragment de la classe client. Ce résultat aura été obtenu par une analyse de l'utilisation logique des fragments normalisés, ce qui constitue bien un prolongement de la notion habituelle de fragmentation verticale.

Avant de passer au second cas étudié, celui des attributs "multi-objets", nous allons, dans la section suivante, faire quelques remarques quant à l'influence de la seconde phase sur les résultats obtenus dans la première.

2.3.2.6. Influence de la seconde phase de fragmentation sur la première

Puisque l'étude que nous proposons se fait pour tout couple de la forme (SUP.fi, INF.fj), nous pouvons, à ce niveau, nous demander si nous ne risquons pas de prendre des décisions qui pourraient remettre en cause d'une façon ou d'une autre les fragments normalisés obtenus dans la phase 1.

A priori, deux situations paradoxales peuvent intervenir. D'une part, l'algorithme peut décider de rattacher un fragment de INF à deux fragments différents de SUP. Nous aboutirions, avec ce cas, à une remise en cause de la fragmentation de SUP puisqu'il nous faudrait réunir physiquement les deux fragments en question pour satisfaire les exigence de la phase 2.

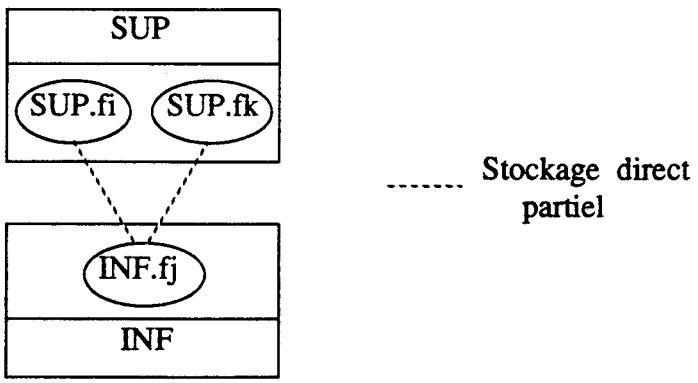


Fig. 2.60 : Remise en cause de la fragmentation de SUP

D'autre part, l'algorithme peut aussi décider de rattacher deux fragments différents de INF à un même fragment de SUP, ce qui reviendrait dans ce cas à remettre en cause la phase 1 de fragmentation de INF.

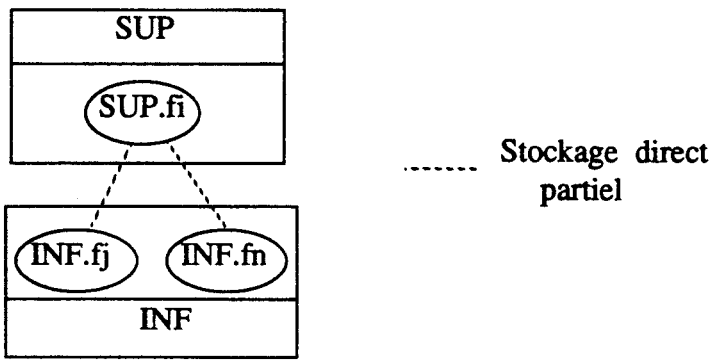


Fig. 2.61 : Remise en cause de la fragmentation de INF

Analysons de plus près l'éventuelle remise en cause de la fragmentation de SUP. Afin qu'un tel cas survienne, il faudrait que les conditions suivantes soient satisfaites:

CLS(SUP.fi , INF.fj) > SEUIL_CLS et CLS(SUP.fk , INF.fj) > SEUIL_CLS

Il faut bien voir cependant, que, de par leur définition, ces coefficients ne sont pas indépendants. Nous représentons ci-dessous cette dépendance issue de relations d'inclusion entre certains éléments des coefficients de liaison.

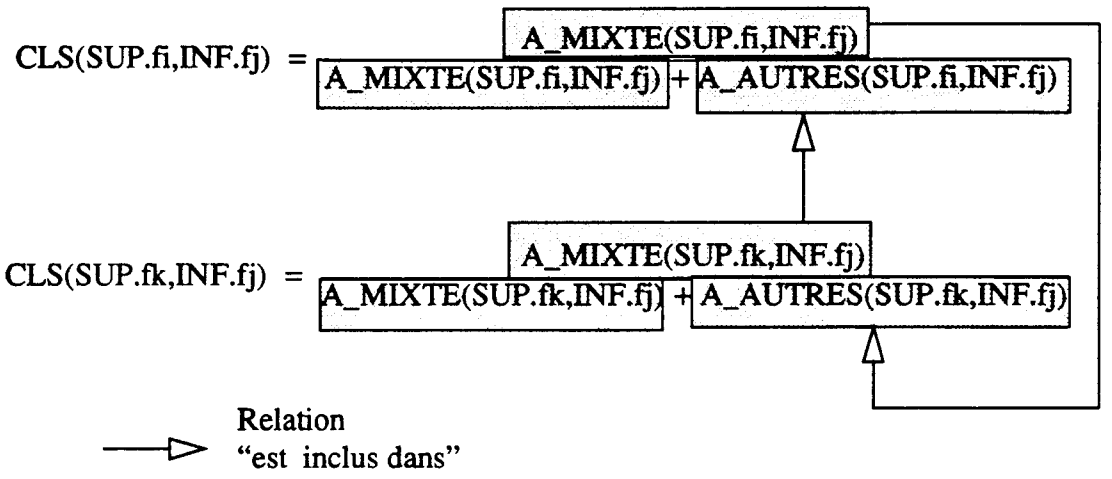


Fig. 2.62 : Dépendance entre coefficients de liaison simple

Pour simplifier la suite de l'exposé, nous appellerons CLS_i et CLS_k ces coefficients et x_i et x_k les accès mixtes qui s'y rapportent. Nous pouvons alors écrire:

$CLS_i = x_i / (x_i + x_k + C1)$ et $CLS_k = x_k / (x_k + x_i + C2)$

Il est assez simple de vérifier que le cas le plus favorable, c'est-à-dire celui qui donne simultanément les plus grandes valeurs possibles pour CLS_i et CLS_k, est obtenu pour des valeurs égales de x_i et x_k. En effet, dès que ces valeurs x_i et x_k diffèrent, l'un des deux coefficients en pâtit et risque alors de ne plus être supérieur au seuil.

Exemples:

. $x_i=2x_k$ et C1=C2=0 (prendre ces constantes égales à 0 va dans le sens d'une augmentation des coefficients) => $CLS_i=2/3$ et $CLS_k=1/3$

. $x_i=3x_k$ et C1=C2=0 => $CLS_i=3/4$ et $CLS_k=1/4$

Cependant, même pour ce cas le plus favorable ($x_i=x_k$, C1=C2=0), les coefficients obtenus ne peuvent excéder 0,5.

Il faudrait donc une valeur de seuil (SEUIL_CLS) inférieure à 0,5 pour pouvoir éventuellement remettre en cause la fragmentation de SUP. Si on se réfère aux nombreux inconvénients attachés au stockage direct partiel (mises à jour à propager, chemins d'accès obli-

gatoires, accès inutiles, etc), il semble peu concevable d'obtenir une telle valeur de seuil. Cela voudrait dire qu'une machine peut avoir de bonnes performances avec le stockage direct même dans le cas où les accès globaux (objet partiel composé + objets partiels composants) sont minoritaires.

Nous sommes une fois de plus confrontés au problème de la machine cible et de la connaissance d'informations qui s'y rapportent.

Notons tout de même, que si vraiment nous pouvions aboutir à ce type de situation (SEUIL_CLS < 0,5 pour une machine donnée), il serait alors nécessaire de prolonger l'étude pour faire un choix parmi les trois alternatives suivantes:

-remise en cause de la fragmentation de SUP, ie fusion de SUP.fi et SUP.fk,

-duplication de INF.fi dans chacun des fragments SUP.fi et SUP.fk,

-stockage normalisé en cas de conflit.

Le second cas, qui concerne l'éventuelle remise en cause de la fragmentation de INF, peut être analysé d'une façon tout à fait symétrique pour en arriver aux mêmes conclusions. Nous ne le détaillerons donc pas.

Il existe enfin un dernier cas qui ne constitue pas une quelconque remise en cause de la phase1, mais plutôt une éventualité de fusion entre fragments normalisés de classes différentes. En effet, la phase 2 de fragmentation verticale peut conclure au stockage direct partiel de INF.fj dans deux fragments de classes différentes et supérieures à INF.

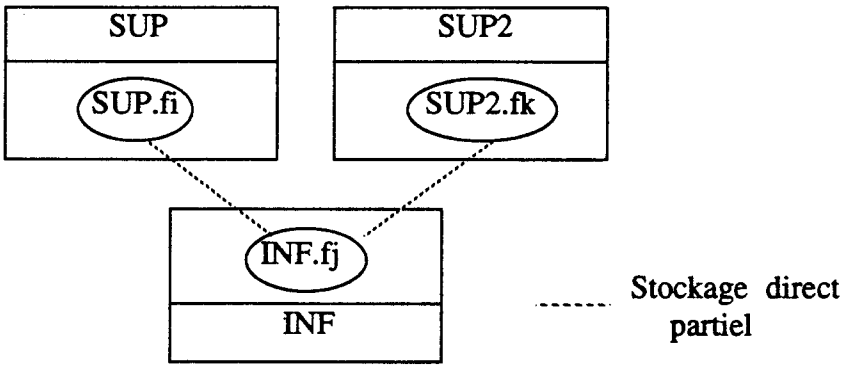


Fig. 2.63 : Stockage direct de INF.fj dans deux fragments de classes supérieures différentes

Toujours pour les mêmes raisons, un tel cas n'est envisageable que pour une valeur de SEUIL_CLS inférieure à 0,5. Il faut cependant signaler qu'il ne s'agit plus ici d'une remise en cause de la phase 1. Cela doit plutôt être vu comme une indication (information) de l'intérêt du regroupement des fragments supérieurs concernés.

Nous allons maintenant étudier le second cas qui concerne celui d'un attribut "multi-objets".

2.3.3. Second cas : l'attribut objet est un ensemble d'objets nuplets

2.3.3.1. Point de vue général

Le cas que nous allons étudier dans cette section peut être schématisé par la figure ci-dessous. Il s'agit donc de prendre en compte un attribut qui est un objet-ensemble composé d'objets nuplets, attribut que, par abus de langage, nous qualifierons de "multi-objets". Dans un tel cas, la seconde classe joue uniquement le rôle de constructeur d'ensembles. D'un point de vue fragmentation verticale, ie stockage des données, c'est le lien entre SUP et INF qui nous intéresse véritablement. Alors que dans le premier cas ce lien était du type n:1 (plusieurs objets de SUP pouvaient partager un objet de INF), il est cette fois de type n:m, puisqu' à un objet de SUP correspond plusieurs objets de INF. De plus, ces objets de INF peuvent être partagés de deux façons différentes par ceux de SUP.

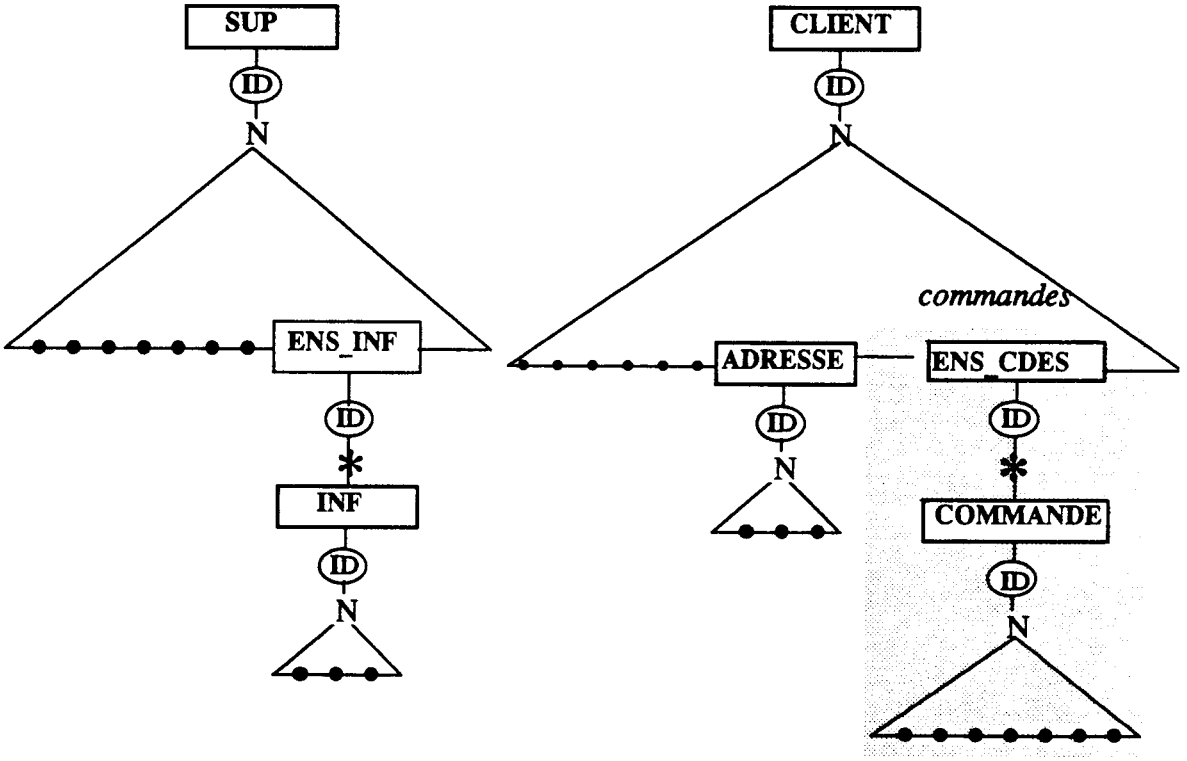


Fig. 2.64 : Représentation du second cas étudié, cas général et exemple

Si nous nous référons à notre exemple, il s'agit donc de prendre en compte l'attribut *commandes* des objets de la classe CLIENT, attribut qui n'est autre qu'un ensemble d'objets nuplets de la classe COMMANDE.

Nous pourrions attribuer à ce second cas le même genre de figure que pour le premier (cf figure 2.50). Il suffirait pour cela de rajouter la classe intermédiaire ENS_INF et les nouveaux types d'accès qui s'y rapportent. La figure obtenue serait quelque peu surchargée, nous ne la présenterons donc pas.

Nous allons donc analyser l'éventuel rattachement d'un fragment INF.fj à un frag-

ment SUP.fi. Il faut cependant bien voir que, dans ce second cas, le stockage direct partiel va consister à regrouper tous les objets partiels composants avec leur objet partiel composé; il s'agira donc d'un stockage direct multi-objets partiels. Nous pouvons représenter cela par la figure ci-dessous.

Le stockage normalisé consistera, lui, comme pour le premier cas, à regrouper toutes les occurrences d'un même fragment dans une même entité physique. Il faudra donc maintenir en plus, dans ce cas, l'information nécessaire à la reconstruction des ensembles d'objets partiels initiaux (couples de la forme: (Identifiant d'objet de INF, Identifiant d'objet de ENS_INF)).Nous donnons ci-dessous une représentation de cette situation au niveau du schéma logique découpé.

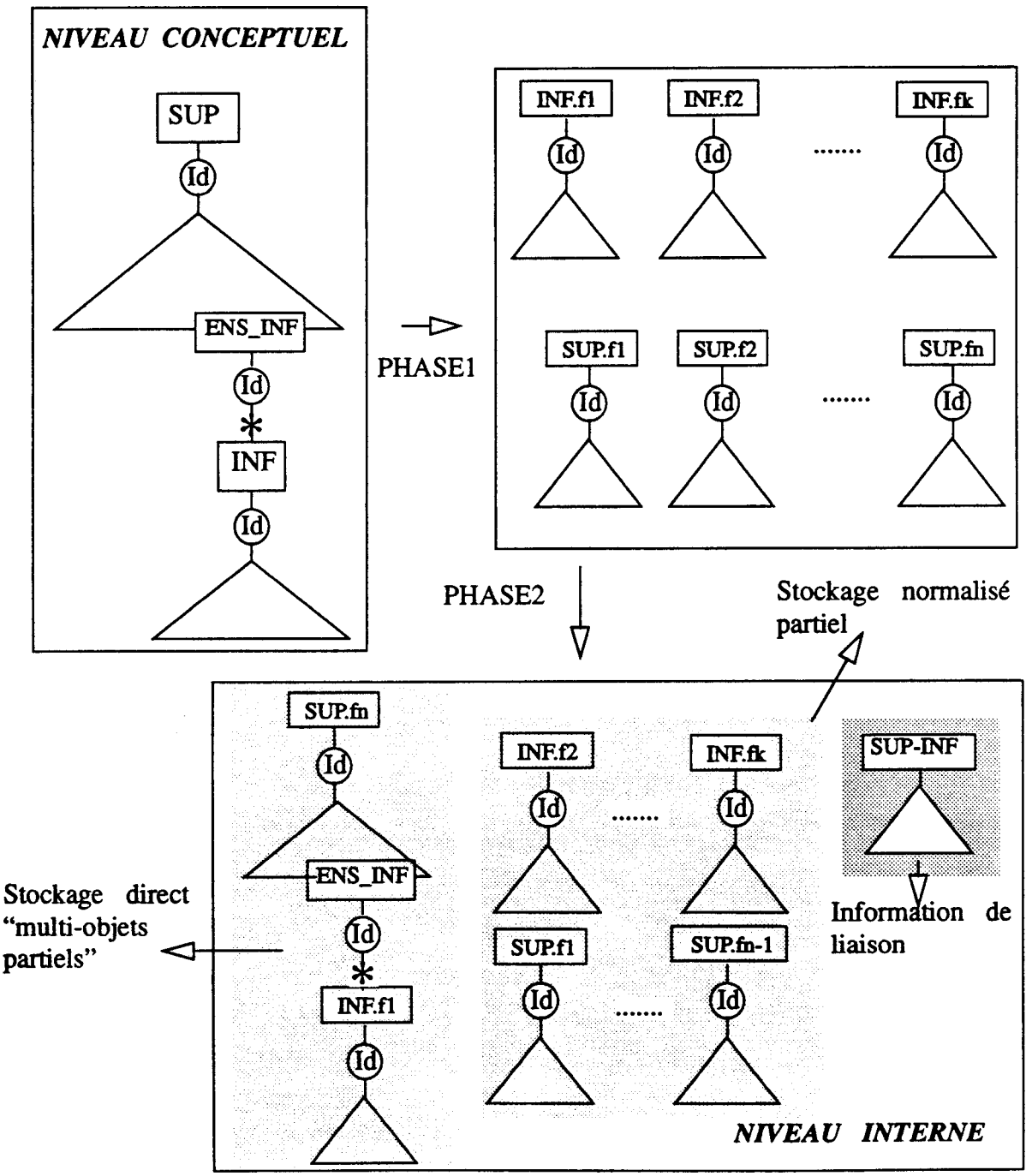


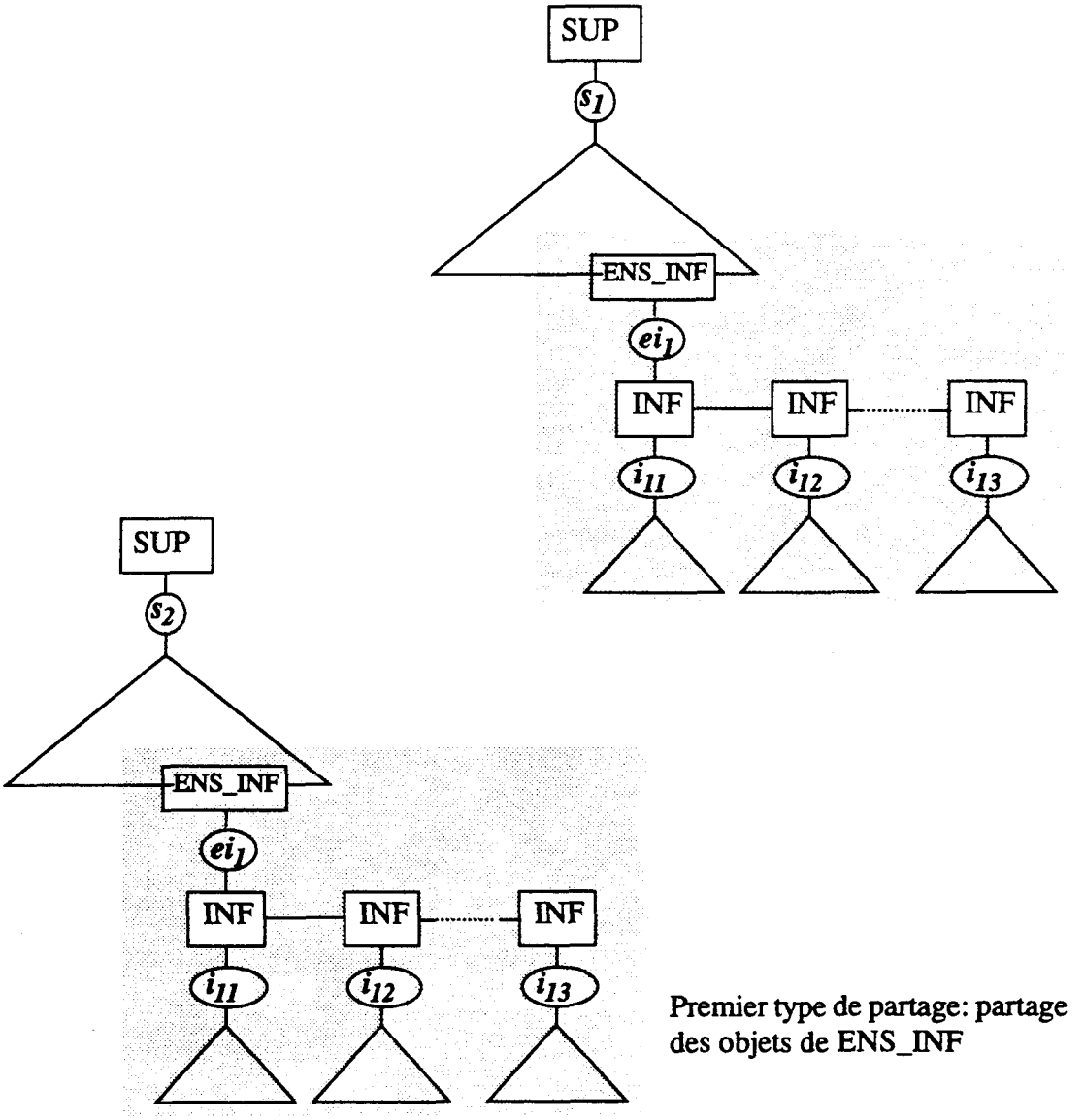
Fig. 2.65 : Phase 2 de la fragmentation verticale appliquée au second cas

Nous utiliserons, pour la présentation de ce second cas, la même démarche que pour le premier. Dans ce qui suit, nous mettrons donc tout d'abord en avant les fonctions utilisées. Nous discuterons ensuite de leur utilisation effective et des résultats obtenus sur notre exemple. Enfin, nous ferons comme précédemment le lien avec la première phase de fragmentation.

2.3.3.2. Evaluation des fonctions utilisées

Les fonctions que nous évaluons, pour ce second cas, sont symétriques à celles qui l'étaient pour le premier cas.

La première d'entre elles concerne donc les mises à jour à propager qui seraient induites par un stockage direct partiel de INF.fj avec SUP.fi. Il existe ici deux niveaux potentiels de partage entre SUP et INF. Un objet de INF peut en effet appartenir à un objet ensemble de ENS_INF partagé par deux objets de SUP, et être alors "transitivement" partagé par ces objets de SUP. Il peut aussi être partagé par deux objets-ensembles différents de ENS_INF relatifs à des objets différents de SUP. Nous donnons, dans la figure suivante, une représentation de ces deux types de partage au niveau des objets.



Premier type de partage: partage des objets de ENS_INF

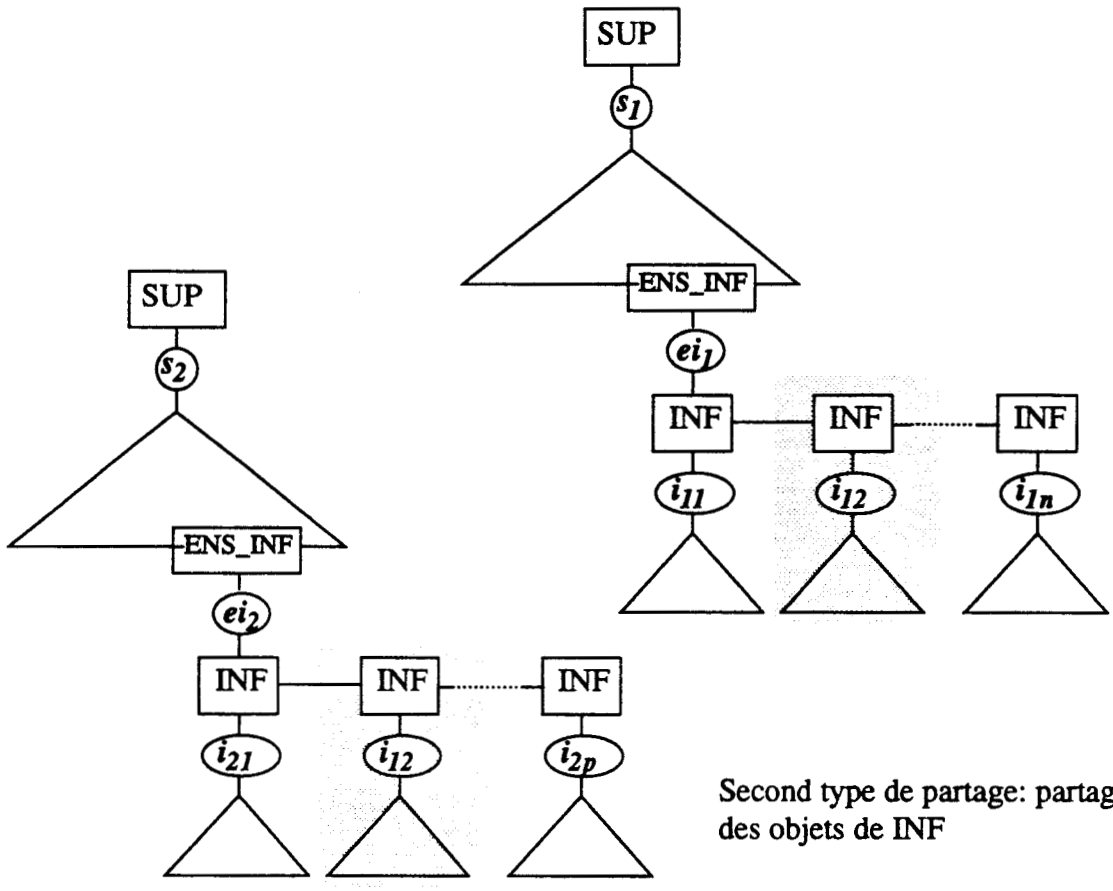


Fig. 2.66 : Les deux types de partage des objets de INF par des objets de SUP

Cette remarque qui concerne le double niveau de partage au niveau des objets, se répercute au niveau des objets partiels. En conséquence, l'évaluation de la fonction relative aux propagations de mises à jour doit en tenir compte. Ainsi la fonction utilisée s'exprime par:

$$MAJ(INF.fj) = \sum_{k\ tq\ R_k \text{modifie } INF.fj} FREQ_k \times N_k(INF) \times PSOP(INF)$$

$$+ \sum_{n\ tq\ R_n \text{modifie } ENS\_INF} FREQ_n \times N_n(ENS\_INF) \times PSOP(ENS\_INF)$$

Rappelons que les valeurs de PSOP dénotent le taux de partage des objets de la classe concernée. Le premier terme de cette fonction calcule donc le nombre de mises à jour des objets partiels de INF.fj à propager. Le second est lui relatif aux mises à jour des objets de ENS_INF, mises à jour induites par l'ajout ou la suppression dans un tel objet partagé d'un objet de INF.

Voyons maintenant les inconvénients attachés au stockage normalisé partiel de SUP.fi et INF.fj, qui ne sont autres que les "recollements" élémentaires à effectuer lors des requêtes qui accèdent simultanément les deux fragments. Ces "recollements" sont comptabilisés

par le nombre pondéré en fonction du taux de partage, d'objets partiels de INF.fj atteints par ces requêtes. La fonction utilisée est donc:

$$RECOL(SUP.fi,INF.fj) = \sum_{k\ tq\ R_k \in A\_MIXTE} FREQ_k \times N_k(INF) \times (1 + PSOP(INF))$$

Remarques:

- Le coefficient PSOP(INF) tient compte des deux types possibles de partage des objets de INF par des objets de SUP.

- Nous ne pouvons plus ici exprimer cette fonction à partir de N_k(SUP) puisque le nombre d'objets de INF conceptuellement attachés à ceux de SUP, est variable.

- Nous conservons pour ce second cas la propriété:

RECOL(SUP.fi,INF.fj) = A_MIXTE(SUP.fi,INF.fj)

Enfin, comme pour le premier cas, nous allons déterminer un coefficient d'utilisation favorable du stockage direct partiel.

Une première remarque pour signaler que le Coefficient de Liaison Simple est ici insuffisant. En effet, dans le cas d'un attribut multi-objets, le fait que les deux fragments SUP.fi et INF.fj soient fréquemment utilisés ensemble, ne nous permet pas de conclure que le stockage direct est la meilleure solution pour eux. Cette utilisation peut être mauvaise en ce qui concerne le nombre moyen d'objets partiels de INF.fj utiles dans chaque objet-ensemble de ENS_INF.

Exemple:

Supposons que nous ayons une majorité d'accès à SUP.fi et INF.fj de type A_MIXTE (accès simultanés) mais que seulement 10% des éléments des objet-ensembles de ENS_INF soient en moyenne concernés par ces accès. Nous aurons un Coefficient de Liaison Simple voisin de 1 (puisque majorité d'accès simultanés) et préconiserons le stockage partiel direct. Néanmoins il en résultera 90% de données inutilement accédées (puisque physiquement regroupées avec les 10% utiles) au niveau de chaque objet-ensemble de ENS_INF.

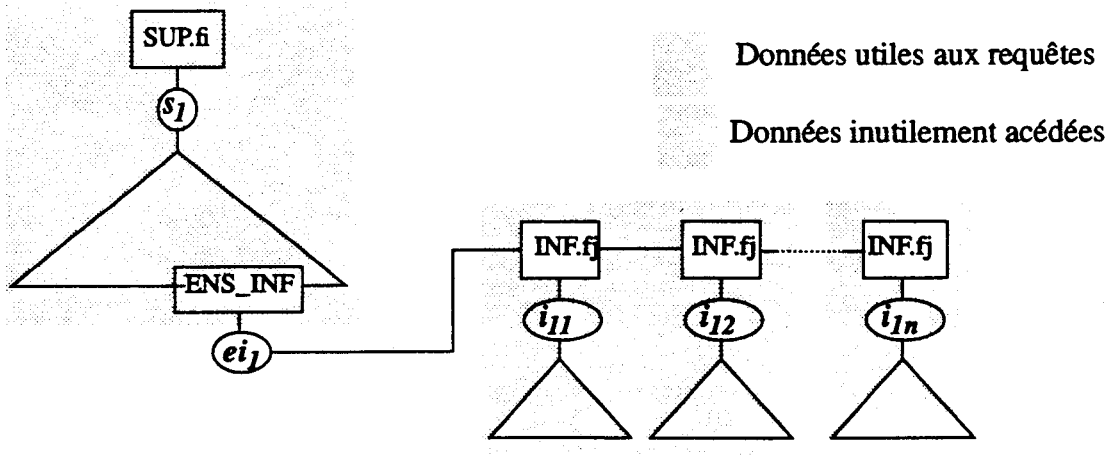
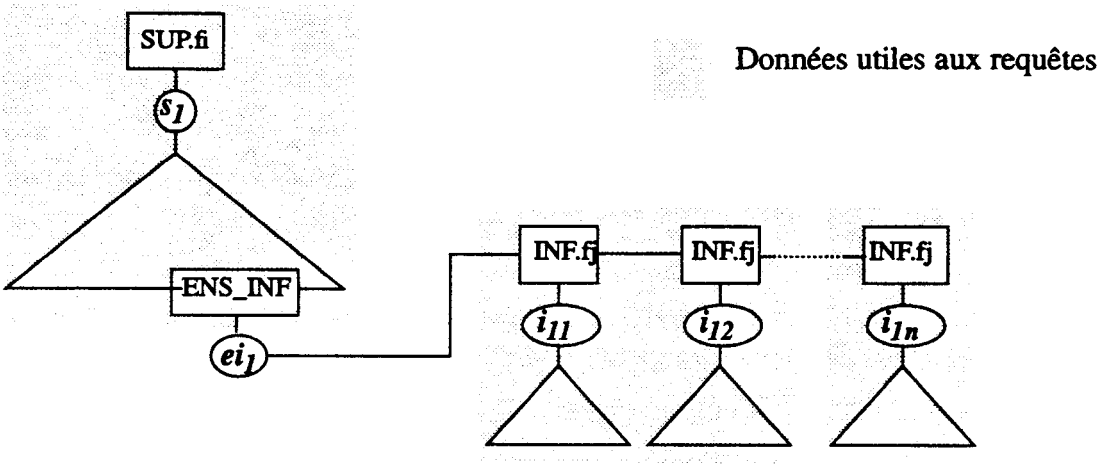


Fig. 2.67 : Stockage direct partiel et données inutilement accédées

En fait, il va de soi que le stockage direct multi-objets partiels n'a d'intérêt que si les requêtes utilisent de façon simultanée pour tout objet partiel composé de SUP.fi, une majorité de ses objets partiels composants de INF.fj.

Afin de prendre en compte cette remarque, nous introduisons un nombre imaginaire d'accès que nous appellons les ***accès simultanés favorables*** (A_FAV_MIXTE(SUP.fi,INF.fj)) et dont le calcul prend comme hypothèse que tous les objets partiels composants sont systématiquement utilisés avec leur objet partiel composé.



=> Aucune donnée inutilement accédée

Fig. 2.68 : Situation considérée pour le calcul des accès simultanés favorables

L'évaluation de ce nombre d'accès simultanés favorables se fera par:

$$A\_FAV\_MIXTE(SUP.fi,INF.fj) = \sum_{k\ tq\ R_k \in A\_MIXTE} FREQ_k \times N_k(SUP) \times \frac{CARD(INF)}{CARD(SUP)}$$

Le facteur CARD(INF) / CARD(SUP) représente la cardinalité moyenne des objets-ensembles de ENS_INF.

Grâce à cette fonction, nous pouvons évaluer un taux d'accès simultanés favorables par le rapport A_MIXTE(SUP.fi,INF.fj) / A_FAV_MIXTE(SUP.fi,INF.fj).

Nous exprimerons alors notre Coefficient de Liaison Multiple entre les fragments SUP.fi et INF.fj par:

$$CLM(SUP.fi,INF.fj) = \frac{A\_MIXTE(SUP.fi,INF.fj)}{A\_FAV\_MIXTE(SUP.fi,INF.fj)} \times CLS(SUP.fi,INF.fj)$$

Ce coefficient prend donc en compte deux choses, d'une part, la part respective de l'utilisation logique simultanée des fragments (facteur CLS() de la fonction), et, d'autre part, la validité de cette utilisation simultanée (rapport A_MIXTE() / A_FAV_ MIXTE()). Il représente

donc bien un coefficient d'utilisation favorable du stockage direct partiel dans le cas d'un attribut "multi-objets".

2.3.3.3. Mise en oeuvre des fonctions proposées

Comme pour le premier cas (cf 2.3.2.4), nous donnons ici un algorithme général qui suppose la connaissance des informations manquantes telles que le poids à attribuer à chaque type d'inconvénients, et le seuil propre aux Coefficient de Liaison Multiple. Ces informations dépendent, rappelons-le, de la machine cible.

Là encore, l'algorithme proposé n'a d'autre but que de donner une interprétation générale au rattachement éventuel des fragments de SUP et INF, ce que nous traduisons par:

```
ANAL(SUP,INF)
    POUR TOUT fragment SUP.fi de SUP FAIRE
      POUR TOUT fragment INF.fj de INF FAIRE
        SI (POIDS_MAJ * MAJ(SUP.fj,INF.fj) < RECOL(SUP.fi,iNF.fj))
        ALORS /* les redondances ne constituent pas un sérieux handicap;
                  on peut envisager le stockage direct partiel de INF.fj et SUP.fi*/
          SI CLM(SUP.fi,INF.fj) > SEUIL_CLM
          ALORS /*forte utilisation simultanée et favorable des fragments
                    stockage direct partiel préconisé*/
             stockage(SUP.fi,INF.fj)=DIRECT_PARTIEL
          SINON /*faible utilisation simultanée des fragments
                    ou (et) utilisation simultanée défavorable,
                    stockage normalisé partiel préconisé*/
             stockage(SUP.fi,INF.fj) =NORMALISE_PARTIEL
          FSI
        SINON /* trop de mises a jour à propager*/
           stockage(SUP.fi, INF.fj)=NORMALISE_PARTIEL
        FSI
      FAIT
    FAIT
  FIN_ANAL
```

Nous ne donnerons pas, pour ce second cas, une interprétation géométrique du processus d'analyse appliqué à deux fragments donnés. Il nous faudrait, pour cela,utiliser une quatrième dimension, celle des accès simultanés favorables. Tel que l'algorithme est présenté, le stockage direct serait géométriquement délimité, d'une part, par le plan correspondant aux mises à jour et aux recollements élémentaires, et, d'autre part, par la surface (hyperplan) relative à l'équation CLM(SUP.fi,INF.fj) = SEUIL_CLM, équation qui fait intervenir les trois dimen-

sions A_MIXTE, A_FAV_MIXTE et A_AUTRES.

Compte tenu de la contrainte supplémentaire attachée au Coefficient de Liaison Multiple (celle de l'utilisation favorable simultanée), la part respective du stockage sera moins importante que dans le premier cas. En d'autres termes, le rattachement d'un fragment INF.fj à un fragment SUP.fi sera plus rare dans le cas "multi-objets partiels" que dans le cas mono-objet partiel. Néanmoins, un tel rattachement restera toujours possible et nous évitera le recours systématique à un mode de stockage précis.

2.3.3.4. Résultats sur notre exemple

Appliqué à notre exemple (lien entre la classe CLIENT et la classe COMMANDE) et en considérant qu'il n'y a pas ou peu de partage des objets de COMMANDE par ceux de CLIENT, nous obtenons les résultats présentés ci-dessous.

Nous donnons dans ce tableau les résultats de l'évaluation de chacune des fonctions utiles et ce, pour chacun des couples de fragments de la forme (CLIENT.fi,COMMANDE.fj), couples que nous notons (CLT.fi,CDE.fj).

Remarque: R_FAV dénote le rapport A_MIXTE / A_FAV_MIXTE.

Fonction / Couple	RECOL	A_FAV_MIXTE	R_FAV	A_INF	A_SUP	CLS	CLM
CLT.f1-CDE.f1	30	60	0,5	6050	1,53	0,005	$2,5\ 10^{-3}$
CLT.f1-CDE.f2	0	0	-	650	2,13	0	0
CLT.f1-CDE.f3	0	0	-	400	2,13	0	0
CLT.f2-CDE.f1	380	7060	0,05	5700	0	0,06	$3\ 10^{-3}$
CLT.f2-CDE-f2	350	7000	0,05	300	0,6	0,53	$2,6\ 10^{-2}$
CLT.f2-CDE.f3	0	0	-	400	70,6	0	0
CLT.f3-CDE.f1	5780	57060	0,10	300	31,5	0,95	0,1
CLT.f3-CDE.f2	350	7000	0,05	300	532	0,30	$1,5\ 10^{-3}$
CLT.f3-CDE.f3	400	40000	0,01	0	200,8	0,66	$6,6\ 10^{-3}$

Fig. 2.69 : Résultats de la phase 2 appliquée à notre exemple

De ces résultats il découle qu'aucun des fragments de la classe COMMANDE ne sera stocké directement avec un des fragments de la classe CLIENT. Deux couples de fragments, (CLT.f3,CDE.f1) et (CLT.f3, CDE.f3), semblaient pourtant de bon candidats puisqu'ils correspondaient à des fragments fréquemment utilisés ensemble (CLS(CLT.f3,CDE.f1)=0,95 et CLS(CLT.f3,CDE.f3)=0,66).

Néanmoins, ces utilisations simultanées sont globalement défavorables puisqu'une fois normalisées par rapport à la notion d'accès simultanés favorables, nous obtenons de très

faibles Coefficient de Liaison Multiple (CLM(CLT.f3,CDE.f1)=0,1 et CLM (CLT.f3, CDE.f3)=6,6 10^{-3}).

Ce résultat unilatéral est bien sûr lié à la nature même de notre exemple. Nous avons attribué une cardinalité moyenne de 100 aux objets de ENS_CDE (ie 100 commandes par client). De ce fait, nous avions peu de chances d'aboutir au stockage direct d'un fragment de COMMANDE avec un fragment de CLIENT. Il eut fallu pour cela qu'il y ait une majorité de requêtes qui utilisent "les 100 commandes avec leur client", ce qui n'est pas le cas pour les requêtes choisies.

Remarquons au passage qu'il suffirait de changer quelques paramètres tels que les sélectivités absolues des requêtes (S_k) et la cardinalité de COMMANDE (ce qui impliquerait une modification de la cardinalité moyenne des objets de ENS_CDES) pour exhiber un cas de stockage direct multi-objets partiels. Nous ne considérerons pas ici de telles modifications dont l'intérêt se limiterait juste à la présentation d'un autre exemple.

En guise de conclusion, nous pouvons souligner, pour le cas général, que le stockage direct multi-objets partiels sera "logiquement" d'autant plus probable que la cardinalité moyenne des objets de ENS_INF sera faible.

Avant d'en terminer avec ce second cas de fragmentation verticale, nous allons, comme pour le premier cas, faire quelques rapprochements avec la phase1.

2.3.3.5. Influence de la seconde phase de fragmentation sur la première

Comme pour le premier cas, nous pouvons nous demander si le processus proposé ne risque pas de remettre en cause les résultats obtenus pour la phase 1.

Deux situations paradoxales peuvent en effet survenir, à savoir la remise en cause de la fragmentation de SUP ou de la fragmentation de INF. Contrairement à ce qui se passait dans le premier cas (cf 2.3.2.6), ces deux cas ne sont plus ici symétriques ;nous étudierons dans ce qui suit chacun d'entre eux.

Considérons tout d'abord la remise en cause de la fragmentation de SUP.

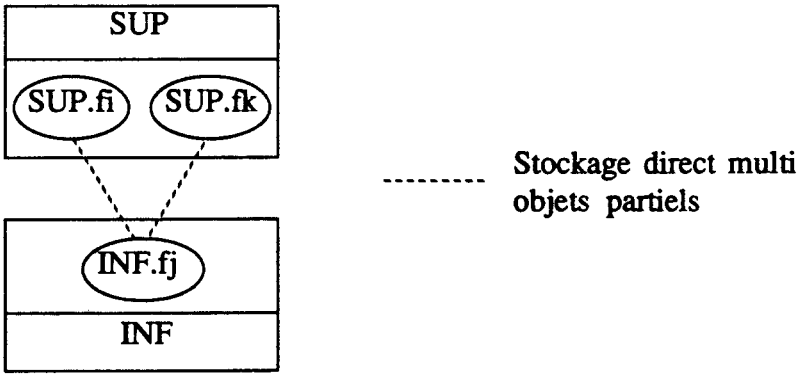


Fig. 2.70 : Remise en cause de la fragmentation de SUP

Il faudrait donc pour cela satisfaire les deux conditions suivantes:

CLM(SUP.fi,INF.fj) ≥ SEUIL_CLM et CLM(SUP.fk,INF.fj) ≥ SEUIL_CLM

Ces coefficients sont cependant liés de la façon suivante:

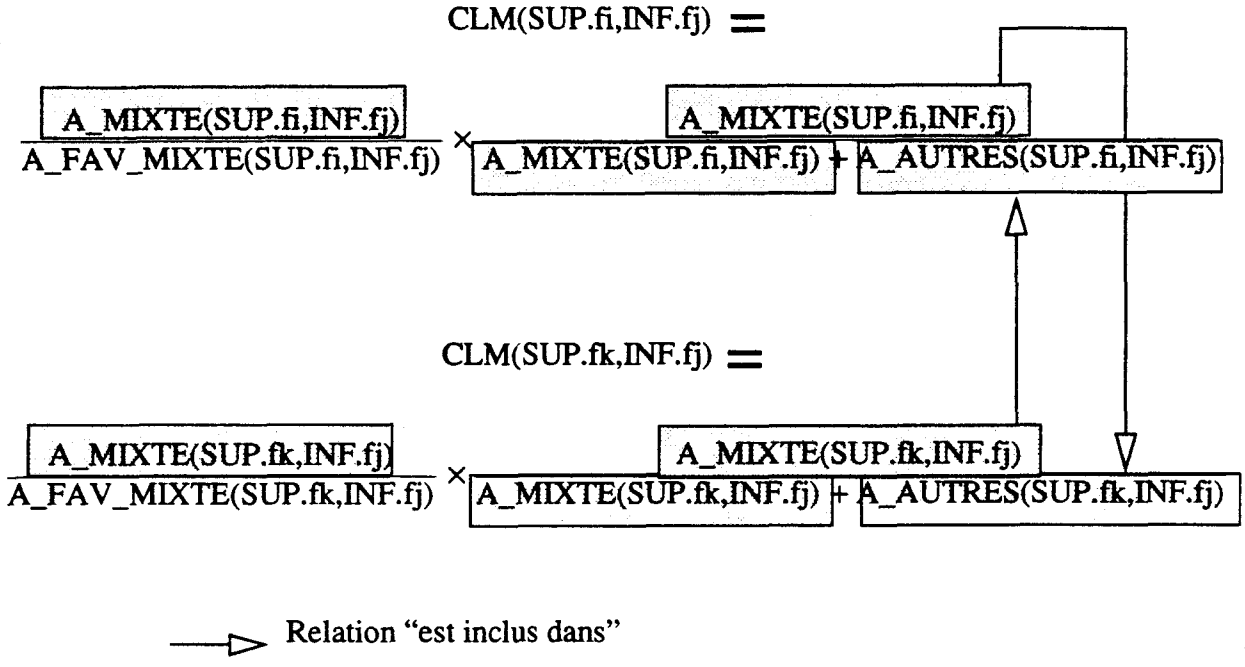


Fig. 2.71 : Dépendance entre Coefficients de Liaison Multiple dans le cas d'une remise en cause de la fragmentation de SUP

Pour simplifier la suite de l'exposé, nous appellerons CLM_i et CLM_k ces coefficients, x_i et x_k les accès mixtes et y_i et y_k les accès simultanés favorables qui s'y rapportent. Ainsi nous écrirons:

$$CLM_i = x_i^2 / y_i\,(x_i + x_k + C1) \text{ et } CLM_k = x_k^2 / y_k\,(x_k + x_i + C2)$$

Etant donné que $y_i \geq x_i$ et $y_k \geq x_k$, nous noterons $y_i = a_i x_i$ et $y_k = a_k x_k$, ce qui nous donnera:

$$CLM_i = \frac{1}{a_i} \times \frac{x_i}{x_i + x_k + C1} \qquad CLM_k = \frac{1}{a_k} \times \frac{x_k}{x_k + x_i + C2}$$

Comme pour les Coefficients de Liaison Simple, il est assez simple de vérifier que le cas le plus favorable, c'est-à-dire celui qui donne les plus grandes valeurs possibles pour CLM_i et CLM_k, est obtenu pour des valeurs égales de x_i et x_k et en considérant $a_i=a_k=1$ (tous les accès simultanés sont favorables) et C1=C2=0.

Une fois de plus, même pour ce cas le plus favorable qui suppose tout de même beaucoup de choses, les coefficients obtenus ne peuvent excéder 0,5.

Il faudrait donc une valeur de seuil (SEUIL_CLM) inférieure à 0,5 pour pouvoir remettre en cause la fragmentation de SUP. Un tel SEUIL qui semblait peu probable pour les Coef-

ficients de Liaison Simple, l'est encore moins pour les Coefficients de Liaison Multiple puisque nous avons rajouté la contrainte liée à l'aspect favorable des accès simultanés.

Voyons maintenant le cas de la remise en cause de la fragmentation de INF, cas où deux fragments de INF seraient directement stockés avec un même fragment de SUP.

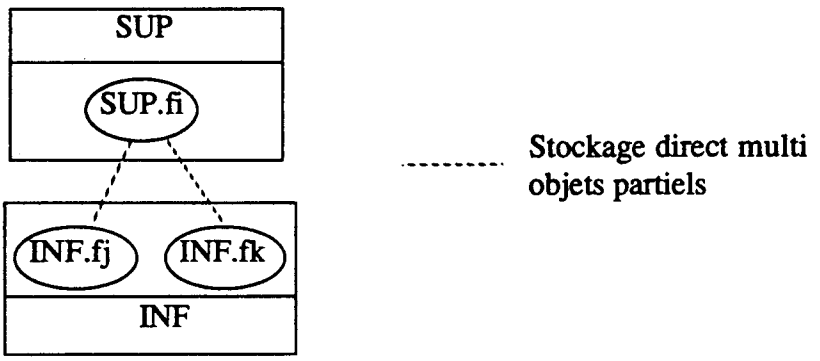


Fig. 2.72 : Remise en cause de la fragmentation de INF

Il faudrait pour cela avoir les conditions suivantes:

CLM(SUP.fi,INF.fj) ≥ SEUIL_CLM et CLM(SUP.fi,INF.fk) ≥ SEUIL_CLM

Cette fois encore, ces coefficients sont dépendants, mais leur dépendance est néanmoins moins forte. En effet, nous n'avons plus une relation d'inclusion entre les accès mixtes de l'un des coefficients et les autres accès (A_AUTRES) de l'autre.

Par exemple les accès mixtes A_MIXTE(SUP.fi,INF.fj) n'ont que leur composante "supérieure" prise en compte dans A_AUTRES(SUP.fi,INF.fk) (plus précisément dans A_SUP(SUP.fi,INF.fk)).

Précisons ce que nous entendons par composante supérieure de A_MIXTE (SUP.-fi,INF.fj). A ces accès mixtes, ie accès simultanés aux deux fragments, nous pouvons associer un certain nombre d'accès à SUP.fi, nombre qui correspond au nombre d'accès réalisés aux objets partiels de SUP.fi lors des accès simultanés considérés, et que nous noterons:

A_SUP_MIXTE(SUP.fi,INF.fj).

Ce nombre d'accès s'exprime par:

$$A\_SUP\_MIXTE(SUP.fi,INF.fj) = \sum_{k \text{ tq } R_k \in A\_MIXTE} FREQ_k \times N_k(SUP)$$

Puisque nous ne prenons en compte que la composante supérieure des accès simultanés, ie nous ignorons l'aspect multi objets partiels induit par INF.fj, la propriété suivante est vérifiée:

A_MIXTE(SUP.fi,INF.fj) ≥ A_SUP_MIXTE(SUP.fi,INF.fj)

Nous écrirons donc:

$A\_MIXTE(SUP.fi,INF.fj) = N_{ij} * A\_SUP\_MIXTE(SUP.fi,INF.fj)$

avec N_{ij}: nombre moyen d'objets partiels composants de INF.fj, atteints avec leur objet partiel composé de SUP.fi.

Exemple: Si nous avons 1000 accès simultanés aux fragments SUP.fi et INF.fj et si en moyenne 4 objets partiels de INF.fj sont atteints simultanément avec leur objet partiel composé, nous aurons alors 1000 / 4 = 250 accès effectifs aux objets partiels de SUP.fi, ce que nous pourrons traduire par:

A_MIXTE(SUP.fi,INF.fj) = 4 * A_SUP_MIXTE(SUP.fi,INF.fj)

Cette décomposition des accès inutiles par rapport à leur composante supérieure va nous permettre d'énoncer la règle de dépendance entre les deux Coefficients de Liaison Multiple. En effet, nous avons une relation d'inclusion entre A_SUP_MIXTE(SUP.fi,INF.fj) et A_AUTRES(SUP.fi,INF.fk) et vice versa.

Notations:

CLM_j = CLM(SUP.fi,INF.fj)

CLM_k = CLM(SUP.fi,INF.fk)

x_j = A_SUP_MIXTE(SUP.fi,INF.fj) => A_MIXTE(SUP.fi,INF.fj) = $N_{ij}\ x_j$

x_k = A_SUP_MIXTE(SUP.fi,INF.fk) => A_MIXTE(SUP.fi,INF.fk) = $N_{ik}\ x_k$

Remarque:

A_FAV_MIXTE(SUP.fi,INF.fj) = a_j A_MIXTE(SUP.fi,INF.fj) = $a_j\ N_{ij}\ x_j$

A_FAV_MIXTE(SUP.fi,INF.fk) = a_k A_MIXTE(SUP.fi,INF.fk) = $a_k\ N_{ik}\ x_k$

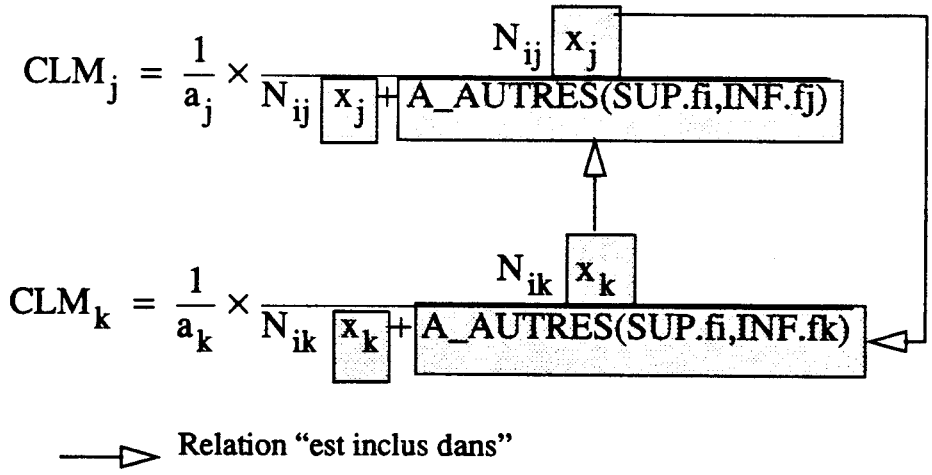


Fig. 2.73 : Dépendance entre les Coefficients de Liaison Multiple dans le cas d'une remise en cause de la fragmentation de INF

$$\Leftrightarrow \quad \boxed{CLM_j = \frac{1}{a_j} \times \frac{N_{ij}\ x_j}{N_{ij}\ x_j + x_k + C1}}$$

$$\text{et} \quad \boxed{CLM_k = \frac{1}{a_k} \times \frac{N_{ik}\ x_k}{N_{ik}\ x_k + x_j + C2}}$$

Le cas le plus favorable obtenu en prenant $a_j=a_k=1$ (tous les accès simultanés sont favorables), C1=C2=0 (il n'existe pas d'autres accès que ceux entre SUP.fi et INF.fj et ceux entre SUP.fi et INF.fj) et $x_i=x_k$ (pour ne pas défavoriser l'un des coefficients), nous donne:

$$CLM_j = \frac{N_{ij}}{N_{ij}+1} \qquad \text{et} \qquad CLM_k = \frac{N_{ik}}{N_{ik}+1}$$

Nous pouvons donc obtenir, théoriquement, des valeurs quelconques de Coefficients de Liaison Multiple et donc la fragmentation de INF peut être remise en cause.

Exemple:

Supposons que $N_{ij}=N_{ik}=100$, c'est-à-dire que les requêtes considérées pour chaque catégorie d'accès simultanés, accèdent en moyenne à 100 objets partiels composants de chaque objet partiel composé. Nous obtenons alors:

$CLM_j=CLM_k=100/101=0,99$, coefficients qui ont toutes les chances d'être supérieurs au SEUIL_CLM.

Remarquons tout de même que, pour en arriver là, bon nombre d'hypothèses draconiennes ont été retenues (uniquement des accès simultanés favorables, aucun autre type d'accès, coefficients N_{ij} et N_{ik} importants).

Ce que nous venons de faire ne constitue qu'une partie de l'étude théorique. Il ne faut pas oublier que si les fragments INF.fj et INF.fk existent, c'est parce que la matrice d'affinité répondait à certaines caractéristiques. En fait, il faudrait pousser l'étude plus loin, en prenant en compte le processus de découpage de cette matrice d'affinité, pour tenter d'établir certaines propriétés en ce qui concerne les limites de la remise en cause de la fragmentation de INF. Nous n'aborderons pas ici une telle étude.

Avant d'en terminer avec ce cas, il est intéressant de remarquer que nous retrouvons, par nos calculs, les résultats qui concernent la remise en cause de la fragmentation de INF dans le cas mono-objet (cf 2.3.2.6). Dans ce cas, en effet, $N_{ij}=N_{ik}=1$ puisqu' un seul objet composant ne peut être accédé avec son objet partiel composé. Nous retrouvons alors comme valeur limite des Coefficients de Liaison Simple $CLM_j=CLM_k=0,5$.

Pour en finir avec la liaison phase2-phase1, signalons enfin un dernier cas, celui du stockage direct de INF.fj dans deux fragments de classes supérieures différentes.

De même que pour le cas qui concernait la remise en cause de la fragmentation de SUP, ce cas n'est envisageable qu'en présence d'une valeur de SEUIL_CLM inférieure à 0,5. Quoiqu'il en soit, il ne constitue pas une remise en cause de la phase 1, mais plutôt une information sur l'utilité de la fusion de fragments de classes différentes.

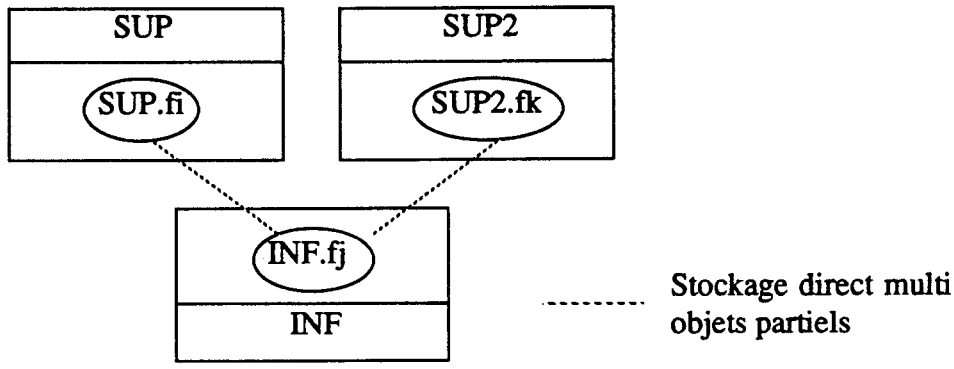


Fig. 2.74 : Stockage direct multi objets partiels de INF.fj dans deux fragment de classes supérieures et différentes

2.4. Synthèse sur la fragmentation verticale

2.4.1. Discussion sur le "pourquoi" d'une décomposition en deux étapes

Nous allons dans cette section justifier de façon plus précise le "pourquoi" d'une décomposition de la fragmentation verticale en deux étapes.

A priori, on peut se demander si une telle décomposition est réellement nécessaire et si le problème ne pourrait pas être résolu en prenant en compte, dès le départ, la structure complexe des objets. Il s'agit là de la première solution que nous avons envisagée lors de nos travaux. "Forts" de la connaissance des méthodes basées sur les matrices d'affinité [HOFF75], [NAVA84], nous pensions pouvoir les étendre facilement pour répondre à nos besoins. C'est donc les différents problèmes que nous avons rencontrés que nous présentons ci-dessous.

Prendre en compte directement la structure complexe des objets, revient donc à calculer une matrice d'affinité entre tous les attributs concernés, c'est-à-dire tous les attributs valeurs simples (ceux relatifs aux objets composés et ceux relatifs à tous les objets composants associés). Le principe du calcul des coefficients d'affinité reste le même, seuls quelques paramètres supplémentaires sont nécessaires. Le problème n'est pas tant la construction de la matrice d'affinité, ni sa taille souvent importante, mais plutôt son interprétation. A priori on pourrait se dire qu'il suffit d'appliquer le Bond Energy Algorithm pour regrouper les attributs semblables d'un point de vue utilisation puis, d'adapter le processus de production récursive de fragments. Le problème n'est, cependant, pas si simple.

Si l'on envisage l'utilisation d'une seule matrice d'affinité, l'utilisation d'un critère de regroupement des attributs, basé sur la "similitude" des coefficients d'affinité qui s'y rapportent, n'a plus de sens. Bien que de tels coefficients d'affinité comptabilisent tous des accès logiques simultanés, ils ne sont pas suffisants pour pouvoir prendre une décision, quant au "regroupements judicieux" d'attributs de sources différentes (relatifs à des classes différentes liées par un lien composés-composants). En d'autres termes, ils ne renferment pas suffisamment d'informations pour juger de l'utilisation favorable de tel ou tel autre mode de stockage (notons qu'il s'agit bien là du problème si on considère le regroupement de tels attributs). Entre autres, ils ne reflètent pas le partage qui peut exister, ni la notion d'accès favorables. Ce manque de précision compromet, bien évidemment l'utilisation du Bond Energy Algorithm afin de regrouper les attributs.

Exemple: Supposons que les objets d'une classe C1 soient composés entre autres, d'objets d'une classes C2, avec en moyenne, 1000 objets composants de C2 "dans" un objet composé de C1 (lien "multi-objets").

Supposons, de plus, que toutes les requêtes relatives à C1 et C2, accèdent simultanément aux attributs a_1 et a_2 de C1 et aux attributs a_3 et a_4 de C2, mais que seulement 1 objet composant de C2 (parmi les 1000) soit accédé avec son composé de C1. Les coefficients d'affinité relatifs aux attributs a_1, a_2, a_3, a_4 seront égaux. L'utilisation du BEA conduira donc au regroupement de ces attributs et donc à un mode de stockage direct partiel. Cependant 999 objets partiels composants seront inutilement et systématiquement accédés à chaque requête.

Bon nombre d'autres petits exemples de ce type pourraient être avancés, afin de mettre en évidence "l'inaptitude partielle" des coefficients d'affinité.

En fait, nous pouvons considérer qu'il y a une perte d'informations lors de leur calcul. Uniquement une partie de l'information pertinente, liée à la nature complexe des objets et aux requêtes concernées, est utilisée pour ce calcul. Il semble donc difficile de pouvoir obtenir en sortie des objets partiels complexes représentatifs. De plus,on risque fort d'aboutir à des incohérences liées à "l'ordre" des classes dans la structure complexe des objets. A ce sujet, nous montrons dans la section suivante, que de telles incohérences ne peuvent survenir avec notre méthode.

Il faudrait, pour remédier à ce problème, donner aux coefficients d'affinité un rôle dual qui serait d'une part, comptabiliser les accès logiques simultanés et, d'autre part, traduire la "qualité' de ces accès. Néanmoins, il nous paraît peu probable de pouvoir traduire dans une seule valeur cette double composante quantitative et qualitative, et surtout de pouvoir l'exploiter facilement.

Autre problème attaché à une telle méthode globale de fragmentation verticale, celui des classes d'objets partagées entre différentes classes. Leurs attributs apparaîtraient dans différentes matrices et donc, certaines décisions contradictoires pourraient être prises pour un même attribut. Il faudrait donc une étape ultérieure pour harmoniser les résultats obtenus à partir de chaque matrice. En conséquence, il y aurait aussi, avec une telle méthode, une remise en cause possible de résultats précédemment obtenus.

Nous pourrions continuer la présentation d'autres problèmes " pointus" attachés à l'utilisation d'une telle méthode globale, notamment en ce qui concerne la réalisation d'un pro-

cessus de production récursive de fragments. Nous en resterons là et soulignerons que nos travaux dans cette voie se sont vite "enlisés", tant l'harmonisation des différents problèmes à prendre en compte simultanément est peu évidente. Tous comptes faits, nos deux étapes répondent à des préoccupations différentes (ne serait-ce que par la "nature conceptuelle" des données qui y sont manipulées), et trouvent là, nous semble-t-il, leur principale justification.

2.4.2. Discussion sur les résultats de la fragmentation verticale

Suite aux deux phases de fragmentation verticale nous obtenons donc, pour chaque classe d'objets nuplets, un ensemble de fragments qui peuvent être ***libres*** ou ***liés***, suivant que la phase 2 ait "décidé" ou pas de les rattacher à un fragment d'une autre classe.

Le schéma logique initial est donc découpé verticalement en ***fragments normalisés*** et ***fragments complexes***, et ce, en fonction de l'utilisation des données.

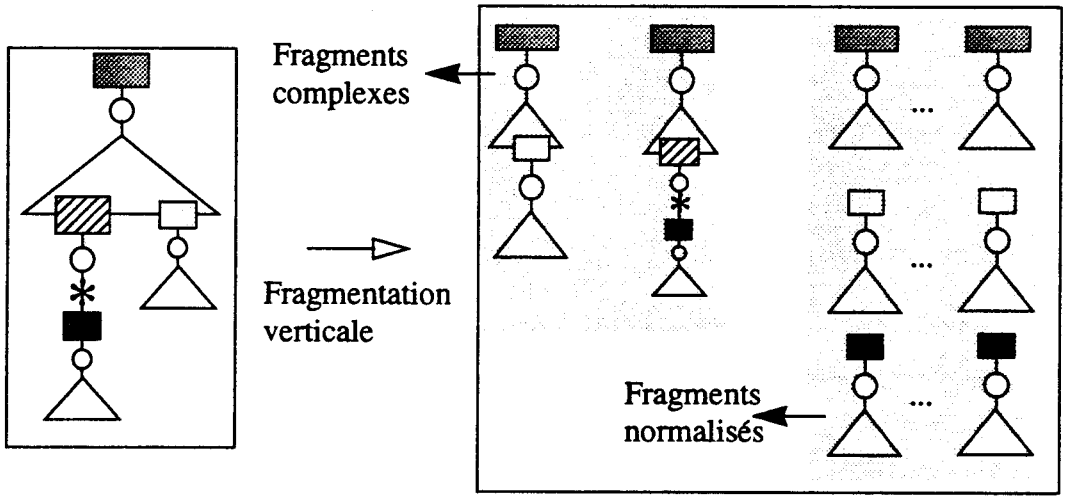


Fig. 2.75 : Fragments complexes et fragments normalisés

A ce niveau, nous pouvons nous demander s'il est nécessaire d'établir un ordre quelconque entre les classes du schéma logique, afin de procéder à la prise en compte des attributs objets (phase 2). Il faut tout d'abord remarquer que la méthode proposée tient compte, lors de l'étude d'un lien entre deux classes, des autres liens sur ces classes, puisque tous les accès logiques sont évalués pour le calcul des coefficients CLS et CLM. Cependant, nous n'utilisons jamais de décisions de stockage qui auraient été précédemment prises pour ces autres liens.

Une prise de décision pour deux fragments se fait donc toujours indépendamment de tout ce qui a pu se passer avant, ie des décisions antérieures prises pour les autres liens entre fragments. L'ordre, dans lequel les couples des classes liées sont étudiés, n'influence donc pas le résultat final.

Un des résultats importants qu'il est utile de rappeler est que, grâce à la méthode proposée, un même objet pourra bénéficier des avantages de différents types de stockage. Un de ses objets partiels pourra en effet être stocké directement avec un objet partiel d'une autre

classe, alors que les autres seront stockés de façon normalisée. Nous descendons donc à un niveau plus fin que celui de l'objet, les avantages attachés à chaque mode de stockage.

En ce qui concerne les "conséquences parallèles" de cette fragmentation verticale, nous pouvons en trouver de deux types. D'une part, nous rendons possible un parallélisme intra-objet et augmentons ainsi le parallélisme inter-requêtes potentiel, parallélisme qui n'est autre que le parallélisme de contrôle [SANS90] propre aux applications bases de données.

D'autre part, le fait d'avoir découpé les objets va nous permettre, au moment de la répartition des objets partiels obtenus, de faire des choix différents au sein d'un même objet.

Nous abordons là le domaine de la distribution des données sur un réseau, notion orthogonale à la notion de fragmentation verticale et qui relève de l'utilisation de techniques regroupées sous le nom de ***fragmentation horizontale***. Cette autre notion qui, dans un contexte parallèle, est avant tout orientée vers le parallélisme intra-requête (forme propre aux applications bases de données du parallélisme de données [SANS90]), fait l'objet du chapitre suivant. Nous pouvons néanmoins déjà souligner que, grâce à la fragmentation verticale proposée, nous pourrons adapter à un niveau fin les différentes techniques retenues pour la fragmentation horizontale et ainsi affiner l'utilisation de la machine par rapport au parallélisme intra-requête.

2.4.3. Quelques mots sur les extensions envisageables

Avant de passer au chapitre sur la fragmentation horizontale, nous allons proposer brièvement quelques extensions que l'on pourrait apporter dans le cadre de la fragmentation verticale.

Tout d'abord, nous pourrions envisager d'utiliser différents modes de stockage pour un même fragment d'objets partiels composants. En effet, dans le cas d'un attribut "multi-objets" (ex: attribut *commande* de notre exemple principal cf fig 2.25), nous pourrions instaurer un stockage partiel direct de certains objets partiels composants "avec" leur objet partiel composé et, un stockage partiel normalisé pour les autres objets partiels composants qui se rapportent au même composé.

A partir de notre exemple, cela pourrait consister à stocker "avec un client" les objets partiels relatifs à ses deux dernières commandes ou à ces commandes du trimestre en cours.

Une telle méthode nécessiterait de prendre en compte un quelconque prédicat, afin de déterminer ceux des objets partiels à stocker directement. Ce prédicat pourrait être relatif à la valeur d'un ou plusieurs attributs ou, relatif au temps. Il est important de noter, que le recours à ces multiples modes de stockage, entraînerait des migrations de données en cas de changement des valeurs utilisées dans les prédicats ou, tout simplement avec le temps dans le cas de prédicats temporels.

Nous ne détaillerons pas plus, ici, une telle extension, à laquelle nous n'avons pas encore sérieusement réfléchi, surtout en ce qui concerne ces incidences sur les solutions que nous avons proposées.

Une autre extension que nous pourrions envisager dans le cadre de la fragmentation verticale, est relative à l'analyse d'autres types de liens entre les classes d'objets (ex: attribut objet ensemble d'ensembles d'objets nuplets!). Plus généralement, nous pourrions mener une

approche similaire pour la fragmentation verticale dans le contexte d'autres modèles de données, tels que ceux supportés par les SGBD Orientés Objets. Ainsi nous pourrions prendre en compte les autres constructeurs de données complexes que sont: les listes, les "bags", etc. Puisqu'il semble, qu'une certaine homogénéité apparaisse, en ce qui concerne les constructeurs de données complexes retenus, un tel travail ne serait certainement pas dénué d'intérêt.

Enfin, le dernier point, que nous citerons ici, concerne l'étude qui pourrait être faite sur la transposition de la notion d'héritage, chère à l'approche Orientée Objet, au niveau interne d'une machine sans-partage. En d'autres termes il s'agirait de répondre à la question: "comment traduire la notion conceptuelle d'héritage (ou de sous-typage des Types Abstraits de Données) dans une méthode de placement de données telle que nous l'avons proposée?" Ce problème "plus qu'intéressant" est totalement ouvert.

Ainsi se termine ce chapitre sur la fragmentation verticale. Nous allons maintenant passer à la seconde grande étape de notre méthode de transformation de schéma, celle qui concerne la ***fragmentation horizontale*** ou, en d'autres termes, la répartition des objets partiels précédemment obtenus.

3. LA FRAGMENTATION HORIZONTALE

3.1. Présentation générale du problème

3.1.1. Définitions

Indépendamment de tout contexte précis, l'idée générale de la fragmentation horizontale est d'éclater un ensemble de données en plusieurs sous-ensembles de données de même nature (même structure), disjoints ou non. Dans tous les cas, ces sous-ensembles doivent permettre de reconstruire l'ensemble des informations de départ. Ce concept est souvent implanté dans le cadre du modèle relationnel. Néanmoins les contextes d'utilisation peuvent varier entre une approche centralisée, répartie ou parallèle. En fonction de son champ d'application, la définition et les objectifs qu'on attribue à la fragmentation horizontale varient sensiblement. Nous nous proposons dans un premier temps de faire un tour d'horizon sur le sujet afin de le situer précisément dans notre contexte.

3.1.1.1. Contexte relationnel centralisé ou réparti[1]

G. Gardarin et P. Valduriez donnent dans [GARD90] la définition suivante de la fragmentation horizontale: "Fonction qui partitionne une relation en sous-ensembles de nuplets, chacun étant défini par une opération de restriction appliquée à la relation". S. Ceri la voit comme "le résultat d'une opération de l'algèbre relationnelle" [CERI85]. Ces deux définitions semblables permettent d'introduire la notion de prédicats de fragmentation qui ne sont autres que les opérations de restriction appliquées.

EMPLOYE

NUM	NOM	AGE	SERVICE
01	DURAND	32	ATELIER1
02	MARTIN	45	ATELIER2
03	DUPONT	30	ATELIER3
04	AIME	43	ATELIER1
..	..	..	..

Prédicats de fragmentation:

EMPLOYE1 <=> $\sigma_{AGE<35}$(EMPLOYE)

EMPLOYE2 <=> $\sigma_{AGE\geq35}$(EMPLOYE)

EMPLOYE1

NUM	NOM	AGE	SERVICE
01	DURAND	32	ATELIER1
03	DUPONT	30	ATELIER3
..	..	..	..

EMPLOYE2

NUM	NOM	AGE	SERVICE
02	MARTIN	45	ATELIER2
04	AIME	43	ATELIER1
..	..	..	..

Fig. 3.1 : Fragmentation horizontale à partir de prédicats de fragmentation

1. "Réparti" se réfère ici à la notion de bases de données réparties, notion sur laquelle nous reviendrons plus en détail par la suite lorsque nous étudierons les relations entre "notre" fragmentation horizontale et les bases de données réparties (cf 3.1.3).

Avec une telle définition, la fragmentation horizontale, encore appelée concaténation verticale, favorise le traitement des requêtes de restriction qui portent sur les attributs utilisés dans la définition des fragments à accéder. Son rôle est donc, dans ce cas, de regrouper les nuplets qui sont ou seront accédés simultanément. Dans un univers centralisé, elle permet de limiter le nombre de pages accédées, ou en d'autres termes, elle permet de réduire le nombre de nuplets accédés inutilement.

Dans un contexte réparti, elle offre la possibilité de découpage d'un ensemble de données entre les différents sites, afin d'établir une localité d'un plus haut niveau entre les nuplets utilisés simultanément. L'exemple couramment cité est celui du découpage d'un ensemble de clients en attribuant à chaque agence le sous-ensemble de ses clients. Bien que ces deux approches soient différentes, le rôle de la fragmentation horizontale y est le même, au niveau près.

Afin de favoriser l'opération de jointure[1] il peut être procédé à une fragmentation horizontale indirecte. Celle ci se définit comme une fonction qui partitionne une relation en sous-ensembles de nuplets, chacun étant défini par une opération de semi-jointure[2] avec un fragment d'une autre relation.

Exemple: En reprenant l'exemple précédent, nous pouvons ajouter la relation PRODUCTION que nous fragmentons alors de la façon suivante:

PRODUCTION

NUM_EMP	NUM_PIECE	DATE	QUTE
01	p1	06/89	12
02	p3	07/89	15
01	p2	06/89	5
04	p1	08/89	3
02	p2	06/89	8

PRODUCTION1 <=> PRODUCTION ⋉ EMPLOYE1 <=> production des employés de EMPLOYE1

PRODUCTION2 <=> PRODUCTION ⋉ EMPLOYE2 <=> production des employés de EMPLOYE2

PRODUCTION1

NUM_EMP	NUM_PIECE	DATE	QUTE
01	p1	06/89	12
01	p2	06/89	5

PRODUCTION2

NUM_EMP	NUM_PIECE	DATE	QUTE
02	p3	07/89	15
04	p1	08/89	3
02	p2	06/89	8

Fig. 3.2 : Exemple de fragmentation horizontale indirecte

1. Les définitions données ici se réfèrent au modèle relationnel, terrain de prédilection de la fragmentation horizontale. Néanmoins les notions de relationnel et de fragmentation horizontale ne sont pas indissociables.
2. Rappelons que la semi-jointure entre deux relations R1 et R2 donne comme résultat les nuplets de R1 qui participent à la jointure des deux relations. La notation utilisée est R1 ⋉ R2.

Le placement "au même endroit" de chaque paire de fragments $EMPLOYE_i$-$PRODUCTION_i$ permet d'améliorer les performances de la jointure entre les relations EMPLOYE et PRODUCTION, puisqu'il la rend exécutable localement. Une fois de plus, cette amélioration peut intervenir à deux niveaux, le niveau des pages mémoire du contexte centralisé ou celui des sites du contexte réparti.

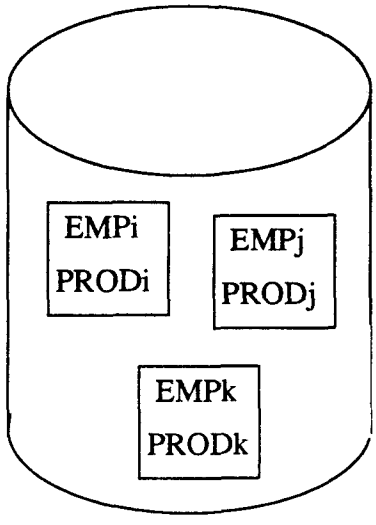


Fig. 3.3 : Fragmentation horizontale indirecte dans un contexte relationnel centralisé

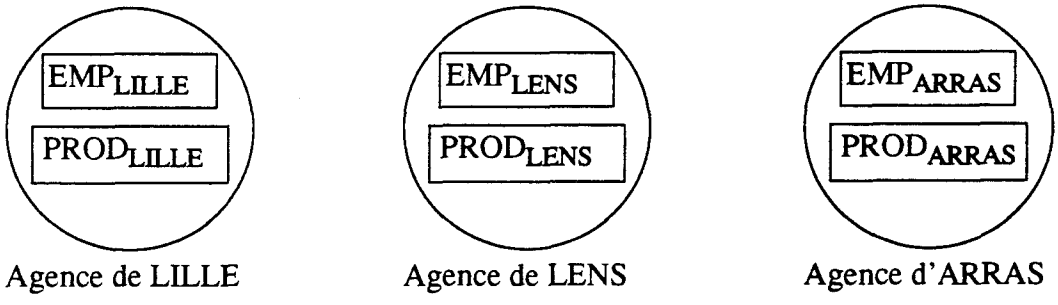


Fig. 3.4 : Fragmentation horizontale indirecte dans un contexte relationnel réparti

Avec ce premier type de fragmentation horizontale, un grand rôle est laissé à l'administrateur qui doit trouver, s'il existe, l'ensemble idéal et minimal [BOUZ87] de prédicats de fragmentation. D'une façon générale, cet ensemble de prédicats pourra rarement favoriser toutes les requêtes; il risquera même d'en désavantager fortement certaines. Il apparait donc nécessaire d'analyser précisément certains paramètres avant de prendre une quelconque décision. Nous ne nous appesantirons pas plus sur le sujet étant donné qu'il sort de notre contexte parallèle et des objectifs autres, que nous voulons accorder à la fragmentation horizontale.

3.1.1.2. La fragmentation horizontale dans un contexte parallèle.

La définition générale qui est de partager un ensemble de données en sous-ensembles, ne change pas. Cependant les objectifs diffèrent totalement. Il s'agit cette fois "d'éclater" les données afin de pouvoir les traiter ultérieurement en parallèle et ainsi diminuer le temps de réponse. Il n'est donc plus question de rassembler les données utilisées simultanément, mais au contraire de les disséminer pour obtenir un taux important de parallélisme pour l'exécution d'une même opération.

Dans ce cas, la fragmentation horizontale va dans le sens d'une augmentation du taux de parallélisme intra-requête. En l'analysant grâce à un autre vocabulaire, nous pouvons dire qu'elle induit un fonctionnement en mode SIMD (Single Instruction Multiple Data) alors que la fragmentation verticale conduit comme nous l'avons vus (cf 2.1) à un fonctionnement MIMD (multiple Instructions Multiple Data).

Si nous reprenons la classification des différents types de parallélisme proposés dans [SANS90], la fragmentation horizontale se réfère donc au parallélisme de données. Rappelons que de par leur nature, les applications bases de données constituent un terrain de prédilection pour ce type de parallélisme.

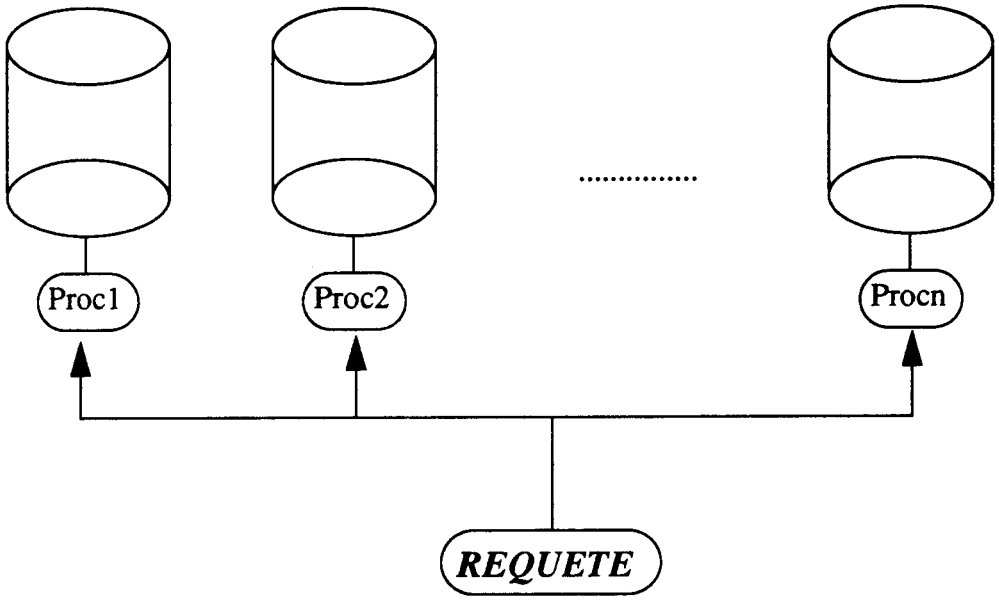


Fig. 3.5 : Fragmentation horizontale dans un contexte parallèle

Une telle utilisation de la fragmentation horizontale sous-entend la manipulation de volumes de données très importants, ce qui est bien sûr l'objectif premier de toute machine bases de données parallèle. Une façon plus imagée de voir les choses, est d'attribuer (comme le fait G. Gardarin [GARD90]) à ces machines la devise "Diviser pour régner", ie répartir les données pour augmenter les performances.

Dans tout ce qui suit, nous nous intéresserons à la fragmentation horizontale dans un contexte parallèle.

3.1.2. Analyse intuitive des avantages et désavantages

La répartition des données entraîne l'exécution d'une requête sur plusieurs noeuds (disque + processeur). De ce fait, toute activité peut être décomposée en, d'une part, les traitements proprement dits et d'autre part, les communications induites par la répartition des données. Un des problèmes majeurs liés à la fragmentation horizontale, est donc de trouver le meilleur compromis entre traitements et communications. Le degré de répartition, c'est à dire le nombre de noeuds sur lesquels la fragmentation s'opère, est de toute importance. Son aug-

mentation diminue le temps de traitement sur un noeud mais augmente les communications entre les noeuds, ainsi que les temps de démarrage et de terminaison d'une requête. Ce problème de détermination d'un juste milieu entre communications et traitements est d'ailleurs commun à toute activité parallèle, quelqu'en soit le domaine d'application.

M. Livny présente de façon intuitive [LIVN87] les avantages que l'on peut attribuer à une répartition des données comparativement à une solution centralisée. Le terme "centralisée" est à prendre ici en tant que regroupement des données d'un même ensemble sur un même disque, le système étant tout de même composé de plusieurs disques. Il discerne différents cas en fonction de la charge du système.

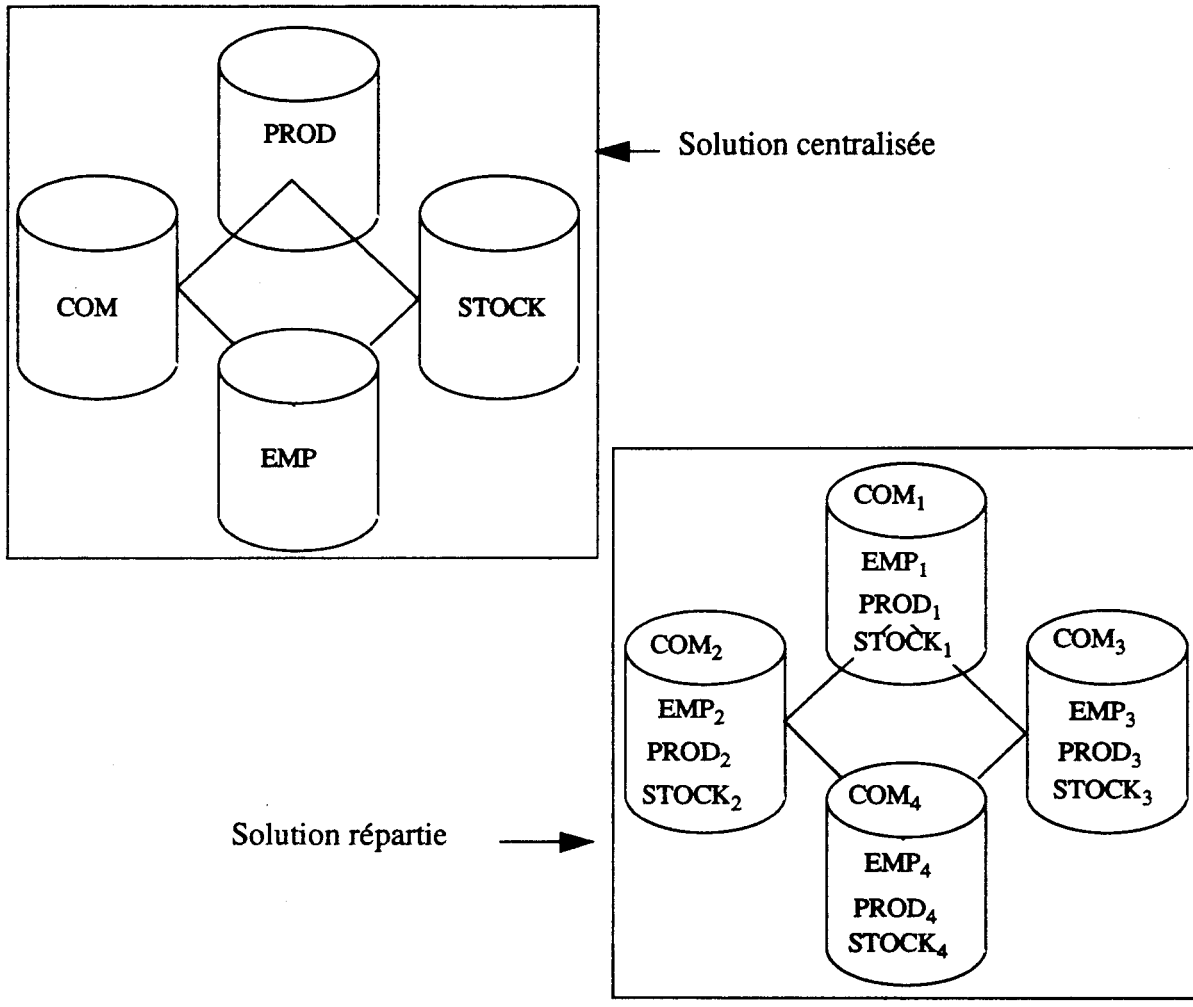


Fig. 3.6 : Approches centralisée et répartie selon M. LIVNY

Pour un système faiblement chargé, il pense que puisqu'il n'y a pas ou peu de compétition pour les ressources, la répartition des données implique une diminution des temps d'exécution et donc une augmentation du débit par rapport à l'autre solution. Dans le cas d'un système moyennement chargé, son idée est de dire que puisque les demandes sont plus "petites" que pour le système centralisé, le fonctionnement sera plus rapide malgré le surplus de communications[1]. Enfin pour un système fortement chargé, bien qu'a priori la solution répartie puisse avoir des "conséquences dramatiques", l'auteur souligne le fait que grâce à elle, la saturation

1. Cette "intuition" s'appuie sur les travaux qui montrent que pour la plupart des systèmes avec file d'attente, la réduction de la taille des demandes entraîne une diminution du temps d'attente normalisé [KLEI75].

de l'un des disques risque moins de se produire.

M. Livny ne se contente pas bien sûr de cette approche intuitive. Il propose aussi un modèle de simulation et un ensemble de résultats qui vont dans le sens de ses suggestions. Nous reviendrons plus en détail sur ces résultats dans ce qui suit (cf 3.1.4 Les travaux existants sur la fragmentation horizontale). Auparavant nous allons présenter tout aussi intuitivement les désavantages que nous pensons pouvoir attribuer à la fragmentation horizontale.

Tout d'abord, qui dit répartition de données, dit surplus d'informations de maintenance que sont les dictionnaires globaux et locaux de données, les fonctions de répartitions, les index locaux et globaux, etc. La gestion de ces informations et les mécanismes parfois complexes qu'elles nécessitent sont bien sûr une charge supplémentaire, comparativement à un système classique.

L'ensemble des problèmes liés à la migration, à l'intégrité et à la sécurité des données constitue un autre point important à la charge de la fragmentation horizontale. Là encore, le surcroît de difficultés entraîné ne doit pas être plus important que les gains obtenus sur les temps de traitement. Enfin, le fait que les données soient réparties peut être gênant en cas de défaillance de l'un des disques. En effet, dans ce cas le nombre de fragments atteints est plus important, ce qui va dans le sens d'une altération plus grave du système, sauf si bien sûr on dispose d'un mécanisme coûteux de duplication de données.

Avant d'aborder la présentation des travaux sur la fragmentation horizontale, nous allons faire une petite parenthèse sur les liaisons qui existent entre cette notion (prise dans le contexte parallèle) et le domaine "voisin" des bases de données réparties.

3.1.3. Liaisons avec le contexte des bases de données réparties.

Bien que les contextes des bases de données parallèles et des bases de données réparties soient différents, il existe entre eux certaines affinités qui les rapprochent. D'ailleurs si nous nous référons aux définitions données par Gardarin et Valduriez [GARD90], nous voyons bien que les deux domaines sont loin d'être disjoints:

Base de données répartie: "Ensemble de bases de données coopérantes, chacune résidant sur un site différent, vu et manipulé par l'utilisateur comme une seule base de données centralisée".

Base de données parallèle: "Base de données répartie homogène dont les sites sont les noeuds d'un calculateur parallèle (multi-processeurs) et communiquent par messages".

Ces définitions un peu rapides considèrent donc les bases de données parallèles comme un sous-ensemble des bases de données réparties. Cependant, il ne faut pas oublier que la répartition ne se fait pas à un même niveau topologique[1] et que, par conséquent, certains des objectifs des bases de données réparties n'ont plus lieu d'être pour une machine parallèle.

Au titre des objectifs "à conserver" nous pouvons citer:

-l'indépendance vis à vis de la fragmentation qui doit cacher à l'utilisateur le fait que les données sont fragmentées,

---

1. Il est en effet difficile de comparer les noeuds d'un hypercube avec les différentes agences départementales d'une société. Nous revenons par ce biais au problème évoqué dans la section précédente et qui est relatif aux objectifs initiaux différents de la répartition des données.

-l'indépendance vis à vis de la duplication qui doit, cette fois, cacher le fait que les données sont éventuellement dupliquées.

-l'extensibilité.

En ce qui concerne l'objectif d'indépendance par rapport à la localisation, qui a pour but de cacher le fait que les données ne résident pas forcément sur le site utilisateur, il est difficile de le replacer dans le contexte des machines parallèles. En effet, le noeud d'une telle machine ne peut être comparé avec un site et n'est en tout cas, aucunement lié à une notion d'utilisateur.

Enfin l'objectif d'autonomie de site, cher aux bases de données réparties, n'a plus de sens pour une machine parallèle. Le but recherché n'est pas de transformer chaque noeud d'un hypercube (ou d'une autre configuration) en un SGBD potentiellement autonome, mais plutôt de s'assurer de la bonne coopération entre les différents noeuds afin d'obtenir de bonnes performances.

Le contexte bases de données réparties et notre contexte bases de données parallèles ont donc, face aux outils de fragmentation qu'ils utilisent, bon nombre d'objectifs communs. Cette liaison partielle entre les deux domaines peut influencer les phases de conception et d'utilisation d'une telle machine parallèle. En effet, de nombreux travaux sur les bases de données réparties, dont une bonne synthèse peut être trouvée dans [CERI85], offrent une source non négligeable d'idées, de remarques et de résultats. Bien sûr cela nécessite une adaptation, un approfondissement ou même parfois une remise en cause en fonction des objectifs fixés. Néanmoins il nous semble nécessaire d'avoir connaissance de ce qui se fait dans ce domaine très proche.

3.1.4. Les travaux existants sur la fragmentation horizontale

3.1.4.1. L'étude comparative avec la solution centralisée: M. Livny

Cette étude, à laquelle nous avons déjà fait allusion (cf 3.1.1.2), est relative au bien fondé de la fragmentation horizontale. Elle compare donc cette méthode avec une solution centralisée qui consiste à toujours mettre les données d'un même ensemble sur un même disque, et cela bien que le système soit composé de plusieurs noeuds (cf figure3.6).

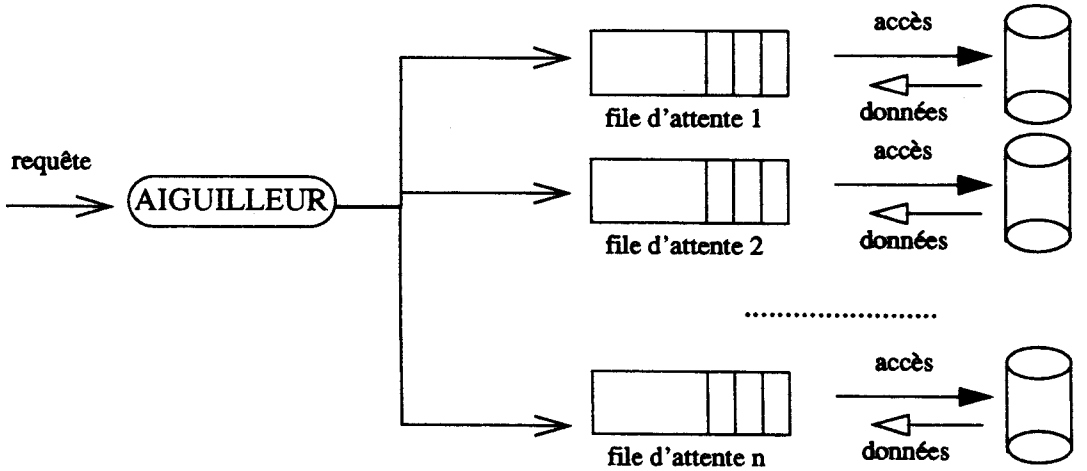


Fig. 3.7 : Principe du modèle de simulation utilisé par M. LIVNY

La comparaison se fait donc entre d'une part, une solution répartie qui favorise les traitements parallèles et, d'autre part, une solution centralisée qui induit une bonne localité entre les données d'un même ensemble. Le modèle de simulation utilisé est composé de n systèmes mono-file d'attente, mono-serveur et d'un aiguilleur (cf figure précédente).

D'une façon générale, les résultats montrent que la répartition de données est quasiment toujours la meilleure des deux solutions. L'auteur distingue deux catégories de résultats suivant que les accès aux données se font ou pas de façon uniforme.

Pour des accès uniformes, qui par ailleurs, favorisent la solution centralisée, la répartition des données s'avère être une moins bonne solution dans le cas d'un système très chargé; dans ce cas en effet, la saturation de l'un des disques intervient plus rapidement. Par contre cette solution répartie profite de l'augmentation du nombre de disques pour diminuer les différences entre les différentes catégories de requêtes testées. En d'autres termes, grâce à un nombre plus important de noeuds, les requêtes habituellement coûteuses "s'en sortent" de mieux en mieux. Ceci n'est bien sûr pas le cas pour la solution centralisée.

Pour des accès non uniformes, ce qui reflète tout de même bien mieux la réalité, la répartition est alors toujours la meilleure solution. Néanmoins l'auteur souligne le fait que ces résultats optimistes dépendent fortement de la façon dont ont été placées les données sur les disques et qu'il est très difficile et surtout coûteux de maintenir une répartition de charge bien équilibrée entre les disques.

Nous nous contenterons ici de donner ces quelques résultats généraux qui confortent notre intention d'utiliser la fragmentation horizontale dans notre contexte. L'article cité renferme des résultats précis agrémentés de nombreuses courbes que nous ne détaillerons pas ici. Nous pouvons citer comme autres références sur le même sujet [KIM85] et [SALE84].

3.1.4.2. Etude de l'influence du degré de répartition, la machine BUBBA.

Dans ce travail, les auteurs [COPE88] étudient entre autres, l'influence du degré de répartition, ie du nombre de noeuds cibles de la répartition, sur le débit d'une machine parallèle. Intuitivement plus ce nombre augmente, plus il y a de communications. De plus, dans un contexte bases de données et indépendamment du modèle utilisé, il existe des opérations pour lesquelles ce surplus de communications n'augmente pas linéairement avec le degré de répartition[1]. Les auteurs n'hésitent d'ailleurs pas à qualifier ce problème de "fléau des traitements parallèles", ce qui ne semble pas être un avis isolé [CVET87], [VRSA85]. Ils optent pour une prise en compte de la répartition dès la conception d'une machine et pour la détermination de mécanismes qui supportent des répartitions variables.

En utilisant l'outil de modèlisation FIRM [BOUG87] et une technique de placement basée sur un ensemble d'heuristiques[2], ils fournissent un certain nombre de résultats sur l'estimation du débit de leur machine. Le plus intéressant de ces résultats (tout au moins en ce qui nous concerne) est relatif à l'importance du degré de répartition et confirme notre intuition. Il met en évidence que pour certaines opérations, il existe un nombre de noeuds au delà duquel le débit diminue de par la trop grande importance des communications.

---

1. La plus tristement célèbre de ces opérations est bien sûr la jointure n:m du modèle relationnel.
2. Le problème de placement de données en fonction d'un certain nombre de contraintes étant NP-complet [MUKK87], les auteurs soulignent que l'utilisation d'heuristiques apparait être la seule solution viable pour tenter de résoudre le problème.

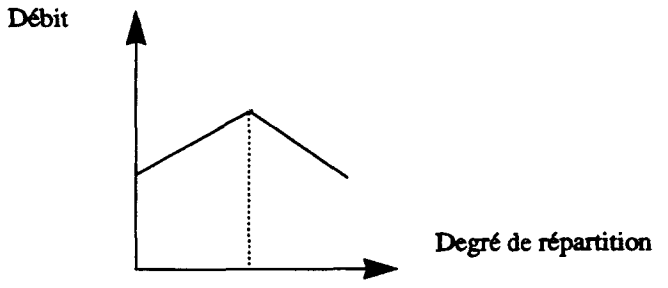


Fig. 3.8 : Allure générale de la courbe du débit en fonction du degré de répartition

En dessous du nombre "idéal" de noeuds, le débit est limité par le goulot d'étranglement que constituent les Entrées-Sorties sur disques; au delà de ce nombre, ce sont cette fois les processeurs qui limitent le débit à cause du trop grand nombre de messages à traiter.

Quelques leçons sont à tirer de ce résultat. Premièrement, il ne faut pas vouloir à tout prix répartir "le plus possible", ce qui au passage est la stratégie utilisée dans des machines telles que GAMMA [DEWI86] et TANDEM [TAND87]. A croire qu'à trop diviser on finit par avoir du mal à régner! Secondement, il faut pouvoir adapter la répartition à son utilisation future et surtout pouvoir détecter les opérations à risques. Nous reviendrons dans la suite sur ce problème traité dans notre contexte (cf 3.2 Importance du degré de répartition dans notre contexte).

Une dernière remarque avant de présenter la fragmentation horizontale sur quelques prototypes ou machines commercialisées qui utilisent tous l'approche "shared-nothing". Le nombre idéal de noeuds obtenu par simulation est de l'ordre de 750, ce qui pour l'instant est loin d'être atteint par les machines proposées sur le marché. D'autre part, ce seuil est issu d'une simulation, ce qui lui confère une certaine marge d'erreur, point sur lequel nous reviendrons aussi par la suite.

3.1.4.3. La fragmentation horizontale sur la machine GAMMA.

Rappelons que cette machine, qui se veut être une machine bases de données relationnelle dataflow à hautes performances, est issue du "Computer Sciences Department" de l'Université du Wisconsin [DEWI86] (cf1.3.3.2).

La fragmentation horizontale y est réalisée par une répartition totale, ie toutes les relations sont "éclatées" sur tous les disques. Il n'y a donc a priori pas d'adaptation en fonction de l'utilisation. Ceci s'explique certainement par le fait que le prototype utilisé ne comporte que 20 noeuds (VAX11/750), nombre "autour" duquel on ne risque pas de rencontrer les problèmes soulevés dans la section précédente et relatifs au compromis traitements-communications.

Quatre méthodes sont proposées à l'administrateur pour répartir ses données:

-Distribution circulaire ("Round robin"): dans ce cas les données sont distribuées au fur et à mesure sur les disques et a priori sans moyen direct de les retrouver ultérieurement.

-Utilisation d'une fonction de hachage aléatoire qui à une clé donnée associe un numéro de disque. Cette méthode est aussi utilisée dans la Teradata Database Machine [TERA84] (cf 1.3.3.4). Ce type de répartition est bien sûr directement dépendant du choix d'une bonne

fonction de hachage qui doit fournir une répartition bien homogène afin d'équilibrer la charge. Ce problème apparemment simple à résoudre, est en fait peu évident; nous lui consacrerons tout une partie dans la suite de l'exposé (cf 3.4 Comment répartir?).

-Utilisation d'intervalles spécifiés par l'utilisateur, ce qui nous ramène aux prédicats de fragmentation présentés auparavant. Cette solution est aussi utilisée dans la machine TANDEM; cependant dans GAMMA l'ordre des données n'est pas préservé, les concepteurs voulant conserver une propriété d'indépendance entre l'attribut de répartition et l'ordre des nuplets sur les noeuds. Deux critiques peuvent être apportées à ce genre de répartition. Tout d'abord elle est dépendante de l'administrateur qui, sans l'aide d'informations quantitatives, risque de définir une répartition non équilibrée. Secondement les prédicats de fragmentation risquent de faire de l'anti-jeu, ie de l'anti parallélisme intra-requête, s'ils regroupent les nuplets utilisés simultanément. Nous en revenons au problème des objectifs de la fragmentation horizontale qui pour nous doivent être avant tout synonymes d'obtention d'un meilleur taux de parallélisme.

-Utilisation d'intervalles de distribution uniformes qui ne sont autres que le résultat d'une distribution circulaire, suivie d'un tri. L'objectif est, dans ce cas, d'envoyer sur chaque disque sensiblement le même nombre de nuplets, ce qui nous ramène bien au rôle de la fragmentation horizontale dans un contexte parallèle, mais qui est ici dépendant d'un tri coûteux et peu simple à remettre en cause.

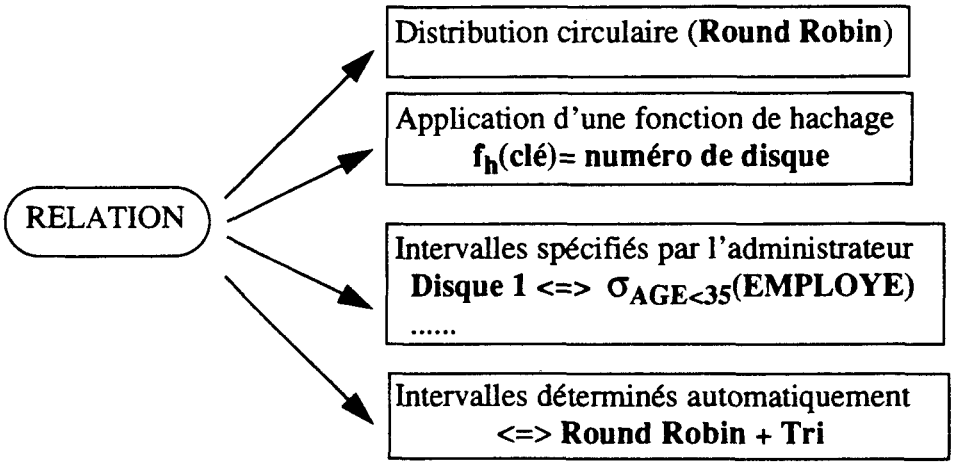


Fig. 3.9 : Méthodes de fragmentation horizontale proposées dans GAMMA

3.1.4.4. La machine ARBRE.

Ce projet, "the Almaden Research Backend Relational Engine", est issu du centre de recherche IBM de San Jose [LORI89] (cf 1.3.3.6). Il ne propose rien de bien original pour la répartition de données puisqu'il préconise l'utilisation d'intervalles de valeurs spécifiés par l'utilisateur ou déterminés automatiquement comme dans la machine GAMMA. Néanmoins le fait qu'il propose une certaine indépendance par rapport aux configurations matérielles laisse entrevoir des possibilités d'adaptation de différents critères dont la fragmentation horizontale

fait partie. Les concepteurs soulignent d'ailleurs dans leur article consacré à l'ajout de parallélisme intra-requête dans un SGBD [LORI89], que leurs travaux doivent maintenant s'orienter vers une étude plus poussée de la répartition de charge, ce qui devrait les amener à étudier plus finement les techniques de répartition de données.

3.1.4.5. Le projet TANDEM.

Ce projet est, rappelons-le, issu d'une association entre différents laboratoires industriels (Bell Telephone Laboratories -UNIX-, Tandem Computers, ...) [TAND87]. Comme nous l'avons soulignés lors de sa présentation (cf 1.3.3.5), il marie les approches bases de données réparties et parallèles. Il s'articule autour d'une version NonStop de SQL. Toute répartition de données est explicitement exprimée en SQL. En d'autres termes, la fragmentation horizontale se fait par l'utilisation d'intervalles de valeurs spécifiés par l'administrateur, méthode sur laquelle nous ne reviendrons pas.

Remarque: Le fait que les aspects réparti et parallèle soient tous deux utilisables offre à l'administrateur les deux classes d'objectifs de la fragmentation horizontale, ce qui ne va pas dans le sens d'une diminution de la difficulté de son rôle.

```
CREATE TABLE site1.disk1.dir.emp
                    (emp-no INTEGER,
                     dep_no INTEGER,
                     PRIMARY KEY emp_no),
PARTITION site2.disk3.dir.emp FIRST KEY 10000,
CATALOG site2.disk2.dir4;
```

Fig. 3.10 : Expression explicite de la fragmentation verticale dans le projet TANDEM

3.1.4.6. Conclusions sur les travaux présentés.

Nous nous permettons d'apporter plusieurs critiques sur l'utilisation de la fragmentation horizontale dans les machines présentées. Tout d'abord, le rôle laissé à l'administrateur est souvent trop important. La détermination de prédicats de fragmentation judicieux pour une relation donnée, risque de ne pas être chose facile sans l'aide d'informations sur la distribution des données, les accès aux données, etc. A ce sujet, aucun projet ne parle de l'utilisation de telles informations.

La solution qui consiste à employer des fonctions de hachage semble être moins dépendante du bon choix à effectuer par la personne responsable. Néanmoins ceux qui l'utilisent (GAMMA, TERADATA) sont peu bavards sur la façon dont ils obtiennent ces fonctions et sur les résultats qu'elles donnent. Comme nous le verrons par la suite le problème n'est pas trivial.

En fait, les seuls projets qui se rapprochent de nos préoccupations dans le domaine de la fragmentation horizontale, sont la machine BUBBA [COPE88] et la machine ARBRE [LORI89]. Néanmoins l'étude des points importants liés à la répartition des données n'y est que partiellement entamée.

Enfin, il faut souligner que tous ces travaux s'articulent autour du modèle relationnel (sauf BUBBA dont une version a été testée avec le modèle de données et le langage FAD), ce qui à notre sens limite les problèmes de répartition comparativement à ceux que nous allons rencontrer dans notre contexte. De plus, les données réparties sur ces machines ne sont pas préalablement fragmentées de façon verticale. Cet autre point va introduire une nouvelle dimension que nous devrons prendre en compte pour la fragmentation horizontale de nos objets partiels.

3.1.5. La fragmentation horizontale dans notre contexte.

3.1.5.1. Influence du modèle de données

Le modèle qui sert de base à notre étude est, comme nous l'avons souligné lors de sa description, sur bien des points différent du modèle relationnel.

Il est important de rappeler que les données que nous manipulons sont des objets complexes identifiés, qui peuvent être éventuellement partagés et qui peuvent avoir été fragmentés verticalement en objets partiels.

Nous sommes là bien loin de simples nuplets propres au modèle relationnel. La notion d'identité d'objet, c'est-à-dire l'existence pour tout objet d'un identifiant interne, permet d'envisager une répartition par rapport à ces identifiants. Comme nous le verrons par la suite (cf 3.4.1 Les différents modes de répartition et leur influence), répartition par rapport aux valeurs et répartition par rapport aux identifiants ont chacune leurs avantages et sont plus ou moins adaptées suivant les cas. En d'autres termes, grâce à cette propriété intéressante du modèle de données, la fragmentation horizontale pourra être réalisée de différentes façons, ce qui permettra de l'adapter au mieux à l'utilisation ultérieure des données.

3.1.5.2. Influence des étapes préalables de fragmentation verticale.

L'utilisation de la notion de fragmentation verticale et surtout son prolongement "vers" le modèle utilisé, va nous obliger à "repenser" quelque peu la notion de fragmentation horizontale.

Par les méthodes proposées, tout objet nuplet est scindé en un ensemble d'objets partiels libres ou liés (cf chapitre 2 sur la fragmentation verticale).Nous donnons ci-dessous, sur un exemple, une représentation au niveau du schéma logique de ces différentes phases de fragmentation verticale.

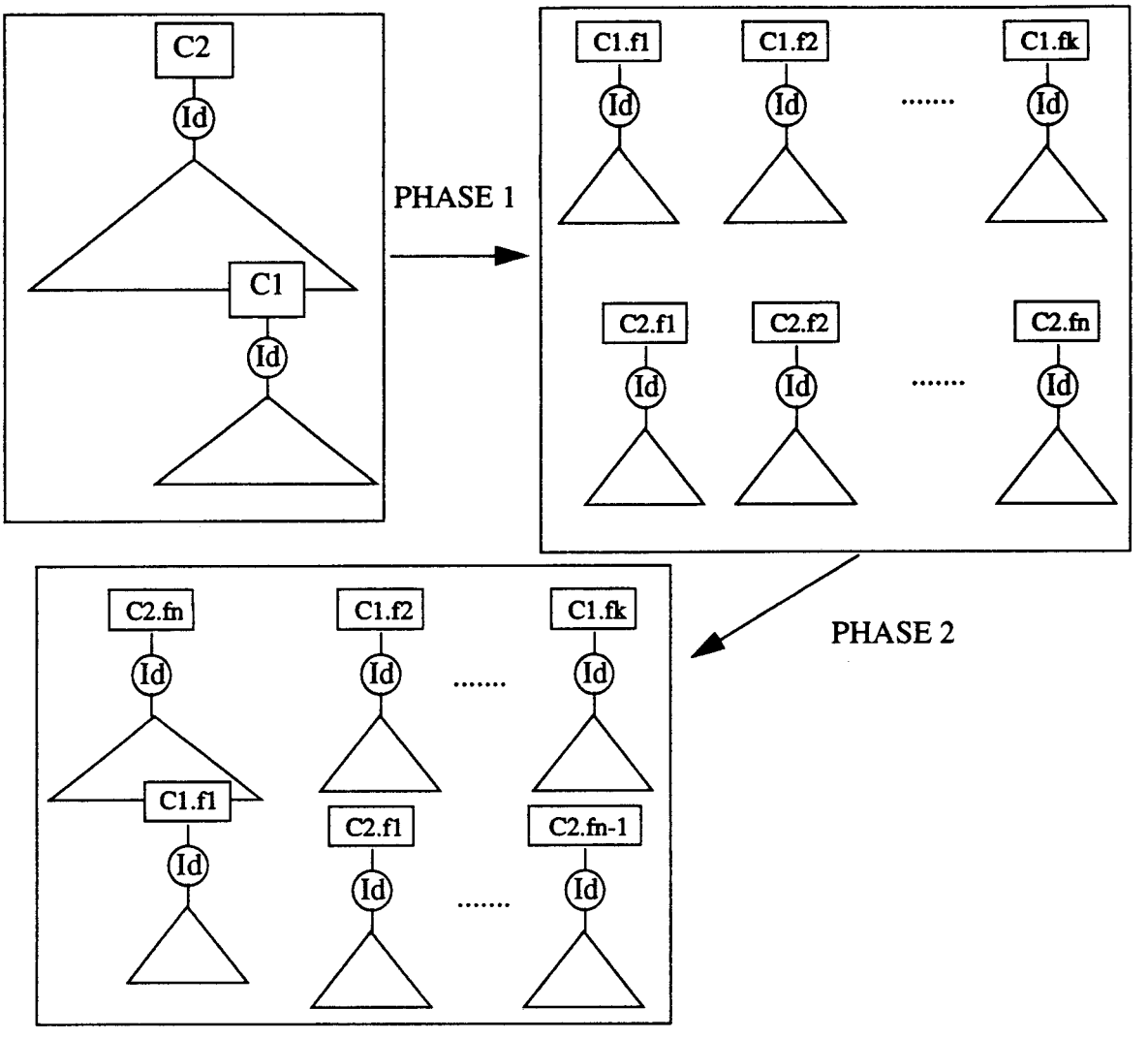


Fig. 3.11 : Schématisation des phases de fragmentation verticale

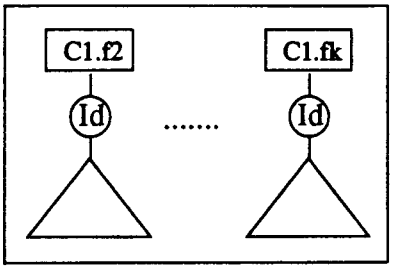
Ainsi, sur cet exemple, une occurrence de C1.f1 est un objet partiel lié puisqu'elle est objet partiel composant de une ou plusieurs occurrences de C2.fn.

Dans notre contexte, les entités manipulées à un niveau physique sont donc des objets partiels dont certains peuvent être composants d'autres.

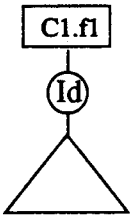
Le fait qu'un fragment soit rattaché à un autre, c'est-à-dire que ses occurrences soient objets partiels composant, dénote de par les fonctions utilisées dans la phase 2 de la fragmentation verticale, une majorité de requêtes qui utilisent ces fragments simultanément et qui "empruntent" donc le chemin: objet partiel composé -> objet partiel composant. En conséquence, il apparait nécessaire que ce soit le placement de l'objet partiel composé qui contraigne celui du composant. Nous pouvons d'ailleurs remarquer que vouloir faire le contraire serait impossible dans le cas de multiples objets partiels composants pour un même composé.

Pour une classe C d'objets nuplets, les seules données à répartir sur le réseau sont donc les objets partiels libres, les objets partiels liés étant traités au moment de la répartition des objets partiels composés dont ils dépendent.

Cela donne pour la classe C1 de notre exemple:



Fragments libres de C1 <=> objets partiels libres d'un point de vue répartition



Fragment lié de C1 <=> objets partiels composants traités par rapport aux occurrences de C2.fn

Fig. 3.12 : Fragments libres et fragments liés de C1

L'idée première pourrait être de pratiquer une fragmentation horizontale indépendante de chaque fragment libre du schéma comme le font les systèmes relationnels classiques qui "éclatent" indépendamment chaque relation.

Cependant, avant toute précipitation, nous devons rappeler que nous travaillons ici à un double niveau: celui des classes d'objets et celui des fragments d'objets partiels. Le découpage en classes d'objets est issu d'une phase de conceptualisation comme le sont les relations dans un système relationnel. Les entités ainsi discernées sont conceptuellement indépendantes et n'ont a priori pas lieu d'être réparties les unes par rapport aux autres[1].

Le second découpage, qui consiste à scinder une classe d'objets en fragments, est d'un ordre totalement différent puisqu'il consiste à découper une même entité conceptuelle, donc cette fois au niveau interne. Bien qu'appartenant à des fragments différents, les objets partiels d'un même objet sont liés au niveau conceptuel. D'ailleurs, cette appartenance à des fragments différents n'implique pas entre eux une totale indépendance logique. Il peut exister des requêtes qui nécessitent la reconstruction de tout ou partie d'un objet à partir de ses objets partiels. Ces requêtes ne peuvent être majoritaires puisque, dans ce cas, il n'y aurait pas eu fragmentation verticale. Néanmoins, il existe une notion de dépendance entre les objets partiels d'un même objet.

Entre deux objets partiels totalement indépendants, ie qui ne seront jamais accédés simultanément, et deux autres issus d'une décision de fragmentation verticale "prise de justesse", se situent bien des nuances qu'il serait peut-être bon de traiter de façon adaptée. En fait, la question est de savoir comment doit se comporter la fragmentation horizontale face à cette dépendance entre objets partiels. En d'autres termes, est-il intéressant de gérer une proximité entre objets partiels d'un même objet?

Nous opterons ici pour une gestion calculée de cette proximité, solution qui répondra de façon logique, ie en fonction de l'utilisation des objets partiels, mais aussi de façon phy-

1. Cet a priori est néanmoins discutable. En effet, nous pourrions envisager de répartir de façon relative des données en provenance de classes différentes. Nous aborderons d'ailleurs cette éventualité dans (cf 3.5 synthèse sur le fragmentation horizontale).

sique, ie par rapport à la configuration matérielle utilisée, au problème soulevé. Notre approche permettra d'éviter le recours systématique à l'une ou l'autre des solutions extrêmes qui consistent soit en une répartition totalement indépendante des objets partiels d'un même objet, soit en leur localisation sur un même noeud.

Si nous reprenons notre approche comparée avec les bases de données réparties, nous pouvons remarquer que, pour nous la notion d'indépendance par rapport à la fragmentation est double puisque nous préconisons l'emploi simultané des techniques de fragmentation verticale et horizontale. De plus, de par le modèle de données que nous considérons, cette indépendance par rapport à la fragmentation est enchevêtrée avec l'indépendance nécessaire par rapport au partage d'objets. Somme toute, un utilisateur ne doit pas se soucier du fait qu'un objet soit découpé verticalement, qu'il soit "éparpillé" sur le réseau et qu'il soit éventuellement partagé par d'autres objets.

3.1.5.3. Influence du contexte parallèle

Comme nous l'avons souligné précédemment (cf 3.1.1.2 La fragmentation horizontale dans un contexte parallèle), notre utilisation de la fragmentation horizontale a pour but de favoriser la répartition de la charge et par là d'augmenter le débit de la machine. Rappelons que pour le type de machine auquel nous nous intéressons, il y a une forte dépendance entre la répartition des données et la répartition de charge(cf chapitre 1 Parallélisme et bases de données) puisque l'exécution des requêtes se fait là où les données se trouvent.

Pour procéder à cette répartition de données, nous n'utiliserons pas de prédicats de fragmentation. Nous proposerons plutôt des solutions qui s'orientent vers l'obtention d'une répartition homogène, ce qui constitue aux yeux de beaucoup la clé d'un fonctionnement optimal.

L'utilisation que nous allons faire de la fragmentation horizontale est donc guidée par, d'une part le souci de gérer habilement la proximité entre les différents objets partiels libres d'un même objet, et d'autre part la volonté d'obtenir une répartition équilibrée. La démarche proposée est basée comme pour la fragmentation verticale, sur l'analyse quantitative périodique de certains paramètres. En conséquence, la répartition pourra être remise en cause pour des raisons logiques: modification de l'utilisation des données, ou pour des raisons physiques: modification de la configuration matérielle.

3.1.6. Présentation de la suite de l'exposé

La suite de l'exposé, qui concerne les solutions que nous proposons, se décompose comme suit. Dans un premier temps, nous nous intéresserons aux problèmes liés à l'importance du degré de répartition dans notre contexte. A ce sujet nous verrons quels sont les critères importants à prendre en compte. Nous discuterons de l'utilisation de différents modèles de simulation afin de déterminer le ou les seuils critiques de ce degré de répartition.

Dans une seconde partie, nous étudierons la répartition relative des occurrences des fragments verticaux d'une même classe. Nous proposerons pour cela l'évaluation d'un éloignement inter-fragments dont nous étudierons d'une part, les propriétés générales et d'autre part, les liens qu'il possède avec la notion de fragmentation verticale. Nous verrons ensuite comment

transformer ces éloignements "deux à deux" en véritables distances qui prennent en compte tous les fragments libres de la classe concernée. Ces deux premières phases nous permettront de répondre de façon logique au problème posé. Nous étudierons ensuite la transformation des positions relatives logiques en positions relatives physiques adaptées à un réseau donné. Cette séparation en un niveau logique et un niveau physique fera de notre approche une solution facilement adaptable. Différents types de réorganisation de données pourront ainsi être discernés, offrant pour la machine un cadre souple d'utilisation.

Dans une troisième partie, nous analyserons les problèmes relatifs à l'obtention d'une répartition homogène. Nous discuterons de l'influence du type de répartition choisi (répartition par rapport aux valeurs ou répartition par rapport aux identifiants). Après avoir fait un survol des techniques classiques de hachage et de leur utilisation potentielle dans notre contexte, nous proposerons une solution simple, adaptable et homogène.

Enfin, dans une dernière partie, nous ferons une synthèse de notre approche et discuterons de son aspect dynamique.

3.2. Importance du degré de répartition dans notre contexte

3.2.1. Point de vue général

D'une façon générale, nous allons nous attacher ici aux problèmes qui concernent la question: "Sur combien de noeuds répartir les objets partiels d'un fragment?". Le but recherché est d'adapter pour chaque fragment[1], ce degré de répartition afin d'obtenir le meilleur rendement possible pour la machine. Il est important de souligner que le fait d'avoir préalablement pratiqué une fragmentation verticale, va nous permettre d'adapter le degré de répartition à un niveau plus fin que celui de l'entité conceptuelle qu'est l'objet. Ainsi un compromis traitements-communications pourra être trouvé pour chaque fragment (ou chaque sous-ensemble) d'une même classe ce qui ne peut être qu'une solution meilleure que d'avoir recours à un compromis "moyen" valable pour toute une classe en l'absence de fragmentation verticale .

Il s'agit donc ici de trouver un compromis entre traitements et communications, problème commun à toute application parallèle, quelqu'en soit le domaine. Pour cela, nous nous basons sur les observations justifiées de Copeland et al [COPE88] qui mettent en évidence le rôle néfaste des opérations pour lesquelles le surplus de communication induit par une répartition des données n'augmente pas linéairement avec le degré de répartition. Rappelons qu'avec une de ces opérations dans un ensemble de requêtes, la courbe du débit en fonction du degré de répartition s'inverse brusquement, mettant ainsi en évidence "un degré optimal" (cf figure suivante).

Il semble donc nécessaire qu'un administrateur dispose d'un outil qui lui permette de déterminer pour chaque fragment (ou chaque sous-ensemble de fragments liés pour une répartition relative de leurs occurrences) un "bon" degré de répartition.

---

1. Cette adaptation pourra se faire comme nous le verrons, au niveau d'un sous-ensemble de fragments d'une classe donnée lorsque nous procéderons à une répartition partiellement relative des objets partiels, point abordé dans la section suivante (cf 3.3 Répartition relative des occurrences des fragments d'une même classe).

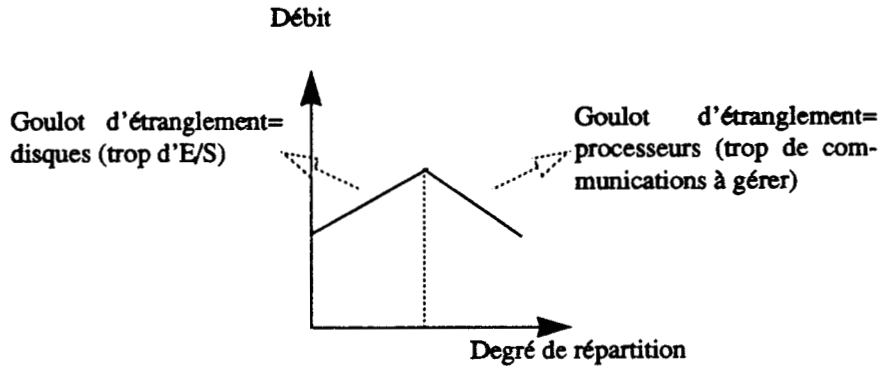


Fig. 3.13 : Influence du degré de répartition sur le débit

Un tel outil peut être schématisé de la façon suivante:

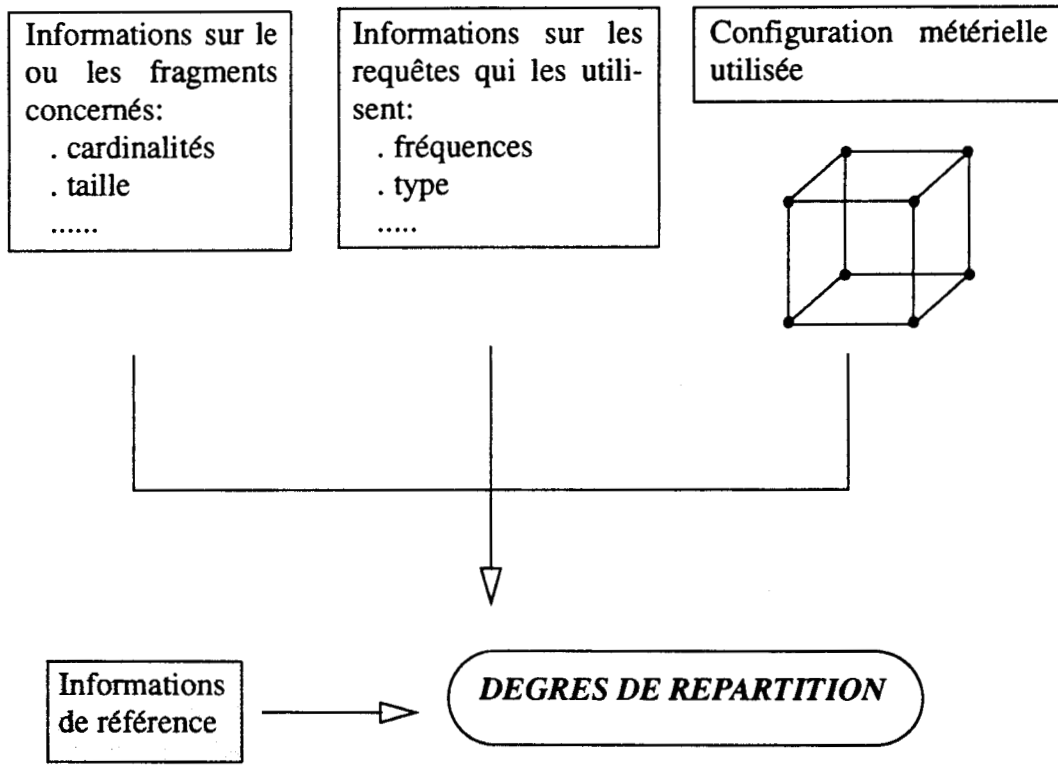


Fig. 3.14 : Principe général de la détermination des degré de répartition

Sa conception soulève deux problèmes. D'une part, la classification des requêtes par rapport à ce problème de détermination d'un degré de répartition et donc le "repérage" des opérations dont l'influence est primordiale. D'autre part, la constitution d'un ensemble d'informations de référence qui permettront, aux vues des requêtes qui utilisent un fragment et de la configuration matérielle disponible, de déterminer un degré de répartition acceptable pour les objets partiels de ce fragment.

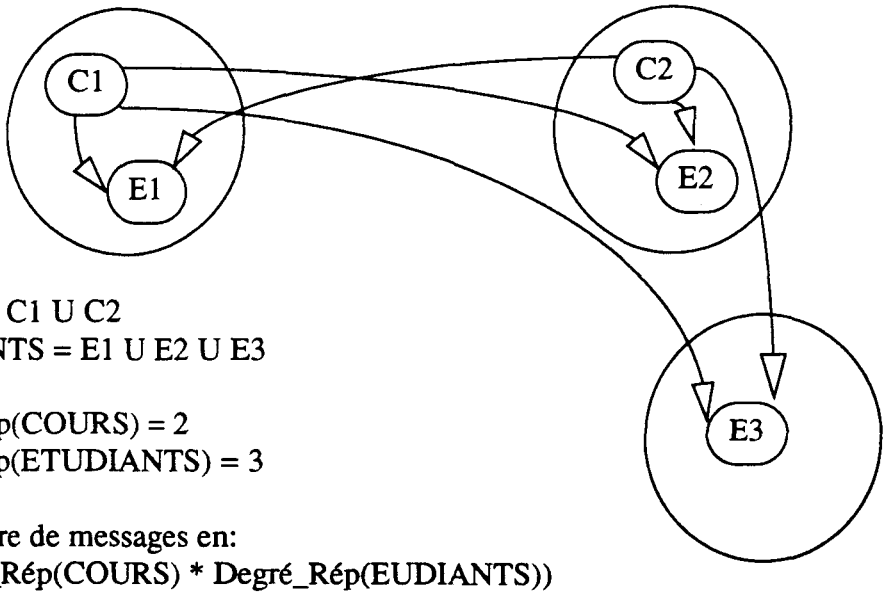
3.2.2. Caractérisation des opérations à risques

Dans un contexte bases de données, les opérations dont les communications n'augmentent pas linéairement avec le degré de répartition, sont celles qui font intervenir deux ensembles de données entre lesquels il existe un lien du type n:m, c'est-à-dire un lien tel que à toute donnée de l'un des ensembles, on puisse "logiquement" associer m données de l'autre, et vice-versa.

L'exemple courant que l'on peut donner d'un tel lien est celui qui règne entre les deux entités ETUDIANTS et COURS. En effet, un cours est suivi par plusieurs étudiants et un étudiant peut suivre plusieurs cours.

Une telle opération nécessite la plupart du temps, le "croisement" des informations des deux ensembles. Dans un contexte réparti, c'est-à-dire lorsque chacun des ensembles est éclaté sur différents noeuds, cela se traduit par la diffusion de l'un des ensembles vers l'autre, ou en d'autres termes, par l'envoi de chaque partie de l'ensemble à diffuser vers toutes les parties de l'autre ensemble.

Si nous reprenons notre exemple précédent du lien n:m ETUDIANTS-COURS et si nous supposons que les données correspondantes sont réparties, nous pouvons schématiser une telle diffusion de la façon suivante:



COURS = C1 U C2
ETUDIANTS = E1 U E2 U E3

Degré_Rép(COURS) = 2
Degré_Rép(ETUDIANTS) = 3

=> Nombre de messages en:
O(Degré_Rép(COURS) * Degré_Rép(EUDIANTS))

Fig. 3.15 : Diffusion de l'ensemble des cours vers l'ensemble des étudiants

De toute évidence, une augmentation linéaire du degré de répartition de chacun des ensembles de données, entraîne une augmentation polynomiale (de degré 2) du nombre de messages[1] qui transitent.

1. Nous considérons ici une notion logique de message, c'est-à-dire qui ne prend pas en compte la taille des "paquets" pour une machine donnée.

Degré_rép(COURS)	Degré_rép(ETUDIANTS)	Proportion/ cas initial	Nb messages	Proportion / nb initial messages
2	3	1	6	1
10	15	5	150	25
20	30	10	600	100
100	150	50	15000	2500

Fig. 3.16 : Conséquence d'une augmentation linéaire des degrés de répartition

Dans le contexte relationnel, ce type de traitement s'exprime explicitement par l'opération de jointure de l'algèbre relationnelle sur laquelle nous ne reviendrons pas. Il s'agit, dans ce contexte, de la seule opération qui peut entraîner une telle augmentation démesurée des communications.

```
SELECT   nom, prenom,
         des_cours
FROM     ETUDIANTS, COURS, INSCRIPTIONS
WHERE    ETUDIANT.num=INSCRPTIONS.num_etud
         AND COURS.code=INSCRIPTIONS.code_cours
```

Fig. 3.17 : Expression de la jointure explicite n:m en SQL

Dans notre contexte, c'est-à-dire celui du modèle de données FAD sur lequel nous pratiquons des fragmentations, mais plus généralement dans celui d'un quelconque modèle de données qui supporte les objets complexes identifiés, nous pouvons discerner deux types d'opérations "à risques": celles qui sont explicitement exprimées par l'utilisateur et celles qui sont induites par les techniques de fragmentation que nous proposons.

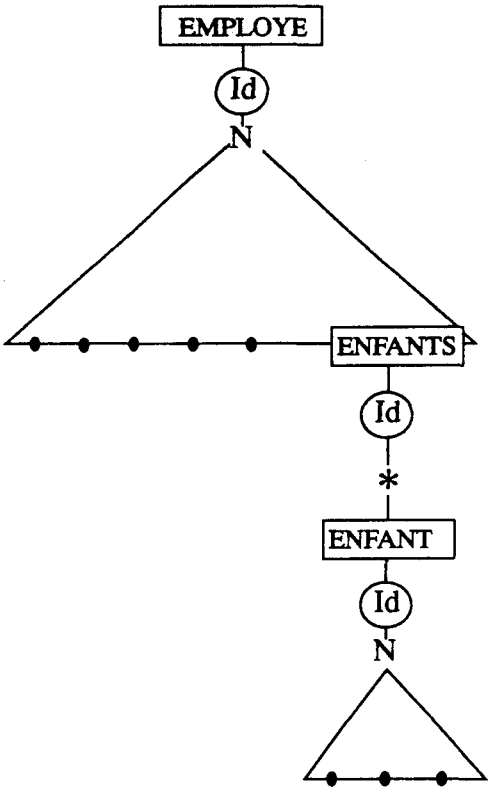
Pour les premières, rien de bien différent des jointures du modèle relationnel. Le principe est le même, seule l'expression change si on fait bien sûr abstraction du modèle d'exécution sous-jacent. Le traitement alors concerné (pour FAD)est le FILTER qui permet d'appliquer une fonction aux éléments du produit cartésien des ensembles passés en paramètres. Bien que son rôle ne se limite pas uniquement à cela, il est le seul à permettre l'expression d'une jointure explicite entre différentes classes d'objets (cf chapitre 1, section 2 relative à FAD).

```
FILTER ( fun (e<etudiant>,c<cours>, i<inscription>)
        is <['nom:strings,'prenom;strings, 'design_cours: strings]
if and ( eq?(e.'num, i.'num_etud), eq?(c.'code, i.'code_cours))
        then ['nom:e.'nom, 'prenom:e.'prenom, 'design_cours:c.'design_cours ],
db.'etudiants, db.'inscriptions, db.'cours)
```

Fig. 3.18 : Expression de la jointure explicite n:m en FAD

Bien que le modèle soit différent, le principe général ou tout au moins le nombre de messages qui transitent est du même ordre. Cela s'explique par le fait que l'on parle toujours du même type d'opération mais vu sous des angles différents.

Par contre, la fragmentation verticale, ou plus précisément le fait de l'avoir prolongée pour intégrer la notion de liens composés-composants (cf chapitre 2 phase 2), ainsi que la notion nouvelle de partage d'objets, vont nous amener à discerner un second type d'opérations à risques. En effet, pour avoir verticalement fragmenté certaines classes, une requête utilisateur pourra se transformer d'une façon interne, c'est-à-dire d'une façon transparente pour l'utilisateur, en une "mauvaise" opération d'un point de vue communications.



Rappelons qu'une telle représentation signifie qu'un objet de la classe ENFANT peut être "transitivement" composant d'un objet de la classe EMPLOYE. Elle relate aussi le fait qu'un tel objet peut être partagé de deux façons différentes par des objets de la classe EMPLOYE: lorsqu'il appartient à deux objets -ensembles différents de la classe ENFANTS (qui sont chacun relatifs à un objet EMPLOYE différent) ou lorsqu'il appartient à un seul de ces objets qui, lui, est partagé par deux objets EMPLOYE différents .

En fait, grâce à la notion de partage d'objets, partage qui ici peut s'appliquer à un double niveau, le lien EMPLOYE: ENFANT est tout à fait exploitable en tant que lien n:m. Il ne s'agit pas d'un lien explicite par rapport à de quelconques attributs (comme il le serait avec le modèle relationnel), mais d'un lien implicite interne qui se fera "le long" des identifiants.

Fig. 3.19 : Exemple de lien composés-composants n:m

Supposons maintenant que la phase 2 de la fragmentation verticale ait conduit à stocker séparément les fragments de EMPLOYE et ENFANT.

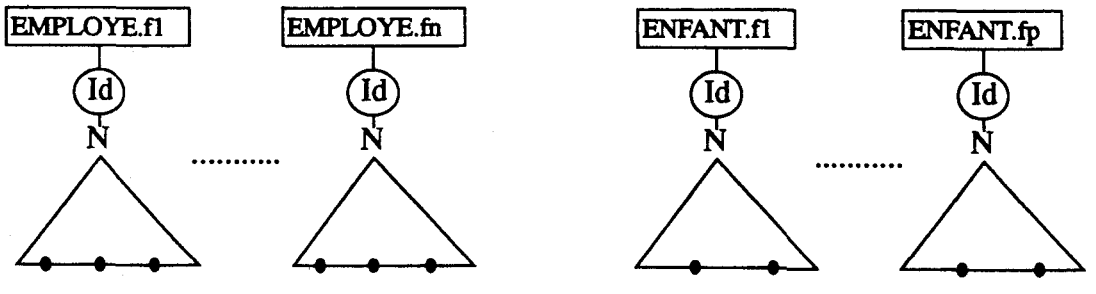


Fig. 3.20 : fragments verticaux normalisés issus de l'exemple

Toute requête qui nécessite la reconstruction d'informations "sur les employés et leurs enfants" se traduit alors par une opération que nous pouvons qualifier de jointure interne, puisqu'elle n'est pas exprimée par l'utilisateur mais est uniquement une conséquence interne de la fragmentation verticale. Une telle opération n'en est pas moins une opération à risques d'un point de vue communications, puisqu'elle prend appui sur un lien n:m.

En conséquence, lorsqu'aucun mécanisme particulier n'est mis en oeuvre pour gérer le partage d'objets [1], toute opération qui utilise simultanément des composés et leurs composants stockés séparément, risque d'avoir des conséquences néfastes sur le bon fonctionnement de la machine. Ces opérations ne se limitent pas, dans notre cas, au traitement FILTER puisque d'autres traitements tels que le PUMP, MATCH, etc (cf 1.4.2 Présentation de FAD chapitre 1), peuvent alors se traduire par le même problème.

Pour le choix d'un bon degré de répartition, la détermination des opérations à risques ne pourra se contenter d'une analyse "externe" des requêtes utilisateurs, c'est-à-dire de ce qui y est exprimé explicitement. Elle devra tenir compte des résultats de la fragmentation verticale pour caractériser le comportement interne de ces requêtes et ainsi les identifier de façon juste par rapport au problème du degré de répartition.

Pour parvenir à ce résultat dans le contexte précis du langage FAD, nous pourrions utiliser la forme ré-écrite des programmes ou requêtes. Nous présentons en détail cette phase de ré-écriture dans le quatrième chapitre qui concerne entre autres l'adaptation du modèle d'exécution de FAD proposé par l'équipe du MCC [HART88] [DANF88b], pour qu'il prenne en compte les fragmentations que nous préconisons. En fait, elle consiste à traduire en FAD+ [2] les requêtes utilisateurs afin de les exprimer par rapport aux fragments verticaux et horizontaux déterminés. C'est donc à ce niveau qu'une juste analyse des conséquences de la répartition des données pourrait se faire.

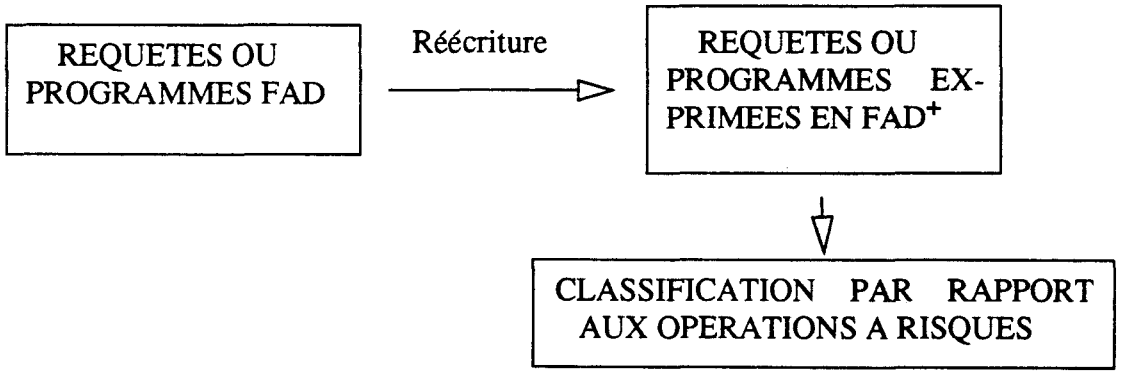


Fig. 3.21 : Prise en compte des opérations à risques dans le cas de FAD

Nous n'irons pas plus loin dans le développement de ce point précis relatif à la caractérisation des opérations à risques. Une telle présentation n'aurait de réel intérêt que pour un contexte précis (modèle de données, langage de programmation et modèle d'exécution connus).

---

1. Nous entendons par là un mécanisme de bas niveau qui permettrait de savoir si un objet est partagé ou non, et qui éviterait ainsi des jointures globales de type n:m. Notre objectif n'est pas ici de décrire de tels mécanismes qui relèvent d'une implantation interne du SGBD.
2. Ce nouveau langage FAD+ n'est autre que le langage FAD dans son intégralité, auquel nous adjoignons des opérations qui permettent de manipuler les identifiants comme des valeurs. Nous reviendrons en détail sur ce point dans le chapitre 4.

Notre objectif se limite ici à mettre en évidence les incidences que peut avoir la fragmentation verticale d'objets complexes, sur le déroulement des requêtes ou programmes.

3.2.3. Problèmes relatifs à la détermination des informations de référence

Nous allons donc nous intéresser ici aux problèmes liés à l'obtention d'un ensemble d'informations de référence, informations qui permettront, en fonction de l'analyse des requêtes (cf 3.2.2) et des informations générales propres à un fragment, de proposer un degré de répartition acceptable et adapté à la configuration matérielle choisie.

Globalement, nous pouvons définir deux démarches pour répondre à ce problème. La première consiste en l'utilisation d'un modèle de simulation et fournit un ensemble de prédictions sur le débit de la machine relativement à différents placements de données, ie différents degrés de répartition, et ce, pour les différentes catégories de requêtes discernées. La seconde, consiste à réaliser pour chaque placement une évaluation des coûts induits, le but du jeu étant alors de minimiser ces coûts. Dans ce qui suit nous présentons tout d'abord l'exemple du modèle de simulation utilisé dans le projet BUBBA. Nous ferons ensuite une comparaison générale entre l'approche modèle de simulation et l'approche évaluation de coûts. Enfin, nous reviendrons dans notre contexte pour discuter de la stratégie à adopter et des travaux futurs qu'elle sous-entend.

3.2.3.1. La solution BUBBA

Dans cette solution, les concepteurs ont choisi comme mesure d'évaluation de leur stratégie de placement, le débit en transactions. Pour estimer ce débit, ils ont implanté un outil de modélisation appelé FIRM et dont une description détaillée peut être trouvée dans [BOUG87].

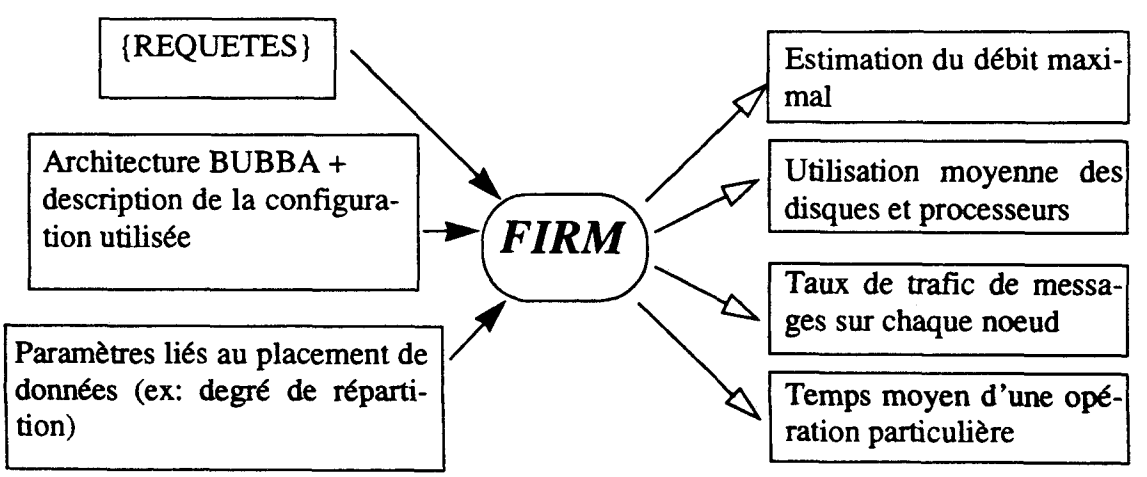


Fig. 3.22 : Principe de l'outil de simulation FIRM

FIRM consiste en trois programmes: le premier fait le placement des données, le second adapte la charge de travail aux différents noeuds et le troisième résoud le modèle analytique de files d'attente qui en résulte. La charge de travail se résume pour leurs essais en un ensemble représentatif de requêtes. Nous pouvons schématiquement représenter leur approche par la figure 3.22.

Puisque les contraintes qui s'y rapportent en font un problème NP-complet, le placement des données (pour un degré de répartition fixé) se fait par l'utilisation d'heuristiques qui mettent en jeu des notions de localité présentées sous le nom de "chaleur" et "température" [COPE88].

Leur outil est général et ne s'arrête pas bien sûr à la caractérisation de bons degrés de répartition. Néanmoins, ils avouent obtenir des résultats forcément optimistes et sont tout à fait conscients de l'incertitude de telles prédictions.

3.2.3.2. Comparaison modèles de simulation / approches par évaluation de coûts.

En fait, le problème du placement optimal des données sur une machine parallèle peut être analysé de deux façons suivant le sens que l'on donne au mot optimal:

La première approche est de considérer comme optimal le placement qui minimise un ensemble de coûts tels que les coûts de stockage, de recherche, de mise à jour, de communications, etc. Le recours à une fonction objective qui "résume" ces coûts est dans ce cas nécessaire. Cette solution se traduit par un problème d'optimisation qui n'est généralement pas calculable d'une façon linéaire. Même pour de petits exemples, l'obtention de solutions précises est "prohibitivement" coûteuse. Les performances du système et les seuils relatifs à ses capacités sont souvent incluses comme contraintes. Par contre, les délais d'attente ne sont généralement pas pris en compte! Pour simplifier le problème on peut envisager des solutions basées sur l'utilisation d'heuristiques. Ces dernières sont néanmoins difficiles à évaluer en terme d'efficacité puisque la distance qui les sépare de la solution optimale est inconnue.

La seconde approche utilisable est celle qui consiste à juger comme optimal un placement qui maximise les performances, c'est à dire qui offre des temps de réponse minimaux et un débit (en transactions) maximal. Pour cela on prend cette fois en compte les temps d'attente et les capacités de routage alternatif du réseau. Par contre, les paramètres liés aux coûts sont considérés comme fixes. Le système est alors modèlisé par un réseau de files d'attente. La borne supérieure des performances est dans ce cas connue grâce aux paramètres tels que la vitesse des organes matériels, les taux de consultation, etc. En conséquence, des heuristiques peuvent alors être évaluées en terme d'efficacité puisque la distance qui les sépare de la solution optimale est calculable.

En fait, ces deux approches sont plus complémentaires que réellement exclusives. Elles peuvent intervenir à des moments différents de la vie d'une machine bases de données. Les fonctions de minimisation des coûts ont plutôt leur place dans la phase de conception de la machine, leur rôle étant de proposer une configuration initiale acceptable. D'ailleurs, durant une telle phase, l'obtention de bonnes performances peut paraître secondaire puisque la connaissance de la charge de travail est, à ce moment là, partielle. Par contre, dans la phase de production, l'augmentation des performances par une adaptation de la charge, prend toute sa valeur. Les coûts sont alors secondaires puisqu'il s'agit là de faire au mieux avec la configuration dont on dispose.

Cette façon dichotomique et usuelle d'aborder le problème n'en reste pas moins frustrante. D'une part les calculs ardus de minimisation de coûts négligent totalement l'aspect dynamique (temps d'attente, routage,etc). D'autre part, les modèles de simulation qui oeuvrent en faveur d'une maximisation des performances considèrent la plupart des coûts comme constants.

Pour W. Dowdy la cause de ce dilemme se résume par le fait que les systèmes informatiques sont si complexes qu'aucune analyse détaillée du comportement de tous les composants à tout moment, n'est envisageable. Dans son article "Comparative models for File assignment Problem" [DOWD82], cette dernière présente une comparaison entre les principales solutions proposées pour la résolution du problème d'assignation de fichiers sur un réseau de processeurs. Bien que le problème soit différent, puisqu'il s'agit là d'assigner les fichiers sans les éclater, la démarche comparative est tout à fait intéressante puisqu'elle met bien en évidence l'inexistence (pour le moment) d'une solution miracle qui prend en compte tous les paramètres.

3.2.3.3. Quelle stratégie dans notre contexte?

Revenons-en à notre problème initial qui était de déterminer de "bonnes" informations de référence pour l'obtention ultérieure de degrés de répartition adaptés. La solution que nous envisageons est une adaptation de l'outil de modèlisation FIRM [BOUG87] pour qu'il puisse prendre en compte nos travaux sur la fragmentation verticale ainsi que les différents éléments de la fragmentation horizontale que nous présentons dans ce qui suit. Un tel travail ne se limitera pas bien sûr à la collecte d'informations sur les degrés de répartition.

Rappelons que nos travaux traitent du problème de placement de données avant tout sous un angle dynamique. Ils proposent en effet des solutions en vue d'obtenir une meilleure utilisation de la machine, ie de meilleures performances en phase de production. En ce sens, la simulation nous semble donc être plus adaptée.

Enfin, il est important de souligner que l'exhibition de telle ou telle information de référence n'a de sens que dans le contexte d'une machine précise, chose dont nous ne disposons pas et vis à vis de laquelle nous voulons rester ici assez indépendants. Notre souci est plus ici de mettre en avant les problèmes dans un contexte général et de proposer des solutions adaptables, que de conclure que la répartition de 100000 commandes et 1000 clients ne doit pas se faire sur plus de 113 disques!

Hormis la solution de facilité qui consiste à adapter FIRM, une approche plus approfondie des simulations à effectuer pourrait être faite à partir de l'analyse et des conclusions de W. Dowdy [DOWD82] qui constituent une excellente synthèse sur un sujet très proche. Nous reviendrons sur cette éventualité dans notre conclusion.

3.3. Répartition relative des occurrences des fragments d'une même classe

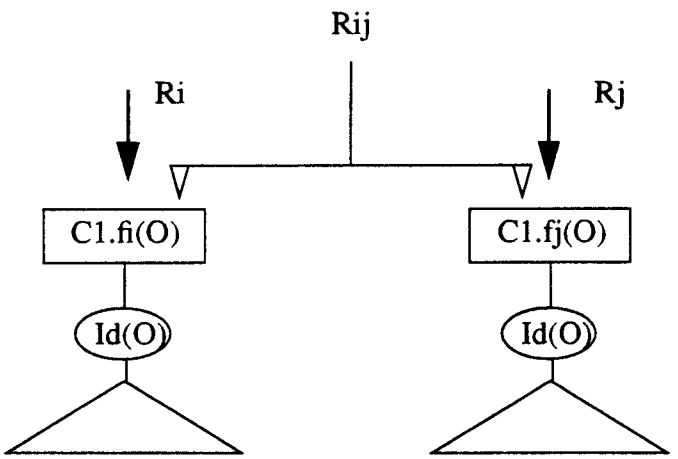
3.3.1. Présentation du problème

Nous allons nous intéresser ici aux problèmes relatifs à la répartition des objets par-

tiels libres d'une classe C. A priori, puisque la fragmentation verticale consiste à fabriquer, pour une classe, des sous-ensembles de données relativement indépendants (d'un point de vue utilisation), nous pourrions être tentés de conclure que les occurrences d'un fragment peuvent être réparties indépendamment de celles des autres fragments. Cependant, entre le cas de deux fragments totalement indépendants (jamais utilisés simultanément par une même requête) et celui de fragments issus d'une décision prise de justesse (cas limite de fragmentation), se situent beaucoup de nuances que nous nous proposons de traiter de façon adaptée.

En effet, deux fragments non totalement indépendants sont forcément utilisés simultanément par une même requête. Bien que cette utilisation ne soit pas prédominante, elle est d'autant plus importante que nous nous approcherons du cas limite de fragmentation. En conséquence, il semble intéressant de pouvoir garder une certaine proximité physique entre les objets partiels d'un même objet afin de favoriser l'exécution de ce type de requêtes.

Alors que l'existence de fragments verticaux favorise les requêtes qui les utilisent séparément (moins d'accès inutiles, parallélisme inter-requêtes au travers du parallélisme intra-objet), une bonne gestion de la proximité entre ces fragments (ie entre les objets partiels d'un même objet occurrence de ces fragments) pénalisera moins celles qui les accèdent simultanément, ce que nous résumons dans la figure ci-dessous.



Ri, Rj requêtes majoritaires qui accèdent séparément les objets partiels

Rij requêtes minoritaires qui accèdent simultanément les objets partiels

Existence des fragments => Ri, Rj favorisées

Gestion de la proximité physique => Rij moins pénalisées

Fig. 3.23 : Intérêt de la gestion de la proximité physique des fragments d'une même classe

Le problème de la gestion de la proximité peut se résumer par deux questions: Premièrement, pouvons-nous déterminer une mesure d'éloignement logique entre les fragments concernés, c'est à dire entre les objets partiels libres d'un même objet? Secondement, comment devons nous utiliser cette mesure logique afin d'établir une notion de proximité physique adaptée à un réseau donné?

Dans les pages qui suivent, nous allons nous efforcer de répondre à ces questions en analysant de près les incidences des solutions proposées. Dans un premier temps, nous proposerons une fonction afin de déterminer une mesure d'éloignement entre deux fragments libres quelconques d'une même classe. Nous étudierons les variations de cette fonction pour nous assurer qu'elle répond bien aux besoins. Nous rapprocherons ensuite cette fonction de cel-

les utilisées dans le processus de fragmentation verticale et énoncerons alors certaines propriétés. Nous verrons enfin comment utiliser ces mesures d'éloignement pour la répartition des objets partiels sur un réseau physique donné.

3.3.2. Evaluation d'un éloignement inter-fragments

3.3.2.1. But

Nous cherchons donc à obtenir une information quantitative qui mesure un écart, un éloignement pour tout couple de fragments libres d'une même classe. Etant donné que les fragments ont été construits par rapport à une utilisation logique des données, nous chercherons, dans un premier temps, à déterminer une mesure logique c'est-à-dire qui sera directement liée à cette utilisation logique des données. Cette mesure devra être d'autant plus importante que les fragments concernés sont moins liés, c'est-à-dire qu'ils sont plus souvent accédés individuellement.

Caractérisation d'un couple de fragments

Si nous nous référons aux phases de fragmentation verticale, tout couple de fragments (fi,fj) peut être caractérisé par une sous matrice d'affinité ainsi que le montre la figure ci-dessous.

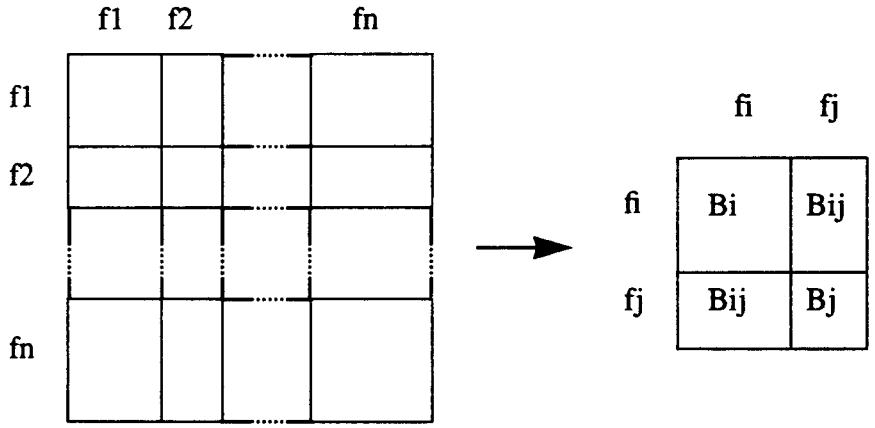


Fig. 3.24 : Sous-matrice d'affinité relative à deux fragments

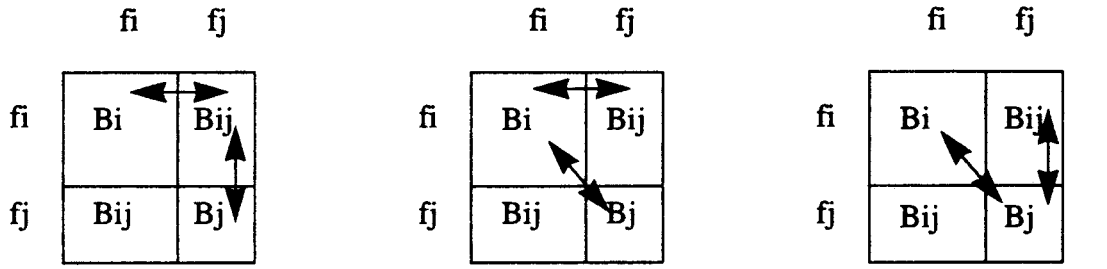
Afin de mettre en valeur la fragmentation la plus intéressante d'une classe, nous avons proposé dans la phase 1 de prendre en compte deux mesures: d'une part, les accès supplémentaires engendrés par la fragmentation et, d'autre part, les accès inutiles qui subsistent après cette fragmentation (cf 2.2.4 Production récursive de fragments). En conséquence, nous pouvons dire qu'une telle fragmentation tient compte à la fois de ce qui se passe entre les fragments (accès supplémentaires)et de ce qui se passe dans chaque fragment (accès inutiles).

Nous voulons maintenant quantifier ce qui lie deux fragments donnés d'une même classe.[1] Pour cela, nous nous intéresserons uniquement à ce qui se passe entre ces fragments.

1. A ce niveau il ne s'agit en aucun cas de remettre en cause les fragments proposés.

En effet, le comportement interne de chacun d'eux n'est d'aucune importance et ne doit modifier en aucun cas leur niveau de dépendance. Nous proposons d'utiliser la sous-matrice d'affinité qui caractérise un couple de fragments afin d'obtenir une mesure d'éloignement entre eux. Pour cela, nous allons étudier les influences respectives des blocs de valeurs Bi, Bj et Bij.

Intuitivement nous pouvons définir trois types de fonctions pour analyser ces influences:



TYPE 1: Etude influence Bi<->Bij et influence Bij<->Bj

TYPE 2: Etude influence Bi<->Bij et influence Bi<->Bj

TYPE 3: Etude influence Bi<->Bj et influence Bij<->Bj

Fig. 3.25 : Représentation des différents types de fonctions d'analyse d'une sous-matrice

Remarque: L'étude de toutes les influences possibles pourrait constituer un quatrième type de fonction; cependant, une telle fonction fournirait inutilement des résultats que nous pourrions déduire par transitivité.

La fonction que nous proposons dans ce qui suit est du type 1. Néanmoins, nous avons réalisé des essais avec des fonctions des autres types qui se sont comportées de la même façon.

3.3.2.2. La fonction proprement dite

Comme nous l'avons souligné lors de la phase 1 (cf 2..2.4 Production récursive de fragments) les valeurs du bloc Bij associé à tout couple de fragments, correspondent aux accès supplémentaires engendrés par l'existence de ces deux fragments. Vis à vis du bloc Bi [1] ces accès supplémentaires peuvent varier de 0 à une valeur maximale ACC_SUP_Bi définie par:

$$ACC\_SUP\_Bi = (n-k) \times \sum_{x=1}^{k} a_{xx}$$

En effet, de par la définition des coefficients d'affinité, les valeurs dans Bij sont forcément inférieures ou égales aux valeurs diagonales de Bi. Le bloc Bij maximal vis à vis de Bi est donc celui obtenu en remplaçant les a_{ij} par a_{ii}.

---

1. Nous nous plaçons par rapport au bloc Bi puisque nous cherchons à étudier l'influence entre Bi et Bij indépendamment du bloc Bj.

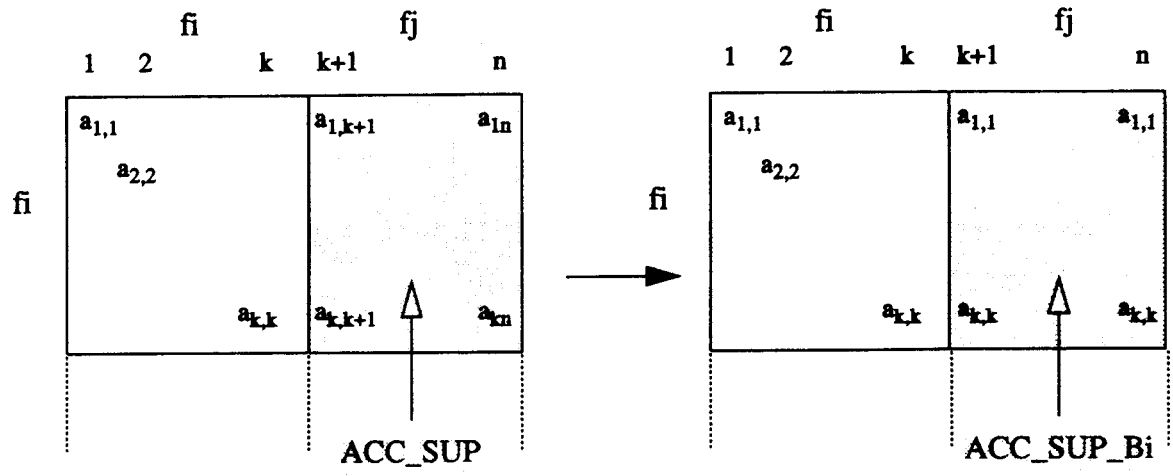


Fig. 3.26 : Principe du calcul de ACC_SUP_Bi

Dès lors, le rapport ACC_SUP / ACC_SUP_Bi mesure un pourcentage d'accès supplémentaires vis à vis du fragment fi. Il mesure, en fait, la part respective de l'utilisation de fi qui se fait simultanément avec fj, ce que nous pourrions comparer à une force d'attraction entre les blocs Bi et Bij. Plus il est proche de 1, plus ces blocs sont dépendants ou, en d'autres termes, plus fi est utilisé simultanément avec fj plutôt qu'indépendamment.

Le rapport inverse Ri = ACC_SUP_Bi / ACC_SUP fournit donc une information quantitative de répulsion entre Bi et Bij. Il est d'autant plus important que fi est moins utilisé de façon simultanée avec fj. Par conséquent, il dénote une information partielle d'éloignement entre les fragments fi et fj, information partielle puisqu'elle ne tient pas compte du "comment" de l'utilisation de fj. Ce rapport Ri varie de 1, cas où le nombre d'accès supplémentaires est maximal par rapport à Bi (utilisation maximale de fi en simultané <=> éloignement minimal entre fi et fj), à l'infini, cas ou il n'y a aucun accès supplémentaire (utilisation nulle de fi en simultané <=> éloignement infini entre fi et fj).

$$Ri = \frac{ACC\_SUP\_Bi}{ACC\_SUP}$$

Avec un raisonnement analogue vis-à-vis du bloc Bj, nous pouvons déterminer une information quantitative de répulsion entre Bj et Bij; le rapport Rj obtenu est d'autant plus important que fj est utilisé simultanément avec fi plutôt qu'indépendamment. Il représente une seconde information partielle d'éloignement entre fi et fj qui tient compte cette fois uniquement du "comment" de l'utilisation de fj.

$$ACC\_SUP\_Bj = k \times \sum_{y=k+1}^{n} a_{yy}$$

$$Rj = \frac{ACC\_SUP\_Bj}{ACC\_SUP}$$

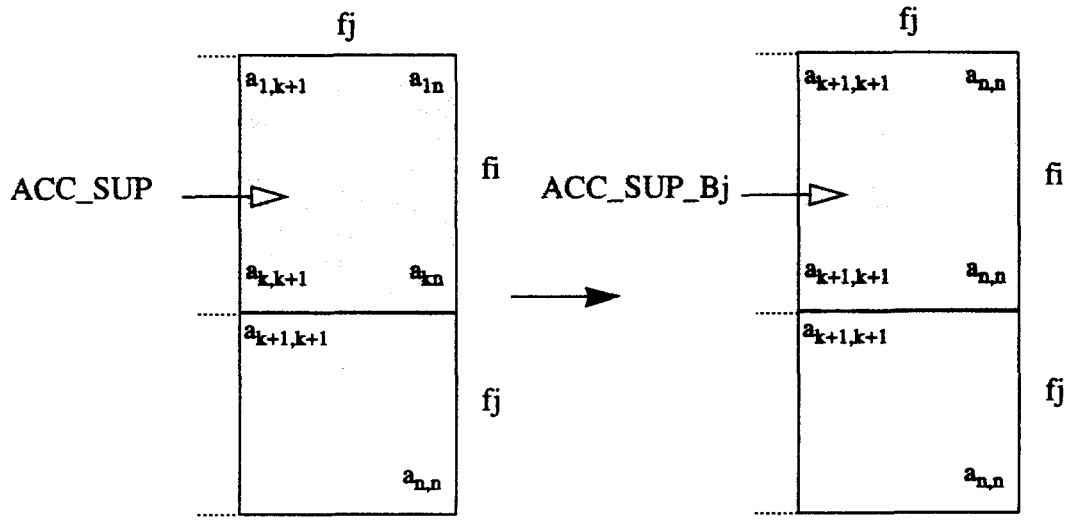


Fig. 3.27 : Principe du calcul de ACC_SUP_Bj

Les deux rapports Ri et Rj ainsi obtenus nous permettent donc de quantifier les influences entre Bi et Bij et entre Bij et Bj.

Nous proposons d'utiliser la fonction suivante afin d'obtenir un éloignement entre deux fragments fi et fj:

$$E(fi,fj) = Ri * Rj$$

Avec une telle fonction, l'éloignement entre deux fragments sera directement proportionnel à l'utilisation de chaque fragment. Il doublera, par exemple,dès lors que l'un des fragments sera utilisé deux fois plus de façon indépendante. Nous aurions pu utiliser une fonction du type e2(fi,fj) = Ri + Rj. Il nous semble cependant qu'une telle fonction traduise moins bien le comment de l'utilisation des fragments.

Prenons par exemple le cas simpliste ci-dessous représenté par une sous-matrice à quatre valeurs:

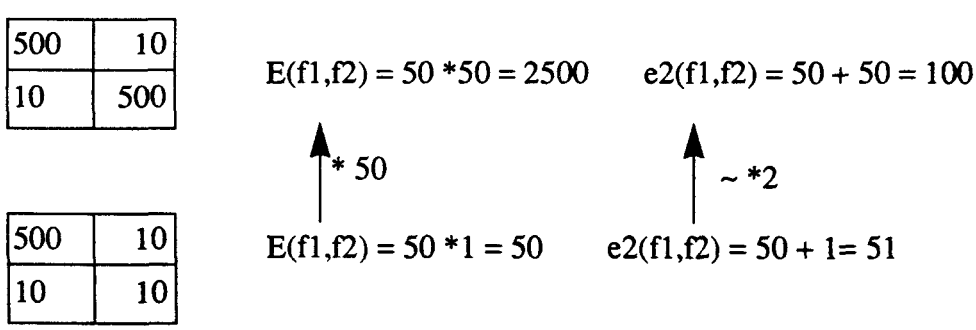


Fig. 3.28 : Comparaison sur un exemple entre deux fonctions d'éloignement

D'une configuration à l'autre e2(f1,f2) n'est divisé que par 2 alors que l'utilisation

de f2 est 50 fois plus favorable. Par contre E(f1,f2) reflète directement l'évolution de cette utilisation de f2.

Remarque: L'utilisation de E(fi,fj) plutôt que e2(fi,fj) va fournir des mesures d'éloignement plus dispersées. D'autre part, ces deux types de fonctions appartiennent à une même famille ce qui leur confère, comme nous le verrons par la suite, des propriétés semblables.

Nous allons maintenant étudier les variations de la fonction proposée afin de nous assurer qu'elle répond effectivement à nos besoins.

3.3.2.3. Etude des variations de la fonction utilisée

. Propriétés générales

$$E(fi,fj) = \frac{ACC\_SUP\_Bi}{ACC\_SUP} \times \frac{ACC\_SUP\_Bj}{ACC\_SUP} = Ri \times Rj$$

L'éloignement est infini en l'absence d'accès supplémentaire, c'est à dire lorsque les fragments sont totalement indépendants. Sur ce point, la fonction mathématique utilisée traduit bien l'idée intuitive que nous avions sur la notion d'éloignement entre fragments.

L'éloignement minimal théorique (valeur 1) est obtenu lorsque les blocs Bi et Bj sont égaux du point de vue valeurs diagonales et que Bij est maximal par rapport à ces valeurs. Nous donnons ci-dessous un exemple d'une telle configuration. Cependant, à ce niveau, nous pouvons nous demander si ce cas est envisageable ou plus précisément si la phase 1 de la fragmentation verticale peut nous fournir de tels fragments. Nous aborderons ce point dans la section "Liaison éloignement-fragmentation verticale" (cf 3.3.3.2).

1	2	k	k+1	n
x	y_{12}	y_{1k}	x	x
	x	y_{2k}		
y_{k1}	y_{k2}	x	x	x
x		x	x $y_{..}$	$y_{..}$
			$y_{..}$ x	$y_{..}$
x		x	$y_{..}$ $y_{..}$	x

Fig. 3.29 : Configuration qui donne l'éloignement minimal théorique

Enfin, l'éloignement ne tient pas compte de ce qui se passe à l'intérieur de chaque fragment puisqu'il n'utilise que les valeurs diagonales des blocs Bi et Bj qui sont les seules significatives dès lors que la fragmentation fi-fj est admise.

. Variations des valeurs dans Bij

Nous allons étudier ici les variations de l'éloignement en fonction des valeurs du bloc Bij pour des blocs Bi et Bj fixés ou, en d'autres termes, en fonction des accès supplémentaires.

Pour deux blocs Bi et Bj donnés, nous pouvons faire varier le nombre d'accès supplémentaires de 0 à ACC_SUP_MAX définie par:

$$ACC\_SUP\_MAX = \sum_{x=1}^{k} \sum_{y=k+1}^{n} \min(a_{xx}, a_{yy})$$

En effet, de par la définition des coefficients d'affinité, toute valeur a_{ij} de Bij est inférieure ou égale aux deux valeurs diagonales a_{ii} et a_{jj}.

Le rapport α=ACC_SUP / ACC_SUP_MAX représente donc le pourcentage d'accès supplémentaires effectifs pour les fragments fi et fj. Il varie de 0 à1. Nous nous proposons d'étudier les variations de l' éloignement en fonction de ce rapport α. Pour cela, nous donnons ci-dessous une expression équivalente de la fonction d'éloignement:

$$E(fi,fj) = \frac{ACC\_SUP\_Bi \times ACC\_SUP\_Bj}{ACC\_SUP\_MAX^2} \times \frac{1}{\alpha^2} = E_0 \times \frac{1}{\alpha^2}$$

L'éloignement minimal qui correspond à la constante E_0 est obtenu pour α=1, c'est-à-dire pour le nombre maximal d'accès supplémentaires. Cet éloignement minimal dépend, bien entendu, des blocs fixes Bi et Bj. Il est important de noter qu'il joue le rôle de coefficient pour la fonction hyperbolique obtenue. En conséquence, au plus il sera important, au plus les variations de l'éloignement seront "rapides".

Nous donnons ci-dessous un pseudo tableau de variation ainsi qu'un exemple de courbes obtenues. Sur la figure, chacune des courbes représente un cas différent, c'est-à-dire des bloc Bi et Bj différents.

α	0	1/4	1/2	3/4	1
E()	∞	$16*E_0$	$4*E_0$	$16/9*E_0$	E_0

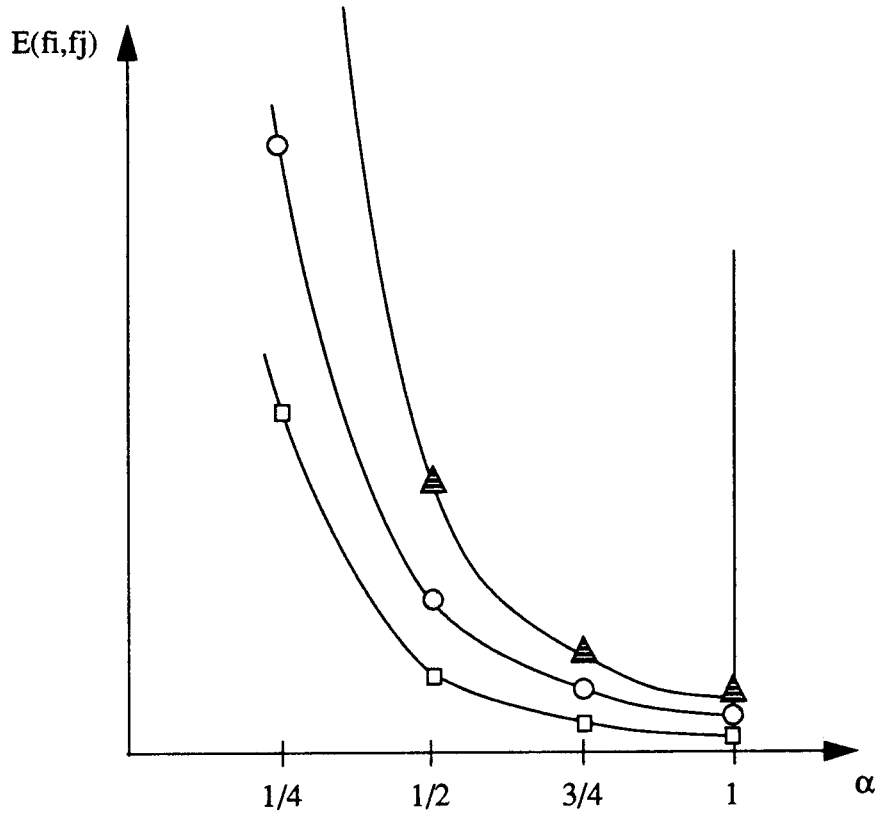


Fig. 3.30 : Variations de l'éloignement en fonction du taux effectif d'accès supplémentaires

En conclusion, pour deux blocs Bi et Bj fixes, c'est-à-dire pour deux fragments pour lesquels nous ne modifions pas les accès indépendants, l'éloignement sera d'autant plus important que les valeurs de Bij seront faibles, ie que ces fragments seront moins accédés simultanément. Là encore, la fonction répond donc bien à nos attentes.

. Variations des valeurs de Bi

Nous allons nous intéresser ici à la variation des valeurs d'un des deux blocs diagonaux, en l'occurrence Bi, ou plus précisément à la variation des valeurs diagonales de ce bloc qui sont les seules significatives. En d'autres termes, nous allons faire varier la part de l'utilisation de fj qui se fait de façon indépendante (ie non simultanée). Nous supposerons donc ici que Bj et Bij sont fixes.

Chaque valeur diagonale de Bi peut varier d'une valeur minimale v_{min} à l'infini. En effet, de par la définition des coefficients d'affinité, toute valeur diagonale est maximale sur sa ligne et sur sa colonne ce qui nous permet de définir v_{min} par:

$$v_{min}(a_{xx}) = \min_t(a_{xt}) \quad \text{pour t variant de k+1 à n}$$

Si nous faisons varier de façon linéaire et homogène ces valeurs diagonales de Bi

(même variation linéaire appliquée à chaque valeur au même moment), seul le rapport Ri varie puisque Rj dépend des blocs Bij et Bj qui sont fixes. En fait ,seul ACC_SUP_Bi va varier. Ceci nous amène à exprimer l'éloignement entre fi et fj avec l'équation équivalente suivante:

$$E(fi,fj) = \frac{ACC\_SUP\_Bj}{ACC\_SUP} \times \frac{ACC\_SUP\_Bi}{ACC\_SUP} = E_0 \times Ri = E(Ri)$$

Pour une variation linéaire et homogène des valeurs diagonales, la variation de Ri sera linéaire, comme celle de ACC_SUP_Bi . L'éloignement variera donc de façon linéaire depuis une valeur minimale $Emin=E_0*Ri\_min$ jusqu'à l'infini. La vitesse de variation (pente de la droite) sera la constante E_0 qui dépendra des blocs fixes Bj et Bij.

Nous donnons ci-dessous un exemple de courbes obtenues pour trois cas différents de blocs Bj et Bij; l'axe des abscisses représente le rapport entre les valeurs diagonales effectives et les valeurs diagonales minimales.

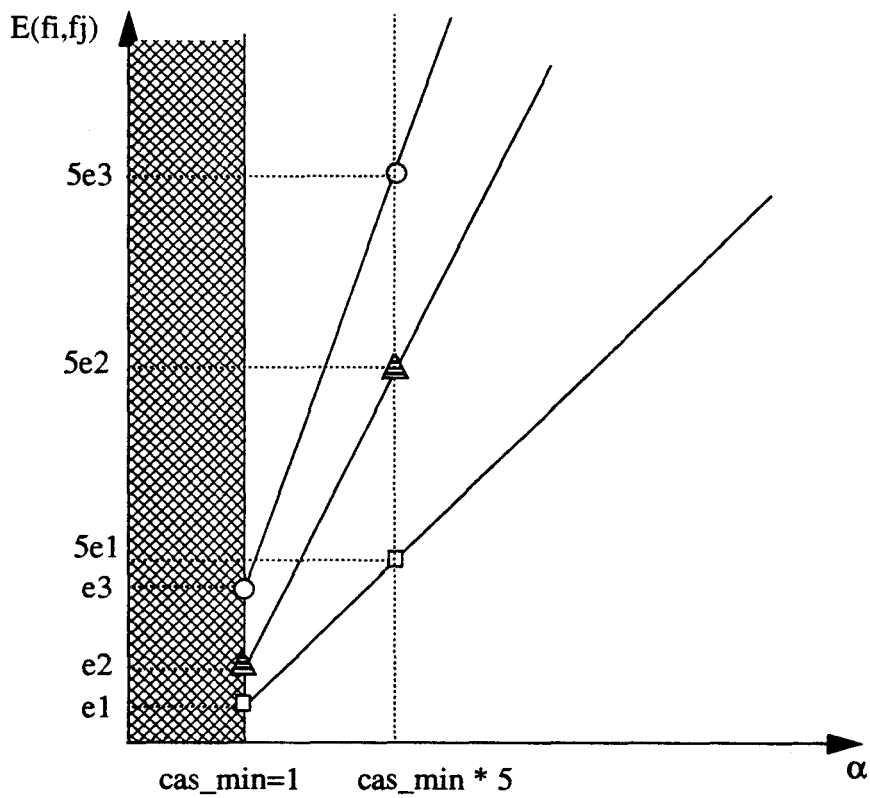


Fig. 3.31 : Variations de l'éloignement en fonction du taux d'utilisation indépendante de l'un des deux fragments

En conclusion, l'éloignement sera d'autant plus important que le fragment fi sera utilisé de façon indépendante. D'autre part, la variation sera d'autant plus rapide que l'éloignement partiel fixe (<=> influence Bj<-> Bij) sera grand.

3.3.3. Liaison éloignement / fragmentation verticale

Nous avons proposé d'utiliser une certaine fonction afin de quantifier l'éloignement entre deux fragments libres quelconques d'une même classe. Nous proposons maintenant de rapprocher cette fonction d'éloignement avec les fonctions qui servent à la construction des fragments lors de la phase 1.

Bien que les notions de fragmentation verticale et d'éloignement ne reposent pas sur les mêmes principes et avant tout n'aient pas les mêmes objectifs, elles utilisent le même support qu'est la matrice d'affinité; à ce titre, il nous semble intéressant d'énoncer quelques règles qui lient ces deux notions.

3.3.3.1. Meilleure fragmentation n'implique pas éloignement maximal

Pour un bloc donné, la phase 1 détermine une meilleure fragmentation binaire, c'est-à-dire propose deux fragments qui entraîneront le moins d'inconvénients possibles (combinaison linéaire d'accès inutiles et d'accès supplémentaires). A priori, il semblerait logique qu'à cette meilleure fragmentation corresponde la meilleure mesure d'éloignement.

Il faut cependant se souvenir que la notion de fragmentation s'appuie sur ce qui se passe dans les fragments (accès inutiles) et ce qui se passe entre les fragments (accès supplémentaires). Par contre, la notion d'éloignement inter-fragments ne s'occupe que de ce qui se passe entre les fragments (accès supplémentaires). Les notions ne sont donc que partiellement liées au travers des accès supplémentaires; en conséquence, il peut exister une moins bonne fragmentation qui donne un éloignement plus grand.

Nous donnons ci-dessous un exemple tiré des simulations réalisées. En attribuant le même poids aux accès inutiles et aux accès supplémentaires, le meilleur découpage s'avère être le deuxième. Cependant ce n'est pas celui qui donne le plus grand éloignement entre les fragments obtenus. En effet, l'éloignement obtenu avec le troisième découpage est plus grand. A ce troisième découpage correspond un nombre important d'accès inutiles qui sont pris en compte pour la décision de fragmentation mais pas pour le calcul de l'éloignement, d'où le résultat.

phase 1

100	90	0	10
90	100	10	0
0	10	10	0
10	0	0	10

100	90	0	10
90	100	10	0
0	10	10	0
10	0	0	10

Découpage	1	2	3
Accès inutiles	310	40	220
Accès sup.	100	20	10
"Inconvénients"	410	60	230
ACC_SUP_Bi	300	400	210
ACC_SUP_Bj	120	40	30
Eloignement	4,2	22	24

Fig. 3.32 : Meilleure fragmentation et éloignement minimal

Le poids attribué aux accès supplémentaires est bien sûr d'une grande importance dans cette liaison partielle entre la fragmentation verticale et l'éloignement entre fragments. En effet, plus il sera important, plus la meilleure fragmentation et l'éloignement maximal auront des chances de coïncider.

Remarque: Dans les simulations réalisées, il était peu fréquent que l'éloignement ne soit pas maximal pour la fragmentation proposée. De façon générale, l'éloignement obtenu pour la meilleure fragmentation était toujours au moins du même ordre de grandeur que l'éloignement maximal.

3.3.3.2. Fragments et éloignement minimal théorique

Nous allons tout d'abord nous intéresser au cas de deux fragments adjacents dans la matrice d'affinité; nous généraliserons ensuite à deux fragments quelconques d'une même classe.Comme nous l'avons souligné précédemment (cf propriétés générales de la fonction proposée), l'éloignement minimal théorique, valeur 1, ne peut être obtenu qu'avec la configuration représentée ci-dessous. Dans ce cas en effet, chacun des rapports Ri et Rj vaut 1, valeur minimale théorique, ce qui aboutit forcément à l'éloignement minimal théorique.

1	2	k	k+1	n
x	$y_{1,2}$	$y_{1,k}$	x	x
	x	$y_{2,k}$		
$y_{k,1}$	$y_{k,2}$	x	x	x
x		x	x $y_{.,.}$ $y_{.,.}$	
			$y_{.,.}$ x $y_{.,.}$	
x		x	$y_{.,.}$ $y_{.,.}$ x	

Fig. 3.33 : Configuration qui donne l'éloignement minimal théorique

Remarque: les y_{ij} sont inférieurs à x de par la définition des coefficients d'affinité et $y_{ij}=y_{ji}$.

Le problème est donc de savoir si la phase 1 de fragmentation verticale peut aboutir à une telle configuration c'est-à -dire à la mise en valeur de ces deux fragments qui seront éloignés de la valeur minimale théorique 1.

Cette phase 1 étudie tous les découpages binaires du bloc n*n et retient le moins défavorable; nous nous posons donc la question suivante: partant de cette configuration générale, le découpage "après k" peut-il être le moins défavorable?

Nous proposons ci-dessous la mise en évidence d'un cas particulier tel que ce découpage "après k" soit le moins défavorable et donc tel que l'éloignement entre les fragments obtenus soit l'éloignement minimal théorique.

Soit S_i la mesure faite lors du découpage "après i". S_i comptabilise donc les accès inutiles de chaque bloc diagonal (BSUPi et BINFi)et les accès supplémentaires pondérés (1 accès supplémentaire = ***C*** accès inutiles) du bloc non diagonal (BMIXTEi).

$$S_i = ACCES\_INUT(BSUPi) + ACCES\_INUT(BINFi) + C * ACCES\_SUP(BMIXTEi)$$

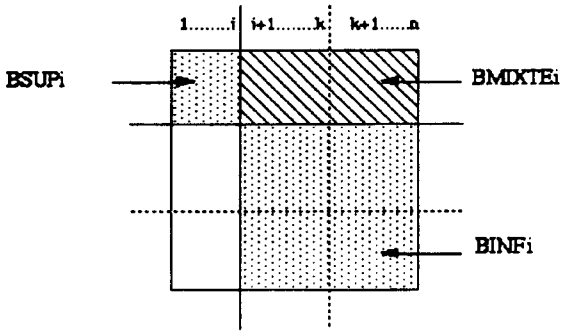


Fig. 3.34 : Principe de l'évaluation de S_i

$\forall i < k$ S_i est calculé par:

$$S_i = \left[(i-1) \times i \times x - 2 \times \sum_{l=1}^{i-1} \sum_{c=l+1}^{i} y_{lc} \right] \longrightarrow ACCES\_INUT(BSUPi) +$$

$$\left[(k-i-1) \times (k-i) \times x - 2 \times \sum_{l=i+1}^{k-1} \sum_{c=l+1}^{k} y_{lc} + (n-k-1) \times (n-k) \times x - 2 \times \sum_{l=k+1}^{n-1} \sum_{c=l+1}^{n} y_{lc} \right]$$

$$+ \quad C \times \left[\sum_{l=1}^{i} \sum_{c=i+1}^{k} y_{lc} + (n-k) \times i \times x \right] \longrightarrow ACCES\_SUP(BMIXTEi)$$

$ACCES\_INUT(BINFi)$

$\Leftrightarrow \forall i < k$ S_i est calculé par:

$$S_i = x \times [(i-1) \times i + (k-i-1) \times (k-i) + (n-k-1) \times (n-k) + C \times (n-k) \times i]$$

$$- 2 \times \sum_{l=1}^{i-1} \sum_{c=l+1}^{i} y_{lc} - 2 \times \sum_{l=i+1}^{k-1} \sum_{c=l+1}^{k} y_{lc} - 2 \times \sum_{l=k+1}^{n-1} \sum_{c=l+1}^{n} y_{lc} + C \times \sum_{l=1}^{i} \sum_{c=i+1}^{k} y_{lc}$$

Pour i=k S_k est calculé par:

$$S_k = \left[(k-1) \times k \times x - 2 \times \sum_{l=1}^{k-1} \sum_{c=l+1}^{k} y_{lc} \right] \xrightarrow[+]{} \text{ACCES\_INUT(BSUPk)}$$

$$\left[(n-k-1) \times (n-k) \times x - 2 \times \sum_{l=k+1}^{n-1} \sum_{c=l+1}^{n} y_{lc} \right] + C \times [(n-k) \times k \times x]$$

$$\downarrow \qquad\qquad\qquad \downarrow$$

ACCES_INUT(BINFk) ACCES_SUP(BMIXTEk)

$\Leftrightarrow$ Pour i=k S_k est calculé par:

$$S_k = x \times [(k-1) \times k + (n-k-1) \times (n-k) + C \times (n-k) \times k] - 2 \times \sum_{l=1}^{k-1} \sum_{c=l+1}^{k} y_{lc} - 2 \times \sum_{l=k+1}^{n-1} \sum_{c=l+1}^{n} y_{lc}$$

Après avoir donné les expressions de S_i (pour i<k) et de S_k, nous allons mettre en évidence un résultat en cherchant à quelle condition le découpage "après k" est moins défavorable que celui "après 1", ce qui d'un point de vue mesure se traduit par $S_k < S_1$.

Supposons donc que $S_k < S_1$.

A partir de l'expression de S_i (pour i<k) et en opérant une petite transformation, nous obtenons l'expression suivante de S1:

$$S_1 = x \times [(k-2) \times (k-1) + (n-k-1) \times (n-k) + C \times (n-k)] - 2 \times \sum_{l=1}^{k-1} \sum_{c=l+1}^{k} y_{lc} - 2 \times \sum_{l=k+1}^{n-1} \sum_{c=l+1}^{n} y_{lc} + (C+2) \times \sum_{c=2}^{k} y_{lc}$$

Dès lors $S_k < S_1$ $\Leftrightarrow$

$$x \times [(k-1) \times k + (n-k-1) \times (n-k) + C \times (n-k) \times k] < x \times [(k-2) \times (k-1) + (n-k-1) \times (n-k) + C \times (n-k)] + (C+2) \times \sum_{c=2}^{k} y_{lc}$$

$\Leftrightarrow$

$$x \times [(k-1) \times (2 + (n-k) \times C)] < (C+2) \times \sum_{c=2}^{k} y_{lc}$$

Comme $\forall l$ et $\forall c$ $y_{lc} \leq x$ nous pouvons majorer l'inéquation précédente de la façon suivante:

$$x \times [(k-1) \times (2 + (n-k) \times C)] < (C+2) \times \sum_{c=2}^{k} y_{lc} \leq (C+2) \times (k-1) \times x$$

$$\Leftrightarrow$$

$$x \times [(k-1) \times C \times (n-k-1)] \leq 0$$

L'indice k variant de 1 à n-1, la valeur (n-k-1) varie de n-2 à 0 et est donc toujours positive ou nulle. Le terme de gauche de l'inégalité sera donc toujours positif ou nul. Nous pouvons donc écrire:

$$x \times [(k-1) \times C \times (n-k-1)] \leq 0 \quad \Leftrightarrow \quad x \times [(k-1) \times C \times (n-k-1)] = 0$$

$$\Leftrightarrow \quad k=1 \text{ ou } k=n-1$$

En conséquence, pour toutes les valeurs de k comprises entre 2 et n-2, nous ne pourrons jamais avoir $S_k < S_1$. Le découpage "après 1" sera alors toujours plus favorable que celui "après k". En d'autres termes, la phase 1 de la fragmentation verticale ne fournira jamais deux fragments d'au moins deux attributs chacun ($1<k<n-1$) qui répondent à la configuration initiale et qui donnent l'éloignement minimal théorique.

Résultat: Deux fragments d'au moins deux attributs ne peuvent être éloignés de la valeur minimale théorique.

Remarque: Il est intéressant de noter que ce résultat est indépendant du poids C attribué aux accès supplémentaires.

Si nous supprimons la restriction sur les deux attributs minimum de chaque fragment, nous pouvons alors trouver des configurations qui donnent l'éloignement minimal théorique. Nous donnons ci-dessous un exemple de ce cas:

10	10	10	10	10
10	10	4	7	10
10	4	10	5	10
10	7	5	10	10
10	10	10	10	10

Découpage	1	2	3	4
S_i	108	112	116	108
Eloignement	1	2,3	2,3	1

Fig. 3.35 : Exemple de config. telle que les fragments obtenus sont minimalement éloignés

Sur un tel bloc d'affinité la phase 1 propose deux fragmentations qui donneront toutes deux l'éloignement minimal théorique entre les fragments construits.

3.3.3.3. Eloignement minimal pour deux fragments d'au moins deux attributs

Soient deux fragments adjacents f1={1,...,k} et f2={k+1,...,n} avec 1<k<n-1. Comme nous l'avons montré précédemment, l'éloignement entre deux fragments de ce type ne peut être égal à l'éloignement minimal théorique. Nous nous proposons ici de donner une borne minimale de cet éloignement qui sera fonction de n et k, c'est-à-dire de la taille des fragments.

Désirant mettre en évidence une borne minimale, nous partirons du bloc d'affinité ci-dessous:

	1	2		k	k+1	k+2		n
1	x	y_{12}	..	y_{1k}	$x-\varepsilon_{1,k+1}$	$x-\varepsilon_{1,k+2}$	..	$x-\varepsilon_{1,n}$
2	y_{21}	x	..	y_{2k}	$x-\varepsilon_{2,k+1}$	$x-\varepsilon_{2,k+2}$	..	$x-\varepsilon_{2,n}$
	..	..	..	..	..	..	..	..
k	y_{k1}	y_{k2}	..	x	$x-\varepsilon_{k,k+1}$	$x-\varepsilon_{k,k+2}$	..	$x-\varepsilon_{k,n}$
k+1	$x-\varepsilon_{k+1,1}$	$x-\varepsilon_{k+1,2}$	..	$x-\varepsilon_{K+1,k}$	x	$y_{k+1,k+2}$	..	$y_{k+1,n}$
k+2	$x-\varepsilon_{k+2,1}$	$x-\varepsilon_{k+2,2}$	..	$x-\varepsilon_{k+2,k}$	$y_{k+2,k+1}$	x	..	$y_{k+2,n}$
	..	..	..	..	..	..	..	..
n	$x-\varepsilon_{n,1}$	$x-\varepsilon_{n,2}$	..	$x-\varepsilon_{n,k}$	$y_{n,k+1}$	$y_{n,k+2}$	..	x

Fig. 3.36 : Configuration considérée pour la détermination d'une borne minimale de l'éloignement entre deux fragments d'au moins deux attributs

Remarque: $\forall i$ et $\forall j$ $y_{i,j}=y_{j,i}$ et $\varepsilon_{i,j}=\varepsilon_{i,j}$ et $y_{i,j}<=x$ et $\varepsilon_{i,j}<=x$

En effet, nous devons prendre les mêmes valeurs diagonales dans chaque fragment afin de ne pas augmenter l'un ou l'autre des rapports Ri ou Rj et, par la même, de s'écarter de l'éloignement minimal recherché.

Entre deux tels fragments l'éloignement vaut:

$$E(f1,f2) = \left(\frac{k \times (n-k) \times x}{k \times (n-k) \times x - \Sigma_{total}} \right)^2 \quad \text{avec} \quad \Sigma_{total} = \sum_{l=1}^{k} \sum_{c=k+1}^{n} \varepsilon_{lc}$$

Σ_{total} représente donc la somme des écarts élémentaires par rapport aux valeurs diagonales. Il calcule un écart global entre les accès supplémentaires effectifs et le nombre théorique maximum de ces accès supplémentaires.

Vis-à-vis de la phase 1, ces fragments ne peuvent exister que si $S_k<S_i$ et $S_k<S_j$

Nous allons donc développer ces deux inégalités ce qui nous donnera une condition sur la valeur minimale de Σ_{total} et, par la même, une borne minimale pour l'éloignement entre les deux fragments.

Avec un calcul analogue à celui utilisé dans les pages précédentes, nous obtenons:

$\forall i < k$ S_i:

$$S_i = x \times [(i-1) \times i + (k-i-1) \times (k-i) + (n-k-1) \times (n-k) + C \times (n-k) \times i]$$
$$- 2 \times \sum_{l=1}^{k-1} \sum_{c=l+1}^{k} y_{lc} - 2 \times \sum_{l=k+1}^{n-1} \sum_{c=l+1}^{n} y_{lc} + (C+2) \times \sum_{l=1}^{i} \sum_{c=i+1}^{k} y_{lc}$$
$$- C \times \Sigma_{total} + (C+2) \times \sum_{l=i+1}^{k} \sum_{c=k+1}^{n} \varepsilon_{lc}$$

$$S_k = x \times [(k-1) \times k + (n-k-1) \times (n-k) + C \times (n-k) \times k]$$
$$- 2 \times \sum_{l=1}^{k-1} \sum_{c=l+1}^{k} y_{lc} - 2 \times \sum_{l=k+1}^{n-1} \sum_{c=l+1}^{n} y_{lc} - C \times \Sigma_{total}$$

$\forall j > k$ Sj:

$$S_j = x \times [(k-1) \times k + (j-k-1) \times (j-k) + (n-j-1) \times (n-j) + C \times (n-j) \times k]$$
$$- 2 \times \sum_{l=1}^{k-1} \sum_{c=l+1}^{k} y_{lc} - 2 \times \sum_{l=k+1}^{n-1} \sum_{c=l+1}^{n} y_{lc} + (C+2) \times \sum_{l=k+1}^{j} \sum_{c=j+1}^{n} y_{lc}$$
$$- C \times \Sigma_{total} + (C+2) \times \sum_{l=1}^{k} \sum_{c=k+1}^{j} \varepsilon_{lc}$$

La première inégalité, $S_k < S_i$ pour tout $i < k$, nous conduit à:

$$x \times [(k-1) \times k + C \times (n-k) \times k - (i-1) \times i - (k-i-1) \times (k-i) - C \times (n-k) \times i]$$
$$< (C+2) \times \left[\sum_{l=1}^{i} \sum_{c=i+1}^{k} y_{lc} + \sum_{l=i+1}^{k} \sum_{c=k+1}^{n} \varepsilon_{lc} \right]$$

Comme $\forall l$ et $\forall c$ $y_{lc} \leq x$ et $\sum_{l=i+1}^{k} \sum_{c=k+1}^{n} \varepsilon_{lc} \leq \Sigma_{total}$ nous pouvons majorer par:

$$x \times [(k-1) \times k + C \times (n-k) \times k - (i-1) \times i - (k-i-1) \times (k-i) - C \times (n-k) \times i]$$

$$- \quad x \times [(C+2) \times i \times (k-i)] \leq (C+2) \times \Sigma_{total}$$

$\Leftrightarrow$

$$\boxed{x \times \left[\frac{C}{C+2} \times (k-i) \times (n-k-i)\right] \leq \Sigma_{total}} \quad \Leftrightarrow \quad \boxed{\Sigma_{total}(i) \leq \Sigma_{total}}$$

Chaque inégalité $S_k < S_i$ nous conduit donc à une valeur minimale $\Sigma_{total}(i)$ fonction de i. Pour vérifier toutes ces inégalités, la valeur effective Σ_{total} devra donc être supérieure ou égale à la valeur $\Sigma_{total}(i)$ maximale pour i variant de 1 à k, ce qui peut s'exprimer par:

$$\boxed{\max_i (\Sigma_{total}(i)) \leq \Sigma_{total}}$$

La seconde inégalité, $S_k < S_j$ pour tout k< j, nous conduit à:

$$x \times [(n-k-1) \times (n-k) + C \times (n-k) \times k - (j-k-1) \times (j-k) - (n-j-1) \times (n-j)]$$

$$- \quad x \times [C \times (n-j) \times k] \leq (C+2) \times \left[\sum_{l=k+1}^{j} \sum_{c=j+1}^{n} y_{lc} + \sum_{l=1}^{k} \sum_{c=k+1}^{j} \varepsilon_{lc}\right]$$

Comme $\forall l$ et $\forall c$ $y_{lc} \leq x$ et $\sum_{l=1}^{k} \sum_{c=k+1}^{j} \varepsilon_{lc} \leq \Sigma_{total}$ nous pouvons majorer et obtenons:

$$\boxed{x \times \left[\frac{C}{C+2} \times (j-k) \times (j+k-n)\right] \leq \Sigma_{total}} \quad \Leftrightarrow \quad \boxed{\Sigma_{total}(j) \leq \Sigma_{total}}$$

Chaque inégalité $S_k < S_j$ nous conduit donc à une valeur minimale $\Sigma_{total}(j)$ fonction de j. Pour vérifier toutes ces inégalités la valeur effective Σ_{total} devra donc être supérieure ou égale à la valeur $\Sigma_{total}(j)$ maximale pour j variant de k+1 à n, ce qui peut s'exprimer par:

$$\boxed{\max_j (\Sigma_{total}(j)) \leq \Sigma_{total}}$$

Les deux fragments f1 et f2 initialement considérés ne peuvent donc exister, c'est-à-dire ne peuvent avoir été mis en évidence par la phase 1, que si la valeur de Σ_{total} est supérieure à la fois à $\max_i(\Sigma_{total}(i))$ et à $\max_j(\Sigma_{total}(j))$, c'est-à-dire si les valeurs de B_{ij} sont suffisamment petites.

Pour toute valeur de n et k il existe donc une valeur de Σ_{total} minimale au-deçà de laquelle les fragments ne pourraient avoir été produits lors de la phase 1. A cette Σ_{total_min} correspond, de par l'équation qui lie l'éloignement et Σ_{total}, un éloignement minimal défini par:

$$E_{min}(f1,f2) = \left(\frac{k \times (n-k) \times x}{k \times (n-k) \times x - \Sigma_{total\_min}} \right)^2$$

Remarque: Il est intéressant de noter que puisque Σ_{total_min} est de la forme Cte*x, cette valeur est indépendante de x, c'est-à-dire de la valeur diagonale utilisée. Elle dépend bien uniquement de la taille des fragments traités, c'est-à-dire de n et k.

Nous donnons ci-dessous quelques exemples d'éloignements minimaux:

n	k	Σ_{total_min}	$E_{min}(f1,f2)$
5	2	1,42*x	1,72
9	5	8.52*x	3,04
12	9	17,75*x	8,52
20	16	57,51*x	97,25

Fig. 3.37 : Exemple d'éloignements minimaux en fonction de n et k

Remarque: Pour une valeur de n donnée, c'est la valeur k=n-2 qui fournit toujours l'éloignement minimum le plus important (par rapport aux éloignements minimum obtenus avec les autres valeurs de k). Ce résultat s'obtient en étudiant les variations des fonctions $max_i()$ et $max_j()$. Nous donnons en annexe, un tableau des différentes valeurs de Emin(f1,f2) suivant les valeurs de n et k. Nous pouvons en conclure que, au plus n est grand, au plus cet éloignement minimum est important ou, en d'autres termes, plus la sous-matrice d'affinité est grande, moins les fragments ont des chances d'être proches. Là encore, nous pouvons retrouver ce résultat par l'analyse des fonctions $max_i()$ et $max_j()$.

Cependant, il existe des cas impossibles, c'est-à-dire des valeurs de n et k pour lesquelles le bloc d'affinité initialement considéré, avec les mêmes valeurs diagonales pour chaque fragment, n'est pas envisageable puisqu'il faudrait alors mettre des valeurs négatives dans B_{ij} pour que la phase 1 produise les bons fragments.

Exemple: Pour n=16 et k=2, c'est-à-dire pour deux fragments de taille respective 14 et 2, nous obtenons: $\Sigma_{total} >= 29,82*x$. Cependant, comme un coefficient d'affinité est toujours positif ou nul, la valeur Σ_{total} est bornée supérieurement par k*(n-k)*x qui en l'occurrence vaut ici 28*x. La condition $\Sigma_{total} > \Sigma_{total\_min} = 29,82*x$ ne pourra donc jamais être vérifiée. La phase 1 ne fournira donc jamais deux fragments de taille 14 et 2 et répondant à la configuration initiale.

Pour de tels cas, il faudrait partir d'une matrice plus générale, avec des valeurs variables sur la diagonale, afin de mettre en évidence un éloignement minimal.

Il est intéressant de remarquer que la condition sur la valeur minimale de Σ_{total} nous permet de retrouver le résultat précédent qui concernait l'éloignement minimal théorique. Pour obtenir cet éloignement minimal théorique, il suffirait que Σ_{total_min} soit inférieur ou égal à 0. En effet, dans ce cas, toute valeur ε_{lc} pourrait être nulle ce qui aboutirait alors à l'obtention de

l'éloignement minimal théorique. Cette condition peut se traduire par:

$$\Sigma_{total\_min} \leq 0 \quad \Leftrightarrow \quad (\max_i (\Sigma_{total}(i)) \leq 0)\ ET\ (\max_j (\Sigma_{total}(j)) \leq 0)$$

$$\Leftrightarrow \quad \left(\max_i \left(x \times \frac{C}{C+2} \times (k-i) \times (n-k-i)\right) \leq 0\right) \quad ET$$
$$\left(\max_j \left(x \times \frac{C}{C+2} \times (j-k) \times (j+k-n)\right) \leq 0\right)$$

$$\Leftrightarrow \quad (\max_i (n-k-i) \leq 0)\ ET\ (\max_j (j+k-n) \leq 0)$$

$$\Leftrightarrow \quad ((n-k-1) \leq 0)\ ET\ (((k-1)+k-n) \leq 0)$$

$$\Leftrightarrow \quad (n-1 \leq k)\ ET\ (k \leq 1) \quad \Leftrightarrow \quad k=n-1\ ET\ k=1$$

Ces deux conditions sur k semblent a priori incompatibles.

Cependant lorsque k=n-1, il n'existe pas de valeur j entre k et n. De ce fait, la condition $\Sigma_{total\_min} <= 0$ se réduit à $\max_i(\Sigma_{total}(i)) <= 0$ qui aboutit à la condition vérifiée k=n-1. De même lorsque k=1, il n'existe pas de valeur i entre 1 et k. La condition $\Sigma_{total\_min} <= 0$ se réduit alors à $\max_i(\Sigma_{total}(i)) <= 0$ qui aboutit à la condition là aussi vérifiée k=1.

Nous retrouvons donc bien le résultat précédent qui disait que deux fragments, dont l'un n'a qu'un attribut, peuvent être éloignés de la valeur minimale théorique.

3.3.3.4. Fragments non adjacents et éloignement minimal théorique

Les résultats précédents étaient obtenus pour des couples de fragments adjacents dans la matrice d'affinité. Nous pouvons maintenant nous intéresser à des fragments quelconques d'une classe et voir si l'éloignement entre eux peut être la valeur minimale théorique.

Pour obtenir l'éloignement minimal théorique entre deux fragments fi et fj non adjacents fi et fj, il faudrait avoir la configuration suivante.

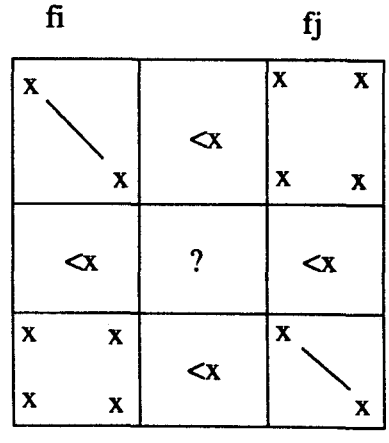


Fig. 3.38 : Configuration pour obtenir l'éloignement minimal entre fragments non adjacents

De par les capacités du BEA à regrouper les données en blocs, une telle configuration semble peu probable, surtout avec l'hypothèse que nous avons ajoutée pour l'application de cet algorithme et qui était de considérer la matrice bordée d'une grande valeur (cf 2.2.3). Sur tous les exemples testés, nous n'avons jamais obtenu ce type de matrice. Nous ne chercherons pas ici à confirmer cette intuition, confirmation qui nécessiterait la prise en compte à la fois du BEA et des fonctions de fragmentation verticale.

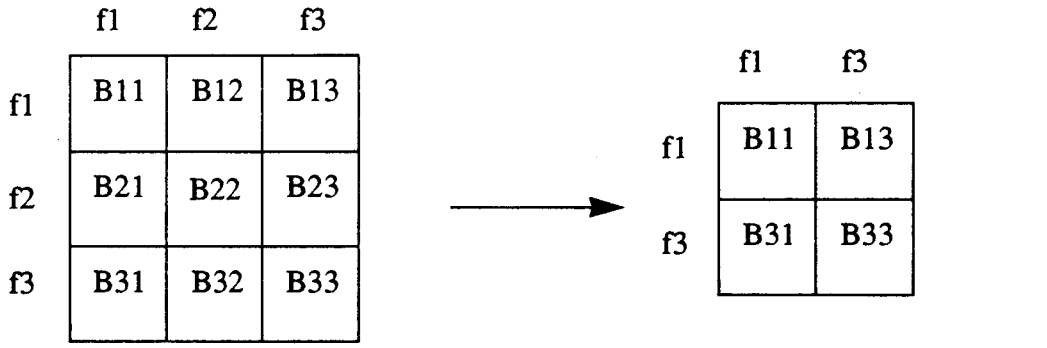
Après avoir proposé une fonction d'éloignement entre fragments d'une même classe et avoir étudié son comportement et ce qui la lie à la fonction de fragmentation verticale, nous allons voir maintenant comment utiliser cet éloignement afin de répartir les occurrences des fragments considérés.

3.3.4. Rectification des éloignements deux à deux

3.3.4.1. Présentation du problème

La fonction précédemment présentée calcule pour tout couple de fragments libres d'une même classe, une valeur d'éloignement. Ce calcul se fait par l'utilisation des informations qui concernent uniquement les deux fragments, c'est-à-dire par l'utilisation de la sous-matrice d'affinité relative à ces fragments. En conséquence, l'éloignement entre deux fragments ne tient pas compte de tout ce qui peut les lier transitivement.

Exemple: Soit une classe C pour laquelle la phase 1 fournit trois fragments C.f1, C.f2, C.f3 qui restent libres suite à la phase 2. Pour calculer l'éloignement entre C.f1 et C.f3, la fonction se base sur la sous-matrice suivante:



Matrice représentant la fragmentation de C

Sous-matrice utilisée pour le calcul de l'éloignement entre C.f1 et C.f3

Fig. 3.39 : Principe du calcul des éloignements entre fragments

Cependant il semblerait logique que cet éloignement C.f1-C.f3 tienne compte des éloignements C.f1-C.f2 et C.f2-C.f3, ce qui n'est pas le cas.

Les valeurs d'éloignement entre fragments libres d'une même classe ne sont donc pas directement exploitables afin de répartir les occurrences. Elles nécessitent d'être "rectifiées par transitivité".

Pour chaque classe C d'objets nuplet nous avons donc un ensemble de fragments libres et un ensemble de mesures d'éloignement entre ces fragments. La situation peut être représentée par un graphe Gc=[Fc,MEc] avec:

Fc: ensemble des sommets = ensemble des fragments libres de C

MEc: ensemble d'arêtes = ensemble des mesures d'éloignements entre fragments

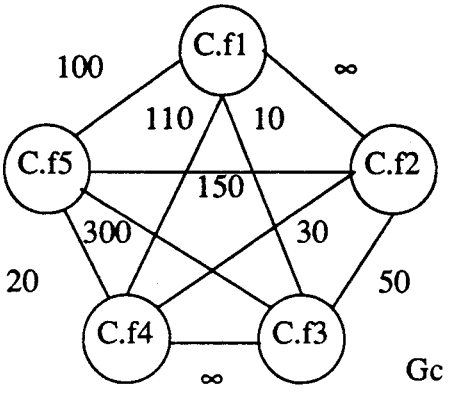


Fig. 3.40 : Graphe Gc représentant la fragmentation d'une classe C

Remarque: Gc est un graphe non orienté, pondéré et complet dont les "longueurs" des arêtes sont strictement positives.

Pour avoir utilisé une fonction d'éloignement "deux à deux", le graphe Gc peut ne pas être cohérent du point de vue inégalité triangulaire. Deux fragments peuvent être par transitivité plus proches que ne l'indique leur mesure d'éloignement. En d'autres termes, les objets partiels qui correspondent à un même objet peuvent être moins indépendants que "prévu".

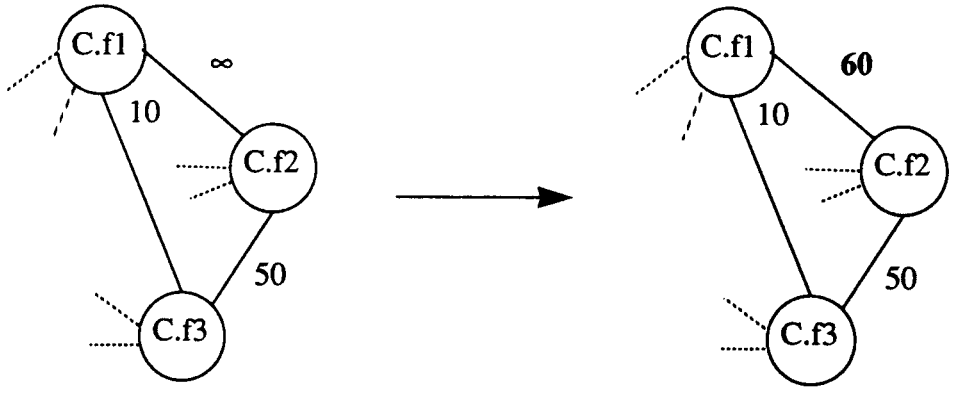


Fig. 3.41 : Exemple de rectification par transitivité

Sur l'exemple, nous nous rendons compte que l'éloignement entre C.f1 et C.f2 ne peut être infini, puisque ces fragments sont indirectement liés au travers de C.f3 par des éloignements finis.

La rectification par transitivité des mesures d'éloignement consiste donc à trouver, pour tout couple de fragments, l'éloignement maximal qui respecte l'inégalité triangulaire. Cet éloignement maximal, que nous pouvons dès lors appelé distance, correspond à la longueur du plus court chemin[1], dans le graphe Gc, entre les deux fragments. Il dénote l'indépendance maximale entre les deux objets partiels associés d'un même objet.

Pour chaque classe C d'objets nuplet, la rectification par transitivité des éloignements entre fragments se ramène donc au problème de la détermination de tous les plus courts chemins dans le graphe Gc correspondant.

Nous savons que le problème du plus court chemin entre deux sommets donnés i_0 et j_0 a une solution si et seulement s'il n'existe pas dans le graphe de circuit de longueur strictement négative pouvant être atteint à partir de i_0 [MINO86]. De plus, à cette condition, il existe toujours un chemin de longueur minimale qui soit élémentaire, c'est-à-dire qui ne passe jamais deux fois par le même sommet. Enfin, lorsque tous les circuits du graphe ont une longueur strictement positive, ce qui est le cas pour les graphes Gc, tout plus court chemin est nécessairement élémentaire.

3.3.4.2. Principe des différents algorithmes utilisables.

Différents algorithmes peuvent être utilisés pour trouver à partir d'un graphe Gc, l'ensemble des distances entre les fragments libres de la classe C, c'est-à-dire tous les plus courts chemins dans le graphe. Nous présentons ci-dessous le principe des deux méthodes les plus fréquemment citées [MINO86] [AHO87].

. L'algorithme de Dijkstra.

Cet algorithme de détermination du plus court chemin entre une source spécifiée i_0 et tout autre sommet du graphe, est l'illustration parfaite d'une méthode "gloutonne". A chaque étape, la solution la meilleure localement est choisie sans se préoccuper des étapes ultérieures. Pour cela, on tient à jour un ensemble E de sommets du graphe dont les distances les plus courtes à la source sont déjà connues. Au départ, E contient uniquement le sommet correspondant à la source.

Pour tout sommet j qui n'est pas dans E, on appelle D[j] ***raccourci*** la longueur du plus court chemin allant de i_0 à j et qui ne passe pas par des sommets de E. Les valeurs de D[j] sont initialisées par la longueur de l'arête i_0j.

A chaque étape de l'algorithme, on ajoute à E le sommet j qui correspond au D[j] minimum et on recalcule les longueurs des raccourcis vers les sommets k par:

D[k]=min (D[k], D[j] + D [k])

L'algorithme est fondé sur l'inégalité triangulaire et n'est bien sûr valable que pour des longueurs d'arêtes positives ou nulles. La complexité est en $O(n^2)$. Le détail et un exemple de l'algorithme peuvent être trouvés en annexe.

En ce qui nous concerne, il faut l'appliquer à tous les sommets du graphe Gc ce qui nous donnerait une complexité en $O(n^3)$.

---

1. En prenant pour longueur d'un chemin la somme des longueurs des arêtes qui le constituent.

. L'algorithme de R. W. Floyd

Il est utilisé pour déterminer la distance minimale entre deux sommets quelconques d'un graphe. Il propose une solution plus directe que l'application à chaque sommet de l'algorithme de Dijkstra.

Il suppose que les sommets du graphe Gc sont numérotés de 1 à n. L'algorithme utilise une matrice L de dimension n*n, qui sert à calculer les longueurs des chemins minimaux. Au départ, on pose L[i,j]= longueur de l'arête ij; n itérations sont alors effectuées sur la matrice L. Après la k-ème itération, la valeur L[i,j] correspond à la distance du chemin minimal entre les sommets i et j qui ne passe pas par un sommet d'indice supérieur à k. On utilise à cette k-ième itération la formule suivante pour mettre à jour L:

$$L_k[i,j]=\min (L_{k-1}[i,j] , L_{k-1}[i,k] + L_{k-1}[k,j])$$

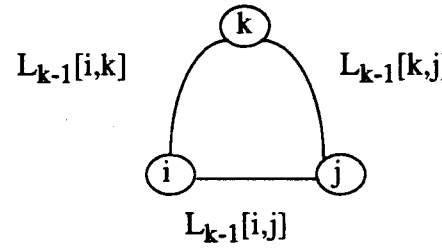


Fig. 3.42 : Insertion de k dans l'ensemble des sommets reliant i à j (Alg. de Floyd)

La complexité de cet algorithme est en $O(n^3)$. Le détail et un exemple de l'algorithme peuvent être trouvés en annexe.

Ce problème des plus courts chemins dans un graphe peut être résolu aussi par l'algorithme de Johnson ([MINO86], [AHO87]) appliqué à chaque sommet. Cet algorithme, qui consiste à représenter le graphe par des listes d'adjacence, n'a cependant un réel avantage que si le nombre m d'arêtes connues est très inférieur à n^2. La complexité élémentaire (application à un sommet du graphe) est alors en O(mlog(n)), d'où une complexité globale en O(m*nlog(n)). Les graphes Gc étant complets (m=n(n-1)/2), nous n'utiliserons pas cette solution.

Par l'un des algorithmes présentés ci-dessus, nous sommes donc capables de transformer un ensemble de mesures d'éloignement en un ensemble cohérent de distances représenté par un nouveau graphe G'c.

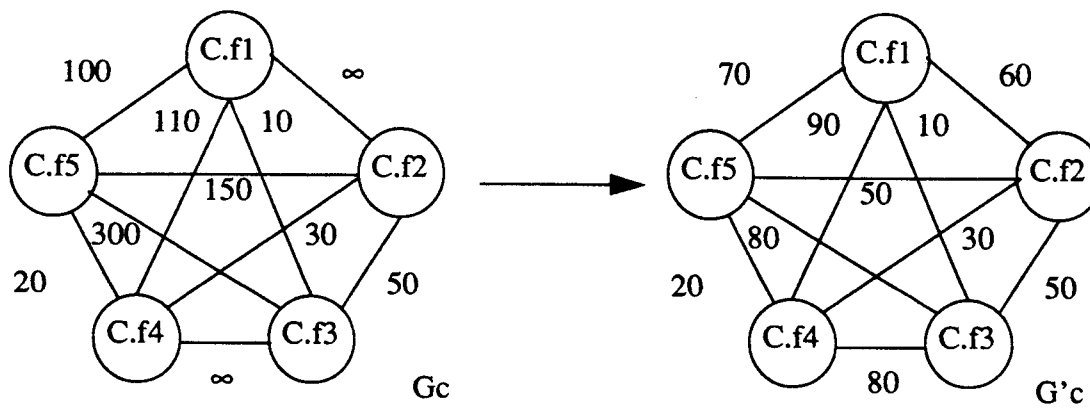


Fig. 3.43 : Exemple complet de rectification par transitivité

Remarque: Il est important de noter que la rectification par transitivité ne s'applique qu'aux éloignements entre fragments libres. Un fragment lié n'influencera donc pas les distances finales ce qui est logique si nous considérons qu'un tel fragment ne fait plus vraiment partie de la classe initiale.

Nous allons maintenant voir comment utiliser ces distances afin de répartir les occurrences des différents fragments.

3.3.5. Utilisation proprement dite des distances inter-fragments.

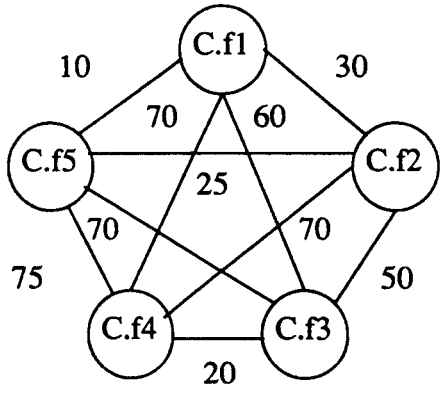
3.3.5.1. Positionnement logique et positionnement physique

Pour chaque classe C d'objets nuplet nous disposons donc d'un ensemble cohérent de distances que nous appelons par abus de langage distances inter-fragments. En fait, celles ci représentent ce qui lie les différents objets partiels libres d'un même objet et, à ce titre, nous devrions plutôt parler de distances inter-objets partiels. Néanmoins, nous emploierons dans la suite de l'exposé l'un ou l'autre des termes.

Un graphe G'c nous donne donc un positionnement relatif des différents objets partiels libres d'un même objet. Ce positionnement est, à ce niveau, totalement indépendant du réseau qui est ou sera utilisé; il est uniquement le fruit de l'exploitation des paramètres qui concernent les principales requêtes qui sont appliquées. En quelque sorte, ce positionnement pourrait être qualifié de logique puisque totalement indépendant du niveau physique.

Le problème qui se pose donc maintenant est celui de l'adaptation d'un tel positionnement logique à un réseau particulier. En d'autres termes, comment traduire les positions relatives logiques des différents objets partiels d'un même objet, en positions relatives physiques adaptées à un réseau donné?

Exemple: Supposons que la situation pour une classe C soit représentée par le graphe G'c suivant:



<=>

.	f1	f2	f3	f4	f5
f1	0	30	60	70	10
f2	30	0	50	70	25
f3	60	50	0	20	70
f4	70	70	20	0	75
f5	10	25	70	75	0

Fig. 3.44 : Exemples de distances inter-fragments pour une classe C

Nous supposerons, pour l'exemple ,que la répartition des objets partiels libres d'un objet O débute par le placement de celui qui est occurrence de f1 que nous noterons $f_1(O)$. Nous appellerons $n_i(O)$ le numéro du noeud dans le réseau où sera localisé $f_i(O)$.

Un exemple d'adaptation du graphe G'c à un hypercube de degré 3 pourrait être[1] décrit par le système d'équations présenté ci-dessous, ce qui donnerait le tableau associé de distances physiques:

$n_2(O) = n_1(O)$ ou_exclusif 011
$n_3(O) = n_1(O)$ ou_exclusif 010
$n_4(O) = n_1(O)$ ou_exclusif 110
$n_5(O) = n_1(O)$ ou_exclusif 001

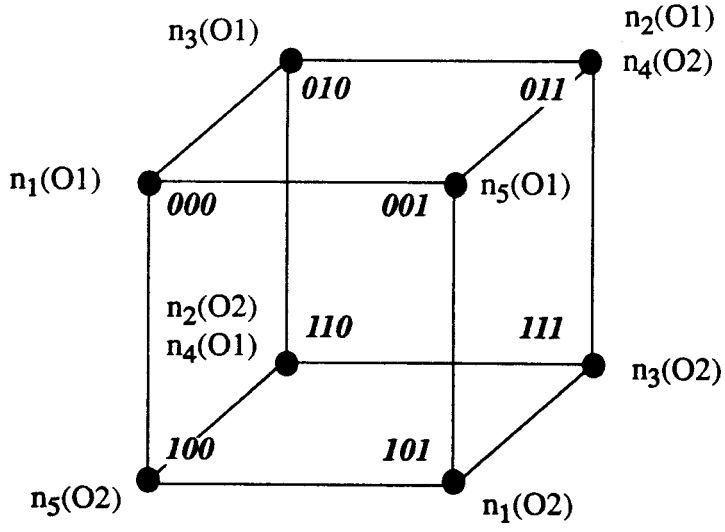
	f1	f2	f3	f4	f5
f1	0	2	1	2	1
f2	2	0	1	2	1
f3	1	1	0	1	2
f4	2	2	1	0	3
f5	1	1	2	3	0

Fig. 3.45 : Transformation en distances physiques adaptées à un hypercube

De par les bonnes propriétés de l'opérateur ou_exclusif (associativité, élément neutre), ces quatre équations sont suffisantes pour obtenir un positionnement relatif physique entre deux sous-objets partiels quelconques d'un même objet O.

Exemple: $n_2(O) = n_1(O)$ ou_exclusif 011 et $n_3(O) = n_1(O)$ ou_exclusif 010
=> $n_2(O) = n_3(O)$ ou_exclusif 001

Pour deux objets particuliers O1 et O2 nous aurions sur l'hypercube une configuration du type:



	O1	O2
$n_1()$	000	101
$n_2()$	011	110
$n_3()$	010	111
$n_4()$	110	011
$n_5()$	001	100

Fig. 3.46 : Exemple de placement des objets partiels de 2 objets sur un hypercube

1. Nous nous contentons ici de donner un exemple afin de bien situer le problème. L'aspect réalisation, algorithme utilisé sera détaillé par la suite.

L'adaptation du même graphe G'c à un réseau en anneau à huit noeuds (degré 2, diamètre 4) pourrait être décrit par cet autre système d'équations:

$n_2(O) = (n_1(O) + 2) \bmod 8$

$n_3(O) = (n_1(O) + 3) \bmod 8$

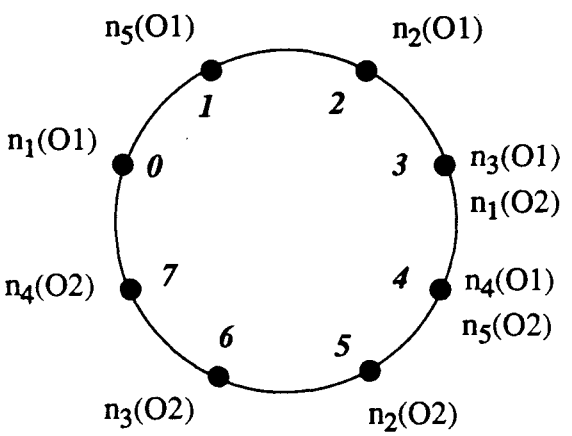
$n_4(O) = (n_1(O) + 4) \bmod 8$

$n_5(O) = (n_1(O) + 1) \bmod 8$

.	f1	f2	f3	f4	f5
f1	0	2	3	4	1
f2	2	0	1	2	1
f3	3	1	0	1	2
f4	4	2	1	0	3
f5	1	1	2	3	0

Fig. 3.47 : Transformation en distances physiques adaptées à un anneau

Ce qui donne pour deux objets particuliers O1 et O2:



	O1	O2
$n_1()$	0	3
$n_2()$	2	5
$n_3()$	3	6
$n_4()$	4	7
$n_5()$	1	4

Fig. 3.48 : Exemple de placement des objets partiels de 2 objets sur un anneau

Il apparait sur cet exemple que les distances physiques inter-objets partiels dépendent du réseau utilisé. Pour un même nombre de noeuds, un réseau avec un degré plus élevé ou un diamètre plus faible donnera des distances physiques plus "courtes" et donc une meilleure disposition générale des objets partiels d'un même objet. Sur l'exemple, l'anneau donne de moins bon résultats que l'hypercube de par son diamètre plus élevé et son degré plus faible.

Avant d'analyser en détail le problème de la transformation des positions logiques en positions physiques et, par la même, de définir ce que nous entendons par meilleure disposition générale des objets partiels d'un même objet, nous aimerions nous appesantir sur la notion de distance inter-objets partiels pour en tirer quelques remarques.

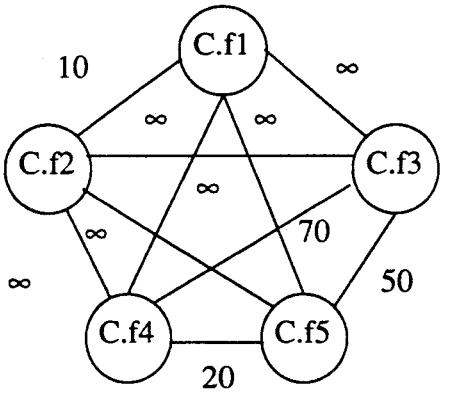
3.3.5.2. Répartition partiellement ou totalement relative?

Un graphe G'c nous donne un positionnement relatif logique de tous les objets par-

tiels libres d'un même objet. A priori, la répartition de ces objets partiels devrait donc être totalement relative, c'est-à-dire que la position de l'un d'eux devrait dépendre de la position de tous les autres. Dans ce cas cependant, tous les objets partiels d'un même objet seraient liés et nous ne disposerions alors pour une classe d'objets nuplet que d'un seul critère de répartition "central" (la fonction de répartition qui sur notre exemple associerait un noeud $n_1(O)$ à $f_1(O)$). En d'autres termes, nous aurions un seul degré de liberté pour répartir les occurrences des fragments de la classe.

Une ***répartition totalement relative*** inhiberait donc systématiquement toute possibilité de multiples critères de répartition au niveau d'une même classe. Nous reviendrons plus en détail sur ce problème dans la section "Comment répartir?" (relative au choix des critères de répartition cf 3.4) pour analyser les incompatibilités entre répartition totalement relative et répartition par rapport aux valeurs.

Revenons maintenant sur la notion même de distance inter-objets partiels. Celles-ci donnent une mesure d'indépendance entre les objets partiels libres d'un même objet. Il nous semble donc raisonnable de penser qu'au delà d'une certaine distance deux objets partiels n'ont plus lieu d'être considérés comme dépendants et que, dans ce cas, ils peuvent être répartis indépendamment. Le cas extrême serait de trouver des objets partiels totalement indépendants:



.	f1	f2	f3	f4	f5
f1	0	10	∞	∞	∞
f2	10	0	∞	∞	∞
f3	∞	∞	0	70	50
f4	∞	∞	70	0	20
f5	∞	∞	50	20	0

Fig. 3.49 : Exemple de cas extrême d'indépendance entre objets partiels

Nous distinguons sur cet exemple deux sous-ensembles de fragments {f1,f2}, {f3,f4,f5} qui n'ont aucun lien entre eux. De toute évidence, $f_1(O)$ peut être placé en ne tenant compte que de $f_2(O)$ (idem pour $f_3(O)$, $f_4(O)$ et $f_5(O)$).

Le graphe initial est scindé en deux sous-graphes $G'c_1$ et $G'c_2$ totalement indépendants. La transformation des positions relatives logiques en positions relatives physiques peut s'appliquer alors séparément sur chacun des sous-graphes. Nous dirons que nous effectuons une ***répartition partiellement relative*** des occurrences des fragments libres de C, puisque relative uniquement dans les sous-graphes.

L'exemple proposé, de par son caractère extrême, ne pose aucun problème pour déterminer les sous-ensembles intéressants de fragments. Nous verrons après avoir énoncé quelques avantages de la répartition partiellement relative, comment nous pouvons mettre en valeur quasi automatiquement ces sous-ensembles.

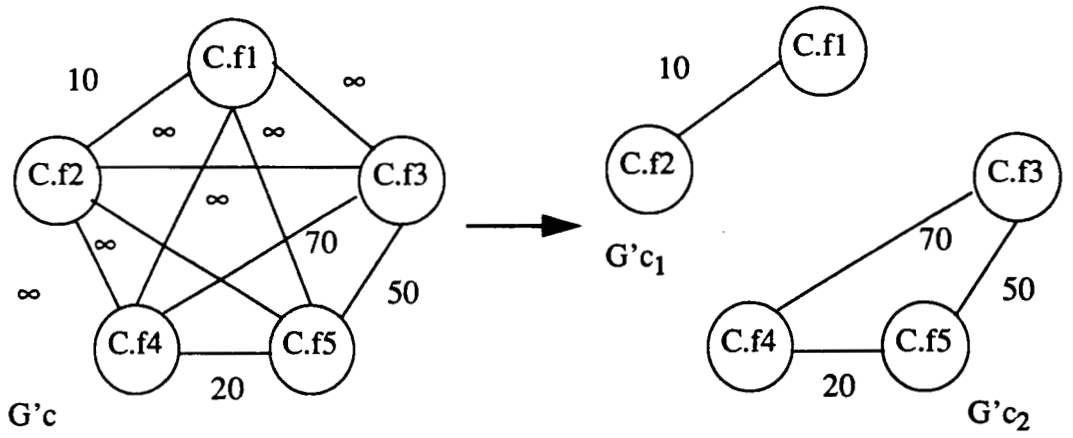


Fig. 3.50 : Décomposition du graphe des distances en sous-graphes indépendants

3.3.5.3. Avantages de la répartition partiellement relative

Outre le fait qu'une répartition partiellement relative permet d'utiliser plusieurs critères de répartition au sein d'une même classe, nous pouvons lui attribuer d'autres propriétés qui ne sont pas sans intérêt dans le contexte de notre étude.

Premièrement, qui dit répartition partiellement relative dit traitement indépendant des différents sous-graphes mis en évidence. Dès lors, le nombre de noeuds cibles qui sont utilisés pour la répartition des occurrences des fragments d'un sous-graphe donné peut être différent des autres. En d'autres termes, les objets partiels issus d'une classe peuvent être répartis sur des nombres différents de noeuds.

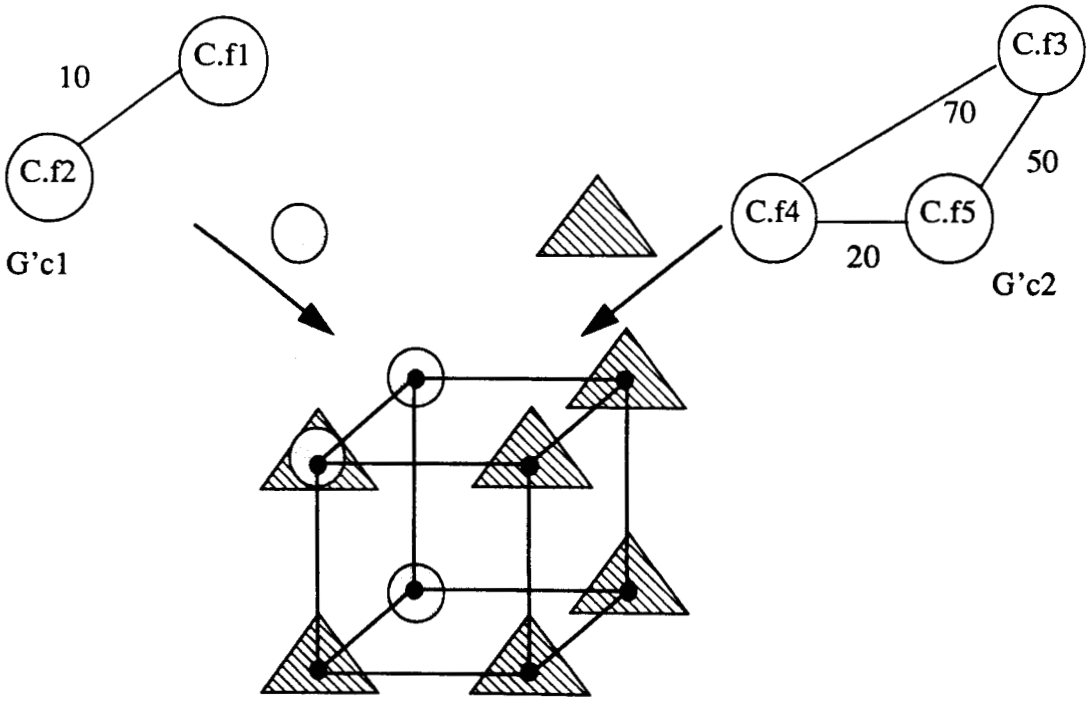


Fig. 3.51 : Répartition partiellement relative et degré de répartition variable

La répartition partiellement relative permet donc de satisfaire, en partie, une des préoccupations que nous avions dans la sous-section précédente (cf 3.2 Importance du degré de répartition dans notre contexte) et qui était d'adapter le nombre de noeuds cibles à l'utilisation qui est ou sera faite du fragment.

Une répartition totalement relative quant à elle forcerait ce nombre de noeuds cibles à être le même pour tous les fragments libres d'une classe. Pour un nombre de noeuds trop élevé, les fragments à risque d'un point de vue communication feraient chuter les performances. A l'opposé, pour un nombre de noeuds sous-estimé, les autres fragments ne seraient pas suffisamment éclatés ce qui limiterait le taux de parallélisme intra-requête. Somme toute, ce nombre unique de noeuds cibles ne pourraient être qu'un compromis qui "limiterait les dégâts" pour l'ensemble des cas.

L'utilisation d'une répartition partiellement relative nous permet, par contre, d'attribuer à chacun des sous-ensembles de fragments, un nombre de noeuds cibles. Nous ne satisfaisons pas entièrement notre préoccupation initiale qui était d'adapter pour chaque fragment ce nombre de noeuds. Néanmoins, nous aboutissons forcément à une répartition plus adaptée en ce sens où le compromis sur le nombre de noeuds est descendu du niveau de la classe au niveau des sous-ensembles de fragments.

Un second avantage que nous pouvons attribuer à la répartition partiellement relative est sa potentialité à conduire à une spécialisation de sous-réseaux physiques vis-à-vis d'une classe d'objets. En effet, nous pouvons très bien envisager d'utiliser pour des sous-ensembles différents de fragments, des ensembles disjoints de noeuds cibles (sous réserve bien sûr que le réseau général soit suffisamment grand). Nous obtenons, dans ce cas, des sous-réseaux physiques spécialisés chacun dans le traitement d'un sous-ensemble de fragments de la classe.

Une telle configuration est d'autant plus intéressante que les sous-ensembles de fragments sont indépendants les uns par rapport aux autres. En effet, si deux sous-ensembles sont fortement indépendants, cela signifie qu'un fragment de l'un a peu de chance d'être utilisé en même temps (par la même requête) qu'un fragment de l'autre. Dès lors, rien n'oblige à maintenir une certaine proximité ou un certain enchevêtrement entre les "régions" du réseau utilisé par chaque sous-ensemble. Au contraire, le fait de dissocier ces régions en spécialisant des sous-réseaux permet d'aboutir à une augmentation du parallélisme entre les requêtes qui utilisent la classe.

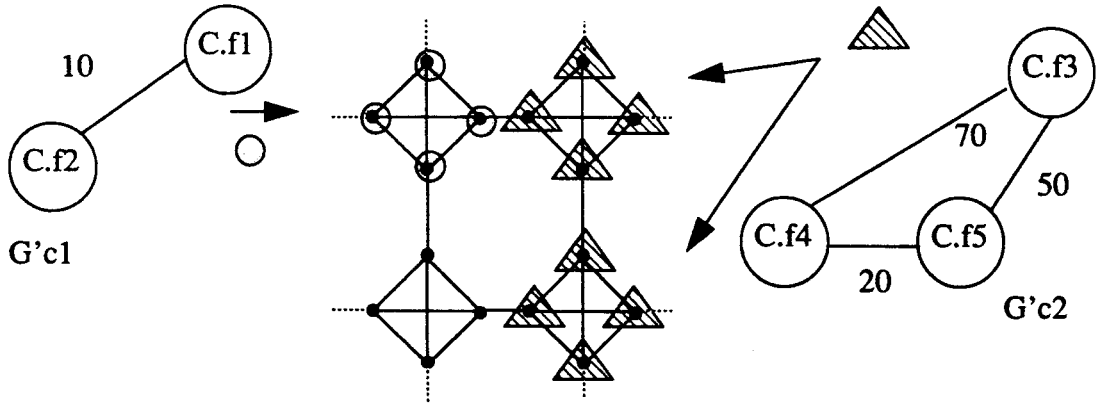


Fig. 3.52 : Répartition partiellement relative et spécialisation en sous-réseaux

Enfin, nous pouvons souligner que la spécialisation de sous-réseaux est d'autant plus intéressante que le réseau initial n'est pas "homogène" c'est-à-dire qu'il est formé de sous-réseaux. Dans ce cas, en effet, cette spécialisation permettra de tirer profit au mieux de la non homogénéité du réseau, sous réserve que cette dernière soit compatible avec le nombre de noeud nécessaires.

Après avoir souligné quelques avantages de la répartition partiellement relative, c'est-à-dire après avoir étudié le "pourquoi", nous allons maintenant proposer une méthode pour déterminer les regroupements de fragments qui seront les plus favorables. Nous nous intéresserons donc, dans ce qui suit, au "comment" de cette répartition.

3.3.5.4. Détermination de la répartition partiellement relative.

Toute classe C est représentée par un graphe G'c ou, ce qui est équivalent, par une matrice carrée de distances inter-objets partiels. Nous voulons donc déterminer ici des sous-ensembles de fragments tels qu'à l'intérieur de chacun d'eux les objets partiels d'un même objet soient "fortement" dépendants ("petites" distances) et tels que deux objets partiels qui appartiennent à deux sous-ensembles différents soient "fortement" indépendants ("grandes" distances).

En quelque sorte, nous cherchons à obtenir, si elle existe, une configuration qui pourrait être représentée par une matrice carrée telle que la présente la figure suivante et qui, bien sûr, serait issue de la matrice initiale.

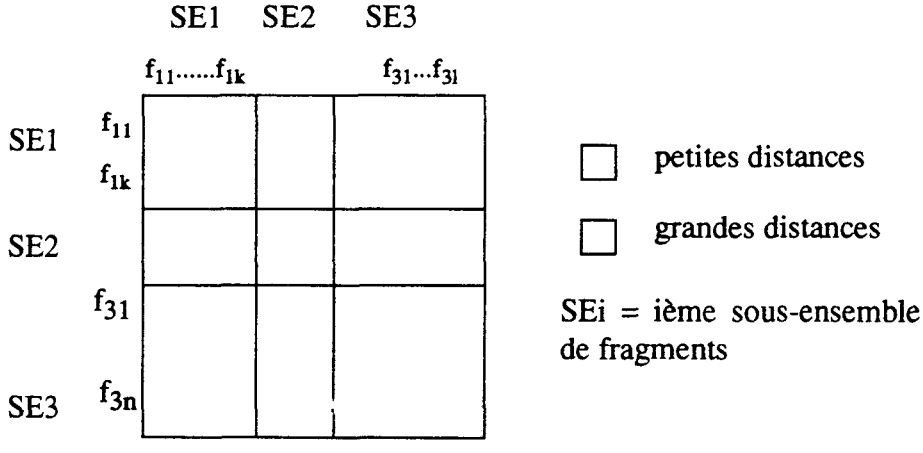


Fig. 3.53 : Configuration "image" de l'intérêt d'une répartition partiellement relative

Il semble donc qu'il ne s'agisse là que d'un problème d'arrangement de matrice que nous pensons solutionner par l'utilisation une fois de plus du Bond Energy Algorithm (BEA) [MCOR72]. Comme nous l'avons souligné précédemment (cd 2.2.3 Transformation de la matrice d'affinité), le but de cet algorithme est de mettre en évidence des regroupements dans des tableaux de données. Il est basé sur l'utilisation d'une mesure d'efficacité appliquée à toute permutation de lignes ou de colonnes du tableau initial. Cette mesure est telle qu'une configuration comportant des blocs denses de données est "meilleure" qu'une autre où les données importantes sont uniformément réparties.

La capacité du Bond Energy Algorithm à produire un tableau ou une matrice sous forme semi-bloc diagonale suggère bien évidemment son utilisation pour des problèmes qui se décomposent en sous-problèmes qui interagissent peu ou pas ensemble. En ce qui nous concerne, nous nous plaçons tout à fait dans un tel contexte puisque nous cherchons à décomposer le problème de l'interprétation physique d'un graphe G'c en sous-problèmes indépendants d'interprétation physique de sous-graphes de $G'c_i$.

Exemple: Nous proposons ci-dessous un exemple d'utilisation de BEA sur un ensemble de fragments d'une même classe.

	f1	f2	f3	f4	f5	f6	f7	f8
f1	0	280	200	290	30	190	40	300
f2	280	0	480	20	300	490	280	30
f3	200	480	0	490	210	20	180	500
f4	290	20	490	0	290	500	270	15
f5	30	300	210	290	0	200	30	300
f6	190	490	20	500	200	0	180	490
f7	40	280	180	270	30	180	0	300
f8	300	30	500	15	300	490	300	0

	f3	f6	f7	f5	f1	f2	f4	f8
f3	0	20	180	210	200	480	490	500
f6	20	0	180	200	190	490	500	490
f7	180	180	0	30	40	280	270	300
f5	210	200	30	0	30	300	290	300
f1	200	190	40	30	0	280	290	300
f2	480	490	280	300	280	0	20	30
f4	490	500	270	290	290	20	0	15
f8	500	490	300	300	300	30	15	0

Fig. 3.54 : Exemple d'application du BEA sur une matrice de distances inter-fragments

Trois sous-ensembles de fragments se détachent aisément du résultat obtenu suite à l'application du BEA. Ils correspondent, si l'on considère qu'ils sont suffisamment indépendants entre eux, aux trois sous-graphes ci-dessous qui seront traités isolément.

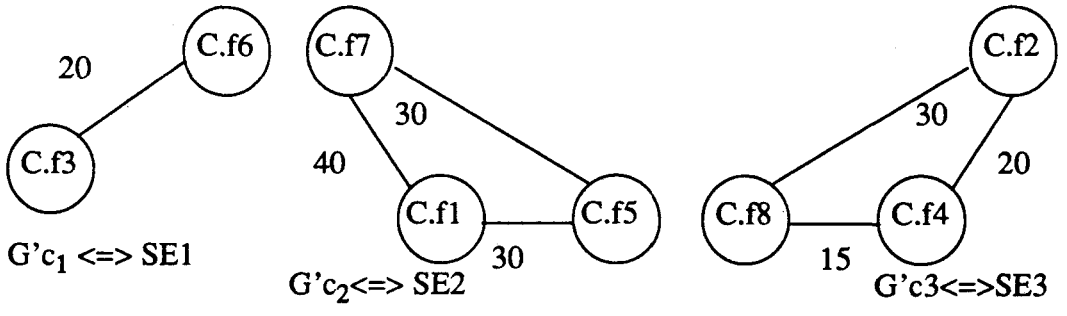


Fig. 3.55 : Sous-graphes obtenus sur l'exemple

Remarque: L'utilisation que nous faisons ici du Bond Energy Algorithm est opposée à celle que nous en faisions dans la phase 1 de la fragmentation verticale (cf 2.2.3). En effet, nous cherchons à construire "sur la diagonale" des blocs de "petites" valeurs et à repousser hors de cette même diagonale les "grandes" valeurs. En ce sens, nous devons adapter l'algorithme qui initialement repousse vers le centre les valeurs importantes puisque la matrice est considé-

rée comme entourée de 0. Pour cela, nous bordons cette dernière par la valeur maximale qu'elle contient ce qui a pour effet d'attirer vers l'extérieur ses valeurs importantes, et donc, de répondre à nos besoins. Une autre solution eut été d'appliquer l'algorithme initial (matrice bordée de 0) à une matrice transformée de la façon suivante:

$$T(mat_1[i][j]) = mat_2[i][j] = valeur\_max(mat_1) - mat_1[i][j]$$

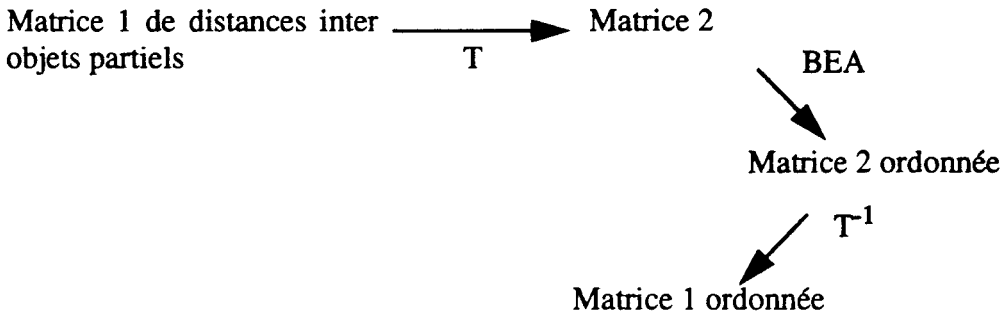


Fig. 3.56 : Adaptation du BEA pour le détermination de la répartition partiellement relative

Un peu comme nous l'avons fait pour la phase 1 de la fragmentation verticale, nous proposons une méthode afin de mettre en évidence automatiquement le meilleur découpage de la matrice ordonnée ou, en d'autres termes, les meilleurs sous-ensembles de fragments. A ce niveau, il faut plutôt voir ce meilleur découpage comme un outil d'aide à la décision proposé à l'administrateur qui, aux vues de l'allure générale de la matrice ordonnée et du découpage proposé, aiguillerait ces choix.

Notre démarche pour proposer ce meilleur découpage est guidée par l'approche intuitive suivante: nous cherchons à mettre en valeur les sous-ensembles de fragments qu'il serait intéressant de traiter indépendamment. Pour cela, nous devons remplir au mieux les deux conditions suivantes. D'une part, les fragments d'un même sous-ensemble doivent être les plus dépendants possible (faible distance).D'autre part, deux fragments de sous-ensembles différents doivent être les plus indépendants possible (distance importante).

Si nous raisonnons en termes de blocs engendrés par un découpage donné de la matrice ordonnée, cela revient à dire que nous devons minimiser les blocs diagonaux et maximiser les autres. Nous proposons donc d'utiliser comme définition du meilleur découpage de la matrice ordonnée, la définition suivante:

maxdécoupage(Σ(valeurs blocs non diagonaux) - Σ(valeurs blocs diagonaux))

Appliquée à l'exemple précédent, cette méthode fournit de toute évidence les trois sous-ensembles escomptés.

Remarque: Cette fonction simpliste n'est valable que pour des sous-ensembles d'au moins 2 fragments. L'obtention d'une méthode "plus fine" mérite reflexion.

Il serait sûrement intéressant de fournir, en plus de ce meilleur découpage, des informations relatives aux interactions entre sous-ensembles ainsi qu'un indice de comportement interne pour chacun d'entre eux. Pour cela, nous pourrions représenter les choses par un nou-

veau graphe où chaque noeud figurerait un sous-ensemble de fragments. Le comportement interne de chaque sous-ensemble pourrait être signalé en attachant au noeud correspondant la valeur moyenne des distances entre les fragments du sous-ensemble ainsi qu'un indice de dispersion de ces distances. L'interaction entre deux sous-ensembles serait dénotée par la pondération de l'arête correspondante du graphe par la valeur moyenne des distances entre ces sous-ensembles ainsi que par un indice de dispersion de ces distances.

Ce "macro-graphe" n'aurait d'autre but que de présenter, de façon résumée, la situation afin de faciliter les prises de décisions; dans le cadre de notre exemple, nous obtiendrions:

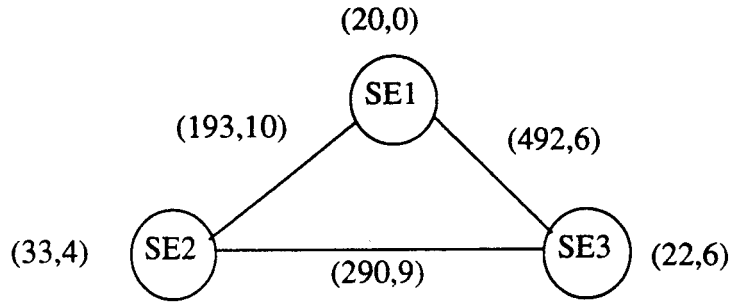


Fig. 3.57 : Exemple de macro-graphe associé à un découpage donné de la matrice des distances inter-fragments

Remarque: Puisque la matrice initiale contient des distances rectifiées, tout triplet de fragments "vérifie" l'inégalité triangulaire. En conséquence, les valeurs associées à notre macro-graphe ne peuvent être quelconques. Bien sûr, la propriété d'inégalité triangulaire ne se transmet pas au niveau des moyennes et écarts types, mais il serait certainement possible d'énoncer certaines règles caractérisant ce macro-graphe.

Somme toute, nous venons de voir pourquoi et comment le problème général de transformation des positions relatives logiques en positions relatives physiques, propre à une classe C, peut être décomposé en sous-problèmes indépendants du même type et propre pour chacun d'eux à un sous-ensemble de fragments de C. Nous allons donc maintenant nous intéresser à la transformation proprement dite, pour un sous-ensemble donné de fragments, des positions logiques en positions physiques.

3.3.6. Transformation proprement dite en position relatives physiques.

Nous voulons donc obtenir pour un sous-ensemble de fragments, ie pour un sous-graphe $G'c_i$, un ensemble de positions relatives physiques. Nous supposons, à ce niveau, que les fragments concernés forment un tout du point de vue répartition de leurs occurrences, c'est-à-dire que les objets partiels concernés d'un même objet doivent être répartis les uns par rapport aux autres. Supposer le contraire reviendrait d'ailleurs à remettre en cause les sous-ensembles proposés de fragments issus du découpage préconisé!

En conséquence, nous cherchons à établir une répartition totalement relative des

occurrences des fragments du sous-ensemble. L'utilisation de la répartition partiellement relative au niveau de la classe C revient donc à n utilisations indépendantes de la répartition totalement relative, si n sous-ensembles de fragments sont mis en évidence par la méthode précédemment présentée. Les contraintes attachées à la répartition totalement relative (cf 3.3.5.2 Répartition partiellement ou totalement relative?) passent donc du niveau de la classe au niveau d'un sous-ensemble de fragments. En outre, nous ne pouvons disposer que d'un seul critère de répartition "central" pour tous les fragments d'un même sous-ensemble, celui qui permettra de placer l'objet partiel qui guidera les autres. Le choix de ce critère de répartition sera discuté plus en avant (cf 3.4 Comment répartir).

Une première technique que nous pouvons utiliser pour obtenir des positions relatives physiques peut être guidée par la volonté de plaquer "toutes proportions gardées" le sous-graphe $G'c_i$ à un réseau donné. Nous désignerons cette première solution sous le nom de ***placement proportionnel***.Dans ce cas, deux fragments sont physiquement et logiquement éloignés dans les mêmes proportions.

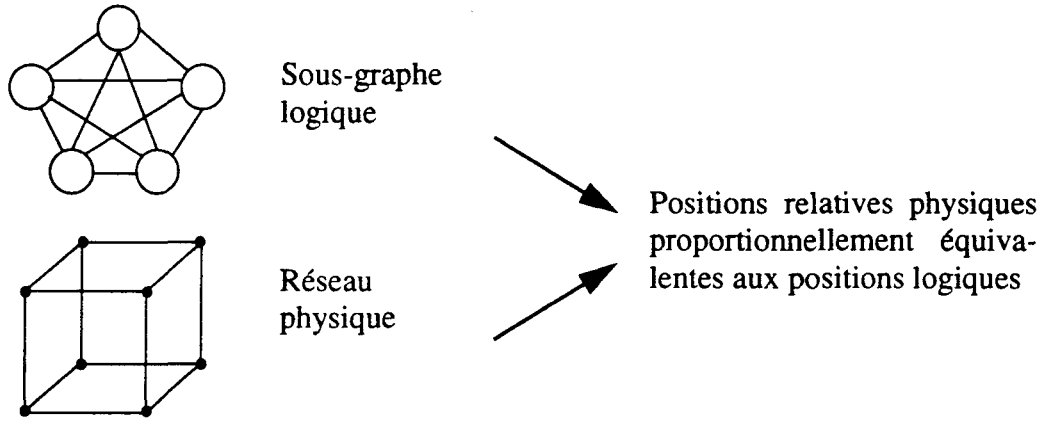
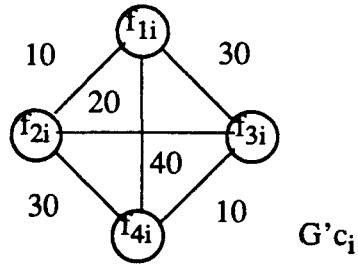


Fig. 3.58 : Principe du placement proportionnel

Exemple: Le plaquage "toutes proportions gardées" du sous-graphe ci-dessous $G'c_i$ à un réseau en anneau à huit noeuds peut se traduire par les équations suivantes:



$$n_{2i}(O) = (n_{1i}(O) + 1) \bmod 8$$

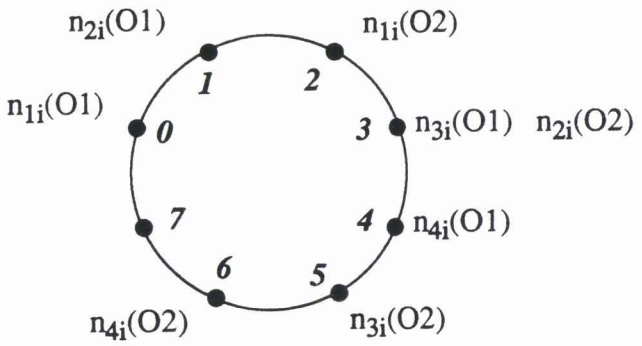
$$n_{3i}(O) = (n_{1i}(O) + 3) \bmod 8$$

$$n_{4i}(O) = (n_{1i}(O) + 4) \bmod 8$$

Fig. 3.59 : Exemple de placement proportionnel

Dans ce cas, les distances physiques inter-objets partiels présentées dans le tableau ci-dessus, reflètent tout à fait les distances logiques initiales.

Ceci donne pour deux objets O1 et O2:



.	f_{1i}	f_{2i}	f_{3i}	f_{4i}
f_{1i}	0	1	3	4
f_{2i}	1	0	2	3
f_{3i}	3	2	0	1
f_{4i}	4	3	1	0

Fig. 3.60 : Placement "proportionnel" sur un anneau des objets partiels de 2 objets

Une première remarque que nous pouvons faire à propos de cette technique est qu'il est certainement utopique de parler de "plaquage toutes proportions gardées". En effet, des paramètres tels que le degré, le diamètre et le nombre de noeuds du réseau limiteront souvent la transformation et modifieront en conséquence ces proportions. La technique se traduira donc par un algorithme qui place proportionnellement au mieux le graphe initial.

Exemple: le plaquage de G'ci à un hypercube de degré 2 donne le tableau suivant de distances physiques qui ne respectent pas les proportion initiales.

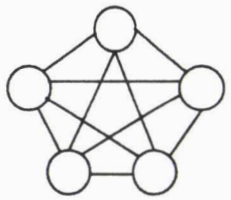
.	f_{1i}	f_{2i}	f_{3i}	f_{4i}
f_{1i}	0	1	2	2
f_{2i}	1	0	1	1
f_{3i}	2	1	0	1
f4i	2	1	1	0

Fig. 3.61 : Placement "proportionnel" sur un hypercube des objets partiels de 2 objets

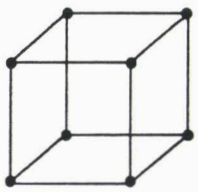
Une seconde remarque, d'un ordre beaucoup plus général, va nous amener à remettre en cause la technique en elle-même. Un sous-ensemble de fragments est constitué de fragments assez fortement dépendants entre eux, c'est-à-dire qui ont des chances d'être utilisés simultanément par une même requête[1]. En conséquence, plutôt que de traduire proportionnellement le sous-graphe logique en graphe physique, nous devons essayer de regrouper au plus physiquement ces fragments, c'est-à-dire les objets partiels concernés d'un même objet. En effet, à quoi bon placer physiquement éloignés deux objets partiels (d'un même objet) que les propriétés physiques du réseau permettent d'être proches? Cela ne veut pas dire pour autant que nous ne devons plus tenir compte du sous-graphe logique. Comme nous le verrons par la suite, celui ci nous servira, dans un algorithme de type glouton, à choisir à chaque étape le meilleur objet partiel à placer.

1. Le niveau de dépendance entre les fragments dépend bien sûr des paramètres de la phase1 ainsi que du choix relatif au découpage de la matrice des distances.

Cette seconde technique, que nous désignerons par ***placement par concentration***, peut être représentée par la figure suivante:



Sous-graphe logique



Réseau physique

Positions relatives physiques qui concentrent au maximum les objets partiels d'un même objet.

Fig. 3.62 : Principe du placement par concentration

Il est important de noter que nous aboutirons à un graphe physique d'autant plus différent (au sens proportions) que les contraintes liées au réseau seront faibles, c'est-à-dire que le nombre de noeuds et le degré seront élevés et que le diamètre sera faible. En effet, dans ce cas les fragments qui étaient logiquement éloignés auront de grandes chances d'être physiquement proches.

Notre exemple précédent appliqué à différents réseaux à huit noeuds donne les résultats suivants:

$n_{2i}(O) = (n_{1i}(O) + 1) \bmod 8$

$n_{3i}(O) = (n_{1i}(O) + 2) \bmod 8$

$n_{4i}(O) = (n_{1i}(O) + 3) \bmod 8$

.	f_{1i}	f_{2i}	f_{3i}	f_{4i}
f_{1i}	0	1	2	3
f_{2i}	1	0	1	2
f_{3i}	2	1	0	1
f_{4i}	3	2	1	0

$n_{2i}(O) = (n_{1i}(O) + 1) \bmod 8$

$n_{3i}(O) = (n_{1i}(O) + 2) \bmod 8$

$n_{4i}(O) = (n_{1i}(O) + 3) \bmod 8$

.	f_{1i}	f_{2i}	f_{3i}	f_{4i}
f_{1i}	0	1	1	2
f_{2i}	1	0	1	1
f_{3i}	1	1	0	1
f_{4i}	2	1	1	0

Fig. 3.63 : Exemples de placements par concentration sur différents réseaux

Nous allons maintenant présenter le principe de l'algorithme qui nous permet d'implanter assez simplement l'approche suggérée. Il est de type glouton en ce sens où, à chaque étape, il choisit la meilleure solution localement sans se préoccuper des étapes suivantes.

Le principe de la méthode est simple. Nous maintenons un ensemble FP de frag-

ments placés. A chaque étape, nous cherchons le fragment fi non encore placé qui est le plus près logiquement d'un des fragments fp de FP. Nous plaçons alors fi physiquement le plus près possible de fp et l'ajoutons à FP. Dans le cas où fi peut être placé à différents endroits qui correspondent à une même distance physique minimale, nous choisissons la solution optimale localement, c'est-à-dire celle qui minimise l'ensemble des distances physiques entre les fragments déjà placés et fi.

L'ensemble FP est initialisé par le fragment qui "guide" les autres. Nous verrons dans la section suivante (3.4 Comment répartir) que, suivant le mode de répartition choisi (par rapport aux valeurs ou par rapport aux identifiants), le choix de ce fragment pourra se faire différemment.

Nous donnons ci-dessous une expression de l'algorithme dans notre pseudo-langage de programmation:

```
PLACER_PHYSIQUEMENT( G'c_i, fragment_guide)
   /*  G'c_i sous-graphe des distances logiques entre fragments
       FP ensemble des fragments déjà placés
       Gc_ip graphe des distances physiques entre fragments
        du sous-ensemble i de la classe C */
       placer (fragment_guide)
   /*initialisation */
   FP <- fragment_guide

   /*traitement */
   TQ (il existe un fragment à placer )
   FAIRE
      /*recherche du fragment fi le plus proche logiquement
          d'un des fragments fp de FP*/
      (fi,fp) <- plus_proche(G'c_i,FP)
      /*positionnement de fi le plus près possible de fp, mise à jour de Gcip */
      positionner(fi,fp,Gc_ip)
      /*ajouter fi à FP */
      ajouter (fi,FP)
   FAIT
FIN
```

Remarques:

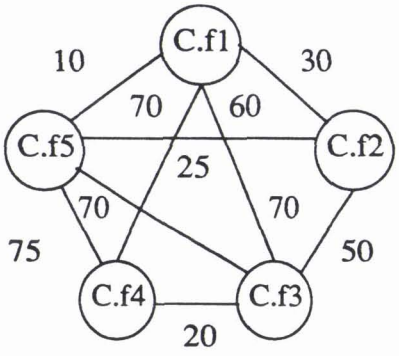
Le graphe physique mémorise les distances physiques entre les fragments mais aussi, pour tout couple de fragments, l'information qui permet de passer de l'un à l'autre.

La procédure ***positionner***(fi,fp,Gc_{ip}) fournit un résultat qui dépend du réseau utilisé. En effet, le positionnement du fragment fi est fonction des voisins disponibles de fp qui sont directement liés aux paramètres du réseau. L'expression de cette procédure dans notre pseudo-langage donne:

```
POSITIONNER ( fap, fpp, graphe)

   /* fap <=> fragment à placer
    fpp <=> fragment placé le plus proche logiquement
   graphe <=> graphe physique courant */

   /*initialisations*/
   ensemble_positions_disp <- { }
   dist_phys <- 0

   /*traitement*/
   REPETER
      dist_phys <- dist_phys + 1
      /*recherche de toutes les positions disponibles qui sont à une
      distance dist_phys du fragment placé le plus proche */
      ensemble_positions_disp <- cherch_pos(fpp,dist_phys,graphe)
   TQ (ensemble_positions_disp = { })

   /*analyse des positions disponibles */
   meilleure_pos <- anal_pos_disp (ensemble_positions_disp, graphe)

   /* mise à jour du graphe physique */
   maj_graphe(fap,meilleure_pos,graphe)

FIN
```

Nous supposons ici que le réseau est suffisamment grand, c'est-à-dire qu'il existe toujours une position disponible. D'un point de vue implantation seule, la procédure ***cherch_pos***() dépend du réseau utilisé.

Nous détaillons ci-dessous le déroulement de l'algorithme sur l'exemple qui nous avait servi à poser le problème :



.	f1	f2	f3	f4	f5
f1	0	30	60	70	10
f2	30	0	50	70	25
f3	60	50	0	20	70
f4	70	70	20	0	75
f5	10	25	70	75	0

Fig. 3.64 : Exemple pour l'illustration de l'algorithme PLACER_PHYSIQUEMENT

Etape	FP avant	fragment à placer	fragment de ref.	distance physique minimale	positions dispo.	meilleure position	FP après	fragments non placés
1	{f1}	f5	f1	1	$\overline{ou}$ 001 $\overline{ou}$ 010 $\overline{ou}$ 100	$\overline{ou}$ 001	{f1,f5}	{f2,f3,f4}
2	{f1,f5}	f2	f5	1	$\overline{ou}$ 011 $\overline{ou}$ 101	$\overline{ou}$ 011	{f1,f5,f2}	{f3,f4}
3	{f1,f2, f3}	f3	f2	1	$\overline{ou}$ 010 $\overline{ou}$ 110	$\overline{ou}$ 010	{f1,f5, f2,f3}	{f4}
4	{f1,f2, f3,f5}	f4	f3	1	$\overline{ou}$ 110	$\overline{ou}$ 110	{f1,f2, f3,f5,f4}	{}

Fig. 3.65 : Transformation des positions logiques en positions physiques adaptées à un hyper-cube de degré 3 et en prenant comme fragment guide f1

Remarque: Dans le tableau ci-dessus, l'opérateur $\overline{ou}$ désigne le ou_exclusif et dans les résultats ci-dessous un couple (x,y) s'interprète comme: (distance physique, équation de passage).

	f1	f2	f3	f4	f5
f1	---	(2,$\overline{ou}$ 011)	(1,$\overline{ou}$ 010)	(2,$\overline{ou}$ 110)	(1,$\overline{ou}$ 001)
f2	(2,$\overline{ou}$ 011)	---	(1,$\overline{ou}$ 001)	(2,$\overline{ou}$ 101)	(1,$\overline{ou}$ 010)
f3	(1,$\overline{ou}$ 010)	(1,$\overline{ou}$ 001)	---	(1,$\overline{ou}$ 100)	(2,$\overline{ou}$ 011)
f4	(2,$\overline{ou}$ 110)	(2,$\overline{ou}$ 101)	(1,$\overline{ou}$ 100)	---	(3,$\overline{ou}$ 111)
f5	(1,$\overline{ou}$ 001)	(1,$\overline{ou}$ 010)	(2,$\overline{ou}$ 011)	(3,$\overline{ou}$ 111)	---

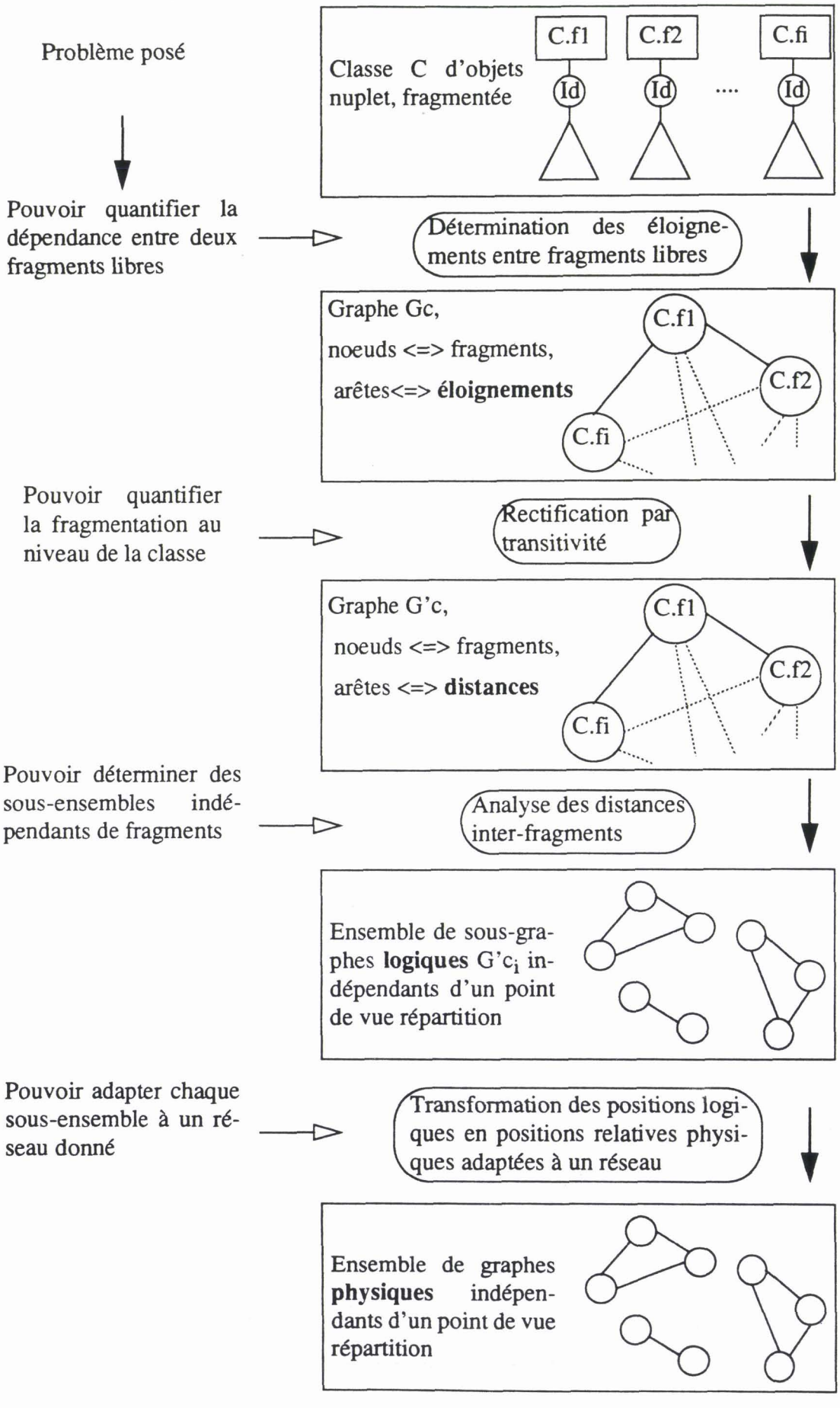
Fig. 3.66 : Représentation sous forme de matrice du graphe physique obtenu

3.3.7. Synthèse sur la répartition relative

Nous venons donc d'analyser, en détail, les différents problèmes relatifs à la répartition des occurrences des fragments d'une même classe. Parallèlement à cela, nous avons proposé un certain nombres d'outils qui permettent de quantifier cette analyse et par la même d'automatiser ou tout au moins de guider les prises de décision importantes.

De tout cela, il ressort une démarche générale applicable à chaque classe d'objets nuplet et que nous pouvons représenter par le diagramme suivant:

Fig. 3.67 : Processus général de répartition relative



En sortie nous obtenons donc un graphe physique par sous-ensemble de fragments. Ce graphe mémorise les positions relatives physiques des fragments concernés. Il sera donc utilisé quand il faudra réunir, dans une même requête, deux objets partiels (occurrences de fragments du sous-ensemble) issus d'un même objet, puisqu'il nous permettra de les localiser l'un par rapport à l'autre. Cependant, ce graphe n'est pas suffisant pour définir totalement la répartition des objets partiels du sous-ensemble. Il nous fournit uniquement des informations de positionnement relatif et ne se préoccupe pas, par conséquent, de la composante "absolue" de la répartition.

Nous allons donc maintenant nous intéresser à cet aspect. Pour cela, nous étudierons les différents modes de répartition utilisables et leur influence ainsi que les problèmes posés pour obtenir une répartition homogène. Cette analyse nous permettra d'ajouter le maillon manquant au processus de répartition des occurrences des fragments d'une même classe.

3.4. Comment répartir

3.4.1. Les différents modes de répartition et leur influence

Nous nous proposons d'étudier ici les différents modes de répartition dont nous disposons afin de placer les occurrences d'un sous-ensemble de fragments d'une classe C. Nous avons précédemment vu comment obtenir un positionnement relatif des objets partiels d'un même objet. Nous avons aussi évoqué le fait qu'un positionnement totalement relatif de ces objets partiels ne permet d'utiliser qu'un seul critère de répartition lié au fragment guide utilisé. Comme nous allons le voir, le choix de ce fragment guide se fera différemment suivant le mode de répartition. La situation peut donc se résumer par la figure suivante:

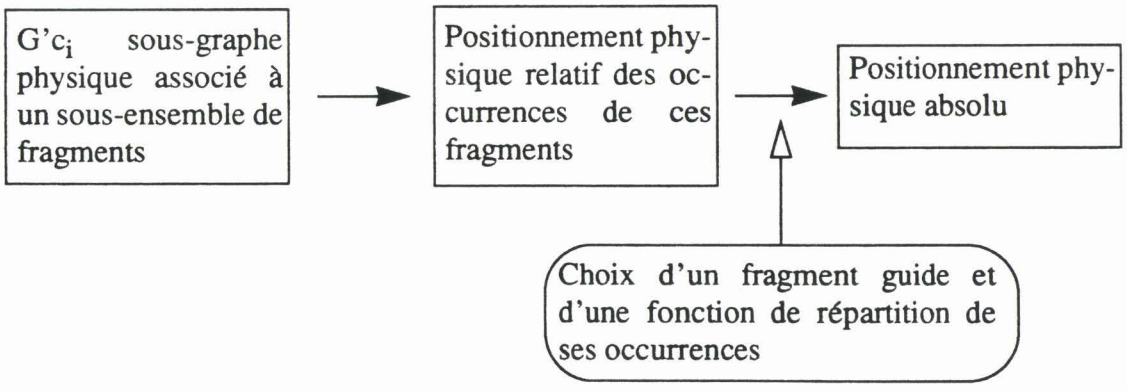


Fig. 3.68 : Principe du positionnement physique absolu

D'une façon générale, dans les systèmes parallèles qui partitionnent horizontalement des données, la répartition se fait par l'utilisation de fonctions de hachage ou d'intervalles de valeurs (cf 3.1.4 Travaux sur la fragmentation horizontale). Dans le cas des fonctions de hachage, celles-ci portent sur les valeurs d'un attribut. Les intervalles de valeurs sont déterminés quant à eux soit par l'utilisateur soit automatiquement comme dans la machine GAMMA [DEWI86]. A un intervalle donné correspond alors un noeud du réseau. Pour l'une ou l'autre des méthodes, la répartition est basée sur les valeurs. Ceci est essentiellement dû au fait que ces machines sont construites soit sur le modèle relationnel soit sur un autre modèle valeur.

Par contre, le modèle que nous utilisons manipule deux types de données qui sont les objets et les valeurs, ce qui est d'ailleurs le cas, au vocabulaire près, de la plupart des modèles orientés objet. Tout objet étant identifié par une valeur interne (identifiant), deux modes de répartition des objets s'offrent à nous: la répartition par rapport aux identifiants et la répartition par rapport aux valeurs.

3.4.1.1. Répartition par rapport aux identifiants

Dans ce cas, un objet partiel du fragment guide[1] est placé par rapport à son identifiant qui, rappelons-le, est le même pour tous les objets partiels d'un même objet. Les autres objets partiels (relatifs au même sous-ensemble de fragments) sont alors placés grâce au graphe G_{cip} des positions physiques relatives.

Notations: Soit O un objet nuplet, nous appellerons:

- $f_n(O)$: l'objet partiel de O occurrence du fragment f_n
  entre autres $f_g(O)$ <=> objet partiel de O occurrence du fragment guide
- $n_n(O)$: le noeud physique où $f_n(O)$ est localisé
  => $n_g(O)$ noeud physique où $f_g(O)$ est localisé
- R_g: la fonction de répartition des occurrences du fragment guide
  <=> R_g: identifiant ----> noeud physique de réseau
  <=> $R_g(Id(f_g(O))) = n_g(O)$
- LR_{gk}: la fonction de localisation de $f_k(O)$ relativement à $f_g(O)$ qui est issue du graphe physique G_{cip}
  <=> LR_{gk}: noeud de $f_g(O)$ ----> noeud de $f_k(O)$
  <=> $LR_{gk}(n_g(O)) = n_k(O)$

Nous donnons ci-dessous une représentation schématique du processus de répartition par rapport aux identifiants:

---

1. Le choix de ce fragment guide sera discuté dans la page suivante.

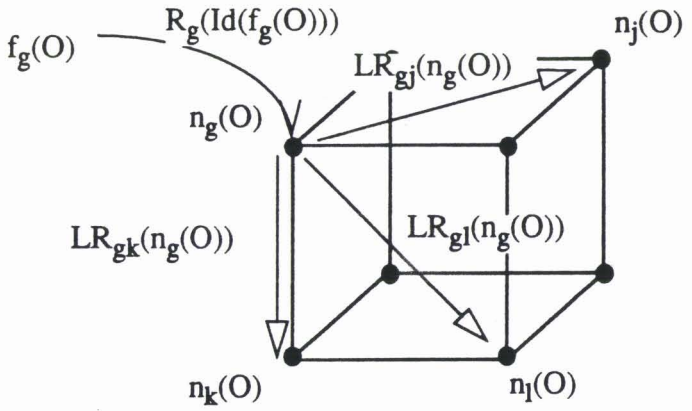


Fig. 3.69 : Représentation du processus de répartition par rapport aux identifiants

Il est important de noter qu'avec une telle répartition la connaissance de la fonction R_g (fonction de répartition des occurrences du fragment guide) et de la fonction LR_{gk} (fonction de localisation de $f_k(O)$ relativement à $f_g(O)$) nous permet de déduire une fonction LD_k de localisation directe des occurrences de f_k applicable directement, c'est-à-dire sans "passer" par le fragment guide. Ceci est dû au fait que les objets partiels d'un même objet ont le même identifiant qui est lui-même la clé de la répartition. Nous retrouvons mathématiquement ce résultat en écrivant:

$$n_g(O) = R_g(Id(f_g(O))) \text{ et } n_k(O) = LR_{gk}(n_g(O))$$

$$<=> n_k(O) = LR_{gk}(R_g(Id(f_g(O)))) = LR_{gk}(R_g(Id(O))) = LR_{gk}(Rg(Id(f_k(O)))$$

$$<=> n_k(O) = (LR_{gk} \circ R_g)\ (Id(f_k(O)))$$

$$\boxed{<=> n_k(O) = LD_k(Id(f_k(O)))}$$

En conséquence, l'utilisation de la répartition par rapport aux identifiants au niveau d'un sous-ensemble de fragments, "met sur le même pied tous les fragments". En effet, dès que nous choisissons une fonction de répartition pour les occurrences de l'un des fragments nous pouvons en déduire une pour chaque autre fragment. Cela ne veut pas dire pour autant que nous avons plusieurs critères de répartition pour le sous-ensemble de fragments; ces fonctions sont en fait liées par l'unique critère commun qu'est l'identifiant. En revanche, l'existence d'une fonction de répartition par fragment garantit un accès direct à tout objet partiel, c'est-à-dire un accès sans "passage obligé" par l'objet partiel associé du fragment guide.

Nous allons maintenant aborber le problème du choix du fragment guide. Nous venons de voir que la répartition par rapport aux identifiants rend "équivalents" les fragments d'un point de vue accès. Nous devons cependant nous souvenir que le résultat de la transformation des positions relatives logiques en positions relatives physiques dépend du fragment guide choisi. En effet, la méthode proposée cherche à placer au plus près les autres fragments et cela dans un ordre qui dépend du fragment guide(cf 3.3.6).

Le choix du fragment guide jouera sur la disposition relative finale des fragments. Cette influence sera d'autant plus forte que le réseau aura de "pauvres" propriétés, c'est-à-dire un faible degré et un fort diamètre.

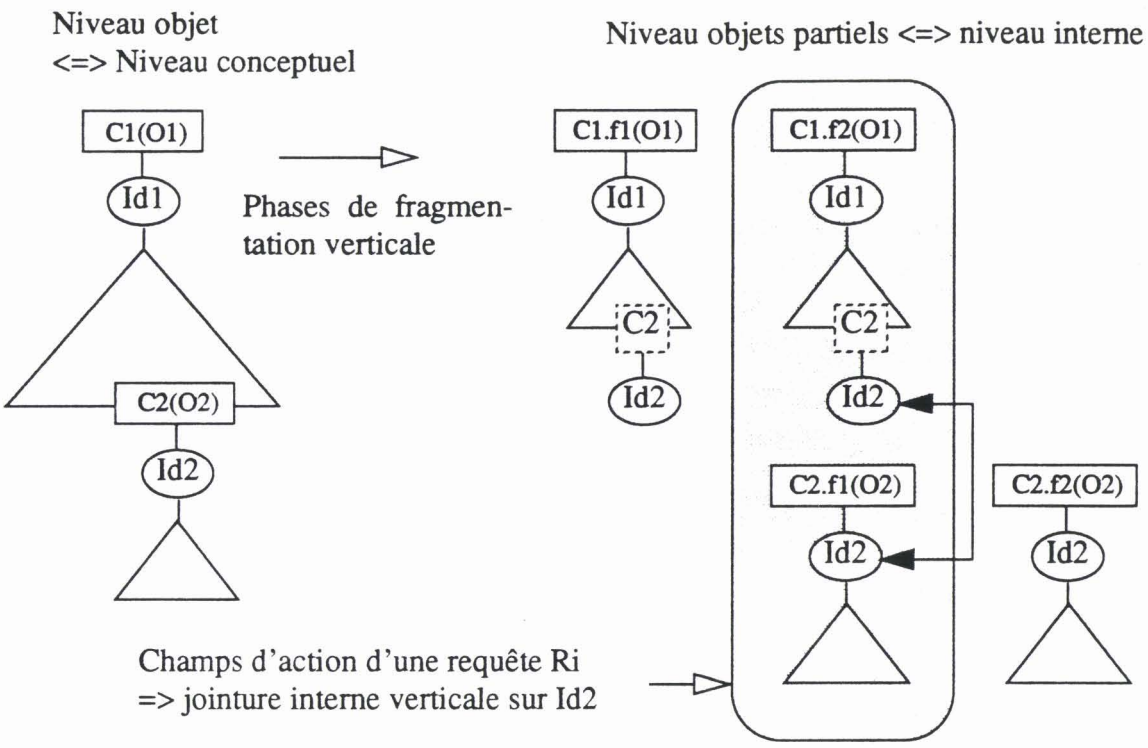
Puisque l'utilisation de la répartition par rapport aux identifiants nous laisse toute liberté sur le choix de ce fragment guide, nous devons opter pour celui qui aboutit à la meilleure disposition relative finale des fragments, c'est-à-dire à celle où les dépendances logiques entre les fragments sont le mieux traduit physiquement. Pour cela, il faut analyser les résultats de la transformation des positions logiques en positions physiques en prenant tour à tour chacun des fragments comme fragment guide. Une telle analyse nécessiterait d'introduire un outil de comparaison des différentes configurations obtenues.

Après avoir analysé le principe de la répartition par rapport aux identifiants, nous allons maintenant en extraire certains avantages. Comme nous l'avons souligné précédemment, (cf chapitre 1) la notion d'identité d'objet confère au modèle utilisé certaines propriétés.Il est donc intéressant d'étudier l'interaction entre ces propriétés et l'utilisation de la répartition par rapport aux identifiants.

Un identifiant interne permet de distinguer clairement les concepts de valeur (ou encore d'état) et d'identité d'un objet [KHOS86]. Une répartition basée sur les identifiants est donc forcément indépendante des valeurs. Puisque l'identifiant est attribué à un objet pour toute son existence (propriété d'indépendance par rapport au temps), une telle répartition est donc figée dans le temps. En conséquence, un objet partiel ainsi placé ne migrera jamais dans le réseau sauf en cas de changement de la fonction de répartition elle-même. Néanmoins, cette dépendance contestable entre un objet partiel et sa localisation se limite à un niveau macroscopique, c'est-à-dire au niveau des noeuds du réseau. D'autre part, cette dépendance partielle n'est que relative au sous-ensemble de fragments traité. Si nous admettons la duplication de fragments dans différents sous-ensembles, un même objet partiel peut être placé sur des noeuds différents de par l'utilisation de fonctions de répartition différentes pour chaque sous-ensemble. Enfin, le fait que les objets partiels ne migrent pas peut être vu comme un avantage dans la mesure où il évite un surplus de communications dû à ces migrations.

Une autre propriété importante de la notion d'identité d'objet est de permettre l'agrégation et le partage d'objets; un objet peut être composant d'un ou plusieurs objets composés. En cas de stockage normalisé des objets partiels composants (cf phase 2 de la fragmentation verticale), l'exécution d'une requête qui utilise un objet partiel composé et l'objet partiel composant associé, nécessite une jointure interne verticale (cf phase 2) sur l'identifiant de cet objet partiel composant. L'utilisation d'une répartition par rapport aux identifiants pour les objets partiels composants rend alors équivalents le critère de répartition et le critère de jointure implicite, ce qui permet un accès direct aux objets partiels composants concernés par une telle requête. Cela évite l'utilisation d'une structure d'accès secondaire ou, pire, le recours à une opération de diffusion pour rechercher ces objets partiels composants à partir de leur identifiant.

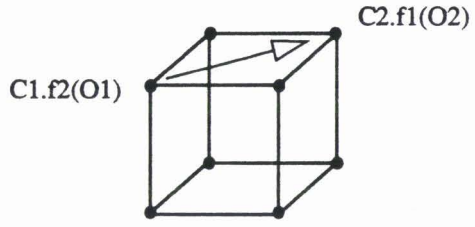
Nous donnons sur la figure suivante, sur un exemple, une représentation imagée de l'opération de jointure interne verticale en fonction de la répartition utilisée.



Cas d'une répartition par rapport aux identifiants des occurrences de C2.f1:

1) accès à C1.f2(Id1)
   => obtention de Id2
2) accès direct à C2.f1(Id2)

<=>



Cas d'un autre type de répartition des occurrences de C2.f1:

1) accès à C1.f2(Id1)
   => obtention de Id2
2) diffusion pour trouver C2.f1(Id2)

<=>

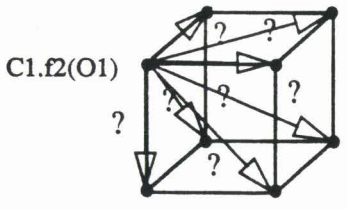


Fig. 3.70 : Répartition par rapport aux identifiants et jointures internes verticales

Un dernier avantage, que nous pouvons attribuer à la répartition par rapport aux identifiants, est le fait que le format fixe de ces valeurs internes rendra plus facile la détermination d'une fonction de hachage homogène. Nous reviendrons en détail sur ce point dans la section suivante qui concerne les problèmes relatifs à l'obtention d'une répartition homogène (cf 3.4.2 Détermination des fonctions de répartition).

Bien sûr, la répartition par rapport aux identifiants a des désavantages; étant donné

que ces derniers ne sont autres que les avantages qui peuvent être attribués à la répartition par rapport aux valeurs, nous allons maintenant nous intéresser à ce deuxième type de répartition.

3.4.1.2. Répartition par rapport aux valeurs

Dans ce cas, un objet partiel du fragment guide est placé par rapport à la valeur de l'un de ses attributs, valeur qui devient donc clé de répartition. Les autres objets partiels du même objet sont alors placés, comme pour la répartition par rapport aux identifiants, grâce au graphe Gc_{ip} des positions physiques relatives.

La fonction R_g de répartition des occurrences du fragment guide a donc, dans ce cas, comme ensemble de départ le domaine des valeurs de l'attribut choisi dans ce fragment guide. Nous noterons $val\_att_g(O)$ la valeur de cet attribut pour un objet O donné. En reprenant les autres notations précédemment utilisées, nous pouvons représenter la répartition par rapport aux valeurs par le schéma suivant:

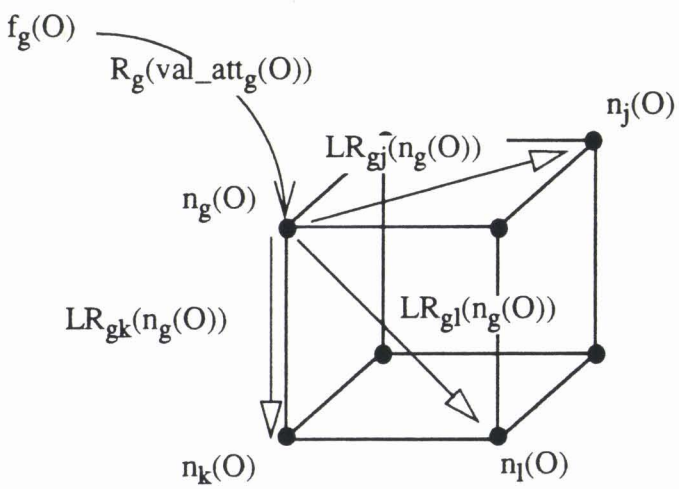


Fig. 3.71 : Représentation du processus de répartition par rapport aux valeurs

Contrairement à ce qui se passait pour la répartition par rapport aux identifiants, la connaissance de R_g et LR_{gk} ne nous permet pas ici de déduire une fonction de localisation directe LD_k des occurrences du fragment f_k . En effet, l'attribut qui sert de clé pour la répartition des occurrences du fragment guide est propre à ce fragment (sauf dans le cas où il est dupliqué dans un autre fragment). En conséquence, une valeur de cet attribut ne peut être connue en dehors de f_g qui, dès lors, devient un passage obligé pour tout accès à une occurrence d'un autre fragment (sauf si l'on dispose d'une structure d'accès secondaire). Ce résultat peut s'exprimer avec les notations utilisées par:

$$n_g(O)=R_g(val\_att_g(O)) \text{ et } n_k(O)=LR_{gk}(n_g(O))$$

$$<=> n_k(O)=(LR_{gk} \circ R_g)(val\_att_g(O)) \quad <=> \quad n_k(O)=LD_k(val\_att_g(O))$$

Il apparait donc que l'argument de LD_k est une valeur propre au fragment guide, inconnue au niveau de f_k. Nous ne pouvons donc déterminer, dans ce cas, une fonction de localisation directe des occurrences de f_k. Un accès direct à une telle occurrence ne pourra donc se faire que s'il existe une structure d'accès secondaire adéquate.

En conclusion, pour un sous-ensemble de fragments donné, seules les occurrences du fragment guide pourront bénéficier d'accès directs. De plus, ces accès directs ne pourront se faire que par rapport aux valeurs d'un seul attribut. Ces restrictions, a priori draconiennes, peuvent néanmoins prendre l'allure d'avantages en faveur de la répartition par rapport aux valeurs lorsque le fragment guide est massivement sollicité par rapport à l'attribut clé choisi. En effet, en présence d'une majorité de requêtes qui opèrent des sélections par rapport aux valeurs de cet attribut clé, la répartition par rapport aux valeurs sera forcément mieux adaptée. Elle le sera d'ailleurs d'autant mieux que le nombre de fragments du sous-ensemble traité sera faible puisque nous aurons alors moins de fragments "défavorisés" autour du fragment guide choisi. A l'extrême, pour les sous-ensembles constitués d'un seul fragment, le problème du manque de fonctions de localisation directe disparaîtra, libérant ainsi la répartition par rapport aux valeurs de ses inconvénients.

Nous allons maintenant analyser, sur un exemple, comment l'opération importante de jointure explicite [1] peut tirer parti d'une répartition par rapport aux valeurs. Comme nous l'avons souligné précédemment, bien que le modèle utilisé permette de construire et manipuler des objets complexes, la conceptualisation peut conduire à traduire certains liens entre classes d'objets comme on le fait avec le modèle relationnel, c'est-à-dire à utiliser des attributs de jointures explicites.

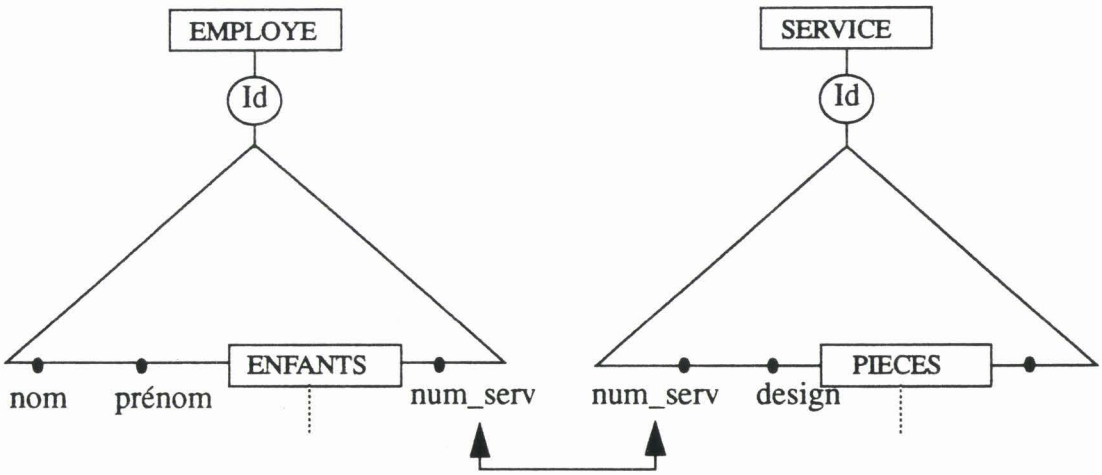


Fig. 3.72 : Exemple d'utilisation d'un attribut de jointure en FAD

Au niveau du langage FAD cité, le constructeur d'action FILTER permet alors à l'utilisateur d'exprimer l'opération de jointure explicite (cf chapitre 1 section sur la présentation de FAD).

Dans un contexte parallèle, le mode de répartition des occurrences des fragments

1. Nous employons ici le terme d'explicite afin de différencier cette opération de jointure exprimée par l'utilisateur des opérations de jointures internes horizontales ou verticales induites par la fragmentation verticale effectuée.

concernés par la jointure va jouer un rôle important sur l'exécution de l'opération. Nous allons illustrer cela à partir de notre exemple (cf figure 3.72).

Supposons que EMPLOYE.f1 et SERVICE.f1 soient les fragments (respectivement de la classe EMPLOYE et de la classe SERVICE) concernés par la jointure exprimée ci-dessus en langage FAD.

Les occurrences de chacun de ces fragments peuvent être réparties par rapport à leur valeur pour l'attribut num_serv (attribut de jointure) ou par rapport à leur identifiant ou à la valeur d'un autre attribut. Nous désignerons par autres répartitions, celles du deuxième type (identifiant ou valeur d'un autre attribut). Puisque deux types de répartition sont offerts à chaque fragment, nous pouvons distinguer quatre cas différents. Nous proposons ci-dessus une étude des coûts de communication associés à l'opération de jointure proposée pour chacun des cas. Pour cela, nous supposerons qu'il n'existe pas, a priori, de structures d'accès secondaire.

Notations:

. ***nb_noeuds_emp***: nombre de noeuds utilisés pour la répartition des objets de type EMPLOYE

. ***nb_noeuds_serv***: nombre de noeuds utilisés pour la répartition des objets de type SERVICE

. ***nb_emp***: nombre d'objets partiels de type EMPLOYE.f1 par noeud[1]

. ***nb_serv***: nombre d'objets partiels de type SERVICE.f1 par noeud

. ***taille_emp***: taille de l'information utile au résultat pour un objet de type EMPLOYE

. ***taille_serv***: taille de l'information utile au résultat pour un objet de type SERVICE

. ***taille_res***: taille d'un résultat élémentaire, ie taille de l'information reconstituée sur un employé et son service

. ***taille_message***: taille maximale d'un message

PREMIER CAS: Les occurrences de EMPLOYE.f1 et de SERVICE.f1 sont réparties par rapport aux valeurs de l'attribut num_serv et grâce à la même fonction de répartition[2].

Dans ce cas, tous les objets partiels de EMPLOYE.f1 qui ont même valeur pour l'attribut num_serv, sont localisés sur un même noeud où se trouve également l'objet partiel de SERVICE.f1 associé. En d'autres termes, les deux parties d'information nécessaires à la jointure sont toujours localisées sur le même noeud.

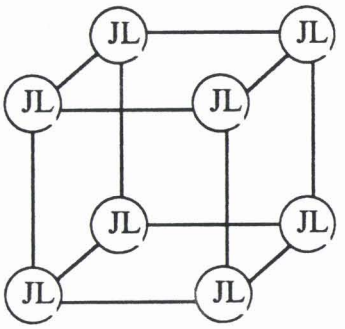
L'opération globale se décompose donc en jointures locales qui ne nécessitent aucune communication. Seule la phase éventuelle de concentration[3] des résultats entraîne un coût de communication en nombre de messages que nous pouvons évaluer par:

---

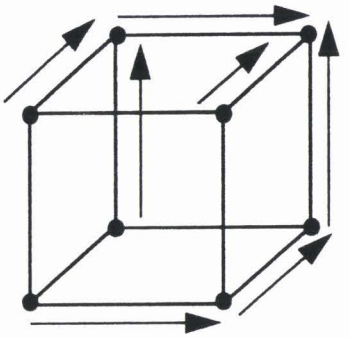
1. Nous supposerons que les objets partiels concernés sont uniformément répartis. Les problèmes relatifs à l'obtention d'une répartition uniforme seront traités dans la section suivante(cf 3.4.2 et 3.4.3)
2. L'utilisation de deux fonctions de répartition différentes serait, dans ce cas, pour le moins absurde et cela autant par la difficulté d'en trouver deux parfaitement homogènes que par la perte de performances qui en découlerait.
3. Cette phase de concentration des résultats peut ne pas exister pour des raisons d'optimisation des opérations ultérieures.

nb_messages(CAS1)= (nb_noeud_serv - 1)*(nb_emp *taille_res / taille_message)

Remarque: dans ce cas, nb_noeud_emp= nb_noeud_serv



PHASE 1: Jointures locales



PHASE 2: concentration des résultats

Fig. 3.73 : Représentation du premier cas

SECOND CAS: Les occurrences de EMPLOYE.f1 sont réparties par rapport aux valeurs de l'attribut num_serv contrairement à celles de SERVICE.f1.

Dans ce cas, il n'y a plus localité entre "un service et ses employés" mais les objets partiels de EMPLOYE.f1 qui ont même valeur pour l'attribut num_service, sont localisés sur un même noeud.

L'opération de jointure peut être décomposée de la façon suivante:

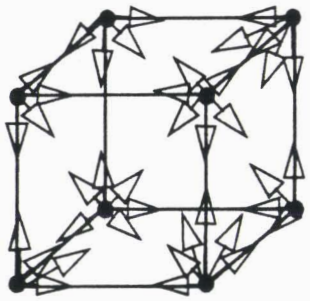
. PHASE1: Envoi des informations utiles des services vers les noeuds concernés.
Ce n'est pas une diffusion puisque les destinations sont connues.

. PHASE2: Jointures locales.

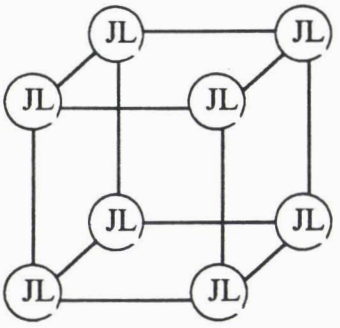
. PHASE3: Concentration des résultats.

Le coût en communication peut être évalué par:

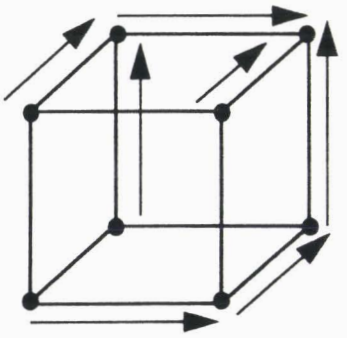
nb_messages(CAS2)= (nb_noeuds_serv * (nb_serv*taille_serv / taille_message)) +
nb-messages(CAS1)



PHASE 1: envoi des informations utiles des services vers les noeuds concernés



PHASE 2: Jointures locales



PHASE 3: concentration des résultats

Fig. 3.74 : Représentation du second cas

TROISIEME CAS: Les occurrences de SERVICE. f1 sont réparties par rapport aux valeurs de l'attribut num_serv contrairement à celles de EMPLOYE.f1.

Il n'y a pas non plus ici localité entre "un service et ses employés" et, de plus, les objets partiels de EMPLOYE.f1 qui ont même valeur pour l'attribut num_serv sont éparpillés sur le réseau.

Puisqu'il y a, a priori, bien moins de services que d'employés, l'opération de jointure peut être schématisée par:

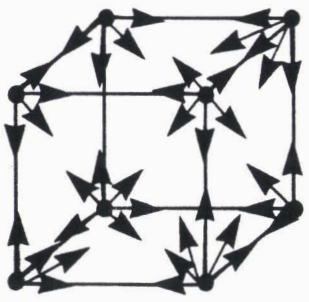
. PHASE 1: Diffusion de toutes les informations sur tous les services, sur tous les noeuds.

. PHASE 2: Jointure locale (mais plus de données à manipuler).

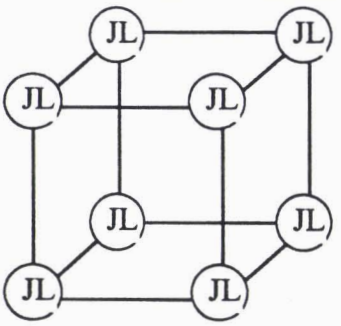
. PHASE 3: Concentration des résultats.

Le coût en communication peut être évalué par:

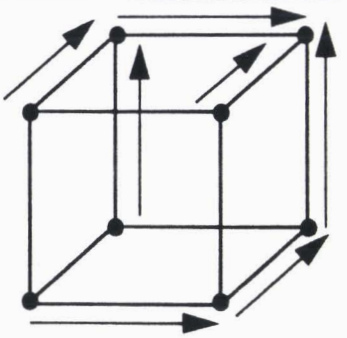
```
nb_messages(CAS3)= (nb_noeuds_serv * (nb_serv * taille_serv / taille_message) *
                    (nb_noeud_serv-1) +    nb-messages(CAS1)
```



PHASE 1: diffusion de toutes les informations sur tous les services vers tous les noeuds



PHASE 2: Jointures locales



PHASE 3: concentration des résultats

Fig. 3.75 : Représentation du troisième cas

Remarque: Le fait que les occurrences de SERVICE.f1 soient réparties par rapport aux valeurs de num_service, ne nous est, dans ce cas, d'aucune utilité puisqu'il vaut mieux diffuser toutes les informations sur les services plutôt que de déplacer les nombreuses "informations Employés" vers les services.

Les jointures locales seront, dans ce cas, plus longues à traiter puisque les volumes de données manipulées seront localement plus importants et plus hétérogènes vis-à-vis de l'attribut de jointure.

QUATRIEME CAS: Ni les occurrences de EMPLOYE.f1, ni celles de SERVICE.f1 ne sont réparties par rapport aux valeurs de l'attribut num_serv.

Nous pouvons appliquer, dans ce cas, la même stratégie que pour le cas précèdent, ce qui nous amène au même coût de communication.

Si nous faisons abstraction du coût de communication dû à la concentration des résultats et qui est le même pour chacun des cas, nous nous rendons compte que, suivant les modes de répartition choisis, le nombre de messages qui transitent sur le réseau varie considérablement.

	Complexité en nombre de messages "logiques"
CAS1	0
CAS2	O(***nb_noeuds_serv***)
CAS3 et 4	O(***nb_noeuds_serv*** ^{2})

Fig. 3.76 : Compléxité des différents cas

Cet exemple simpliste met donc bien en évidence l'adéquation qui peut exister entre les opérations de jointures explicites et la répartition par rapport aux valeurs. Bien sûr, cette dernière est à relativiser par rapport à l'utilisation globale des fragments concernés.

Pour la répartition par rapport aux valeurs, nous retrouvons au rang des inconvénients les avantages précédemment cités pour la répartition par rapport aux identifiants. Dans ce cas en effet, les objets partiels peuvent être amenés à migrer lors de mises à jour de la valeur qui sert à la répartition; il est d'ailleurs important de remarquer qu'une telle mise à jour n'entraînerait pas uniquement la migration d'un objet partiel mais celle de tous les objets partiels (du même objet) qui appartiennent au sous-ensemble de fragments concerné[1]. D'autre part, la répartition par rapport aux valeurs d'un ensemble d'objets partiels composants rend plus complexes donc plus coûteuses les éventuelles opérations de jointures internes verticales qui s'opèrent par rapport aux identifiants. Enfin, le fait que les clés de répartition soient des valeurs,a priori, de nature et de formats irréguliers rendra plus difficile la détermination d'une fonction de répartition homogène qui assure une répartition de charge équilibrée.

1. Cela est, bien sur, dû au fait que les objets partiels d'un même objet qui appartiennent à un même sous-ensemble de fragments sont répartis les uns par rapport aux autres.

Nous proposons ci-dessous un tableau récapitulatif des avantages de chaque mode de répartition analysé.

Pour un sous-ensemble donné de fragments:

Répartition par rapport aux identifiants	Répartition par rapport aux valeurs
. Existence d'une fonction de localisation directe pour chaque fragment du sous-ensemble . Pas de migration des objets partiels (sauf en cas de changement de la fonction de répartition) . Adéquation avec les opérations de jointures internes verticales . Homogénéité de la clé de répartition en taille et "nature"	. Adéquation avec les opérations qui sollicitent le fragment guide par rapport à la clé . Adéquation avec les opérations de jointures explicites

Fig. 3.77 : Récapitulatif des avantages de chaque mode de répartition

Ce tableau fournit les éléments clés utiles à la prise de décision en faveur de l'un ou l'autre des modes de répartition pour un sous-ensemble donné de fragments. Là encore, il serait intéressant qu'un administrateur dispose d'informations quantitatives relatives à chacun de ces points critiques afin d'orienter son choix. Nous ne développerons pas ici l'analyse d'un tel outil mais nous pouvons néanmoins en donner les grandes lignes. Il faudrait, dans un premier temps, établir une classification des différentes requêtes envisageables, et cela, par rapport à leur adéquation avec les modes de répartition. Pour un sous-ensemble donné de fragments, il suffirait d'analyser les requêtes qui s'y rapportent et de quantifier les proportions relatives pour chacune des classes de référence.

Pour conclure sur les différents modes de répartition, nous pouvons remarquer que la répartition par rapport aux valeurs n'est une solution intéressante que pour des cas bien précis en dehors desquels elle risque d'être très pénalisante. A l'opposé, la répartition par rapport aux identifiants reste une solution moyenne même dans les mauvais cas de figure puisqu'elle met sur le même pied les différents fragments d'un sous-ensemble. Cette dernière peut donc être vue soit comme une solution adaptée (beaucoup de jointures internes verticales), soit comme la solution "la moins pire" lorsque la répartition par rapport aux valeurs n'est pas envisageable.

Après avoir analysé l'influence des différents critères de répartition, nous allons maintenant nous intéresser aux problèmes relatifs à la détermination de "bonnes" fonctions de répartition.

3.4.2. Détermination des fonctions de répartition

Le problème qui nous est posé ici est donc: étant donné un ensemble d'objets partiels occurrences d'un même fragment, un critère de répartition (identifiant ou attribut donné) et un ensemble de noeuds du réseau, comment établir une fonction de répartition globalement homogène qui à tout objet partiel associe un noeud, ou en d'autres termes, une fonction qui répartisse à peu près le même nombre d'occurrences par noeud?

Ce problème d'homogénéité est bien sûr lié à la répartition de charge que l'on veut être la plus équilibrée possible. Dans notre contexte, c'est-à-dire pour une machine bases de données parallèle sans mémoire partageable, les notions de répartition de données et de répartition de charge sont fortement liées puisque ce genre de modèle va dans le sens d'une exécution là où les données se trouvent (cf chapitre 1).

Une telle fonction qui à une valeur donnée (identifiant ou valeur d'un attribut) associe "l'adresse" d'un noeud du réseau, peut être qualifiée de fonction de hachage. Elle permet de placer un objet partiel mais aussi de le retrouver en cas de besoin. Elle constitue donc au niveau macroscopique des noeuds du réseau, un outils d'adressage par le contenu[1].

De nombreuses études ont été menées dans le domaine des fonctions de hachage. Bien que le contexte usuellement considéré soit quelque peu différent du nôtre, il nous semble nécessaire d'évoquer les différents résultats obtenus ainsi que les différentes classifications des fonctions proposées. Nous présentons donc dans ce qui suit une synthèse de ces résultats agrémentée d'une discussion sur leur utilisation potentielle dans notre contexte.

3.4.2.1. Principe général d'une fonction de hachage

Comme nous l'avons évoqué précédemment, le rôle d'une fonction de hachage est d'associer à une valeur souvent appelée clé et issue d'une information à placer, une adresse parmi un espace d'adressage connu. Une telle fonction a donc comme ensemble de départ le domaine des valeurs clé et comme ensemble d'arrivée l'espace d'adressage.

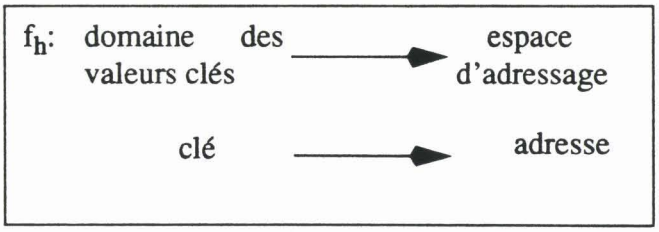


Fig. 3.78 : Principe d'une fonction de hachage

Le contexte, habituellement considéré dans les études de ces fonctions, est celui de placement des enregistrements d'un fichier sur des pages mémoire. Il s'agit donc là d'un niveau

1. L'identifiant doit pour cela être considéré comme faisant partie du contenu d'un objet partiel.

d'adressage plus fin que le nôtre et qui, de ce fait, requiert la mise en oeuvre de certains mécanismes qui ne nous seront pas nécessaires. Dans ce contexte, l'utilisation d'une technique de hachage peut être décomposée selon deux aspects que constituent le placement et la recherche des enregistrements.

. Placement d'un enregistrement

La fonction retenue est appliquée à la clé extraite de l'enregistrement et fournit ainsi une adresse. Deux cas peuvent alors survenir suivant que l'adresse proposée peut ou ne peut pas recevoir l'enregistrement. Dans le premier cas, l'opération de placement est terminée. Dans le second cas, lorsque la page mémoire est pleine[1], on parle de collision; la mise en oeuvre d'un mécanisme de résolution des collisions est alors nécessaire pour achever le placement.

Deux techniques sont utilisables pour résoudre les collisions: le chaînage et l'adressage progressif. Avec la première technique, l'enregistrement est placé dans une page chaînée à celle initialement désignée. Suivant le nombre de collisions, le chaînage peut être plus ou moins long, ce qui fournit des chemins d'accès de longueurs différentes et constitue ainsi un des principaux désavantages de la méthode. La seconde technique, c'est-à-dire l'adressage progressif, consiste à trouver une autre place libre pour l'enregistrement. Cette recherche peut se faire de façon séquentielle, par l'application d'une autre fonction ou par tout autre méthode hybride. Nous donnons ci-dessous une figure qui résume le placement d'un enregistrement.

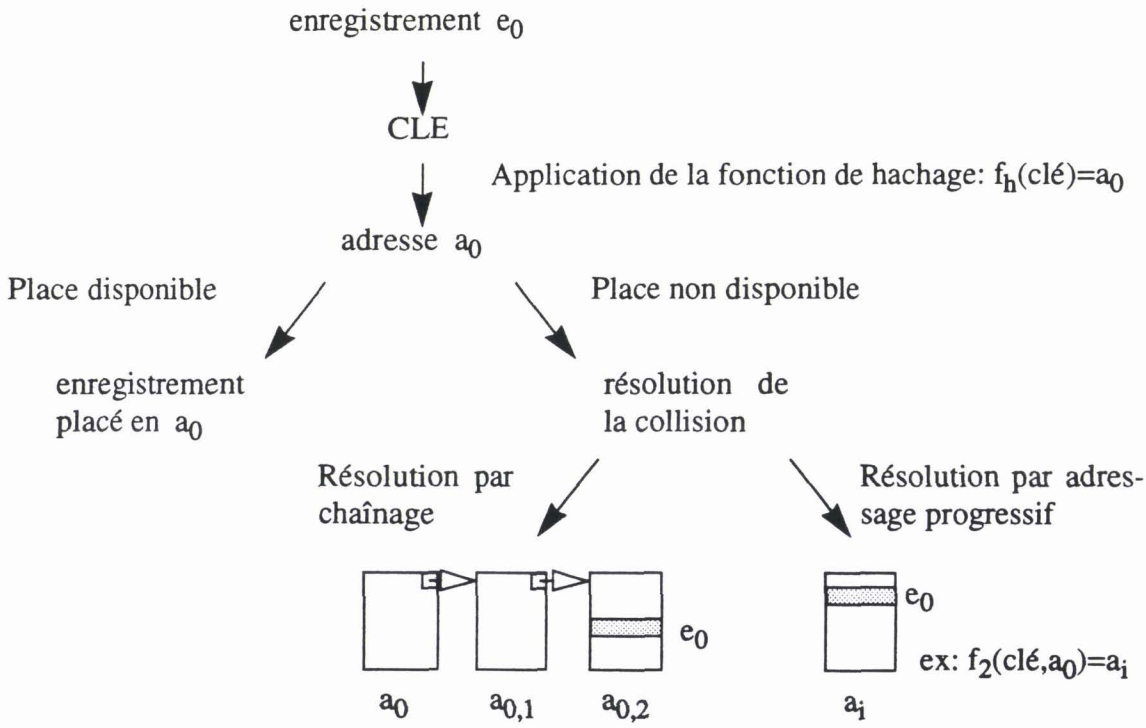


Fig. 3.79 : Principe du placement d'un enregistrement

. Recherche d'un enregistrement

1. "Pleine" est à prendre ici comme "remplie au delà du seuil limite fixé".

Pour retrouver un enregistrement e_0 il suffit de recalculer à partir de la clé l'adresse a_0 et de consulter la page désignée. Si e_0 ne s'y trouve pas c'est qu'il a été placé sur une autre page par le mécanisme de résolution de collisions, qu'il suffit alors d'appliquer à nouveau de la même façon.

Avant de présenter les différentes classifications usuellement citées dans la littérature, il est intéressant de faire quelques remarques à propos de la notion de collisions. En fait, les mécanismes de résolution de collisions ne sont là que pour répondre à la non-homogénéité de la fonction utilisée. En effet, si cette dernière répartissait de façon homogène les enregistrements sur les pages, de tels mécanismes deviendraient superflus. Dès lors nous pouvons envisager deux démarches pour trouver une fonction de hachage, la première en utilisant une fonction classique[1] et en lui adjoignant un mécanisme efficace de résolution de collisions et la seconde en cherchant "directement" une fonction homogène[2].

Dans notre contexte où l'adresse à calculer n'est qu'un numéro de noeud, il est absolument nécessaire de choisir la seconde démarche. En effet, nous ne pouvons envisager de rechercher inutilement sur un noeud un objet partiel qui aurait été envoyé ailleurs faute de place. Notre utilisation des techniques de hachage se fait à un niveau trop élevé pour permettre d'éventuelles collisions qui, d'une part, surchargeraient les processeurs avec des recherches inutiles et qui, d'autre part, augmenteraient le trafic sur le réseau.

Après avoir vu le principe général des fonctions de hachage, nous allons maintenant présenter différentes façons de les classer et leur adaptation éventuelle à nos besoins.

3.4.2.2. Hachage statique et hachage dynamique

Les techniques de hachage dynamique se distinguent des autres par le fait qu'elles modifient dynamiquement l'espace d'adressage cible en changeant[3] la fonction utilisée. Ainsi les performances d'accès ne diminuent pas avec l'augmentation du nombre d'enregistrements placés puisqu'il n'y a plus de collisions. Dès qu'une page est pleine (respectivement sous-utilisée) l'espace d'adressage est agrandi (respectivement rapetissé) et les enregistrements qu'elle contenait sont "re-hachés".

Dans son article "Dynamic Hashing", Per-Ake Larson [LARS78] présente en détail le principe de cette technique, une implantation qui utilise un index sous forme d'arbre binaire et une analyse des résultats obtenus. Une technique de hachage dynamique multidimensionnelle, c'est-à-dire basée sur l'utilisation de plusieurs clés pour un enregistrement donné, est proposée dans [OTOO85]. K. Kawagoe donne dans son article "Modified dynamic hashing" [KAWA85] une classification des différentes solutions proposées agrémentée d'une importante bibliographie. Il propose aussi une optimisation qui garantit de retrouver un enregistrement en un seul accès sans avoir recours à une table ou à un index. Enfin, on pourra trouver dans [PEPIN85] une implantation du "hachage extensible" que l'on peut considérer comme une forme particulière de hachage dynamique. Ces quelques éléments bibliographiques sur le sujet ne constituent bien sûr en rien une liste exhaustive de tous les travaux qui s'y rapportent.

---

1. Ces fonctions classiques souvent utilisées sont présentées dans la partie qui concerne la classification hachage statique, hachage dynamique(cf 3.4.2.2).
2. Cette notion de fonction homogène se rapproche de la notion de hachage parfait que nous présentons dans la suite (cf 3.4.2.3).
3. La modification peut consister à appliquer plusieurs fonctions comme le propose P.A LARSON [LARS78].

L'extrapolation de ce type de hachage à notre contexte va nous amener à faire plusieurs remarques.

Tout d'abord, il est important de rappeler que nous ne travaillons pas au même niveau; il est en effet difficile de comparer la page mémoire usuellement considérée, avec un noeud constitué d'un processeur et d'un disque. Dans le premier cas, l'adressage par le contenu est complet, c'est-à-dire qu'il permet de localiser précisément l'enregistrement recherché. Dans le second cas, il ne fournit qu'un numéro de noeud et doit être relayé par un autre mécanisme pour trouver "dans" le noeud l'objet partiel voulu[1]. Le hachage dynamique dont l'utilisation est habituellement basée sur le taux de remplissage des pages mémoire, doit donc être adapté à nos besoins. Le critère simpliste de remplissage est à reconsidérer.

Comme nous l'avons souligné précédemment, ce n'est pas uniquement le nombre d'occurrences d'un fragment qui permet de déterminer un nombre optimal de noeuds pour leur répartition. En effet, la prise en compte du type et des propriétés des requêtes qui utilisent ces occurrences est souhaitable pour arriver à une bonne utilisation de la machine (cf 3.2 Importance du degré de répartition). En ce qui nous concerne, le critère de modification de l'espace d'adressage doit donc être lié à l'obtention d'un meilleur taux global de parallélisme, ie à l'obtention d'une meilleure utilisation de la machine. Cela implique l'évaluation de nombreux paramètres (nombres d'occurrences, type des requêtes, fréquence, etc). Nous sommes là bien loin de l'évaluation d'un simple taux de remplissage. Les modifications de l'espace d'adressage ne peuvent être que périodiques. En effet, les paramètres qui entrent en jeu varient dans le temps bien moins vite que ne peut le faire le taux de remplissage d'une page mémoire. De plus, les conséquences d'une augmentation ou d'une diminution du nombre de noeuds utilisés sont suffisamment importantes, de par le nombre d'objets partiels qui risquent de migrer, pour qu'on ne le fasse que s'il y a vraiment nécessité.

Le mot "dynamique" de l'expression "hachage dynamique" est donc, en ce qui nous concerne à relativiser. Néanmoins, puisque seuls diffèrent le critère et la fréquence de modification de l'espace d'adressage, toutes les techniques proposées dans la littérature sont a priori utilisables dans notre contexte.

Cependant une remarque d'ordre général sur ces techniques s'impose. Dans toutes celles que nous avons rencontrées, la première étape consiste à transformer la clé de répartition en un nombre sur lequel est ensuite appliquée la solution proposée.

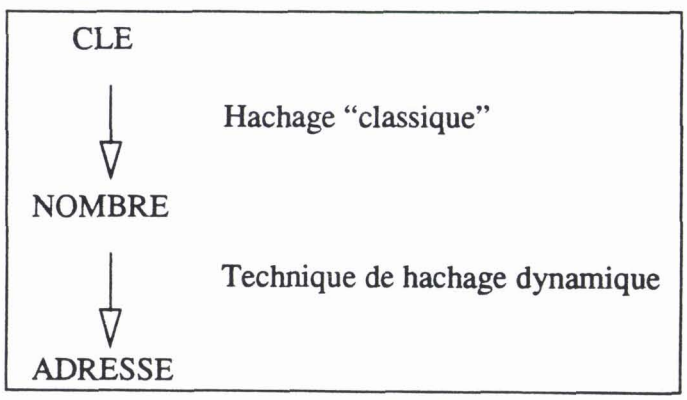


Fig. 3.80 : Principe général d'une technique de hachage dynamique

1. A ce niveau nous ne faisons aucune hypothèse quant à la façon dont seront adressés les objets partiels sur un noeud.

Cette transformation de la clé en nombre est toujours effectuée par une des fonctions que l'on peut qualifier de fonctions de hachage classiques. Nous présentons dans ce qui suit ces fonctions mais, d'ores et déjà, nous pouvons insister sur le fait qu'elles sont rarement homogènes. La non-homogénéité des nombres obtenus est d'autant plus accentuée que les valeurs de la clé sont non uniformément distribuées sur leur domaine, ce qui au passage, est souvent le cas dans les applications bases de données. Le biais introduit par cette première étape ne peut que se répercuter dans les étapes ultérieures[1], ce qui entraîne finalement beaucoup de collisions. Bien sûr, avec l'aspect dynamique de la fonction il n'y a pas de collisions, mais on risque fort de passer son temps à "rehacher" les mêmes objets partiels.

En fait, les techniques de hachage dynamique proposées répondent avant tout au problème de la gestion de la taille de l'espace d'adressage, ce qui, pour nous, constitue un point bien moins important que celui de l'homogénéité nécessaire de la répartition.

Nous proposons dans la section suivante une méthode de hachage que l'on peut qualifier de dynamique mais dont la principale préoccupation est l'homogénéité.

Avant de terminer sur la classification hachage dynamique-hachage statique, nous allons présenter les principaux types de fonctions dites statiques, c'est-à-dire avec un espace d'adressage connu et fixe. Ces dernières sont d'ailleurs souvent utilisées partiellement dans les solutions dynamiques. Pour cela nous nous appuierons sur l'article "Hashing functions" de G. D. Knott [KNOT72] qui constitue une excellente référence sur le sujet.

L'auteur propose trois types de fonctions: celles qui dépendent de la distribution des clés, celles qui éclatent les amas et celles qui sont indépendantes de cette distribution. Il est à noter que les fonctions de la seconde catégorie peuvent appartenir à l'une ou l'autre des deux autres.

. Fonctions qui dépendent de la distribution des clés.

Il faut dans ce cas connaître la fonction F_X de répartition (loi probabiliste -Normale, Poisson- ou autre) des valeurs de la clés. Il faut de plus que cette fonction soit facilement calculable. Dès lors la fonction $H(x) = \lceil nF_X(x) \rceil - 1$ semble être un bon choix pour un espace d'adressage égal à $\{0,1,...,n-1\}$.

Dans un contexte bases de données, une telle fonction n'est envisageable que si on collecte dynamiquement les informations nécessaires à la connaissance de la fonction de répartition F_X. La méthode que nous proposons dans la section suivante sera basée sur ce principe.

Enfin il faut noter qu'avec de telles fonctions la répartition des objets partiels peut facilement devenir homogène puisque l'on connaît tout de leur distribution initiale dans leur domaine.

. Fonctions qui éclatent les amas

1. Ces étapes du hachage dynamique reposent d'ailleurs souvent sur l'hypothèse "ambitieuse" qu'elles reçoivent en entrée des nombres uniformément répartis!

Celles-ci sont basées sur l'idée qu'une fonction de hachage doit éclater les amas de valeurs clés, c'est-à-dire "gommer" leur non uniformité initiale. Une telle fonction doit donc donner une répartition relativement uniforme étant donné certaines hypothèses générales à propos de la distribution des valeurs clés. Le mot "hypothèse" est ici de toute importance; la fonction repose uniquement sur des suppositions relatives à la distribution des valeurs clés.

Différentes hypothèses sont fréquemment utilisées pour ce type de fonction:

. les valeurs clés forment un échantillon aléatoire de toutes les valeurs possibles (=> fonctions de hachage balancées)

. les valeurs clés apparaissent suivant une progression arithmétique [TOYO66]

. les valeurs clés proches au sens des distances de Hamming sont équiprobables.

Fondés sur de telles hypothèses, des calculs souvent complexes permettent d'aboutir à des fonctions acceptables. Cependant nous voyons mal comment utiliser une telle technique dans notre contexte. En effet, la nature variable des clés de répartition et leur non uniformité quasi-certaine rendra impossible la détermination d'une quelconque hypothèse. Ce type de fonction semble avoir été étudié uniquement pour des cas bien précis, ce qui explique les calculs pointus qui en découlent. Il en résulte peu de liberté pour leur utilisation potentielle.

Malgré tout, il existe un type de clé que nous pouvons contrôler. Il s'agit de l'identifiant. La construction de ces valeurs internes pourrait en effet répondre à certaines des hypothèses précédentes, ce qui nous permettrait de réutiliser les travaux cités pour trouver de bonnes fonctions de répartition par rapport à ces identifiants.

Etant donné que le mécanisme que nous proposons dans ce qui suit (cf 3.4.3) offre une solution simple aussi bien pour les identifiants que pour les attributs valeur, nous ne développerons pas plus cette alternative.

. Fonctions indépendantes de la distribution des valeurs clés.

Nous abordons ici les fonctions les plus fréquemment citées. Elles sont donc applicables sans connaissance spécifique de la distribution initiale des valeurs clés et sont basées sur le calcul de nombres pseudo-aléatoires à partir de ces valeurs. Nous donnons ci-dessous une liste des différentes méthodes utilisables accompagnée des points importants à prendre en compte. Nous donnons en annexe un tableau comparatif de ces différentes méthodes appliquées au même ensemble de données. Pour toutes ces méthodes, la valeur clé x est assimilée à une chaîne de bits.

- Méthodes par extraction

Dans ce cas, k bits sont extraits de la clé x et concaténés dans un certain ordre ce qui fournit une adresse parmi $\{0,1,...;2^{k-1}\}$. Il va sans dire que le choix des bits extraits et de l'ordre de concaténation est de toute importance. La méthode que nous proposons dans la section suivante est aussi basée sur une extraction mais qui est cette fois "réfléchie" (cf 3.4.3).

- Méthodes par multiplication

Dans ce cas, k bits sont extraits de la valeur de x^2 ou de C*x. Ce genre de méthode est très sensible à la distribution des valeurs clés et au choix de la constante C [1]. A propos de cette constante, des résultats très (trop?) précis existent; cependant, ces derniers ne souffrent pas la moindre modification de la distribution des clés et digèrent très mal la non uniformité de ces dernières . Enfin, l'opération de multiplication risque, de par sa nature, d'introduire de la non uniformité même si les clés sont uniformément distribuées sur leur domaine.

- Méthodes par ou-exclusif

Ici, les k bits sont le résultats d'un ou exclusif entre différents segments de la valeur clé. Une fois de plus, le choix des segments est très important. L'avantage, cette fois, est qu'une équiprobabilité des 0 et 1 dans les clés est conservée dans le résultat. De plus, l'opération de ou exclusif ne peut introduire de non homogénéité à partir de clés homogènes.

- Méthodes par addition

Ici, les k bits sont le résultat de l'ajout de différents segments. Comme pour la multiplication, l'addition peut créer de la non homogénéité. De plus, si les segments ne sont pas indépendants, les résultats seront biaisés.

- Méthodes par division

L'opération modulo est dans ce cas utilisée.

Exemple: (premier caractère + dernier caractère + longueur chaîne) mod k

Des résultats épars ont été énoncés quant aux valeurs que doit ou ne doit pas prendre k [KNUT73]. Ils reposent souvent sur des hypothèses draconiennes qui risquent bien peu d'être vérifiées, dans notre cas, vu la diversité des clés que nous pouvons avoir.

Toutes ces méthodes, a priori intéressantes, puisqu'elles ne nécessitent pas de connaissance sur la distribution des valeurs, ne sont donc hélas pas miraculeuses d'un point de vue uniformité du résultat obtenu. Le lecteur pourra se reporter au tableau en annexe pour ce convaincre qu'elles risquent fort de ne pas être adaptées à nos besoins. De ce fait, les techniques usuelles de hachage dynamique qui les utilisent , risquent elles aussi de ne pas être adaptées.

Avant de terminer cette synthèse sur les fonctions de hachage, nous allons présenter la notion de hachage parfait, notion qui se rapproche de nos préoccupations.

3.4.2.3. Le hachage parfait

Il constitue une forme particulière de hachage statique, c'est-à-dire avec un espace d'adressage fixe. Dans son article "External perfect hashing" [LARS85], P.A. LARSON en

1. Des calculs précis et des remarques sur le codage interne des caractères conduisent FLOYD (souvent cité mais il n'existe aucune référence hormis une communication "personnelle" non publiée)à préconisé l'utilisation de la fonction $H(x)=\lceil n(158x/255 \bmod 1)\rceil$!

donne la définition suivante:

Soient n le nombre d'enregistrements à placer et m un nombre de pages d'une contenance de b enregistrements. Une fonction de hachage est dite parfaite si aucune page ne reçoit plus de b enregistrements. Si de plus $m=\lceil n/b \rceil$ on parle alors de fonction minimalement parfaite.

De telles fonctions évitent donc d'emblée les problèmes liés aux collisions, ce qui, dans notre contexte, est de toute importance. Cependant, il est important de noter que seules les fonctions minimalement parfaites assurent l'homogénéité du placement des enregistrements. Pour les autres, la seule certitude que l'on ait, c'est qu'il n'y aura jamais saturation d'une page, ce qui ne constitue pas une preuve d'homogénéité.

Passée la définition, le problème principal reste la détermination de telles fonctions. Deux types de méthodes se distinguent à cet effet: les méthodes systématiques et les méthodes par tâtonnements.

. Les méthodes systématiques.

Dans ce cas, des classes de fonctions paramétrées sont proposées; les meilleures valeurs pour les paramètres sont alors déterminées systématiquement ce qui conduit souvent à une forte complexité en temps. Sprugnoli [SPRU77] propose les deux classes de fonctions suivantes:

$h_1(x)=\lfloor (x+s) / N \rfloor$ et $h_2(x)=\lfloor ((xq + d) \bmod M) / N \rfloor$.

La détermination des paramètres est en $O(n^3)$ (n: nombre d'enregistrements). De plus, le taux de remplissage obtenu est, dans ce cas, assez faible; en d'autres termes, ces fonctions sont loin d'être minimalement parfaites. La seule solution que l'auteur propose, pour diminuer la complexité du problème, est de construire, dans un premier temps des sous-ensembles d'enregistrements à l'aide d'une fonction classique, puis de déterminer une des fonctions parfaites proposées pour chaque sous-ensemble!

Jaeschke propose [JAES81] des fonctions du type $h(x)=(\lfloor C/(Dx+L) \rfloor \bmod m)$ pour trouver des fonctions minimalement parfaites. Cependant, la complexité est exponentielle. Chang [CHAN84], quant à lui, soumet une approche qui préserve l'ordre des enregistrements[1] mais qui a le désavantage de manipuler des grands nombres.

. Les méthodes par tâtonnements

Nous retombons ici parmi les méthodes classiques présentées auparavant. Cichelli [CICH80] propose une fonction du type h(x)=g(premier caractère) + g(dernier caractère) + longueur(x), g étant une fonction de "transformation" du caractère en un entier. Fredman, Komlos et Szemeredi [FRED82] soumettent une fonction du type $h(x)=((kx \bmod p) \bmod s)$ avec p nombre premier et s la taille de l'espace d'adressage. Il va sans dire, qu'avec de telles fonctions,

1. Cette préservation de l'ordre des enregistrements peut être très intéressante pour certaines opérations propres aux bases de données telles que la jointure.

nous retrouvons les problèmes d'homogénéité déjà cités. En fait le diversité des clés que nous pouvons utiliser, risque fort de rendre inutilisable une telle approche par tâtonnements.

Enfin, il est intéressant de citer l'article de Berman "Collections of functions for perfect hashing" [BERM86] qui aborde mathématiquement le problème en comptant le nombre de fonctions qui doivent être recherchées pour être sur de trouver une fonction de hachage parfaite.

Dans notre contexte, le hachage parfait n'a d'intérêt que s'il nous fournit des fonctions minimalement parfaites. Or comme nous l'avons vu, l'approche systématique n'arrive pas à ce résultat malgré une complexité souvent importante. Par ailleurs, l'approche par tâtonnements risque fort de souffrir du manque d'uniformité de nos clés de répartition.

Faute d'avoir trouvé dans tous ces travaux un type de fonctions qui réponde vraiment à nos besoins, nous proposons dans ce qui suit une démarche qui permet de déterminer facilement de "bonnes" fonctions de hachage. Les fonctions produites sont minimalement parfaites et s'intègrent aisément dans un mécanisme que l'on peut qualifier de mécanisme de hachage dynamique.

3.4.3. Une solution particulière de hachage.

Relativement à la synthèse que nous venons d'effectuer, les fonctions que nous proposons peuvent être classées dans la catégorie des fonctions statiques. Néanmoins, la simplicité de leur construction nous permettra de les interchanger facilement pour aboutir à un mécanisme de hachage dynamique. Elles sont basées sur l'extraction de certains caractères dans la clé. Elles évitent la non-uniformité du résultat par la connaissance d'informations sur la distribution des caractères extraits. En conséquence, elles peuvent aussi être qualifiées de minimalement parfaites.

Le principe général de notre approche est le suivant: pour chaque attribut valeur susceptible de devenir une clé de répartition, ainsi que pour les identifiants, nous construirons, à partir des valeurs existantes, le coeur de la fonction de répartition relative à cette clé. Nous appellerons cette information centrale le ***générateur de la clé***. Un tel générateur renfermera des informations sur la distribution des caractères extraits, ce qui nous permettra d'obtenir de très bonnes propriétés pour la fonction. Avec l'évolution des données, il pourra être remis en cause et reconstruit afin de conserver ces propriétés. Nous reviendrons plus en détail sur cet aspect dynamique de l'approche.

3.4.3.1. Notion de générateur.

Avant de chercher ce qu'il faut mémoriser sur une clé donnée pour pouvoir obtenir une répartition homogène par rapport à ses valeurs, nous pouvons restreindre le problème en remarquant que, dans une même application bases de données, certains attributs se "ressemblent". Par là, nous entendons qu'ils sont définis sur un même domaine (au sens de CODD[1]) et qu'ils obéissent aux mêmes règles de construction, règles relatives aux caractères utilisés, à leur

1. Bien que nous ne nous plaçons pas dans le contexte relationnel[CODD70], la notion de domaine pour nos attributs valeur peut être définie de façon similaire.

fréquence ou à leur position. Nous donnons ci-dessous un petit exemple:

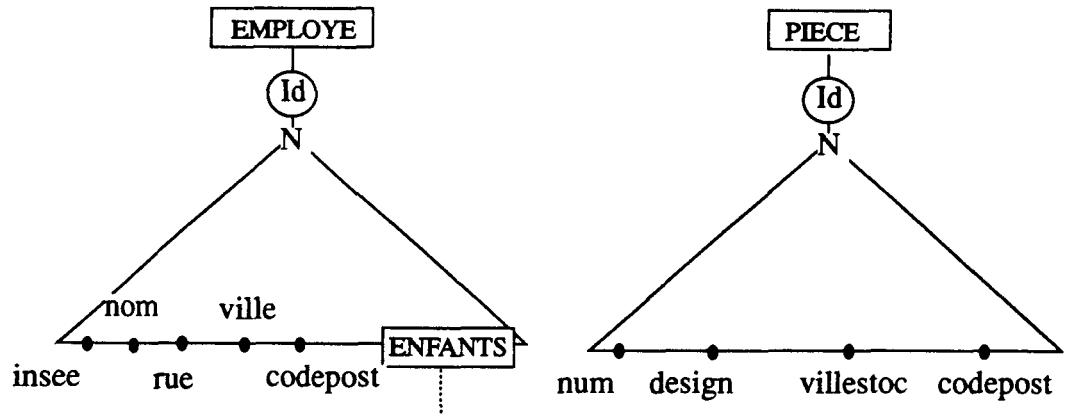


Fig. 3.81 : Exemple utilisé pour la notion de générateur

Les attributs valeurs EMPLOYE.nom et EMPLOYE.rue peuvent être regroupés si nous considèrons la façon dont leurs valeurs sont construites. En effet, ils utilisent le même jeu de caractères; de plus, la fréquence d'apparition et la disposition relative de ces caractères dans les valeurs est semblable. Il en est de même pour EMPLOYE.ville et PIECE.villestock ainsi que pour EMPLOYE.code_post et PIECE.code_post.

Nous nous ramenons donc au problème de savoir ce qu'il faut stocker sur une famille d'attributs (de clés) semblables. En conséquence, nous appellerons générateur d'une clé l'ensemble des informations utiles à un placement par rapport à cette clé ou par rapport à une clé de la même famille.

Notre approche consiste donc, dans une phase préliminaire, à construire les générateurs nécessaires, ces derniers étant utilisés ensuite pour tout placement ou accès à un objet partiel. Ces générateurs seront stockés dans la métabase.

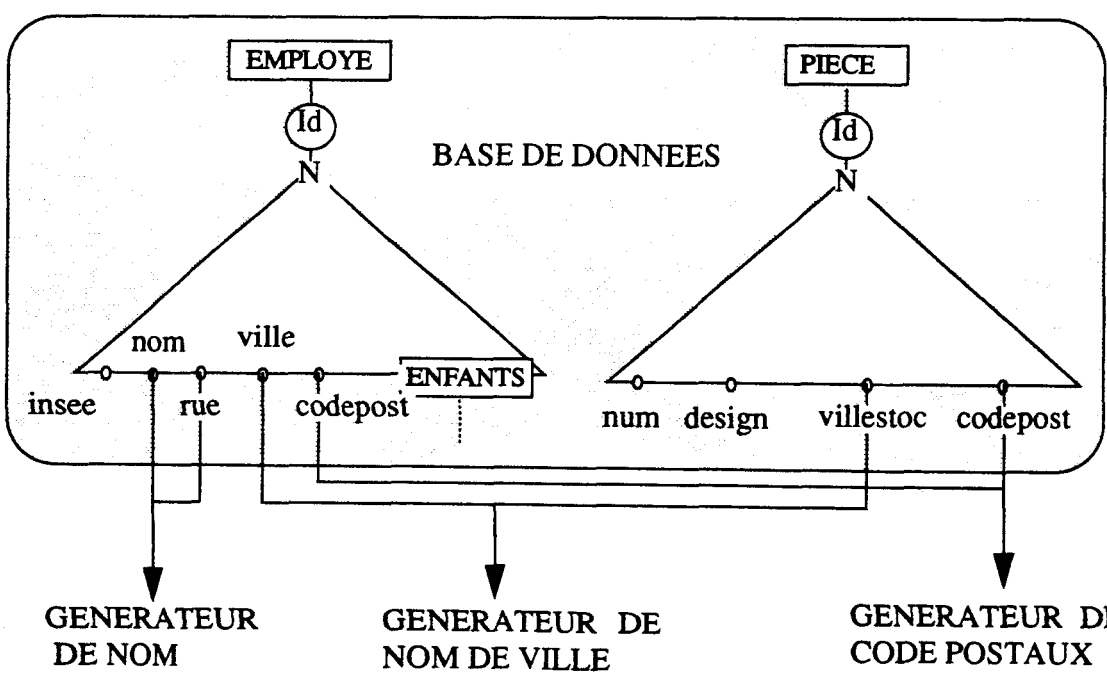


Fig. 3.82 : Principe des générateurs d'attributs

3.4.3.2. Construction des informations à ranger dans les générateurs

Comme nous l'avons souligné précédemment, si nous voulons obtenir à terme une répartition homogène, les informations contenues dans un générateur doivent tenir compte du "comment" de la clé, ou, en d'autres termes, de la distribution des valeurs de cette clé sur leur domaine. Néanmoins, nous ne pensons pas nécessaire de connaître tout de cette distribution. A partir d'une remarque simple, notre méthode propose à "moindre frais" une solution qui s'avère tout à fait efficace.

Avant de la présenter en détail, deux dernières contraintes sont à poser. Les informations des générateurs doivent être simples à construire et facilement utilisables. Nous ne devons pas en effet aboutir à une complexité prohibitive comme dans les approches systématiques du hachage parfait. Enfin, ces informations doivent être aisément adaptables en vue de leur utilisation dans un mécanisme de hachage dynamique.

La construction d'un générateur peut se décomposer en trois phases que sont l'extraction, le comptage et la répartition du tableau de comptage.

. L'extraction

Dans un premier temps, nous allons donc extraire de chaque valeur de clé existante un certain nombre de caractères. Le fait de pratiquer une extraction plutôt que d'utiliser l'ensemble de la valeur, peut, a priori, sembler insuffisant pour connaître la distribution des valeurs sur leur domaine. Cependant, comme nous l'avons fait remarquer à plusieurs reprises, cette distribution est rarement homogène, ce qui se répercute souvent sur les résultats des fonctions ensuite utilisées.

La remarque simple que nous pouvons faire est que nous avons tout intérêt à utiliser ce qu'il y a de plus homogène dans une clé si nous voulons obtenir de bons résultats. En fait, il est inutile d'alourdir le mécanisme en voulant utiliser tous les caractères d'une valeur clé et, de plus, il est indispensable d'évincer ceux qui introduisent de la non uniformité.

Exemples: Dans un code postal, les deux premiers caractères risquent de varier peu et d'introduire ainsi des amas dont on aura bien du mal à se débarrasser. Le dernier peut être considéré comme pire encore puisqu'il est pratiquement toujours égal à 0 ou 5. Par contre, le troisième et le quatrième balaient plus régulièrement l'ensemble des valeurs possibles. De même, les deux premiers caractères d'un nom de famille constituent un mauvais choix puisque certains couples prédominent (LE.., DE.., DU..,)

Dans ce qui suit, nous proposons l'extraction de deux caractères uniquement. La plupart du temps, ce sera suffisant. Néanmoins la méthode peut être étendue dans certains cas précis que nous analyserons ensuite (cf 3.4.3.4 Résultats obtenus). Les positions de ces caractères dans la valeur clé seront choisies de façon à ce qu'ils balaient le plus régulièrement et le plus fréquemment possible leur domaine[1].

Exemples:

-Troisième et quatrième caractère d'un code postal ou d'un nom de famille,

-Deux derniers caractères d'un numéro insee.

---

1. Nous entendons par domaine d'un caractère, l'ensemble des valeurs qu'il peut prendre.

. Comptage par rapport aux positions de caractères choisies.

Pour toute valeur clé, le couple de caractères extraits est transformé en un entier. Cette transformation peut, par exemple, être effectuée par une simple concaténation:

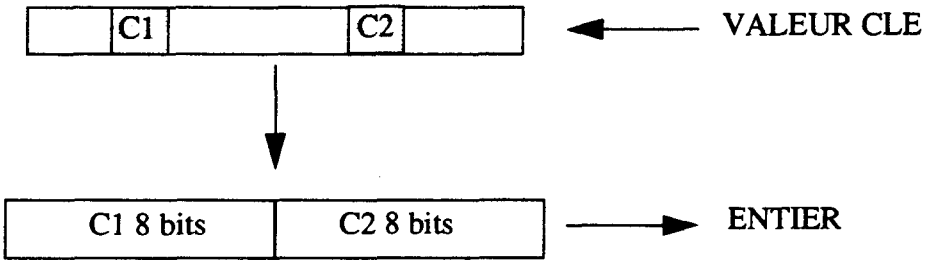


Fig. 3.83 : Exemple de transformation des caractères extraits en un entier

L'idée est de parcourir toutes les valeurs clé de la famille étudiée (un générateur pouvant se rapporter à plusieurs clés) et de compter le nombre "d'apparitions" de chaque entier. Lorsque les emplacements choisis dans la clé peuvent prendre N caractères différents comme valeur, l'opération de comptage fournit un tableau d'au plus N^2 éléments puisque nous utilisons ici une concaténation. Comme nous le verrons par la suite, ce nombre N^2 aura une grande importance sur l'efficacité de la méthode.

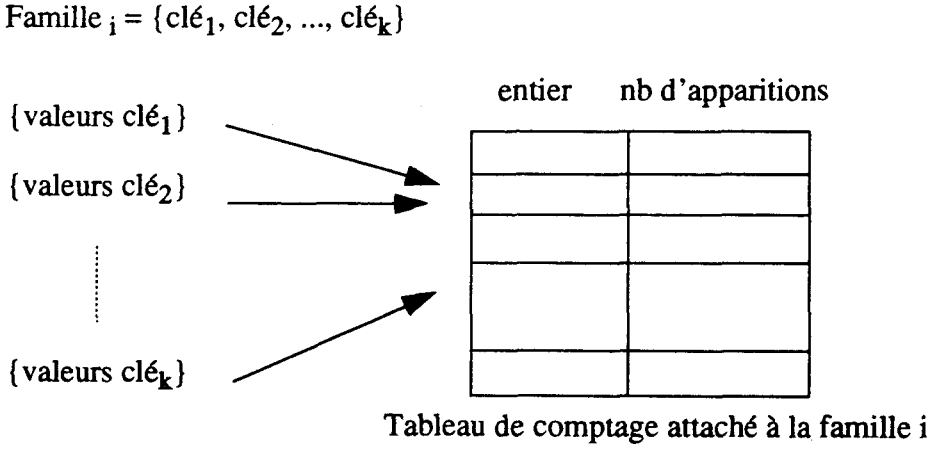


Fig. 3.84 : Principe de la phase de comptage

Cette analyse exhaustive des valeurs clé, relativement à deux caractères, va nous donner une information quantitative sur la distribution de ces valeurs mais uniquement par rapport à ces deux caractères. Le nombre d'apparitions de chacun des entiers sera d'autant plus équilibré que le choix des emplacements aura bien été fait. Néanmoins, certains déséquilibres pourront encore subsister à ce niveau, malgré le meilleur choix effectué. La dernière phase, celle de répartition du tableau de comptage, s'occupera d'effacer au mieux ces anomalies.

. Répartition du tableau de comptage

La phase finale de préparation d'un générateur consiste à répartir le tableau de comptage "sur" les noeuds[1] ou, en d'autres termes, à attribuer à chacun des N^2 entiers précédents un numéro de noeud. Nous supposons ici que ce nombre N^2 est largement supérieur au nombre de noeuds cibles. Lorsque ce n'est pas le cas, il y a toujours moyen de retourner à la phase précédente pour obtenir un plus grand tableau en utilisant plus de deux caractères.

Si nous considérons que, par cette opération, un noeud cumule le nombre d'apparitions des entiers dont il est l'image, nous devons bien sûr oeuvrer pour que ce nombre cumulé soit presque le même pour chaque noeud.

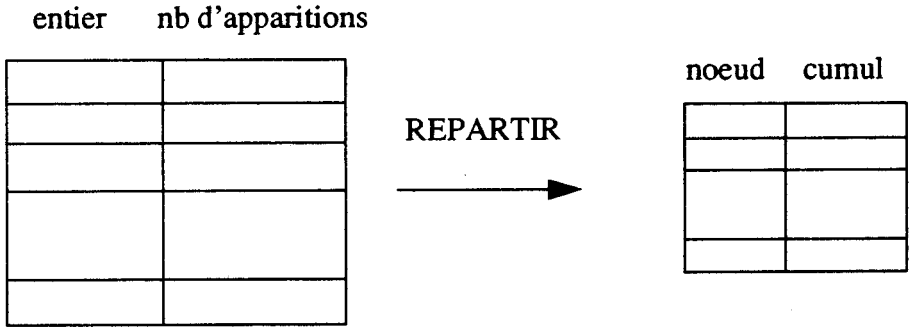


Fig. 3.85 : Principe de la phase de répartition du tableau de comptage

Nous sommes ici confrontés à un problème d'optimisation pour lequel il n'est pas envisageable d'étudier les quelques k^p possibilités (avec $p=N^2$) [2].

Nous proposons ci-dessous une solution simple mais qui, cependant, donne de très bons résultats (cf 3.4.3.4 Résultats obtenus):

TQ (il existe un entier à répartir) FAIRE
-chercher l'entier non encore traité qui a le nombre maximal d'apparitions
-attribuer à cet entier le noeud qui a le nombre cumulé le plus faible
FAIT

L'idée de cette solution est de "caser" d'abord les gros morceaux (entiers qui ont un grand nombre d'apparitions) pour affiner de plus en plus avec les petits. Si on considère que le tableau de comptage n'est pas trié, la complexité est en O(k*p), ce qui est plus que raisonnable[3] devant k^p.

En sortie de cette dernière étape, nous disposons donc de l'information à stocker dans le générateur, c'est-à-dire du tableau qui, à tout entier, associe un numéro de noeud.

---

1. A ce niveau le nombre de noeuds est supposé connu.
2. k=16, N=10 => $16^{100}=2^{400}$ possibilités!
3. $16*100 << 16^{100}$!

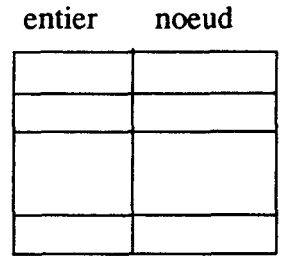


Fig. 3.86 : Information à stocker dans le générateur

Après avoir vu comment construire un générateur, il ne nous reste plus qu'à parler de son utilisation pour le hachage proprement dit.

3.4.3.3. Le hachage proprement dit.

Pour le placement ou la recherche d'un objet partiel donné, il suffit d'extraire les deux caractères utiles dans la valeur de sa clé de répartition, qui, au passage, peut tout à fait être l'identifiant, puis de calculer l'entier associé ce qui permet de trouver le numéro du noeud spécifié dans le tableau du générateur correspondant.

L'opération de hachage se ramène donc à la transformation d'un couple de caractères en un entier, suivie d'une indirection par une table.Cette table doit être présente sur chacun des noeuds de la machine afin d'éviter des consultations à distance qui surchargeraient le réseau. Cette condition n'est en elle-même pas un très gros problème étant donné la petite taille de cette table (N^2 éléments).

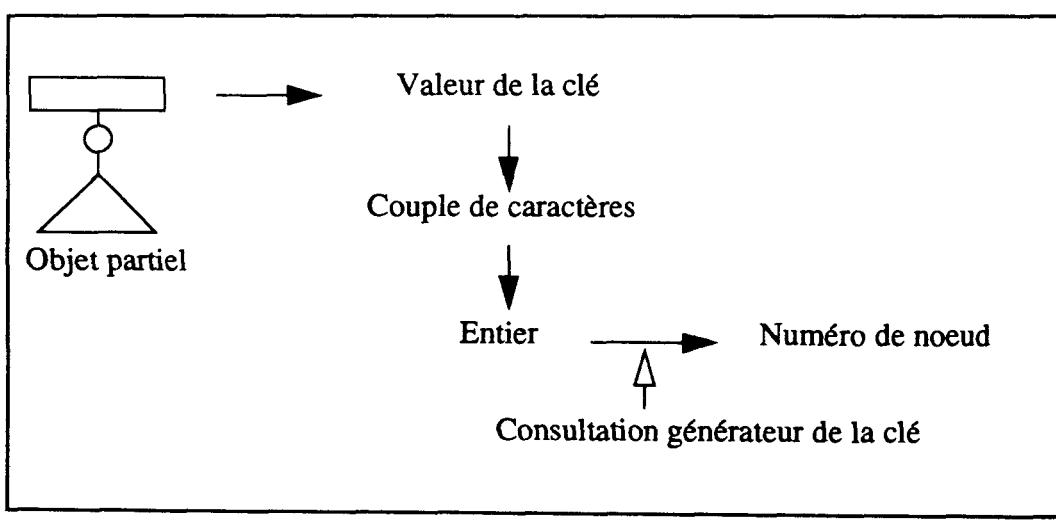


Fig. 3.87 : Le hachage proprement dit

3.4.3.4. Les résultats obtenus.

. Influence des emplacements d'extraction choisis

Nous présentons dans les pages suivantes deux tableaux qui résument les résultats obtenus pour la répartition d'une part des 11000 objets partiels d'un fragment d'une classe EMPLOYE et, d'autre part, des 1000 objets partiels d'un fragment d'une classe ETABLISSEMENT_SCOLAIRE[1]. Le nombre de noeuds cibles est pour l'exemple fixé à 16.

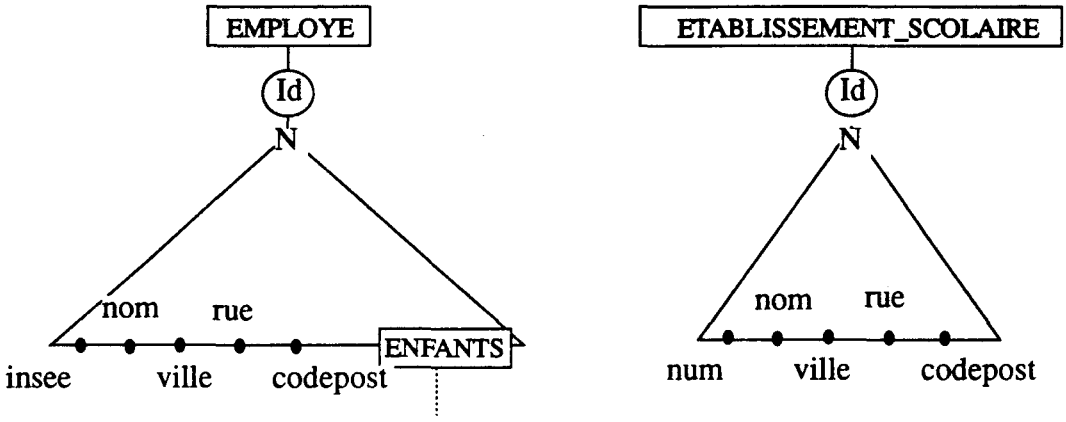


Fig. 3.88 : Description des objets répartis

Dans le cas1 des tests résumés par la figure 138, nous obtenons une très mauvaise répartition de par le mauvais choix des deux positions de caractères utilisées qui implique un nombre très faible d'entiers utilisés (13 sur 100). Le cas2, par contre, fournit une répartition tout à fait acceptable. En ce qui concerne le cas3, la répartition est homogène pour 15 noeuds sur 16, le dernier ayant reçu les apparitions de l'entier qui correspond au couple "DE", couple qui prédomine comme début de nom de famille. Enfin, dans le cas4, en évitant les emplacements 1 et 2 du nom, nous obtenons une répartition parfaitement équitable.

Le même genre de remarques peut s'appliquer au second exemple qui se réfère à un fragment de la classe ETABLISSENT_SCOLAIRE (cf figure 3.90).

1. Il est à noter que ces données sont issues d'une application réelle implantée avec le modèle relationnel. Ce changement de modèle n'influe cependant en rien sur l'exemple.

	CAS1	CAS2	CAS3	CAS4
Clé de répartition	num_insee	num_insee	nom	nom
Positions caractères	1 et 2	12 et 13	1 et 2	3 et 4
Nb théorique d'entiers	100	100	841	841
Nb d'entiers utilisés	13	100	138	441
Nombre d'objets				
noeud 0	2342	694	1461	688
noeud 1	2322	697	671	688
noeud 2	2137	696	668	688
noeud 3	1910	698	631	688
noeud 4	741	674	631	688
noeud 5	584	661	631	688
noeud 6	483	659	631	688
noeud 7	377	660	631	688
noeud 8	44	698	631	687
noeud 9	24	697	631	687
noeud 10	13	696	631	687
noeud 11	10	696	631	687
noeud 12	7	692	631	687
noeud 13	2	696	630	687
noeud 14	2	696	630	687
noeud 15	2	690	630	687
Ecart moyen				
en nombre	414	12	97	0.5
en pourcentage	60%	0.6%	14%	0.07%
Ecart maximal par rapport à la moyenne	1654	28	773	1

Fig. 3.89 : Résultats obtenus pour la répartition de 11000 objets EMPLOYE

	CAS1	CAS2	CAS3
Clé de répartition	code_postal	code_postal	code_postal
Positions caractères	1 et 2	4 et 5	3 et 4
Nb théorique d'entiers	100	100	100
Nb d'entiers utilisés	3	82	95
Nombre d'objets			
noeud 0	677	165	63
noeud 1	322	62	63
noeud 2	1	61	63
noeud 3	0	61	63
noeud 4	0	58	63
noeud 5	0	56	63
noeud 6	0	54	63
noeud 7	0	54	62
noeud 8	0	54	62
noeud 9	0	54	62
noeud 10	0	54	62
noeud 11	0	54	62
noeud 12	0	54	62
noeud 13	0	53	62
noeud 14	0	53	62
noeud 15	0	53	62
Ecart moyen			
en nombre	113	13	0.5
en pourcentage	179%	20%	0.79%
Ecart maximal par rapport à la moyenne	614	102	1

Fig. 3.90 : Résultats obtenus pour la répartition de 1000 objets ETAB.

Ces résultats mettent en évidence l'importance primordiale du choix des emplacements d'extraction. Pour un administrateur familier avec les applications qu'il gère, il devrait être aisé de déterminer le meilleur choix. De plus, cet administrateur peut prévoir ce choix lors de la création d'attributs valeur susceptibles de devenir une clé de répartition. Enfin, en cas d'incertitude, il y a toujours moyen de trouver systématiquement ces meilleurs emplacements

par une analyse exhaustive des valeurs clé existantes.

. Influence de la cardinalité de l'ensemble des valeurs clé.

Quelques mots pour souligner que l'homogénéité du résultat sera d'autant meilleure que la cardinalité de l'ensemble des valeurs clé sera élevée. De toute évidence, avec une forte cardinalité, le nombre d'entiers utilisé sera plus élevé, ce qui ne pourra conduire qu'à une meilleure répartition.

En conséquence, une fonction de hachage par rapport à une clé risque de s'améliorer à chaque reconstruction du générateur associé. Néanmoins, il subsistera toujours des attributs valeur gênants pour lesquels il sera impossible d'obtenir un résultat homogène. Nous voulons parler de ceux qui prennent un faible nombre de valeurs et, de plus, de façon non équilibrée.

Nous donnons ci-dessous l'exemple de l'attribut valeur "fonction" d'un fragment de la classe EMPLOYE. Le nombre de valeurs différentes est environ 20. De plus, ces valeurs ne sont pas équiprobables. En fait, le choix d'un tel attribut comme clé de répartition est très discutable. Les chiffres donnés ci-dessous sont obtenus en extrayant les caractères des meilleurs emplacements.

Noeud	0	1	2	3	4	5	6	7	8	9	10	11	...
Nb d'objets	2649	2479	2100	901	637	373	301	252	221	186	151	149	...

Fig. 3.91 : Résultats à partir d'un attribut de faible cardinalité

3.4.3.5. Aspects dynamiques de la méthode proposée

La construction d'un générateur se décompose, comme nous l'avons indiqué, en trois phases qui sont l'extraction, le comptage et la répartition du tableau de comptage.

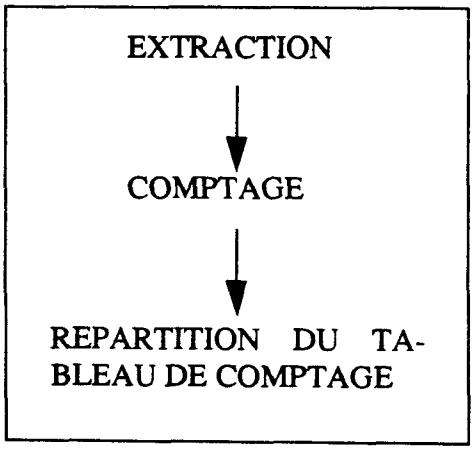


Fig. 3.92 : Construction d'un générateur

Par rapport au temps, deux types d'opérations peuvent intervenir: la reconstruction complète du générateur ou son adaptation à un nouveau nombre de noeuds.

. Reconstruction d'un générateur

Cette opération interviendra uniquement au delà d'un certain nombre de modifications des données qui avaient servies à la construction précédente. Un outil "d'espionnage" des modifications (ajouts, suppressions, mises à jour) des données sera donc nécessaire pour détecter le dépassement du seuil. Nous pouvons tout de même remarquer qu'avec un bon choix des emplacements d'extraction, nous devrions vite arriver à une stabilisation du générateur. En d'autres termes, il y a fort à penser qu'une augmentation de ce nombre n'ajoute plus, à partir d'un certain moment, d'informations sur la distribution des caractères extraits.

Par exemple, l'embauche d'une centaine d'employés risque bien peu d'influencer le générateur de noms construit précédemment à partir de 11000 noms connus.

Enfin, pour les attributs valeurs courants et pour les identifiants dont nous pouvons contrôler la création, il serait intéressant de disposer de générateurs "prêts à l'emploi" dans ce qui pourrait être vu comme une bibliothèque particulière.

. Adaptation d'un générateur

Il s'agit, cette fois, d'adapter un générateur à un nouveau nombre de noeuds cibles. Cette décision peut être prise suite à une modification prévue de la configuration matérielle ou suite à la détection du besoin d'adapter ce nombre afin d'obtenir une meilleure utilisation de la machine(cf 3.2 Importance du degré de répartition). Nous entrons donc là dans le domaine de l'utilisation des fonctions proposées dans un mécanisme de hachage dynamique.

Pour une telle adaptation, seule la phase de répartition du tableau de comptage doit être reprise. La simplicité de ce traitement permet d'envisager la préparation à l'avance d'un certain nombre de fonctions pour des nombres de noeuds différents.

Qu'il s'agisse de la reconstruction ou de l'adaptation d'un générateur, l'ensemble des objets partiels des fragments concernés, sera affecté. Nous pourrions envisager des mécanismes d'adaptation ou de reconstruction partielle des générateurs, en empruntant les idées retenues pour le hachage extensible [PEPIN85]. De tels mécanismes n'affecteraient pas la totalité des objets partiels mais rendraient par contre le hachage proprement dit beaucoup moins trivial. Enfin nous rappelons que l'aspect dynamique est pour nous à considérer de façon assez élevée. Il n'est pas question en effet de réorganiser fréquemment les données. En conséquence, notre solution semble suffisante.

3.4.3.6. Conclusions sur la méthode proposée

Deux mots peuvent principalement caractériser la méthode que nous venons de proposer: ***homogénéité*** et ***adaptabilité***. Homogénéité car elle permet d'obtenir une répartition très

équilibrée des données, ce qui rappelons-le, ira dans le sens d'une bonne répartition de charge. Adaptabilité car elle répond aisément au problème de la diversité des clés de répartition, mais aussi, parce qu'elle permet facilement d'obtenir un mécanisme de hachage dynamique.

Il est à noter qu'elle peut tout à fait être utilisée dans d'autres contextes. Nous proposons dans [NICO89b], son utilisation en vue d'opérer, de façon parallèle et à l'aide de tableaux de bits, l'opération de jointure du modèle relationnel. Le domaine des bases de données n'est bien sûr pas le seul où notre approche pourrait être suivie. Bon nombre d'autres applications parallèles, où la répartition de charge est un point crucial, pourrait en tirer profit.

3.5. Synthèse sur la fragmentation horizontale.

3.5.1. Discussion sur les points traités.

Dans ce chapitre, nous avons principalement abordé les deux points que sont la répartition relative des fragments d'une même classe et l'obtention d'une répartition homogène.

En ce qui concerne la répartition relative des fragments d'une même classe, nous avons proposé une méthode qui, par l'utilisation d'une notion de distances inter-fragments, permet de gérer, si besoin est, la proximité physique entre les différents objets partiels d'un même objet. Il est important de signaler que cette méthode prend en compte la configuration matérielle disponible. Ainsi, cette répartition relative tient compte à la fois de l'utilisation logique des données et du type de machine utilisé. Rappelons enfin que, grâce à une telle gestion de la proximité physique entre objets partiels, nous limitons les inconvénients attachés à la fragmentation verticale puisque nous répondons, lorsque c'est nécessaire, au problème de la reconstruction de tout ou partie d'un objet.

Pour ce qui est de l'obtention d'une répartition homogène, nous avons mis en avant une famille de fonctions de hachage qui répond tout à fait au problème. Les fonctions obtenues sont simples à construire et s'adaptent tout à fait à la diversité des clés de répartition que l'on rencontre dans les applications bases de données. De plus, elles peuvent facilement être interchangées, ce qui permet d'envisager leur intégration dans un mécanisme de hachage dynamique. Enfin, les résultats obtenus sont d'une homogénéitè plus que satisfaisante. Rappelons à ce titre que, dans le contexte considéré, une répartition homogène est pratiquement synonyme d'une répartition de charge équilibrée et donc d'un parallélisme intra-requête "idéal".

Nous terminerons cette partie en faisant allusion au rôle que peut jouer la fragmentation horizontale pour la récupération en cas de panne. Afin de ne pas bloquer l'ensemble de la machine en cas de défaillance de l'un des noeuds, il est nécessaire de prévoir, sous une forme ou sous une autre, une duplication des fragments horizontaux[1] .

---

1. Rappelons qu'un fragment horizontal correspond à un sous-ensemble d'objets partiels d'un fragment vertical, alloués à un même noeud. Par la fragmentation horizontale, un fragment vertical est donc éclaté en différents fragments horizontaux.

En ce qui nous concerne, il suffit juste de prévoir une "translation physique" de la répartition relative précédemment proposée. Notons à ce sujet, la solution originale retenue par la machine BUBBA [COPE88], où, pour minimiser les redondances, seules les informations non "retrouvables" au travers des structures d'accès secondaires (index) sont effectivement dupliquées. Néanmoins, cette solution suppose le placement, sur des noeuds différents, d'une donnée et des "informations indexées" qui s'y rapportent.

Nous allons maintenant discuter de quelques extensions applicables dans le cadre de la fragmentation horizontale.

3.5.2. Les extensions envisageables.

Nous aborderons ici les deux extensions que sont la ***répartition relative des fragments de classes différentes non liées*** et l'utilisation du ***stockage normalisé relatif***.

Pour chacune d'elles, nous donnerons le principe et les outils nécessaires à son intégration.

3.5.2.1. Répartition relative de fragments de classes différentes non liées

Comme nous l'avons fait pour les fragments libres d'une même classe, nous pouvons envisager la répartition relative de fragments de classes différentes et non liées dans le schéma logique. Rappelons que, par abus de langage, nous parlons de répartition relative de fragments mais qu'en fait, il s'agit de répartition relative des occurrences (objets partiels) de ces fragments.

Lorsque nous nous intéressions à une même classe, il s'agissait de la répartition relative des objets partiels d'un même objet. Si nous considérons maintenant des fragments de classes différentes, il s'agit de répartition relative d'objets partiels qui ne sont pas directement liés au niveau conceptuel.

Une telle répartition n'est donc justifiée que s'il existe une liaison entre ces objets partiels issus de classes différentes, cette liaison étant matérialisée par leur utilisation simultanée. L'exemple qui vient immédiatement à l'esprit est celui d'une jointure explicite entre deux classes différentes d'objets. Dans ce cas en effet, il existe, au travers de l'opération de jointure, une liaison entre objets partiels de provenances différentes. Nous pouvons alors envisager de répartir relativement de tels objets partiels.

Prenons l'exemple de deux classes d'objets EMPLOYE et SERVICE, non liées dans le schéma logique, et dont les attributs respectifs *num_emp* et *num_emp_responsable* font figure d'attribut de jointure (cf figure 3.93).

Pour des opérations de jointure explicite, certains fragments verticaux de ces classes peuvent être utilisés simultanément. Dés lors, il peut être intéressant de répartir relativement les objets partiels correspondants.

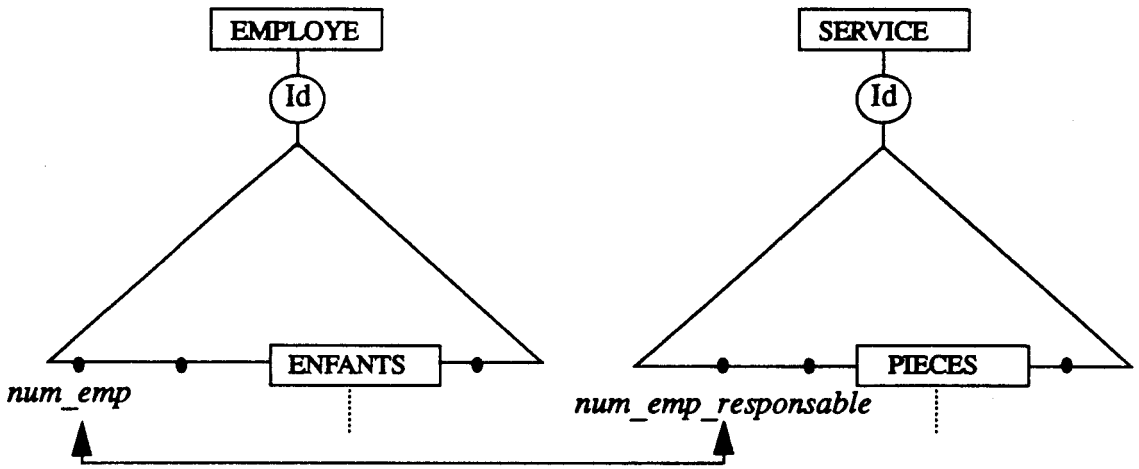


Fig. 3.93 : Exemple d'utilisation d'attributs de jointure.

Nous donnons ci-dessous une représentation, au niveau des objets partiels, d'une telle utilisation simultanée.

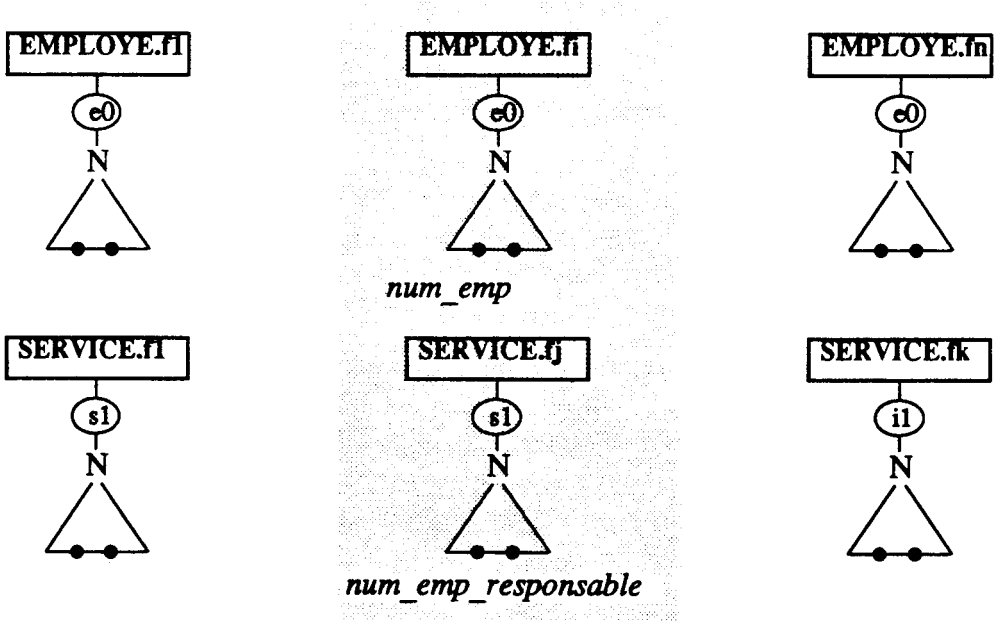


Fig. 3.94 : Utilisation simultanée d'objets partiels issus d'objets différents.

Cette répartition relative d'objets partiels provenant de différentes classes peut être vue comme un prolongement de la ***notion de regroupement*** (***clustering***), proposée dans cer-

tains SGBD Relationnels tels que ORACLE et SYSTEM/R, et qui consiste à regrouper physiquement certains nuplets de différentes relations suivant un prédicat donné. En effet, elle peut consister tout simplement à placer sur le même noeud les objets partiels concernés (e0 de EMPLOYE.fi et s1 de SERVICE.fj de notre exemple).

Notons néanmoins, qu'une telle répartition relative nécessite la définition d'une notion d'éloignement inter-fragments_de_classes_différentes. A ce sujet, la démarche générale utilisée pour les fragments de même classe est tout à fait ré-utilisable.

Nous disposons, pour chacun des fragments, d'un bloc d'affinité qui correspond aux coefficients d'affinité entre ses attributs. Il suffit donc de calculer le bloc manquant, c'est-à-dire celui qui correspond aux coefficients d'affinité entre attributs de fragments différents. Ce calcul doit, bien sûr, prendre en compte uniquement les accès simultanés issus de l'opération de jointure considérée.

Dés lors, à l'aide de la sous-matrice d'affinité obtenue, un éloignement entre les deux fragments peut être calculé.

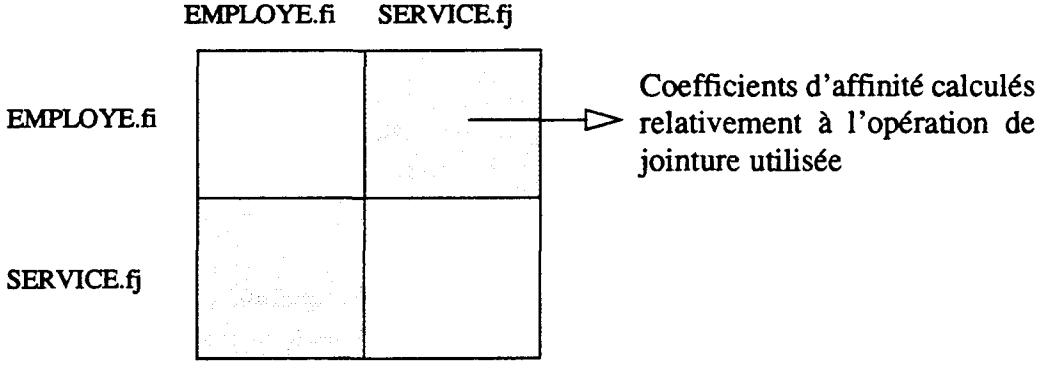


Fig. 3.95 : Sous-matrice d'affinité relative aux fragments concernés.

Cependant, la transformation des éloignements ainsi obtenues doit prendre en compte la totalité des éloignements calculés, à savoir les éloignements entre fragments d'une même classe et les éloignements entre fragments de classes différentes. Nous obtenons alors, une matrice d'éloignement inter-fragments qui regroupe les deux classes concernées, et sur laquelle, la suite du processus (transformation en distances relatives logiques, détermination de la répartition partiellement relative et transformation en positions relatives physiques) peut être adapté.

Nous terminerons cette présentation de la première extension envisageable en émettant quelques réserves. L'utilisation d'une telle répartition relative est, bien entendu, soumise au type de lien qui existe entre les classes considérées. Sur notre exemple, il s'agissait d'un lien 1:1; un employé ne pouvant être responsable que d'un service, et un service ne pouvant avoir qu'un responsable. Dans le cas de lien 1:n ou même n:m, le problème de la répartition relative de plusieurs objets partiels de chaque fragment se poserait. En effet, nous n'aurions plus, dans ce cas, une correspondance 1:1 entre les objets partiels concernés. Une telle situation imposerait, alors, l'utilisation de l'attribut de jointure comme clé de répartition afin de pouvoir appliquer la répartition relative voulue.

3.5.2.2. Sur l'utilisation du stockage normalisé relatif.

La seconde extension que nous pouvons envisager est l'utilisation d'un mode de stockage normalisé relatif.

Lors de la phase 2 de fragmentation verticale, une décision relative au mode de stockage est prise pour tout couple de fragments de la forme (SUP.fi, INF.fj), c'est-à-dire pour tout couple dont les fragments qui sont issus de classes SUP et INF liées, dans le schéma logique, par un lien composé(s)-composant(s). Cette décision peut être de deux types, à savoir, préconisation du stockage direct ou du stockage normalisé.

Dans le cas du stockage direct, il n'y a rien à signaler lors de l'étape de fragmentation horizontale, puisque les objets partiels composants sont stockés avec leur(s) objet(s) partiel(s) composé(s).

Par contre, pour le stockage normalisé, nous pouvons envisager de maintenir, si besoin est, une certaine proximité physique entre un objet partiel composant (de INF.fj) et son (ou ses) objet(s) partiel(s) composé(s) (de SUP.fi). Cela nous amène à considérer un nouveau mode de stockage dont les deux caractéristiques sont l'aspect normalisé et l'aspect relatif, et que nous désignerons sous le terme: ***stockage normalisé relatif.***

Notons qu'une telle extension n'est envisageable que lorsque le lien qui existe entre SUP et INF est de type mono-objet, c'est- à-dire lorsqu'un objet-nuplet de INF est attribut d'un objet-nuplet de SUP (cf 2-3-3). Dans l'autre cas, lorsque le lien entre SUP et INF est de type multi-objets (cf 2-3-3), la répartition normalisée relative d'un objet partiel composé et de ses objets partiels composants poserait trop de contraintes au niveau du choix des critères de répartition.

Même dans le cas d'un lien mono-objet entre SUP et INF, les problèmes à résoudre ne sont pas simples, puisqu'il faut évaluer un ***éloignement vertical***[1] pour chaque couple (SUP. fi, INF. fj) pour lequel le stockage normalisé a été préconisé.

La difficulté de l'évaluation de ces éloignements verticaux réside, avant tout, dans l'obtention d'une fonction d'éloignement vertical compatible avec celle utilisée pour les éloignements horizontaux (ie éloignements entre fragments d'une même classe). Par fonction compatible, nous entendons : fonction qui donne des éloignements verticaux comparables aux éloignements horizontaux.

En effet, si nous voulons réaliser un stockage normalisé relatif, nous devons appliquer le processus de répartition relative (cf figure 3.67) à l'ensemble des fragments libres de SUP et de INF. Ainsi, lors de la transformation des éloignements en distances, les éloignements verticaux et horizontaux seront simultanément pris en compte.

Là encore, passé ce problème de détermination d'une fonction d'éloignement vertical, l'approche utilisée pour les fragments d'une même classe (cf 3.3), reste valable pour déterminer le placement effectif des objets partiels.

---

1. Nous parlons ici d'éloignement vertical puisqu'il s'agit de quantifier le lien entre deux fragments liés par un lien composé(s)-composant(s), lien que, de façon imagée, nous pouvons qualifier de vertical. Ces éloignements verticaux n'ont donc rien à voir avec les éloignements entre fragments d'une même classe (cf 3-3-2) qui "ressemblent" plus à des éloignements horizontaux.

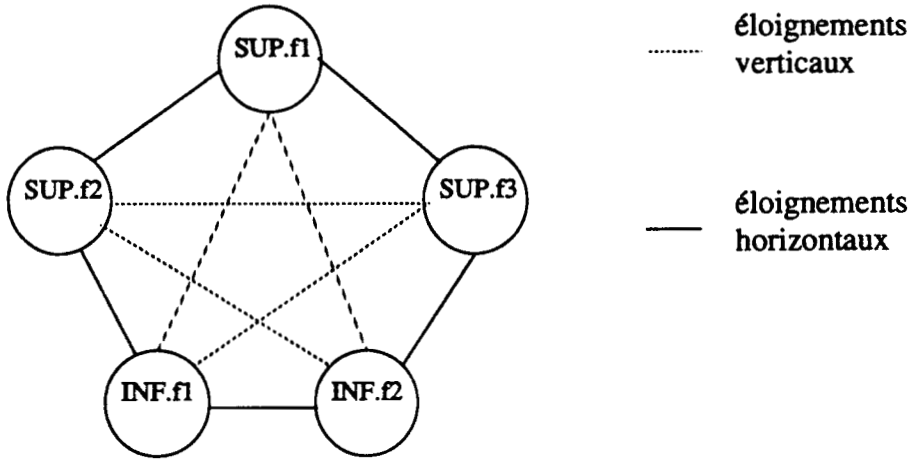


Fig. 3.96 : Exemple de graphe des éloignements inter-fragments pour la mise en oeuvre du stockage normalisé relatif

Nous terminerons cette présentation du stockage normalisé relatif en faisant remarquer, qu'un tel mode de stockage sera plus délicat à mettre en oeuvre, lorsqu'il existera des partages d'objets entre SUP et INF (partage d'objets composants par des objets composés). Dans ce cas, en effet, nous retomberons sur un lien de type multi-objets entre SUP et INF, ce qui imposera certaines contraintes pour le choix des critères de répartition. De telles contraintes risquent fort alors, de rigidifier la répartition obtenue.

3.5.3. Conclusions sur l'aspect dynamique attaché à notre méthode

Pour terminer ce chapitre sur la fragmentation horizontale, nous allons revenir quelque peu sur l'aspect dynamique attaché à la méthode de transformation de schéma logique, que nous préconisons. Les remarques qui suivent ne sont pas uniquement relatives à l'étape de fragmentation horizontale. Elles concernent plutôt l'ensemble du processus de transformation.

La principale caractéristique de notre méthode est donc l'évaluation de paramètres relatifs à l'utilisation logique des données, mais aussi relatifs à la configuration matérielle utilisée. En d'autres termes, la fragmentation et la répartition des données s'orientent à chaque niveau, vers une adaptation fine en fonction des conditions.

Nous pouvons résumer cela par le tableau suivant:

Points liés à l'utilisation logique des données	*Points liés à la configuration matérielle utilisée*
Calcul des coefficients d'affinité	
Production récursive des fragments	Poids relatifs aux accès inutiles et aux accès supplémentaires
Prolongement de la fragmentation verticale <=> choix entre stockage direct et normalisé	Poids et seuils pour les coefficients de liaison
Détermination des degrés de répartition	Détermination des degrés de répartition
Evaluation d'un éloignement inter-fragments	
Détermination de la répartition partiellement relative	
Obtention des positions relatives physiques	Obtention des positions relatives physiques
Choix du mode de répartition	
Construction des générateurs	Construction des générateurs

Fig. 3.97 : Les différents niveaux d'évaluation de notre méthode

Remarque: Notons que pour la construction des générateurs, il s'agit d'utiliser des informations relatives à la nature des données plutôt que sur leur utilisation.

Cette démarche permet de distinguer deux causes possibles de réorganisation des données, à savoir une cause logique liée à l'utilisation des données, et une cause physique liée à la configuration matérielle utilisée.

Dans le premier cas, la réorganisation interviendra suite à des modifications relatives à l'utilisation logique des données, ou, en d'autres termes, suite à des modifications des principales requêtes soumises au système.

Une telle réorganisation pourra être assez profonde, puisque la nature des fragments et le mode de stockage qui leur est associé risquent d'être modifiés. De ce fait, les distances lo-

giques entre fragments seront modifiées, ce qui entraînera un nouveau calcul des distances physiques relatives.

Par contre, de telles modifications n'altéreront pas les générateurs d'attributs. Ainsi, les fonctions de hachage utilisées pour la répartition resteront valides.

Dans le second cas, la réorganisation des données sera liée à une modification de la configuration matérielle utilisée. Si cette modification ne concerne qu'un changement de l'organisation ou du nombre des noeuds d'une même machine, la réorganisation des données sera alors "bénigne" puisque, seuls l'obtention des positions relatives physiques et les fonctions de répartition seront affectées. La nature des fragments et le mode de stockage associé seront dans ce cas invariant.

Enfin, le passage d'une machine à l'autre, entraînera, quant à lui, une réorganisation "profonde" des données, puisque, les différents poids et seuils utilisés risquent alors de changer, entraînant avec eux, une modification de la nature des fragments et des modes de stockage associés.

Quelque soit la cause de la réorganisation des données, notre travail s'oriente vers l'utilisation de ***machines bases de données parallèles***, qui reposent sur l'approche ***sans-partage*** et, sur lesquelles, le ***placement des données est dirigé par leur utilisation et par la configuration matérielle disponible***.

4. CONCLUSIONS

Ce dernier chapitre de conclusion se décompose en deux parties. Tout d'abord, nous y aborderons le problème de la prise en compte de nos solutions, au sein d'une machine sans-partage. Pour cela, l'exemple de l'intégration de ces solutions dans les différentes phases de compilation du langage FAD, sera utilisé.

Nous traiterons ensuite, dans une seconde partie, de l'évolution de ces solutions en vue de leur utilisation effective. Nous discuterons à ce titre, de l'évaluation des paramètres liés à une machine donnée.

Remarque: Nous ne reviendrons pas, dans ce chapitre, sur les conclusions particulières attachées à chacune des étapes de fragmentation, puisque ces dernières ont fait l'objet d'une section de synthèse dans chaque chapitre (cf 2.4 et 3.5).

4.1. Exemple d'intégration de nos solutions dans les phases de compilation du langage FAD

4.1.1. Les différentes phases de compilation du langage FAD

Nous allons donc nous baser ici sur l'implantation du langage FAD pour la machine BUBBA, pour analyser les conséquences des solutions que nous préconisons (DANF88b).

4.1.1.1. Généralités

Les différentes phases de compilation du langage peuvent se représenter par le diagramme ci-dessous :

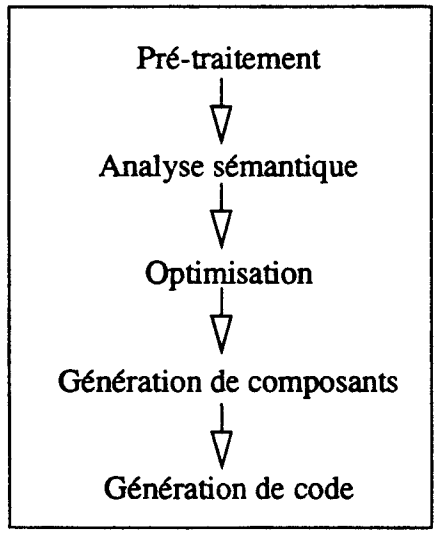


Fig. 4.1 : Les différentes phases de compilation du langage FAD

La phase de ***pré-traitement*** s'occupe de la vérification syntaxique et des problèmes relatifs à l'importation de modules dans les programmes.

La phase d'***analyse sémantique*** opère la vérification et l'inférence des types dans les programmes issus du pré-traitement.

La phase d'***optimisation*** convertit les traitements FILTER n-aires, ainsi que les autres opérations ensemblistes du langage, en séquences optimales d'opérations plus simples.

La phase de ***génération de composants*** produit "un programme physique" en PFAD (Parallel FAD), programme qui tient compte du schéma de répartition des données sur BUBBA.

Enfin, la phase de ***génération de code*** produit un module de chargement qui en plus du code, contient une table d'informations sur les arcs dataflow. Cette table est nécessaire à BUBBA pour le support des opérations *send* et *receive* de PFAD.

Nous allons, dans ce qui suit, détailler les phases d'optimisation et de génération de composants, puisque, c'est à ce niveau que nos solutions vont perturber le "processus" de compilation.

4.1.1.2. La phase d'optimisation

Nous donnons, dans ce qui suit, les grandes lignes de la phase d'optimisation, dont une description précise peut-être trouvée dans [VALD88].

Par cette phase, un programme FAD est transformé en un autre programme FAD. Le programme en entrée de l'optimiseur est exprimé en fonction du schéma conceptuel. Celui en sortie, l'est en fonction du schéma "interne", dans le sens où, il prend en compte les différents index disponibles. Notons néanmoins, qu'il ne s'agit là que d'un "premier niveau" de schéma interne. A ce stade, la répartition des données n'est aucunement prise en compte.

Le programme optimisé est exprimé en FAD et comporte, en plus, les "annotations" relatives aux décisions d'optimisation qui ne peuvent être traduites en FAD. Par exemple, l'ordre des jointures et la sélection des index peuvent être exprimées en FAD, contrairement au choix d'un algorithme particulier de jointure, qui sera spécifié par une telle annotation.

Pour réaliser une telle optimisation, le modèle de coût de la machine BUBBA est utilisé. Il consiste en une liste de méthodes d'accès et de fonctions de coûts qui s'y rapportent.

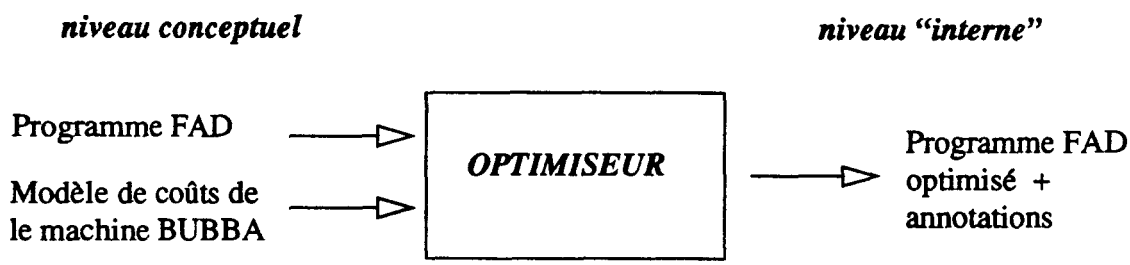


Fig. 4.2 : Principe général de la phase d'optimisation

De façon un peu plus précise, l'optimiseur opère les trois types de transformations que sont : l'optimisation des opérations conceptuelles, la "traduction" des opérations exprimées au niveau conceptuel des données en opérations exprimées au niveau "interne" des données et, le maintien de la consistance des données (exemple: propagation des mises à jour au niveau des index secondaires).

Ce travail est assuré par deux modules: le ***module de ré-écriture*** et le ***module d'optimisation***.

Le ***module de ré-écriture*** est basé sur un compilateur de grammaire d'attribut [AHO86]. Il est responsable de la construction d'un arbre représentant le programme FAD en entrée et, de la collecte d'informations concernant les arguments des opérations ensemblistes[1] du programme. Pour celles qui justifient une optimisation, ces informations sont envoyées au ***module d'optimisation*** qui détermine alors une stratégie efficace pour les supporter. Ce module renvoie alors au module de ré-écriture l'information nécessaire à la construction d'un arbre représentant un programme efficace sémantiquement équivalent.

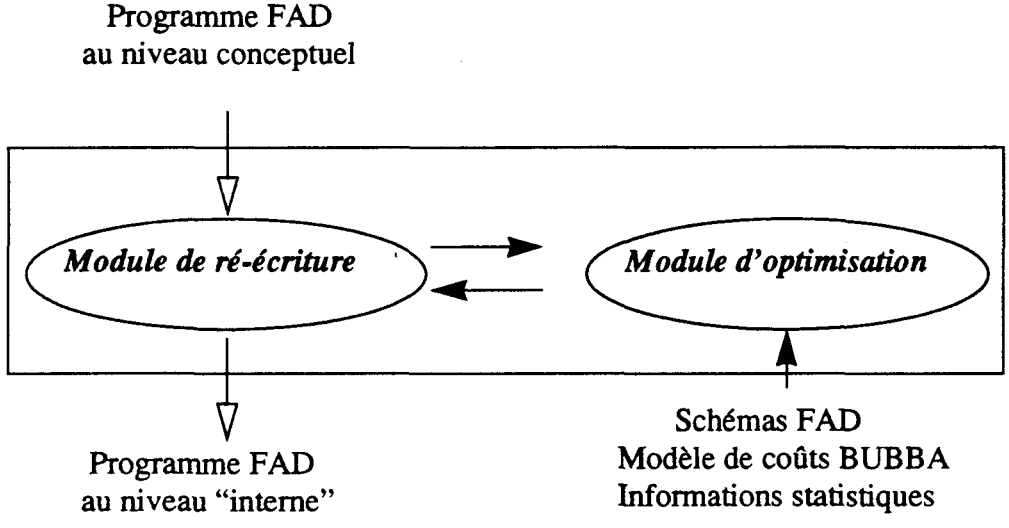


Fig. 4.3 : Les différents modules de l'optimiseur FAD

Nous ne descendrons pas plus dans le détail de la présentation de cette phase d'optimisation, les points présentés ci-dessus seront suffisants pour la prise en compte de nos solutions. Par contre, nous donnons ci-dessous, un petit exemple du résultat d'une telle optimisation.

```
database = [clientèle= { [code_client, nom_client, remise ] },
           commandes={ [ numéro, code_client, ristourne,
                         contenu= { [code_pièce, quantite] } ] },
           pièces= { [code-pièce, désignation, prix] } ]
```

Fig. 4.4 : Exemple de schéma au niveau conceptuel

1. Dans la version étudiée de l'optimiseur, [VALD88] seuls les traitements FILTER étaient traités.

Un tel schéma logique représente les données au niveau conceptuel. Si nous supposons, maintenant, l'existence d'un index secondaire sur le nom des clients, nous pouvons donner le schéma "interne" suivant:

```
database = [clientèle= { [code_client, nom_client, remise ] },
           commandes={ [  numéro, code_client, ristourne,
                          contenu= { [code_pièce, quantite] } ] },
           pièces= { [code-pièce, désignation, prix] },
           clientèle_idx= { [nom_client, code_client] } ]
```

Fig. 4.5 : Exemple de schéma au niveau "interne"

A partir de ces données, nous donnons ci -dessous, l'expression d'un programme FAD au niveau conceptuel et au niveau "interne" optimisé.

```
define changer_une_commande
    fun (Commande, Montant, Remise)
        let tupleassign (Commande, Ristourne + (Ristourne, Montant))
        in [Commande.numéro, Commande.ristourne, Remise]


define changer_toutes_les_commandes_d'un_client
    prog (Nom_du_client, Montant)
        filter      (fun (client, cde))
                        if and? (eq?(client.code_client, cde. code_client),
                                 eq? (client.nom_client, Nom_du_client))
                        then
                        changer_une_commande (cde, Montant, client.remise),
                    database.clientèle,
                    database.commandes)
```

Fig. 4.6 : Exemple de programme FAD au niveau conceptuel

Un tel programme aura donc pour effet d'augmenter d'un même montant, la ristourne attribuée à chaque commande d'un client donné, spécifié par son nom.

Une fois passé dans l'optimiseur, le programme FAD exprimé en fonction du ni-

veau "interne" est le suivant :

```
define changer_toutes_les_commandes_d'un_client
    prog (Nom_du_client, Montant)
        filter ( fun (A,B) (# jointure par boucles imbriquées #)
                    if eq? (A.code_client, B.1)
                    then changer_une_commande (A, Montant, B.2),
                database. commandes,
                filter( fun (C,D) (# jointure associative #)
                            if eq? (C.code_client, D.1)
                            then [C.code_client, C.remise],
                        database.clientèle,
                        filter ( fun(E) (# sélection exacte#)
                                    if eq? (E.nom_client, Nom_du_client)
                                    then [E.code-client],
                                 database. clientèle_idx )))
```

Prise en compte de l'index sur les noms qui existe

annotation

Fig. 4.7 : Même programme FAD mais optimisé et exprimé au niveau "interne".

Le fonctionnement de ce programme "interne" se décompose donc en trois étapes:

- recherche, grâce à l'index sur les noms, du code du ou des clients dont le nom est passé en paramètre (sélection exacte)

- recherche de l'attribut remise (nécessaire au résultat) par une opération de jointure associative puisque le code du ou des clients concernés est connu et que ce code est supposé être la clé de répartition des objets clients

- jointure par boucles imbriquées avec l'ensemble des commandes afin de retrouver les commandes du ou des clients concernés, et de leur appliquer le traitement *changer_une_commande* ; la jointure ne peut, dans ce cas, être associative, puisque nous ne disposons pas des index secondaires nécessaires.

Nous pouvons donc représenter ce programme optimisé par l'arbre suivant:

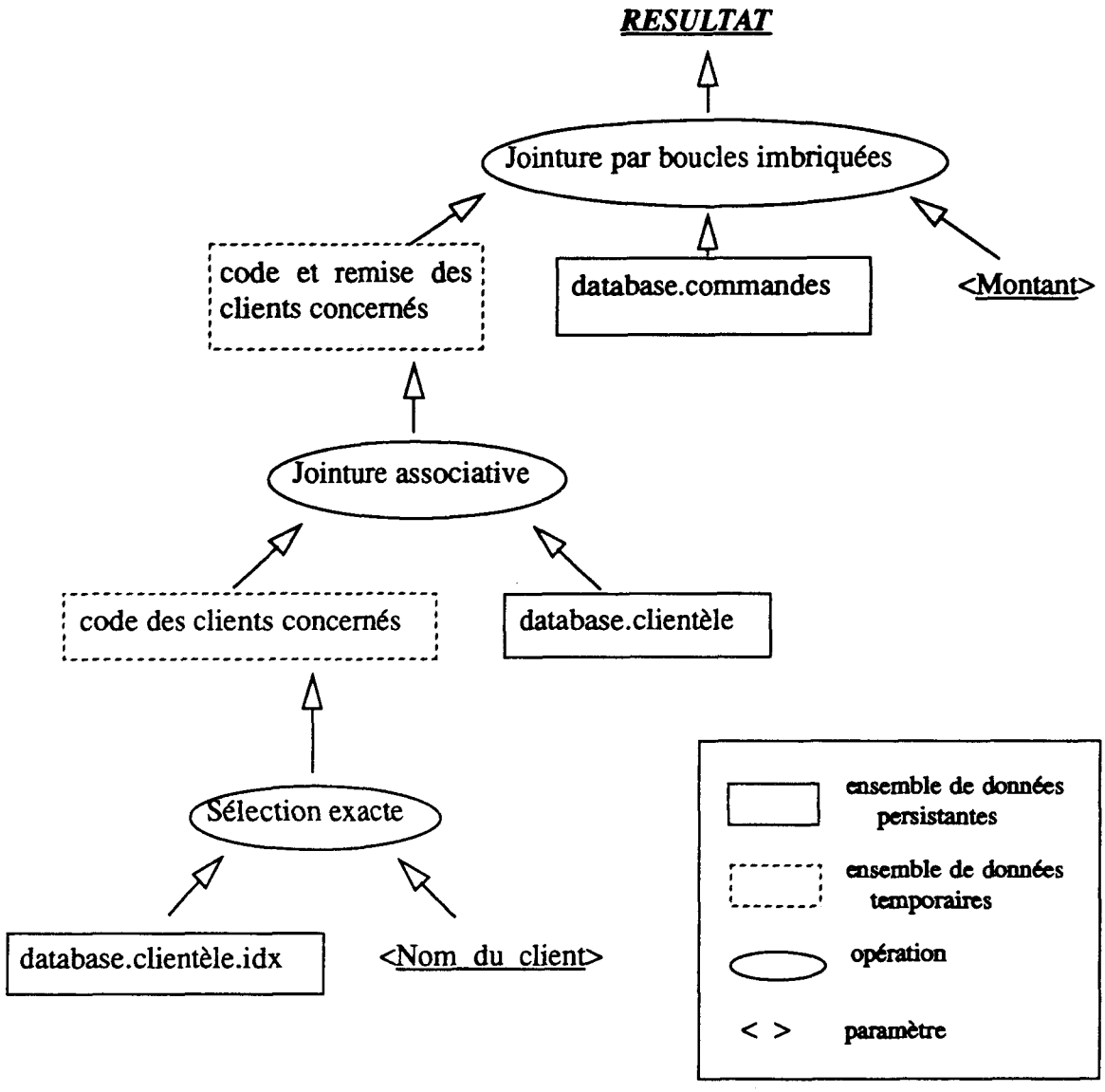


Fig. 4.8 : Représentation du programme sous forme d'arbre

4.1.1.3. La phase de génération de composants.

Suite à la phase d'optimisation, le programme FAD est donc soumis à la phase de génération de composants, qui consiste à le transformer en un programme exprimé en PFAD (Parallel FAD). PFAD est une extension de FAD dans laquelle l'exécution parallèle devient explicite en prenant la forme de ***composants*** et de ***communications par flots de données (data-flow) entre ces composants*** [DANF88b].

Un tel programme PFAD est généré automatiquement à partir du programme FAD optimisé, de la connaissance de la localisation des données et d'autres facteurs tels que les méthodes d'accès, les statistiques d'utilisation, etc.

Pour un programmeur, le modèle d'exécution de FAD est tel, qu'il voit toutes les données centralisées. Rappelons néanmoins, que les arguments des fonctions FAD sont conceptuellement évalués en parallèle, et que, les opérations ensemblistes reposent sur une idée d'exé-

cution parallèle. De ce fait, même si la possibilité de contrôle explicite du parallélisme, n'est pas fourni dans le langage, ce modèle d'exécution de FAD peut-être qualifié de MIMD.

Le langage ***PFAD*** peut, quant à lui, être vu comme ayant deux modèles d'exécution, l'un au ***niveau logique***, et l'autre au ***niveau physique***.

Au ***niveau logique***, chaque ensemble de données est considéré comme non éclaté et donc comme localisé en un seul endroit qui est appelé "***résidence***" de l'ensemble. Néanmoins les résidences de ces ensembles de données sont considérées comme différentes. Ce niveau correspond donc à l'approche centralisée que considère M.Livny [LIVN 87] (cf figure 3.6). Dès lors, les composants d'un programme PFAD au niveau logique, correspondent aux morceaux, d'un programme FAD, qui s'exécutent dans différentes résidences, et qui communiquent "d'une façon dataflow", par envois de messages, lorsque cela est nécessaire. Un programme PFAD logique peut donc être décrit sous la forme d'un macro-graphe dataflow où les noeuds sont les composants et les arcs représentent les canaux de communication utilisés par les instructions *send* et *receive* du code de ces composants. Le mode d'exécution obtenu à ce niveau logique est, cette fois "de façon sure", MIMD.

Au ***niveau physique***, niveau qui correspond à l'architecture de la machine, les résidences sont elles-mêmes distribuées ; en d'autre termes, tout ensemble de données est considéré comme "éclaté" sur différents noeuds de la machine et donc, comme soumis à la fragmentation horizontale. Le code d'un composant qui accède un tel ensemble éclaté, est en général dupliqué et exécuté dans chaque "sous-résidence", ce qui conduit à un mode de fonctionnement MIMD/SIMD (encore appelé MSIMD ou SMIMD). A ce niveau, les composants représentent potentiellement, plusieurs activations sur des noeuds différents, et les arcs dataflow entre composants dénotent un transfert d'informations plus complexe que celui fourni par un simple canal de communication.

Avant de préciser, sur un exemple, chaque niveau du langage PFAD, nous proposons de résumer ce que nous venons d'aborder par la figure ci-dessous.

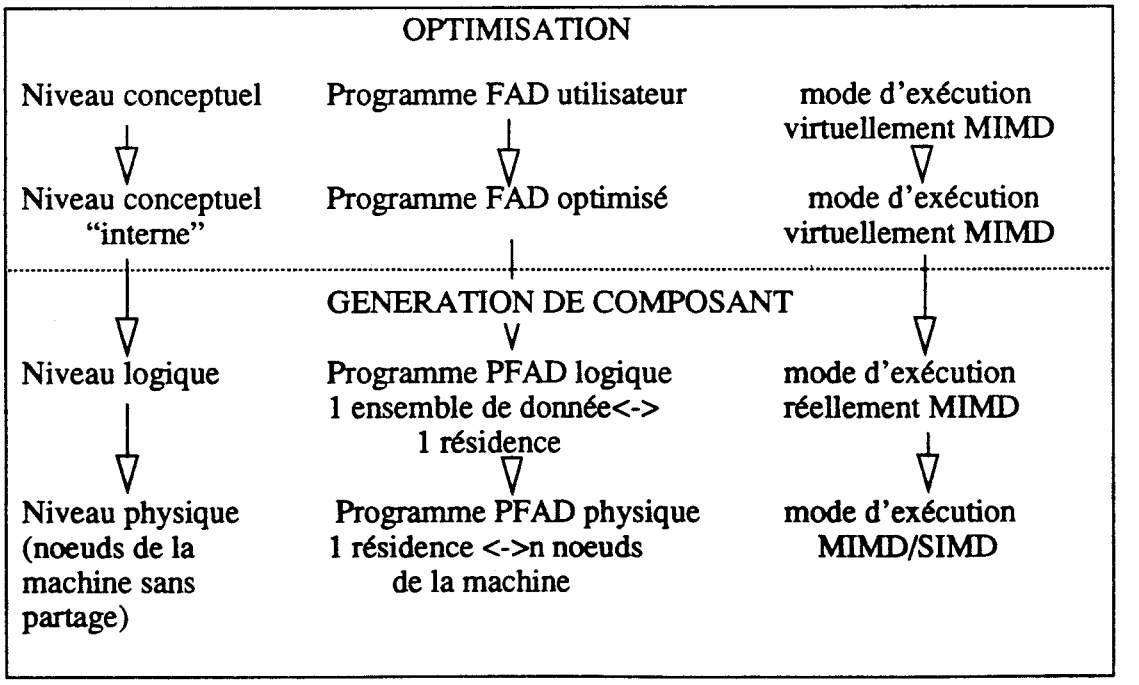


Fig. 4.9 : Principe de la phase de génération de composants

L'exemple que nous allons développer, repose sur le schéma logique suivant:

```
database= [personnel ={ [non_employe, age, salaire] },
          structure = {[ désignation_service, nom_employe_responsable]}]
```

Fig. 4.10 : Schéma logique utilisé pour l'exemple de génération de composants

Nous supposerons que nous disposons des données suivantes:

database.personnel

[Martin, 38, 7500]
[Dupont, 18, 4600]
[Durand, 31, 6150]
[Dubois, 49, 5200]

database.structure

[Compta, Dupont]
[Production, Martin]
[Production, Dubois]
[Expédition, Durand]

Fig. 4.11 : Données relatives à l'exemple

Nous supposerons enfin que ces deux ensembles de données, sont répartis de la façon suivante sur les 3 noeuds de la machine:

database.personnel:

salaire <= 5500 => noeud 1

5500 < salaire <= 7000 => noeud 2

7000 < salaire => noeud 3

database.structure:

nom_employé_responsable<= "Duhamel" => noeud 1

"Duhamel" < nom_employé_responsable<= "Lelong"=> noeud 2

"Lelong" < nom_employé_responsable => noeud 3

Cette répartition des ensembles de données, peut se résumer par la figure suivante:

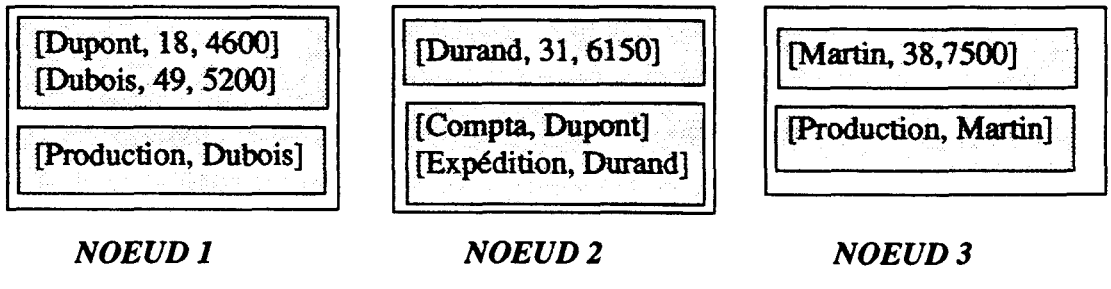


Fig. 4.12 : Répartition des données de notre exemple

Afin d'illustrer la phase de génération de composants, nous considérerons le programme ci-dessous qui consiste à donner les caractéristiques des responsables d'un service passé en paramètre.

```
define liste_responsables
    prog (Service)
        filter (fun (emp,ser)
                    if and? (   eq? (Service, serv.désignation_service),
                                eq? (ser.nom_employé_responsable, emp.nom_ employé))
                    then [Service, emp.nom_employé, emp.age, emp.salaire],
                database.personnel,
                database.structure)
```

Fig. 4.13 : Programme FAD utilisé pour l'illustration de la génération de composants.

Au ***niveau PFAD logique***, la base de données est vue comme partitionnée de telle façon que chacun des ensembles de données (database.personnel et database.structure) soit sur un "noeud" différent. Nous aurons donc un composant "logiciel" par ensemble de données. Pour réaliser le test d'égalité entre les données de ces ensembles, il devra donc y avoir un mouvement de données de l'un de ces ensembles vers l'autre.

Nous donnons ci-dessous, l'expression de ce programme PFAD logique ainsi que le macro-graphe dataflow qui lui est associé.

```
define liste_responsables
    prog (Service)

    composant 1 "sur" database.structure
        send ( filter (fun (serv)
                    if eq? (Service, serv.désignation_service)
                    then serv
               database.structure),
               2a)

    composant 2 "sur" database.personnel
        let ensemble_resp = receive (2a)
        in send (   filter (fun (emp, resp)
                        if eq? (resp. non_employe_responsable, emp.nom_employe)
                        then [Service, emp.nom_employé, emp.age, emp.salaire],
                    database.personnel,
                    ensemble-resp),
                   3a)

    composant 3 "sur" résultat
        receive (3a)
```

Fig. 4.14 : Programme PFAD logique associé au programme FAD

Remarques:

- A ce niveau logique, les deux opérations de communication entre composants, sont donc: ***send (données, canal)*** et ***receive (canal)***.

- Le troisième composant est là pour récupérer les résultats du programme. Puisqu'il n'agit pas sur un ensemble de données persistantes, il est, ce que les concepteurs de FAD appellent, un ***composant symbolique***.

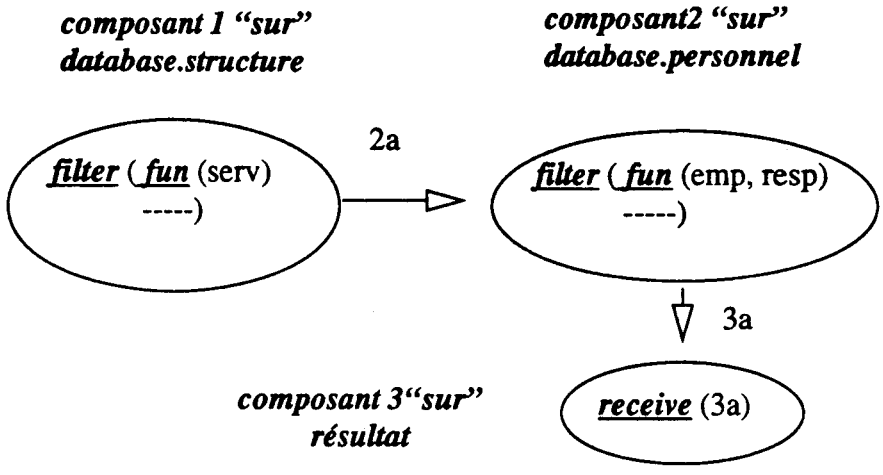


Fig. 4.15 : Macro-graphe dataflow associé au programme PFAD logique

Au ***niveau PFAD Physique***, la base de données est vue comme partitionnée, chaque ensemble de données étant cette fois "éclaté" sur différents noeuds de la machine. Un composant représente donc plusieurs exécutions du même code sur différents noeuds. Le modèle simple de communications inter-composants, du niveau logique, est généralisé pour prendre en compte le fait qu'une simple opération logique ***send*** peut induire des transmissions de données à partir de multiple sources vers de multiples destinations.

Le programme PFAD logique est donc transformé en un programme PFAD physique, de façon à prendre en compte l'éclatement des ensembles de données et l'exécution SIMD des composants, qui en résulte. En particulier, il peut être nécessaire de combiner les résultats des différentes exécutions d'un même composant ; cela peut impliquer la création de composants symboliques intermédiaires. De même, l'utilisation de conditions pour éviter l'exécution systématique d'un composant dans toutes ses "résidences" peut-être utile. Quoiqu'il en soit, ce problème d'obtention du programme PFAD physique, n'est pas simple. Une description précise peut en être trouvée dans [HART88].

Nous terminerons cette présentation synthétique de la phase de génération de composants, en donnant le programme PFAD physique obtenu sur notre exemple.

Remarque: Nous pourrions, à ce niveau, détailler l'exécution de ce programme sur les trois noeuds considérés mais cela n'apporterait pas d'informations intéressantes supplémentaires. En conséquence, nous en resterons là.

```
define liste_responsables
    prog (Service)

     composant 1 "sur" database.structure
          sendassoc ( filter (fun (serv)
                         if eq? (Service, serv.désignation_service)
                         then serv
                    database.structure),
                    2a: nom_employé_responsable)

     composant 2 "sur" database.personnel
          let ensemble_resp = receive (2a)
          in sendall (   filter ( fun (emp, resp)
                              if eq? (resp. non_employe_responsable, emp.nom_employe)
                              then [Service, emp.nom_employé, emp.age, emp.salaire],
                         database.personnel,
                         ensemble-resp),
                        3a)

     composant 3 "sur" résultat
          receive (3a)
```

Fig. 4.16 : Programme PFAD physique obtenu

La forme de ce programme PFAD physique est peu différente de celle du programme PFAD logique. Seule les opérations d'envois de messages sont changées.

L'opération ***sendassoc*** est utilisée pour un routage associatif, c'est-à-dire, lorsque l'on connaît la destination des données. Dans ce cas, il faut préciser sur quel critère (attribut *nom_employé_responsable* sur notre exemple) se base ce routage associatif (***sendassoc*** (données, 2a: nom-employé-responsable)). L'opération ***sendall*** est utilisée pour envoyer une union des données de tous les expéditeurs vers tous les destinataires, ce qui peut-être assimilé à une opération de diffusion multiple.

Notons, que cette forte ressemblance des deux programmes est en partie liée à la simplicité de notre exemple. Pour des programmes plus complexes, il intervient de plus amples modifications.

Après avoir présenté les différentes phases de compilation du langage FAD, nous allons maintenant discuter de la prise en compte, dans ce processus de compilation, des solutions que nous préconisons pour le placement de données.

4.1.2. Conséquences des solutions que nous proposons.

Pour ce qui nous concerne, les programmes FAD, exprimés au niveau conceptuel par l'utilisateur, restent les mêmes. Par contre le niveau interne, qui correspond aux données

effectivement manipulées par la machine, est différent de celui considéré sur la machine BUBBA. Les différences sont essentiellement induites par la fragmentation verticale des classes d'objets que nous avons proposée. Le processus de fragmentation horizontale soumis, n'a pratiquement aucune influence sur la compilation des programmes, puisqu'il ne concerne que l' obtention d'une répartition relative et homogène des données.

Nous nous intéresserons donc ici aux problèmes relatifs à la prise en compte de la fragmentation verticale dans les phases de compilation.

Les conséquences d'une telle fragmentation verticale sont de deux ordres. Tout d'abord un objet peut être découpé verticalement en différents objets partiels (phase 1 de la fragmentation verticale, cf 2.2). Ensuite, un objet partiel composant peut-être stocké avec son objet partiel composé (stockage partiel direct) ou de façon normalisée (stockage partiel normalisé, cf 2.3 phase 2 de la fragmentation verticale).

Dés lors, deux opérations peuvent intervenir, au niveau interne, pour reconstruire tout ou partie d'un objet ainsi découpé, à savoir l'opération de ***jointure interne horizontale*** et l'opération de ***jointure interne verticale***.

La ***jointure interne horizontale*** consiste en la reconstruction d'un objet ou d'une partie d'un objet, à partir de ses objets partiels. Elle peut donc être vue comme une jointure basée l'identifiant identique des objets partiels (cf figure 2.19).

La ***jointure interne verticale*** consiste en le "recollement" d'un objet partiel composant avec son objet partiel composé. Elle se fait donc par rapport à l'identifiant de l'objet partiel composant[1] (cf figure 3.70).

Un programme FAD donné risque donc de nécessiter au niveau interne l'utilisation de l'une de ces opérations. En conséquence, les phases de compilation du langage, doivent intégrer, à un niveau ou à un autre, la notion de fragmentation verticale.

4.1.3. A quel niveau intégrer la notion de fragmentation verticale?

Deux possibilités nous sont offertes pour l'intégration de la notion de fragmentation verticale dans les phases de compilation du langage: intégration dans la phase d'optimisation ou intégration dans la phase de génération de composants.

Nous allons considérer ici, la seconde solution pour mettre en avant les différents problèmes qu'elle pose. L'intégration dans la phase d'optimisation, solution que nous retiendrons, fera l'objet d'une présentation générale dans les sections suivantes.

L'intégration de la notion de fragmentation verticale dans la phase de génération de composants, pourrait consister à ajouter un niveau PFAD intermédiaire, entre le niveau logique et le niveau physique. Ce niveau, que nous pourrions qualifier de ***logique fragmenté***, consisterait à considérer la base de données comme constituée de fragments dont les données seraient stockées de façon centralisée, chacun des fragments étant assigné à un noeud "virtuel" différent.

---

1. Notons que nous ne donnons ici, qu'une définition synthétique de cette opération de jointure interne verticale, qui suivant les cas (partage, type de liens) peut être plus ou moins complexe.

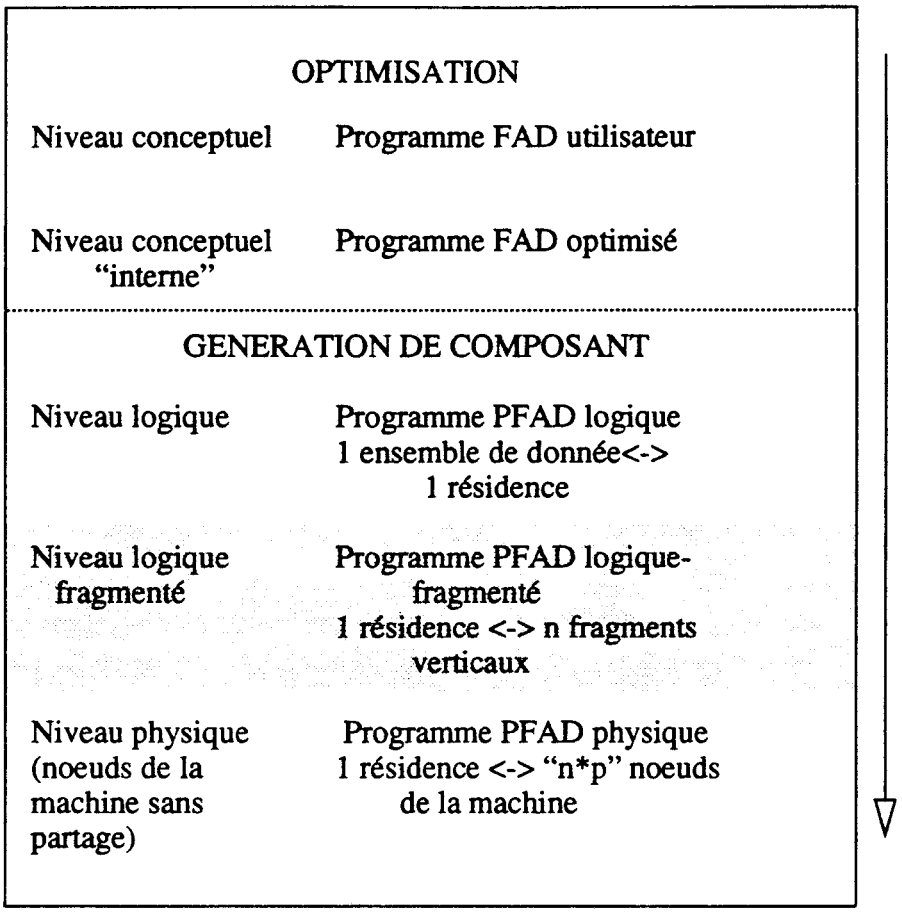


Fig. 4.17 : Intégration de la fragmentation verticale dans la phase de génération de composants

Cette solution, en apparence élégante, pose néanmoins le problème du rôle de la phase d'optimisation. A l'origine, ie dans la version implantée sur la machine BUBBA, cette phase d'optimisation a pour rôle de fournir un programme FAD optimisé et exprimé par rapport à un certain niveau interne de données. Entre autres, les index sur les données y sont pris en compte. L'intégration de la fragmentation verticale proposée ci-dessus, limiterait cette phase à une optimisation par rapport aux ensembles de données non fragmentés et ne prendrait donc pas en compte les opérations de jointure interne, ni les index, qui au niveau interne sont bien évidemment relatifs aux fragments verticaux. Somme toute, cette phase d'optimisation serait incomplète et surtout inutile, puisque "trop loin" des données effectivement manipulées par la machine.

Afin de conserver la même approche, à savoir obtenir, en entrée de la phase de génération de composants, un programme FAD optimisé en fonction du niveau "interne", il est donc nécessaire d'envisager l'intégration de la notion de fragmentation verticale dans la phase d'optimisation.

Nous allons discuter dans ce qui suit, d'une telle intégration. Nous verrons tout d'abord, ce que cela implique au niveau du langage FAD lui-même. Nous discuterons ensuite des

différentes façons de lier l'optimisation des programmes FAD et la fragmentation des données utilisées.

4.1.4. Sur l'influence de l'intégration au niveau du langage

Afin de pouvoir fournir, grâce à la phase d'optimisation, un programme FAD optimisé et exprimé en fonction du niveau interne que constituent les fragments verticaux et les index associés, nous allons devoir ajouter quelques fonctionnalités au langage FAD. Le programme obtenu ne sera donc plus vraiment écrit en FAD ; nous appellerons ***FAD+***, la version enrichie du langage.

Ces nouvelles fonctionnalités sont exclusivement liées à l'expression des opérations de jointure interne (horizontale ou verticale) et à la prise en compte de ces opérations dans le processus d'optimisation.

Comme nous l'avons signalé précédemment, l'utilisation de la fragmentation verticale entraîne, au niveau interne, des jointures internes afin de pouvoir reconstruire tout ou partie d'un objet. Si nous voulons exprimer les programmes optimisés en FAD, il nous faut donc être capable de traduire ces jointures internes dans le langage[1].

Puisque la traitement FILTER (cf 1.4.2.2 et figure 1.35) permet, entre autres, d'exprimer des jointures, il peut être utilisé, au niveau interne, pour répondre à notre problème. Néanmoins une telle utilisation nécessite l'ajout de certaines fonctionnalités au langage.

En fait, nous devons pouvoir récupérer l'identifiant d'un objet partiel pour pouvoir le manipuler comme tout autre valeur. Le langage FAD ne permet pas à l'utilisateur d'avoir accès à l'identifiant d'un objet. Bien que certains langages, qui supportent de la même façon l'identité d'objets (ex: SMALLTALK [GOLD83]), le permettent, une telle limitation n'est pas un handicap tant que l'on reste au niveau conceptuel. En effet, les opérations de manipulation de données de FAD sont suffisamment riches, pour éviter l'accès à ces identifiants.

Par contre, si nous voulons descendre au niveau interne des fragments verticaux, une telle possibilité est absolument nécessaire. Sans elle, nous ne pouvons exprimer les jointures internes, puisque nous ne sommes pas capables de comparer deux identifiants. Outre ce problème de comparaison des identifiants, nous pouvons aussi envisager de leur appliquer, à des fins d'optimisation, certaines opérations initialement réservées aux valeurs. Nous détaillerons ce second aspect dans la section suivante.

Pour l'heure, nous allons donner un exemple de programme FAD et sa ré-écriture en fonction du niveau interne relatif aux fragments verticaux. Afin de ne pas mélanger différents aspects, nous ne prendrons pas en compte dans cet exemple, les éventuels index qui peuvent exister parallèlement aux fragments. Notre but est, avant tout, de mettre en avant la traduction des opérations de jointure interne et les outils supplémentaires qu'elle nécessite.

Notre exemple sera basé sur le schéma logique suivant:

---

1. A ce niveau, il est important de remarquer que les opérations de jointure interne horizontale se feront uniquement entre fragments d'une même classe non-répartis relativement. Dans le cas contraire, pour des fragments répartis relativement, la reconstruction de l'objet à partir des objets partiels, sera triviale puisqu'il suffira de consulter le graphe physique de répartition associé (cf figure 3.67).

```
database = obj [  clientèle = {CLIENT}, commandes = {COMMANDE},
                  production = {ARTICLE}]
CLIENT = obj [num, nom, ville, CA_sem1, CA_sem2]
COMMANDE = obj [num_cde, num_client, num_vendeur, date, contenu = {LIGNE} ]
LIGNE = obj [ref_article, quantité]
ARTICLE = obj [ref_article, prix_unitaire, qute_stock, seuil_stock]
```

Fig. 4.18 : Schéma logique utilisé pour l'exemple relatif aux jointures internes

Pour l'exemple nous supposerons qu'il existe deux fragments pour la classe CLIENT, définis par:

CLIENT. f1 = obj [num_client, nom, ville] et

CLIENT. f2 = obj [CA_sem 1, CA_sem2]

Fig. 4.19 : Fragments verticaux de la classe CLIENT

Enfin, le programme que nous utiliserons est celui qui, pour un numéro de vendeur donné (Numvend), renvoie les commandes enregistrées par ce vendeur, pour des clients qui habitent une ville donnée (Ville) et qui ont un chiffre d'affaire pour le premier semestre (CA_-sem1) supérieur à un chiffre d'affaire donné (CA_min).

Les informations désirées pour chaque commande sont : sa date, son numéro et, le numéro, le nom et le chiffre d'affaire cumulé du client concerné. Nous donnons ci-dessous l'expression d'un tel programme:

```
define commandes_vendeur
    prog (Numvend, Ville, CA_min)
        filter ( fun ( client, cde)
                    if and? (and? (and? (  eq? (client.ville, Ville),
                                           >? (client.CA-sem1, CA-min)),
                                           eq? (client.num, cde.num-client)),
                                           eq? (cde.num-vendeur, Numvend)),
                    then [cde.date, cde. Mum-cde, client.num, client-nom,
                          + (CA_sem1, CA_sem2)],
                database.clientèle,
                database.commandes)
```

Fig. 4.20 : Programme FAD utilisé pour l'exemple relatif aux jointures internes

Notons que cette traduction du programme en est une parmi d'autres. Il est important de signaler qu'à ce niveau conceptuel, le choix de l'une ou l'autre des traductions sémantiquement équivalentes, ne devrait pas jouer sur le temps de réponse. En effet, si l'optimiseur remplit efficacement son rôle, quelque soit la forme conceptuelle du programme, les performances doivent être les mêmes.

Nous donnons ci-dessous l'expression du même programme par rapport au niveau interne. Il ne s'agit peut-être pas là, de la version la mieux optimisée. Elle n'a d'autre but que de mettre en évidence les "nouveautés" au niveau du langage FAD+.

```
define commandes_vendeur
    prog (Numvend, Ville, CA_min)
        filter (fun (A, B)
            if eq? (A.3, B.1)
            then [A.1, A.2, B.1, B.2, B.3],
            filter ( fun (cde)
                if eq? (cde. num_vendeur, Numvend)
                then [cde.date, cde.num_cde, cde.numclient],
                database.commandes|f
                filter ( fun (C, D)
                    if eq? (C.3, D.2)
                    then [C.1, C.2, D.1],
                    filter ( fun (cl_f1)
                        if eq? (cl_f1.ville, Ville)
                        then [cl-f1. Mum-client, cl-f1. nom, ID (cl_f1)]
                        database.clientèle|f1),
                    filter (fun (cl_f2)
                        if >? (cl_f2.CA_sem1, CA-min)
                        then [+(cl_f2.CA_sem1, cl_f2.CA_sem2), ID (cf_f2)]
                        database.clientèle|f2 ))))
```

Fig. 4.21 : Programme FAD+ exprimé par rapport au niveau interne

Deux nouvelles notations apparaissent dans un tel programme. Elles sont relatives, d'une part, à la fonction de récupération d'un identifiant : ***ID( )*** et, d'autre part, à la désignation des fragments verticaux: ***database. <ensemble>|fi***. Pour la deuxième, même si la classe correspondante n'est pas fragmentée verticalement, la même notation est utilisée, le fragment n'est alors pas indicé (exemple: database.commandes|f).

Notons que les identifiants récupérés (ID(cl_f1) et ID(cl_f2)) sont manipulés com-

me les autres valeurs puisqu'ils sont "insérés" dans des résultats temporaires et, ensuite, comparés entre eux. Somme toute, ce langage FAD+ est très peu différent de FAD. Mise à part la nature des ensembles de données accédées, seule la fonction ID( ) est réellement nouvelle.

Nous terminerons cette partie en donnant l'expression du programme FAD+ précédent sous forme d'arbre. Cette représentation, indépendante de toute syntaxe précise, va nous permettre d'aborder les problèmes relatifs à l'optimisation de tels programmes FAD+.

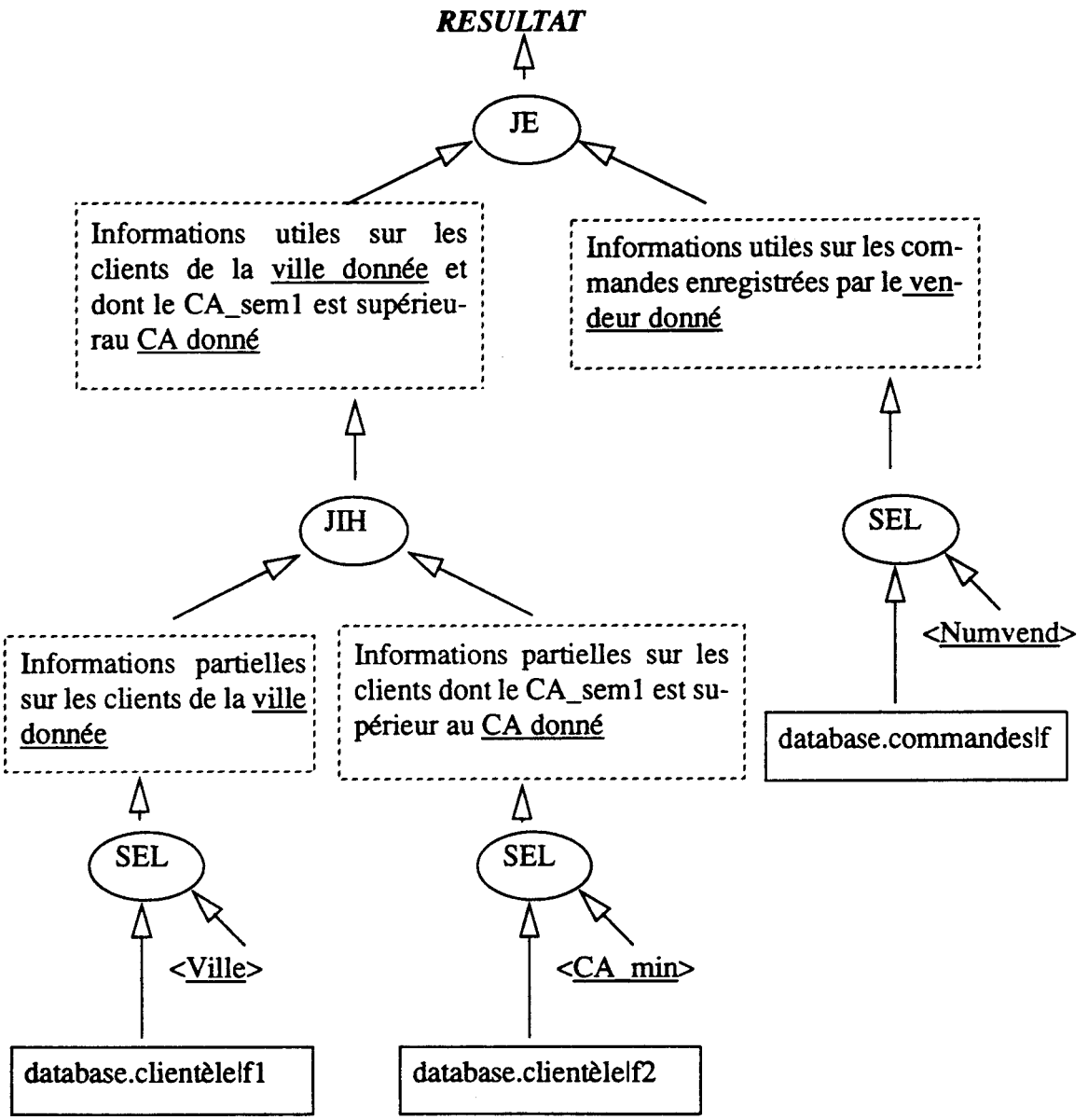


Fig. 4.22 : Représentation du programme FAD+ sous forme d'arbre.

4.1.5. Sur l'optimisation des programmes FAD+.

Par rapport à l'optimisation réalisée sur la machine BUBBA, la principale nouveauté à prendre en compte réside dans les opérations de jointure interne.

Une première solution, que nous pouvons qualifier de ***solution naïve***, pourrait consister à effectuer les jointures internes le plus tôt possible, c'est-à-dire à les considérer comme des opérations "non sécables". Par cette méthode, les opérations faisant intervenir des fragments de différentes classes interviendraient toujours après les jointures internes (verticales ou horizontales) relatives à chacune des classes concernées. En d'autres termes, les données partielles (issues des fragments) seraient toujours "recollées" avant d'être effectivement utilisées.

L'exemple que nous avons donné dans la figure 4.22 repose sur cette méthode puisque la jointure externe (JE) intervient après la jointure interne horizontale (JIH) relative à la classe CLIENT.

L'avantage de cette solution naïve est qu'elle ne "perturberait" pratiquement pas l'optimiseur qui existe sur BUBBA. Néanmoins, par cette méthode, les opérations de jointure interne ne seraient, en aucune façon, prises en compte dans le processus d'optimisation. Une telle solution s'avère donc tout à fait insuffisante.

La solution sérieuse, pour répondre au problème de l'optimisation, consiste à mettre sur le même plan, les opérations explicites et les opérations internes. Ainsi, une fois le programme FAD ré-écrit en programme FAD+, l'ensemble des opérations passe dans le module d'optimisation. Dés lors, les jointures internes se trouvent optimisées au même titre que les opérations explicites exprimées par l'utilisateur.

Suivant les paramètres attachés à chacune des opérations concernées, l'application de cette seconde méthode à notre exemple peut conduire à différents programmes FAD+. Nous donnons dans la figure suivante (4.23), et sous forme d'arbres "abrégés", les deux solutions autres que celle déjà présentée dans la figure 4.22.

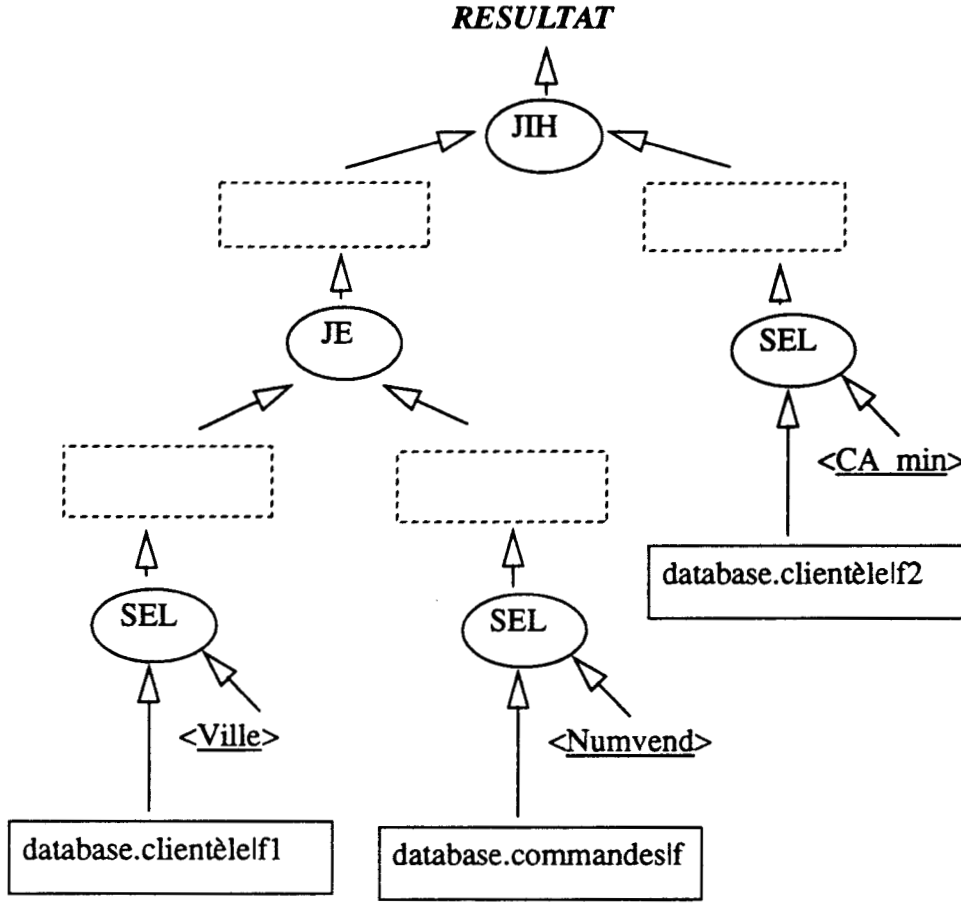
Dans chacune de ces solutions, la jointure interne horizontale ne se fait pas entre les deux fragments de la classe CLIENT, mais entre un seul de ces fragments et un ensemble de données issu d'autres opérations. Bien que le programme FAD utilise des données issues de chacun des fragments de la classe CLIENT, il n'y a pas, au niveau du programme FAD+, de reconstruction immédiate des objets de CLIENT à partir de leurs objets partiels. Le "recollement" des données a été optimisé au même titre que toute autre opération et n'est effectué qu'en dernier lieu.

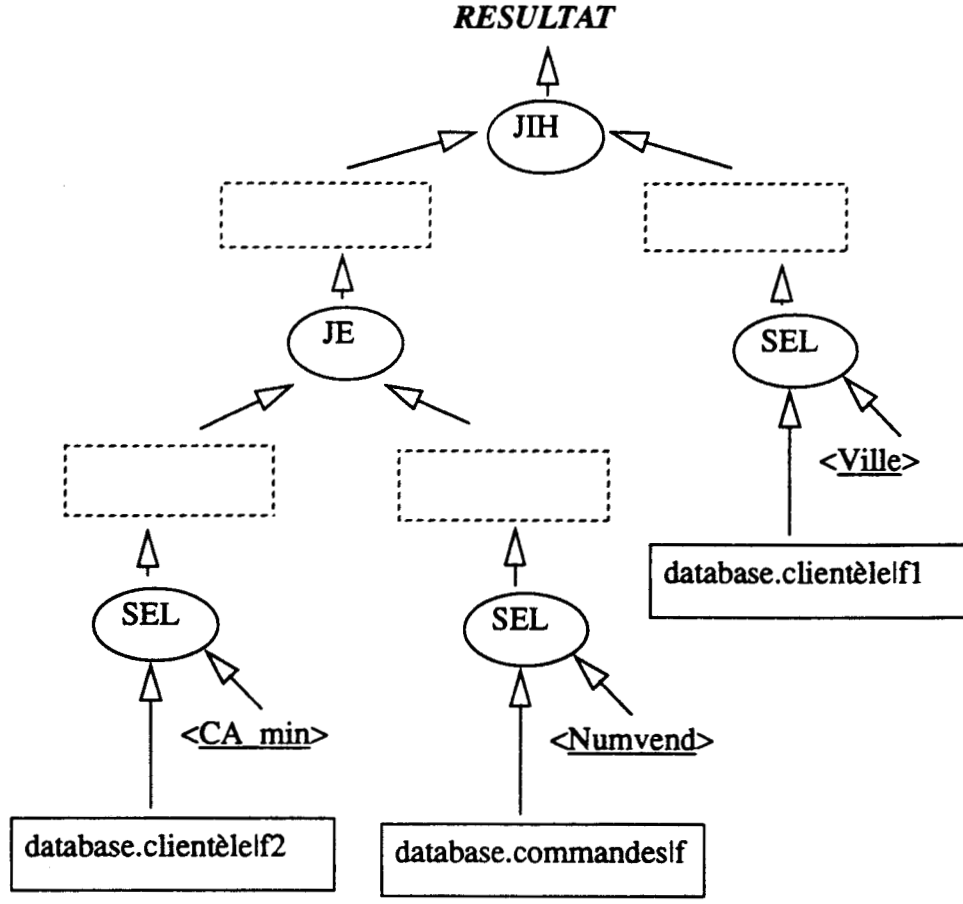
La généralisation de l'optimiseur FAD actuel, afin qu'il puisse réaliser de telles optimisations, consisterait, en partie, en l'incorporation dans la table des méthodes d'accès de celles qui concernent les jointures implicites.

Par ailleurs, l'utilisation, à des fins d'optimisation, d'opérations de semi-jointure est tout à fait envisageable grâce aux fonctionnalités supplémentaires de FAD+.

Enfin, certaines informations quantitatives, issues de notre transformation de schéma, pourraient être prises en compte dans le processus d'optimisation. Entre autres, les distances physiques inter-fragments pourraient fournir des informations très utiles pour l'optimisation des jointures internes.

Fig. 4.23 : Autres possibilités de programmes FAD+ à partir de notre exemple





Un tel travail, qui peut être vu comme une des extensions possibles de nos propositions, aurait beaucoup à tirer des nombreux travaux relatifs à "l'optimisation relationnelle". Bien que nous ayons uniquement fait allusion aux éventuelles duplications de fragments verticaux ou horizontaux, "il y a fort à penser" que l'optimisation des bases de données réparties constituerait aussi une bonne source de réflexion.

4.1.6. Conclusions

Nous terminerons cette partie, relative à la présentation d'un exemple de prise en compte de nos solutions, par quelques remarques d'ordre général.

Revenons, tout d'abord, sur l'ensemble du processus de compilation du langage FAD. Le fait d'avoir intégré la notion de fragmentation verticale dans la phase d'optimisation n'entraîne pratiquement aucune modification de la phase de génération de composants (cf4.1.1.3). Seule la nature des ensembles de données manipulés change, puisqu'il s'agit maintenant de données partielles. La démarche générale reste la même; ainsi, le principe d'un découpage en deux niveaux "logique" et physique peut être conservé. L'expression des composants logiciels ne change pas (hormis l'ajout des manipulations d'identifiant) et la représentation sous forme de macro-graphe dataflow est invariante.

Considérons maintenant les autres utilisations potentielles du langage FAD+. Certaines étapes de la méthode de transformation de schéma que nous préconisons nécessitent une connaissance "précise" des requêtes soumises au système.

Ainsi, la détermination du mode de répartition à utiliser pour un ensemble de données (cf 3.4.1 Les différents modes de répartition et leur influence) et du degré de répartition à adopter (cf3.2 Importance du degré de répartition dans notre contexte) demande une connaissance des requêtes ou programmes utilisateur au niveau interne, c'est-à-dire au niveau des fragments verticaux effectivement manipulés par la machine. En effet, à cause des opérations internes induites par la fragmentation verticale, il n'est pas possible de se baser sur la version conceptuelle des programmes pour déterminer de tels paramètres.

Dans ce cadre, la forme ré-écrite en FAD+ de ces programmes est tout à fait adaptée à ce type de travail. En effet, de tels programmes FAD+ prennent bien en compte toutes les opérations qui seront effectivement exécutées (explicites et internes) et traduisent ainsi la connaissance "précise" des requêtes dont nous avons besoin.

Ainsi se termine la présentation de cet exemple de prise en compte de nos solutions. Bien qu'il ne s'agisse que d'un exemple, et surtout d'une ébauche dans le domaine, il est fort probable que l'intégration de ces solutions sur d'autres machines, et à partir d'autres langages, soulèverait les mêmes problèmes. Ainsi, nous retrouverions inévitablement le problème des opérations internes induites par la fragmentation verticale. De même, l'optimisation devrait aussi être revue.

Néanmoins, quelque soient la machine et le langage retenus, l'aspect ensembliste des opérations de manipulation de données, conjugué à notre méthode de transformation de schéma, nous conduirait certainement à appréhender ces problèmes de façon analogue.

4.2. Sur l'évolution de nos travaux.

Nous consacrerons cette dernière partie du document à l'évolution, ou plutôt aux évolutions, que nous pouvons envisager pour nos travaux. Tout d'abord, nous discuterons des évolutions dans un cadre général. Nous aborderons, ensuite, celles relatives à l'utilisation de nos solutions pour une machine et un langage précis. Enfin, nous terminerons par une conclusion générale.

4.2.1. Evolutions dans un cadre général.

Nous entendons, par évolutions dans un cadre général, les extensions de nos travaux, autres que celles qui consistent en leur utilisation sur une machine et pour un langage précis. De telles évolutions concernent des solutions générales et adaptables dont l'analyse se ferait en suivant une démarche analogue à celle utilisée pour les points traités dans le document.

Pour des raisons d'homogénéité, et surtout pour profiter de l'environnement "courant", ces évolutions ont été présentées dans la section de synthèse de chacun des chapitres 2 et 3. Nous nous contenterons simplement d'en rappeler les "désignations".

Ainsi, au titre des extensions envisageables au niveau de la fragmentation verticale (cf2.4.2), nous avons proposé, l'***utilisation de différents modes de stockage pour un même fragment vertical***, la ***prise en compte d'autres types de liens entre classes d'objets*** et l'***étude de la transposition de la notion d'héritage à un niveau interne***.

En ce qui concerne la fragmentation horizontale, nous nous sommes arrêtés sur deux extensions (cf 3.5.2). Tout d'abord, nous avons traité de l'***utilisation d'une répartition relative de fragments de classes différentes non liées*** (cf 3.5.2.1). Nous avons, ensuite, discuté de l'***utilisation du stockage normalisé relatif*** (cf 3.5.2.2).

Chacun de ces points mériterait une analyse détaillée afin d'enrichir l'ensemble des solutions générales et adaptables que nous avons proposé.

4.2.2. Evolutions dans le cadre d'une machine et d'un langage précis.

Nous allons discuter, ici, des évolutions de nos travaux en vue de leur utilisation dans un cadre précis. Avant d'entrer dans le détail de telles évolutions, nous ferons une remarque sur les problèmes liés à l'entreprise d'un tel projet.

Ces problèmes, locaux à notre environnement de travail, sont de deux ordres puisqu'ils concernent à la fois "l'aspect humain" et l'aspect matériel. "L'aspect humain" est relatif à la notion d'équipe de recherche axée sur les bases de données. Une telle équipe n'existe pas, pour l'instant, au LIFL, ce qui limite fortement les réalisations envisageables dans ce domaine.

L'aspect matériel est, quant à lui, lié à l'absence d'une machine parallèle sans-partage, machine qui servirait de support aux travaux envisagés.

Passés ces problèmes locaux et espérant une évolution favorable dans l'une ou (et?)

l'autre des directions, nous pouvons tout de même aborder ici les problèmes relatifs aux extensions de nos travaux en vue de leur utilisation dans un cadre précis.

Comme nous l'avons signalé à plusieurs reprises, notre méthode de transformation de schémas logiques en schémas physiques répartis repose sur la connaissance de certains paramètres liés à la machine-cible (cf Tableau récapitulatif dans 3.5.3). Les solutions que nous avons proposé s'arrêtent à la frontière que représentent ces paramètres. L'implantation de telles solutions, sur une machine et pour un langage précis, nécessite donc leur détermination.

Un tel travail relève de l'utilisation de ***techniques de modèlisation quantitative*** [FDID82]. En effet, l'étude d'un système réel étant pratiquement irréalisable (de par sa complexité), la modélisation est le seul moyen pour en analyser le comportement et pour déterminer les paramètres-clés qui s'y rapportent. Différentes techniques de modélisation quantitative sont utilisables. A ce titre, il est important de signaler que ces techniques s'adaptent plus ou moins bien aux situations qu'elles doivent modéliser. Nous en donnons, ci-dessous, une brève énumération:

- L'***émulation*** qui consiste à extraire des résultats en utilisant le même environnement de programmation mais, sur une machine virtuelle ; notons que, dans notre cas, puisque nous ne disposons pas de machine sans-partage, cette solution pourrait être de mise.

-Les ***mesures*** qui consistent à placer des sondes matérielles ou logicielles afin d'espionner le système réel auquel sont soumises différentes charges de travail. En ce qui nous concerne, cette solution semble peu probable.

-Les ***méthodes analytiques*** qui reposent sur la théorie des files d'attente. Ces méthodes sont limitées par le nombre restreint de modèles associés qui possèdent une solution exacte. De plus, elles sont mal adaptées à la modèlisation de certains comportements "informatiques" tels que la synchronisation, la concurrence pour l'accès à des ressources partagées, etc. Néanmoins, elles constituent l'outil le plus souvent utilisé, et certains langages de description et d'étude de modèles ayant la structure de réseaux de files d'attente ont été proposés [GELE82]. Notons que l'outil de modèlisation FIRM, utilisé par l'équipe du MCC pour la machine BUBBA, est basé sur une méthode analytique [BOUG87].

-La ***simulation*** qui consiste, après avoir modèlisé le système sous une certaine forme, à simuler son comportement à partir d'un grand nombre d'échantillons. Ces échantillons peuvent être issus d'une trace préalablement enregistrée sur le système existant (on parle alors de simulation par trace) ou d'une génération aléatoire (simulation par événements discrets). Pour chaque simulation, on obtient un point de fonctionnement du système; l'optimisation de ce système peut donc devenir très "onéreuse".

Le choix de techniques, pour répondre à notre problème de détermination de paramètres liés à la machine-cible, est donc vaste. Une connaissance sérieuse des théories sur lesquelles elles reposent (notamment en ce qui concerne les méthodes analytiques) nous semble nécessaire afin d'opter pour la solution la mieux adaptée. Les techniques de modèlisation constituent bien un domaine à part entière pour lequel, il faut bien l'avouer, nous ne connaissons, pour l'instant, que l'aspect superficiel.

Comme nous l'avons déjà signalé, nous envisageons, dans un premier temps, d'adapter à nos besoins l'outil de modèlisation FIRM développé au MCC[BOUG87]. La construction "de toutes pièces" d'un outil de modélisation général permettant de prendre en compte différentes configurations matérielles, différents modèles d'exécution, différentes techniques de placement de données, etc, pourrait constituer une seconde phase mais demanderait, sans aucun doute, des bases plus sérieuses dans le domaine de la modélisation.

La seconde évolution possible de nos travaux dans un cadre précis pourrait concerner la prise en compte de nos solutions au niveau de l'optimisation des programmes. Nous avons précédemment abordé ce problème par rapport au langage FAD (cf 4.1.5). Là encore, l' optimisation des programmes de manipulation de bases de données est pratiquement un domaine à part entière. Les outils qui y sont associés sont loin d'être triviaux. Sous réserve de fixer une machine et un langage précis, le sujet reste ouvert.

4.2.3. Conclusion générale

En fait, nous pouvons résumer les différentes évolutions possibles de nos travaux par la figure suivante:

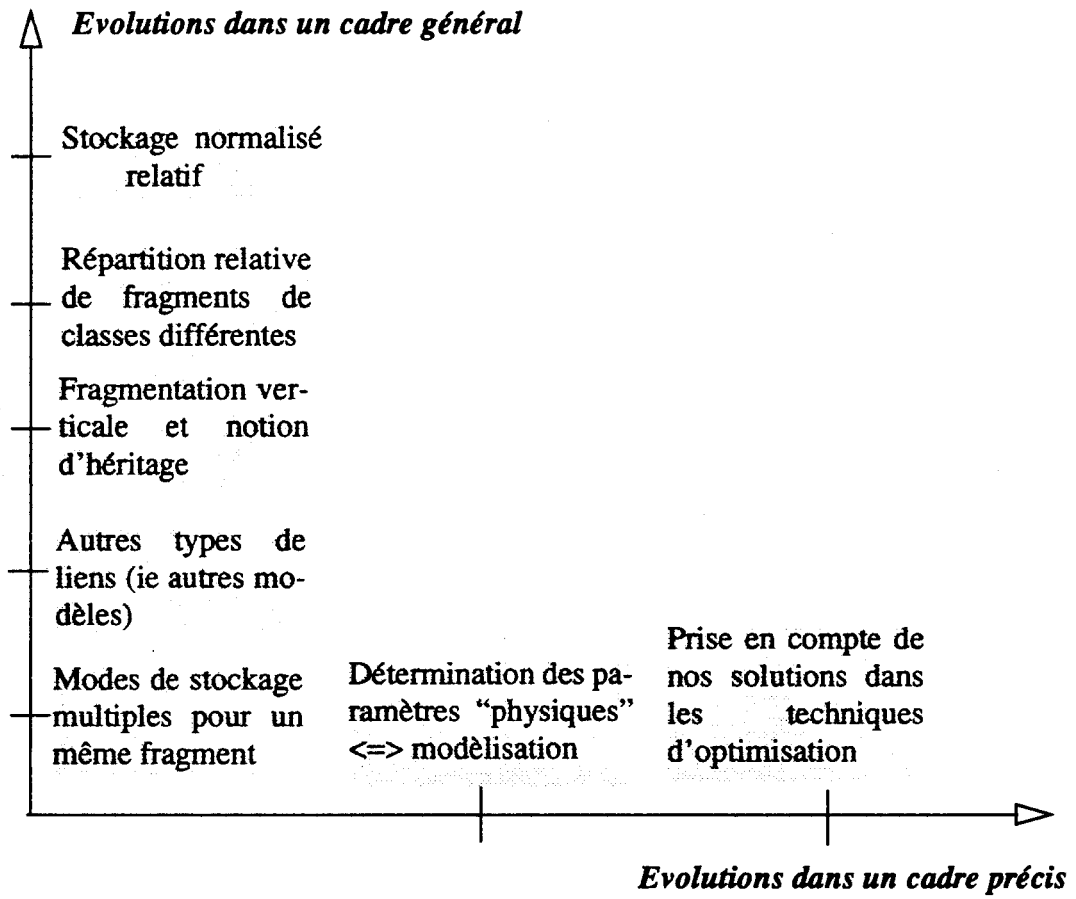


Fig. 4.24 : Représentation des différentes évolutions possibles.

Ces quelques propositions laissent suffisamment de points ouverts pour ne pas "périr d'inactivité". De par la variété des domaines concernés, de telles évolutions ont tout intérêt à être considérées dans le cadre d'un véritable travail d'équipe "Bases de Données".

C'est donc sur cet espoir, de voir se former une telle équipe dans la région Nord Pas-de-Calais, que nous terminerons ce document.

BIBLIOGRAPHIE

[ABIT86] S. ABITEBOUL, N. BIDOIT
Non First Normal Relations: an algebra allowing data reconstructing
Journal of Computer and System Sciences Vol 33 1986 p361-393

[ABIT87] S.ABITEBOUL, S. GRUMBACH
Bases de Données et Objets Structurés
Techniques et Sciences Informatiques Vol 6 Num. 5 1987

[ADIB86] M. ADIBA, G. GARDARIN, C. ROLLAND, R. DEMOLOMBE, M. SCHOLL, J. RHOMER
Nouvelles perspectives des bases de données
Ed. Eyrolles 1986

[AHO86] A. AHO, R. SETHI, J.D ULLMAN
Compilers: Principles, Techniques and Tools
Add Wesley Publishing 1986

[AHO87] A. AHO, J. HOPCROFT, J. ULLMAN
Structures de données et algorithmes (traduit par J.M. Moreau)
Add Wesley Europe Inter-Editions 1987

[ALEX88] W. ALEXANDER, G. COPELAND
Comparison of dataflow control techniques in distributed data intensive systems
ACM SIGMETRICS Conf. Santa Fe New Mexico may 88

[ANDR88] T. ANDREW, C. HARRIS, K. SINKEL
The ONTOS Object Database
??

[ANSI86] American National Standard for Information Systems
Database Language SQL
ANSI X3.135-1986 octobre 1986

[BABB79] E. BABB
Implementing a Relational Database By means of specialized hardware
ACM Transactions on Database Systems Vol 4 Num. 1 march 79 p1-29

[BACK78] BACKUS
Can programming be liberated from the Von Neuman style?
A functional style and its algebra programs.
CACM Vol 21 Num. 8 August 78

[BANC83] F. BANCILHON et al
VERSO: a Relational Backend Database Machine
in "Advanced Database Machine Archirecture" Prentice Hall 1983 p 1-17

[BANC87] F. BANCILHON, T. BRIGGS, S. KHOSHAFIAN, P. VALDURIEZ
FAD, a powerful and simple Database Language
Proc. of the 13th VLDB Conf. Brighton 1987

[BANC88] F. BANCILHON et al
The design and implementation of O2, an Object Oriented Data Base System
Workshop on OODBS. Badmunster West-Germany 1988

[BANE78] J. BANERJEE, D.K. HSIA, R. I. BAUM
Concepts and capabilities of a database computer
ACM Transactions on Database Systems Vol 3 Num. 4 1978 p347-384

[BERM86] F. BERMAN, M.E. BOCK, E. DITTERT, M.J. O'DONNELL, D. PLANK
Collections of functions for perfect hashing
SIAM Journal Computer Vol 15 Num. 2 may 1986

[BORA88] H. BORAL
Parallelism in BUBBA
MCC Tech. Report Num. ACA-ST-296-88 August 1988

[BOUG87] E. BOUGHTER, W. ALEXANDER, T. KELLER
A tool for performance-driven design of parallel systems
MCC technical report ACA-ST-312-87 1987

[BOUZ87] M. BOUZEGHOUB, J. COMYN, D. RICHARD
Conception de bases de données Etat de l'art
Publication interne 190 Lab. MASI Université P et M Curie PARIS 1987

[BRIT82] BRITTON-LEE Inc.
IDM 500 Ref. Manual
Britton Lee Inc Los-Gatos California 1982

[CERI85] S. CERI G. PELAGATTI
Distributed databases Principles and systems
Mc Graw-Hill International editions. Computer Science Series

[CHAM76] D. D. CHAMBERLIN et al
SEQUEL2: A Unified Approach to Data Definition, Manipulation and Control
IBM J. R&D, Vol 20 Num. 6 nov. 76

[CHAM80] D.D. CHAMBERLIN
A Summary of User Experience with the SQL Data Sublanguage
Proc. Int. Conf. on Databases, Aberdeen, Scotland juillet 1980

[CHAN80] S. K. CHANG, W. H. CHENG
A methodology for structured database decomposition
IEEE Transactions on Software Engineering Vol SE 6 Num. 2 march 80

[CHAN84] C.C. CHANG
The study of an ordered minimal perfect hashing scheme
CACM 27 Num. 4 p 384-387 1984

[CHEN76] P. CHEN
The Entity-Relationship Model: Toward a unified view of data
ACM TODS Vol1 Num. 1 mars 76

[CICH80] R.J. CICHELLI
Minimal perfect hash functions made simple
CACM Vol 23 Num. 1 1980 p17-19

[CODD70] E.F. CODD
A relational model of data for large shared data banks
CACM Vol 13 Num. 6 1970 p 377-387

[CODD79] E.F. CODD
Extending the Relational Model to capture more meaning
ACM TODS Vol 4 Num. 4 dec. 79

[COPE82] Société COPERNIQUE
DORSAL32: Documentation technique
Paris 1982

[COPE84] G. COPELAND, D.MAIER
Making Smalltalk a Database System
ACM SIGMOD Int. Conf. BOSTON june 1984

[COPE85] G.P. COPELAND, S.N. KHOSHAFIAN
A decomposition storage model
ACM-SIGMOD 85 Int. Conf. on Management of Data Austin may 85

[COPE88] G. COPELAND, W. ALEXANDER E. BOUGHTER T. KELLER
Data placement in BUBBA
ACM SIGMOD Conf. Chicago Illinois May 1988

[CVET87] Z. CVETANOVIC
The effect of problem partitioning, allocation, and granularity on the performances of a multiprocessor system
IEEE Transactions on Computers Vol C36 Num. 4 Avril 1987

[DANF87] S. DANFORTH, S. KHOSHAFIAN, P. VALDURIEZ
FAD, a database programming language, revision 2
MCC Tech. Report Num. DB-151-85 Rev 2 Sept 87

[DANF88a] S. DANFORTH, P. VALDURIEZ
The data model of FAD, a database programming language
MCC Tech. Report Num. ACA-ST-059-88 Feb. 88

[DANF88b] S. DANFORTH, B. HART, P. VALDURIEZ
Database components. Parallelism based on data location
MCC Technical Report Num. ACA-ST-035-88 October 1988

[DATE85] C. DATE
Introduction au standard SQL
Inter-Editions 1985

[DEWI79] D.J. DEWITT
DIRECT: a multiprocessor organization for supporting a Relational Database Management System
??

[DEWI86] D.J. DEWITT, R.H. GERBER & al
GAMMA, a high performance dataflow database machine
12th International conference on Very Large Data Bases Kyoto août 1986

[DOWD82] L.W. DOWDY, D.V. FOSTER
Comparative models for file assignment problem
ACM computing surveys vol14 Num. 2 1982 p287-313

[EISN76] M.J. EISNER, D.G. SEVERANCE
Mathematical techniques for efficient record segmentation in large shared databases
JACM Vol 23 Num. 4 p619-635 1976

[FDID82] S. FDIDA, G. PUJOLLE
Modèles de systèmes et de réseaux Tome 1 Performances
Ed EYROLLES 1982

[FISC83] P.C. FISCHER, S.J. THOMAS
Operations on Non First Normal relations
IEEE Computer Software and application Conference New-York Oct 1983

[FISH88] D. FISHMAN et al
Overview of the IRIS DBMS
Hewlett Packard Int. Report Palo-Alto (California) 1988

[FLYNN66] M.J. FLYNN
Very High Speed Computing Systems
Proc. IEEE Vol 54 Num. 12 Dec. 66 p1901-1909

[FRED82] M.L. FREDMAN, J. KOMLOS, E. SZERMEREDI
Sorting a sparse table with O(1) worst case access time
Proc. of 23rd Symposium on Foundations of Computer Science IEEE 1982

[GARD81] G. GARDARIN
An introduction to SABRE, a multi-processor Database Machine
6th workshop on Computer Architecture for Non Numeric Processing Hyères Juin 81

[GARD84] G. GARDARIN, E. GELEMBE
New Applications of Data Bases
Academic Press 1984

[GARD88] G. GARDARIN, P. VALDURIEZ
Bases de données relationnelles. Analyse et comparaison des systèmes
Ed EYROLLES 1988

[GARD88b]
Bases de données Orientées Objets
Support de cours, Université d'été AFCET juillet 1988

[GARD90] G. GARDARIN, P. VALDURIEZ
SGBD avancés, bases de données objets, déductives, réparties
Editions Eyrolles 1990

[GELE82] E. GELENBE, G. PUJOLLE
Introduction aux réseaux de files d'attente
Ed EYROLLES 1982

[GOLD83] A. GOLDBERG, D. ROBSON
Smalltalk-80: the language and its implementation
Livre Addison Wesley 1983

[GRAP87] GRAPHAEL S.A.
G-BASE 3.2 Manual 2nde Edition
Avril 88 Université de Compiegne

[HAMM79] M. HAMMER, B. BIAMIR
An heuristic approach to attribute partitioning
ACM SIGMOD Int. Conf. on Management of Data 1979 p93-100

[HART88] B. HART, S. DANFORTH, P. VALDURIEZ
Parallelizing a database programming language
MCC Tech. Report Num. ACA-ST-257-88 August 88

[HILL85] W.HILLIS
The Connection Machine
MIT Press, 1985

[HOFF75] J.A. HOFFER
The use of cluster analysis in physical database design
First Int. Conference on Very Large Databases Fragmingham sept 77

[IBM69] IBM
Information Management System/360. Application, Description manuals
IBM form Num H20-0524, 1969

[ICL82] International Computers Limited
CAFS-ISP Documentation commerciale
ICL 1982 Londres

[INTE81] Intel. Corp
IDBP DBMS Ref. Manual
Documentation Intel. Austin Texas

[JAES81] G. JAESCHKE
Reciprocal hashing: a method for generating minimal perfect hashing functions
CACM Vol 24 Num. 12 1981 p829-833

[JAES82] B. JAESCHKE, H.J. SCHEK
Remarks on the algebra of N1NF relations
SIGACT-SIGMOD Symp. On Principle of Database Systems Los Angeles1982

[KAWA85] K. KAWAGOE
Modified dynamic hashing
Proc. ACM-SIGMOD Int. Conf. On Management of Data May 1985 p201-213

[KHOS86] S. KHOSHAFIAN, G. COPELAND
Object Identity
Proc. OOPSLA'86 Portland Oregon Setp 86

[KIM85] M. KIM
Parallel operation on magnetic disk storage devices: synchronised disk interleaving
Proceedings of the fourth international workshop on databases machines. Mars 1985

[KIM87] W. KIM, N. BALLOU, J. BANERJE, J. CHOU
Features of the ORION Object Oriented Database System
MCC Tech. Report Num ACA-ST-308-87 sept 87

[KIM88] W. KIM, H.T. CHOU, J. BANERJEE
Operations and implementation of complex objects
IEEE Transactions on Software Engineering Vol 14 Num. 7 july 88

[KING85] R. KING, D. Mc LEOD
Semantic database models Database Design
S.B. YAO Ed. Springer-Verlag 1985

[KLEI75] L. KLEINROCK
Queing systems Theory and applications
Ed Wiley 1975

[KNOT72] G. D. KNOTT
Hashing functions
Computer Journal Vol 18 Num. 3 1972

[KNUT73] D.E. DNUTH
The art of computer programming Vol 3 Searching and sorting
Addison-Wesley, Readding Mass

[KUPE85] G.M. KUPER, M.Y. VARDI
On the expressive power of the Logic Data Model
ACM SIGMOD Int. Conf. Austin Texas mai 85

[LARS78] P.A. LARSON
Dynamic hashing
Bit Vol 18 Num. 13 1978

[LARS85] P. A. LARSON, M.V. RAMAKRISHMA
External perfect hashing
Proc. ACM-SIGMOD Int. Conf. On Management of Data may 85 p190-199

[LECL88] C. LECLUSE, P.RICHARD, F. VELEZ
"O2, an object oriented data model"
ACM-SIGMOD Int. Conf Chicogo mai 1988

[LIVN87] M. LIVNY
Multi-disk management algorithms
ACM SIGMETRICS conference, Alberta Canada 1987

[LORI89] R. LORIE, J.J. DAUDENARDE, G. HALLMARK, J. STAMOS, H. YOUNG
Adding intra-transaction parallelism to an existing DBMS, early experience
IEEE data engineering vol 12 mars 1989 p2-8

[MAIE86] D. MAIER, J. STEIN
Development of an Object Oriented DBMS
Proc. OOPSLA'86 Portland Oregon Sept 86

[MARC77] S.T. MARCH, D.G. SEVERANCE
The determination of efficient record segmentations and blocking factor for shared files
ACM TODS Vol 2 Num. 3 1977 p279-296

[MARC84] S.T. MARCH, G.D. SCUDDER
On the selection of efficient record segmentation and backup statregies for large databases
ACM Transactions on database Systems Vol 9 Num. 3 sept 84 p409-438

[MCOR72] W.T. Mc CORMICK and al
Problem decomposition and data reorganization by a clustering technique
Operations Research Vol 20 Num. 5 sept-Oct 1972 p993-1009

[MINO86] M. MINOUX, G. BARTNIK
Graphes, algorithmes et logiciels
Dunod Informatique 1986

[MIRA86a] S. MIRANDA, J.M. BUSTA
L'art des bases de données. Tome1 Introduction aux bases de données
Editions Eyrolles 1986

[MIRA86b] S. MIRANDA, J.M. BUSTA
L'art des bases de données. Tome2 Les bases de données relationnelles
Editions Eyrolles 1986

[MISS83] M. MISSIKOF, M. TERRANOVA
The architecture of a Relational Database Computer known as DBMAC
in "Advanced Database Machine Archirecture" Prentice Hall 1983 p 87-130

[MRI72] MRI Systems Corporation
System 2000 publications
MRI Systems Corporation, Austin, TEXAS 1972

[MUKK87] R. MUKKAMALA
Design of partially replicated distributed database systems: an integrated methodology
Technical report 87-04 Dept of computer science University of IOWA juillet 1987

[MURA83] K. MURAKAMI et al
Relational Database Machine: first step to Knowledge Base Machine
ICOT Tech. Report TR-012 Japan may 83

[NAVA84] S. NAVATHE, S. CERI, G. WIEDERHOLD, J. DOU
Vertical partitioning algorithms for database design
ACM Transactions on Database Systems Vol 9 Num. 4 Dec 1984 p680-710

[NECH87] Ph. M. NECHES
The anatomy of a database computer system
Proc of the 2nd Int. Conf. on Supercomputing. 1987

[NICO89] J-Ch NICOLAS, B. TOURSEL
Parallel distribution of databases with complex objects
IFIP (W.G. 10.3) Working Conf. on Decentralized Systems Dec. 1989 LYON

[NICO89b] J-Ch. NICOLAS, B. TOURSEL
Une technique de hachage adaptée à la répartition d'une relation sur un réseau de processeurs
Publication Interne ERA 68 mars 89 LIFL USTL LILLE

[NICO89c] J-Ch NICOLAS
Description et analyse du langage FAD, un langage de programmation pour bases de données
Publication Interne ERA 69 LIFL USTL Lille Mars 89

[NICO90] J-Ch NICOLAS, B. TOURSEL
Une méthode de placements d'objets complexes pour maschines bases de données parallèle
Com. aux 3eme rencontres sur les architectures et algorithmes massivement parallèles.
CIRM Luminy Octobre 90

[NICO91] J-Ch NICOLAS, B. TOURSEL
Performances on databases machines: the data placement issue
9 th International Symposium on Applied Informatics 18-21 february 1991

[OKI84] OKI LTD
The relational database machine DB-1
OKI ltd documentation 1982

[ONTO88] ONTOLOGIC Inc.
VBASE for object Applications
Ontologic publication, Cambridge (Massachusetts) 1988

[OTOO85] E.J. OTOO
A multidimensional digital hashing scheme for file with composite keys
Proc. ACM-SIGMOD Int. Conf. On Management of Data May 1985 p214-229

[PATT87] D.A PATTERSON, G. GIBSON, R.H. KATZ
A case of Redundant Arrays of Inexpensive Disks (RAID)
Report Num. UCB/CSD87/391 Uni. of Berkeley California. Comp. Science Division

[PEPIN85] PEPIN
Introduction aux systèmes de gestion de bases de données
Informatique et entreprises Ed Eyrolles

[PIER89] P. PIERCE
A concurrent file system for a highly parallel mass storage subsystem
Intel Scientific Computer internal publication

[SALE84] K. SALEM, H. GARCIA-MOLINA
Disk stripping
Department of Electrical Enginnering and Computer Science Princeton University

[SANS90] J.P. SANSONNET
Cours de DEA Université de PARIS VI
Photocopies de transparents

[SCHK77] M. SCHKOLNICK
A clustering Algorithm for hierarchical structures
ACM Transactions on Database Systems Vol 2 Num. 1 p27-44 march 77

[SCHW83] H. SCHWEPPE et al
RDBM a Dedicated Multiprocessor system for Large Data Base Management
in "Advanced Database Machine Archirecture" Prentice Hall 1983 p36-86

[SPRU77] R.J. SPRUGNOLI
Perfect hashing functions: A single probe retrieving method for static sets
CACM Vol 20 Num. 11 1977 p841-850

[STON86a] M. STONEBRAKER
The case of shared nothing
IEEE Database Engineering Vol 9 Num 1 March 86

[STON86b] M. STONEBRAKER
The design of Postgres
SIGMOD Record Vol 15 Num 2 p340-355 june 86

[STON86c] M. STONEBRAKER
The Ingres paper: anatomy of a relational database system
Ad. Wesley Record March 86

[SU75] S.Y. SU, G.J. LIPOWSKI
CASSM: a cellular system for Very Large Data Bases
First VLDB conf. Boston 1975

[TAND87] The TANDEM Database group
Non stop SQL, a distributed, high performance, high availibility implementation of SQL
2nd Int. Workshop on High Perf. Trans. Systems Pacific-Grove 28-30 septembre 1987

[TERA84] J. SHEMER, P. NECHES
The genesis of a database computer
IEEE transactions on computers novembre 1984

[TOYO66] J. TOYODA, Y. TEZUKA, Y. KASHARA
Analysis of the address assignment problem for clustered keys
JACM Vol 13 Num. 4 p526-532 1966

[TSUR84] S. TSUR, C. ZANOLIO
On the implementation of GEM: Supporting a Semantic Data Model on Relationnal Back-End
ACM SIGMOD Int. Conf. Austin Texas mai 84

[VALD86] P. VALDURIEZ, S. KHOSHAFIAN, G. COPELAND
Implementation techniques of complex objects
Proccedings of the 12th Int. Conf. on Very Large Data Bases Kyoto August 86

[VALD87a] P. VALDURIEZ
Objets complexes dans les systèmes de bases de données relationnels
Technique et Science Informatique Vol 6 Num. 5 1987

[VALD87b] P.VALDURIEZ
Join Indices
ACM Transactions on Database Systems Vol 12 Num. 2 June 87 p218-246

[VALD88] P. VALDURIEZ, D. DANFORTH
Query optimization in FAD, a database programming language
MCC Tech. Report Num. ACA-ST-316-88 sept 88

[VERS86] J. VERSO
VERSO: a database machine based on NINF relations
Rapport de recherche Num. 523 INRIA Rocquencourt

[VRSA85] D. VRSALOVIC, E.F. GEHRINGER, R.R. SEGAL, D.P. SIEWICREK
The influence of parallel decomposition strategies on the performances of multiprocessor systems
IEEE/ACM Symposium on computer architecture BOSTON Juin 1985

[WOEL87] D. WOEL, W. KIM
Multimedia Information Management in an Object-Oriented Database System
Proc. of the 13th VLDB Conf. Brighton 1987

[WONG83] E. WONG, R.H. KATZ
Distributing a database for parallelism

[YAO85] S. B. YAO
Principles of database design Vol 1: Logical organization
Prentice Hall 1985
IEEE Trans. on Comp. Vol C28 Num. 6 June 79 p395-406

[ZANO83] C. ZANOLIO
The database language GEM (E/A)
Proc ACM/SIGMOD Int. Conf. on the management of data May 83

Résumé :

Nous nous intéressons au problème de la fragmentation des objets complexes d'une base de données et à la distribution des objets partiels obtenus sur une machine parallèle. Ce travail a comme cadre celui des Machines Bases de Données "sans-partage", sur lesquelles, les opérations sont exécutées là où les données se trouvent. Pour ce type de machine, la répartition de charge est donc fortement dépendante du placement des données. Le modèle de données considéré supporte les notions d'objets complexes, d'identité et de partage d'objets.

La fragmentation des objets est réalisée par une extension originale des concepts de fragmentation verticale. Cela nous permet d'une part, d'augmenter le parallélisme inter-requêtes en introduisant un parallélisme intra-objet et, d'autre part, de gérer de façon adaptée, les agrégations d'objets induites par le modèle de données.

La distribution des objets fragmentés (objets partiels) est réalisée par une technique particulière de fragmentation horizontale, qui assure, en relation avec la configuration matérielle disponible, la gestion de la localité entre les différents objets partiels d'un même objet. De plus, l'utilisation de fonctions de hachage simples et adaptables nous permet d'obtenir une distribution uniforme, source d'un taux de parallélisme intra-requête "optimal".

A la fois pour la fragmentation et pour la distribution des objets, nous utilisons des méthodes quantitatives basées sur l'analyse des principales requêtes soumises au système. Néanmoins, les contraintes physiques sont aussi prises en compte à différents niveaux. En conséquence, le placement obtenu peut être modifié, aussi bien pour des raisons logiques, relatives à l'utilisation des données, que physiques.

Ainsi, nous proposons avec cette approche une solution pour la résolution du délicat problème du placement des données sur une Machine Bases de Données "sans-partage". L'utilisation d'une telle solution s'oriente vers un type de machine où le placement des données serait dirigé par leur utilisation et par la configuration matérielle.

Mots-clés :

Bases de données, machines parallèles, placement de données,
fragmentation verticale, fragmentation horizontale, distribution,
objets complexes.