Utilisation des vues matérialisées, des index et de la fragmentation dans la conception logique et physique d'un entrepôt de données

Ladjel Bellatreche

N d'ordre: D.U. 1255

EDSPIC : 255

UNIVERSITÉ DE CLERMONT-FERRAND II

ÉCOLE DOCTORALE

SCIENCES POUR L'INGÉNIEUR DE CLERMONT - FERRAND

THÈSE

présentée par

Ladjel BELLATRECHE

pour obtenir le grade de

DOCTEUR D'UNIVERSITÉ

Spécialité: INFORMATIQUE

Utilisation des Vues Matérialisées, des Index et de la Fragmentation dans la Conception Logique et Physique d'un Entrepôt de Données

Soutenue publiquement le 18 Décembre 2000 devant le jury:

M.	Djamel	ZIGHED	Président
M.	Henri	BRIAND	Rapporteur
M.	Jacques	KOULOUMDJIAN	Rapporteur
M.	Michel	SCHNEIDER	Directeur de thèse
M.	Michel	SIMONET	Co-Directeur de thèse

Remerciements

Je tiens à remercier toutes les personnes qui ont contribué de manière directe ou indirecte à l'aboutissement de ce travail :

En premier lieu, je remercie vivement Monsieur Michel Schneider qui m'a permis d'effectuer cette thèse sous sa direction. J'ai particulièrement apprécié son enthousiasme, sa disponibilité et sa gentillesse. Son sens critique et son optimisme ont fait de cette thèse un travail motivant.

Je tiens à remercier Michel Simonet et Ana Simonet Directeur et Directeur adjoint de l'équipe OSIRIS du laboratoire TIMC-IMAG, pour m'avoir permis de trouver un bon créneau de recherche et pour m'avoir encadré tout au long de cette thèse.

Je remercie Monsieur Jacques Demongeot, Professeur à l'université Joseph Fourier, Directeur du laboratoire TIMC-IMAG, pour m'avoir facilité mon séjour à Hong Kong.

Je remercie également Kamalakar Karlapalem qui m'a accepté dans son équipe de recherche au Department of Computer Science, Hong Kong University of Science and Technology durant deux années avec un financement.

Je remercie Djamel ZIGHED, qui me fait l'honneur de présider ce jury.

Je remercie Messieurs Jacques KOULOUMDJIAN et Henri BRIAND d'avoir accepté d'être les rapporteurs de ma thèse. Leurs lectures et leurs remarques m'ont beaucoup aidé à finaliser ce document.

Je remercie Messieurs Mukesh Mohania, Qing Li, Gopal K. Bassak, Dimitris Papadias, Fred Lochovsky, et Zohra Bellahsène avec lesquels j'ai pris grand plaisir à travailler.

Je remercie mes collègues du laboratoire TIMC et tous les membres de l'équipe Osiris pour leur présence et leur soutien cordial.

Des pensées particulières vont à tous mes amis qui m'ont soutenu durant ces années.

Mille merci à mon père, ma mère, mes frères, mes soeurs, ma tante, son mari, mes cousins pour le soutien et l'aide qu'ils m'ont apportés durant mes études et sans qui ce travail n'aurait pas pu avoir lieu.

Enfin, une pensée très forte à ma femme et ma fille qui m'ont soutenu durant les moments difficiles.

Que tous ceux et celles que j'ai oubliés n'en recoivent pas moins ma gratitude.

Résumé

Un entrepôt de données est une collection de données orientées sujet, intégrées, non volatiles et historisées, organisées pour supporter un processus d'aide à la décision. Typiquement ce processus est mené par l'intermédiaire de requêtes de type OLAP (On-Line Analytical processing). Ces requêtes sont généralement complexes car elles contiennent de nombreuses opérations de jointure et de regroupement et induisent des temps de réponse très élevés.

Dans ce contexte, nos travaux s'intéressent à diverses techniques d'amélioration des performances des entrepôts de données pour favoriser au mieux les requêtes. Ils interpellent deux niveaux de la conception des entrepôts : le niveau logique et le niveau physique. Au niveau logique, nous suggérons une méthodologie de fragmentation des structures de données de l'entrepôt. Au niveau physique, nous nous intéressons (1) à la définition et à la sélection d'index de jointure en présence des vues matérialisées et (2) à la distribution de l'espace disque entre les vues matérialisées et les index.

En ce qui concerne l'indexation, nous proposons une nouvelle technique d'indexation de jointure appelée index de graphe de jointure. Ces index permettent de réduire considérablement le coût d'exécution des requêtes. Une stratégie d'exécution des requêtes en présence de ces index est décrite, et un modèle de coût évaluant le coût d'exécution d'un ensemble de requêtes est développé. Nous formulons ensuite le problème de sélection d'index de jointure en présence d'une contrainte d'espace disque et nous proposons trois algorithmes de résolution optimaux ou quasi-optimaux (un algorithme exhaustif et deux algorithmes gloutons).

Le problème de la distribution de l'espace disque entre les vues matérialisées et les index a été posé assez récemment et peu de travaux l'ont interpellé. Dans ce mémoire, nous formulons ce problème dans le cas statique et dans le cas dynamique. Nous proposons un algorithme approché de résolution basé sur l'interaction entre deux agents, l'un opérant pour le compte des vues et l'autre pour le compte des index.

L'intérêt de la fragmentation est bien connu dans les bases de données relationnelles. Pour les entrepôts, nous justifions que la fragmentation horizontale apparaît plus spécialement appropriée et nous proposons une méthodologie de fragmentation horizontale pour décomposer un schéma d'entrepôt en étoile. Nous montrons ensuite que les algorithmes de fragmentation basés uniquement sur les fréquences d'accès des requêtes ne sont pas les plus intéressants. Nous suggérons deux nouveaux types d'algorithmes. Le premier type est dirigé par les affinités entre prédicats et le deuxième est dirigé par un modèle de coût. Pour cette dernière catégorie nous comparons un algorithme exhaustif recensant tous les schémas de fragmentation et un algorithme approximatif.

Mots-Clefs : Entrepôt de données, sélection des vues matérialisées, sélection d'index, interaction entre la sélection des vues et la sélection des index, distribution de l'espace entre les vues et les index, fragmentation, modèle de coût.

Abstract

A data warehouse is a subject-oriented, integrated, time-variant, non-volatile collection of data in support of a decision process. This process is usually performed with OLAP (On-Line Analytical processing) queries that aggregate, filter, and join the data in a variety of ways. Because the queries are often complex and the warehouse database is often very large, processing queries rapidly is a critical issue in the data warehousing environment.

Our work focuses on several optimization techniques to enhance query execution in data warehousing environments. These techniques concern the logical and the physical design levels of the data warehouse. At the logical design level, we suggest a fragmentation methodology of the data warehouse schema. At the physical design level, we are interested in (1) the definition and selection of join indexes in the presence of materialized views and (2) the problem of space distribution among views and indexes.

Concerning indexing, we propose a new join indexing technique called graph join index. These indexes reduce significantly the query processing cost. A query execution strategy using these indexes is described and a cost model to compute the cost of a set of queries is developed. We formulate the graph join index selection problem, and present three algorithms (one exhaustive and two greedy) which can provide good query performance under limited storage space for the indexes.

The space distribution problem among views and indexes was only recently introduced and therefore little work has been done on the subject. In this thesis, we formulate this problem in the static and dynamic cases. We propose an iterative approach to automatically distribute the storage space among views and indexes. This approach considers the interaction between indexes and views. Two agents (one for views and another for indexes) cater for this interaction. We extend this approach to handle changes dynamically.

Partitioning is a well known logical database design technique which was first applied to relational databases. We propound horizontal fragmentation as a better adapted technique for ROLAP data warehouses. To do this, we develop a fragmentation methodology to decompose the star schema of a data warehouse. Furthermore, we show that existing fragmentation algorithms are not suitable because of their complexity. We develop two families of algorithms: the first is based on affinities between predicates and the second is driven by a cost model. For the latter, we compare an exhaustive algorithm which enumerates all fragmentation schemas with an approximative algorithm.

Keywords: Data warehouse, materialized view selection, index selection, index, interaction between index and view selection, space distribution between views and indexes, fragmentation, cost model.

Table des matières

Liste des figures

Liste des tables

Liste des algorithmes

Notations

Description des symboles utilisés dans la thèse.

Opérations relationnelles:

$\bowtie$	L'opération de jointure
$\ltimes$	L'opération de semi-jointure
σ	L'opération de sélection
π	L'opération de projection

Notations:

$\|\|T\|\|$	Cardinalité de la table T
$\|T\|$	Nombre de page de la table T
PSV	Problème de sélection de vues
PSI	Problème de sélection d'index
ASV	Algorithme de sélection des vues
ASI	Algorithme de sélection des index
IGJ	Index de graphe de jointure
IJE	Index de jointure en étoile
$PMEV$	Plan multiple d'exécution des vues
$AGS1$	Algorithme glouton sélectionnant un seul IGJ
$AGSK$	Algorithme glouton sélectionnant K IGJs
CT	Coût total d'exécution des requêtes
CT_G	Coût total d'exécution des requêtes en utilisant l'index G
S	Espace global
S^V	Espace réservé aux vues matérialisées
S^I	Espace réservé aux index
W	Matrice de réécriture
RI	Matrice requête-index
$VPPT$	Vue de plus petite taille
$VMFU$	Vue la moins fréquemment utilisée
$IMFU$	Index le moins fréquemment utilisé
PGI	Plus grand index
SDS	Sous domaine stable
$Dist$	Distance de Hamming

Logique des prédicats :

$\vee$	disjonction
$\wedge$	conjonction
$\Longrightarrow$	implication

Chapitre 1

Introduction générale

L'informatique a connu et connaît aujourd'hui encore un essor spectaculaire. Cette évolution est due aux nombreux services qu'elle offre aux usagers et aux entreprises. Grâce à l'informatique, ces dernières ont pu améliorer leur contrôle sur leurs informations et leurs processus de traitement de l'information. Elles ont également augmenté leur productivité en gérant efficacement et rapidement les diverses informations relatives à la production, au marketing et aux ressources humaines. Les aspects transactionnels sont au coeur de ces systèmes informatiques développés depuis un demi-siècle, et de vastes volumes de données ont été constitués. Ces données représentent maintenant une véritable mine de connaissances sur les produits, les clients, les fournisseurs et les opérations de l'entreprise. Cependant, la nature éphémère et hétérogène des données opérationnelles amassées par ces systèmes prive l'entreprise d'une grande partie des connaissances. Pour accroître ses performances, l'entreprise doit mettre en place de nouvelles technologies décisionnelles pour obtenir une meilleure compréhension des informations disponibles et définir des indicateurs pertinents pour faciliter les prises de décision opérationnelles.

Une solution consiste à utiliser des médiateurs permettant d'accéder à toutes les données de l'entreprise depuis un poste client [GO96]. Cependant cette approche est limitée par la taille du poste client et par l'accès aux données distribuées nécessaires. Une autre approche, l'entrepôt de données (data warehouse) consiste à regrouper les informations disséminées, après les avoir prétraitées, au sein d'une base de synthèse unique, intégrée par sujet. En extrayant par **des requêtes** des parties de cette base, il devient possible pour les dirigeants de l'entreprise **d'analyser** les données et de prendre les **meilleures décisions rapidement** [GO96].

Les exemples les plus parlants pour ce genre d'applications se trouvent dans le domaine de la grande distribution, les ventes par correspondance et les compagnies téléphoniques (comme AT&T). Il est important dans ces domaines de regrouper les informations sur les ventes pour déterminer les produits à succès, mieux suivre les tendances, détecter les habitudes d'achat des clients et leurs préférences par secteur géographique. De telles applications nécessitent un accès interactif à des données issues d'applications variées, intégrant l'historique, à partir de logiciels d'aide à la décision.

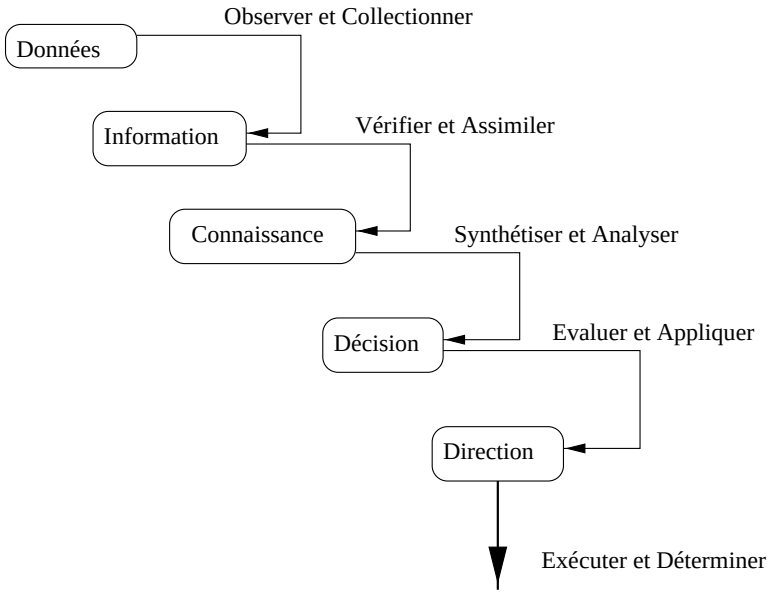


FIG. 1.1: L'organigramme d'analyse des données dans une entreprise

1.1 Entrepôt de données et data mining

C'est une nécessité pour toute entreprise de recueillir des informations variées sur le comportement de leurs clients (préférences, habitudes) afin de mener diverses analyses et d'en déduire des modalités d'actions (voir Figure 1.1). L'analyse par exemple des transactions de vente d'un supermarché permettra d'étudier les habitudes des clients, en fonction de quoi il sera possible de réorganiser les rayons afin d'améliorer les ventes.

L'extraction de connaissances dans les bases de données (data mining) [1] désigne le processus non-trivial d'extraction d'informations implicites, précédemment inconnues et potentiellement utiles concernant les données stockées dans les bases de données [PS91]. Les travaux en ce domaine sont motivés par l'évolution très rapide des techniques de génération (telles que les techniques de lecture des codes barres des articles achetés en supermarché) et de stockage de données (augmentation de la capacité et diminution des coûts des disques) qui ont permis la création par de nombreuses entreprises de bases de données volumineuses (entrepôts) concernant leurs activités commerciales. L'extraction des données d'un entrepôt de données se fait à l'aide des requêtes ad-hoc, complexes, exigeant beaucoup d'agrégations et de jointures. Pour conclure, nous pouvons dire que l'entrepôt de données constitue une source indispensable pour l'extraction des connaissances.

1.2 OLTP et OLAP

Les bases de données des systèmes existants, de type On-line Transaction Processing (OLTP) (appelés systèmes transactionnels), permettent d'enregistrer, de chercher et de gérer des données variées et de réaliser des transactions sur ces données

[1] Le terme data mining est fréquemment utilisé comme synonyme de Knowledge Discovery in Databases, introduit par Shapiro en 1989.

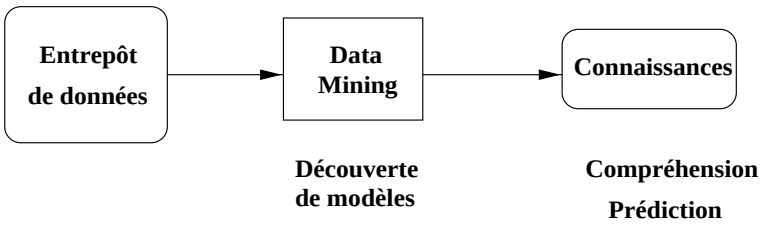


FIG. 1.2: Le data mining et les entrepôts de données

(lectures, mises à jour, etc). Ces systèmes ont bien répondu aux besoins en gestion transactionnelle de l'information. Les bases de données de type OLTP sont généralement créées pour une large communauté d'utilisateurs. Chaque utilisateur connaît ses besoins, et avec une seule transaction il accède à un petit nombre de n-uplets. L'objectif principal de ces bases de données est de maximiser le débit des transactions et de minimiser les conflits concurrentiels.

Avec l'évolution de la société, les besoins ont évolué: l'accroissement des volumes de données a induit des changements dans la vision de l'information et suscité une sophistication des traitements [Mar98]. Maintenant, les entreprises souhaitent avoir un point de vue global pour manipuler des sources de données hétérogènes, sur lesquelles des traitements analytiques sont associés aux traitements transactionnels. Cependant, les systèmes transactionnels sont limités au niveau des fonctions d'aide à la décision [Inm92] et ne peuvent pas extraire toutes les informations nécessaires pour prendre une décision rapide et intelligente permettant une gestion efficace d'une organisation. Par exemple, il est difficile pour les systèmes transactionnels de répondre à la question suivante: *quels sont les types de fournisseurs du produit 'x' à Grenoble en 1997 et comment se différencient-ils de ceux de l'année dernière?*

L'objectif principal des entrepôts de données est d'être une plate-forme commune pour supporter les opérations de type OLAP (*On-Line Analytical Processing*). Codd et al. [CCS93] ont décrit ces opérations comme *des analyses dynamiques nécessaires pour la création, la manipulation, l'animation et la synthèse des informations reflétant la vision de l'analyste.* Les entrepôts de données sont souvent conçus pour les opérations (ou requêtes) de type OLAP. Ces requêtes accèdent généralement à un *grand volume de données* et effectuent beaucoup d'opérations de *jointure* et *d'agrégation.* Les entrepôts de données opèrent dans un environnement où les requêtes de lecture sont plus fréquentes que les requêtes d'écriture (query-centric). Dans les entrepôts le *temps de réponse* est le facteur de performance primordial à satisfaire.

Le data mining a une forte relation avec OLAP, dont les objectifs sont similaires, mais il faut cependant souligner une différence essentielle. OLAP traite les données, alors que le data mining vise également à extraire des connaissances à partir de ces données (règles d'association par exemple [Pas00]).

La Table 1.1 résume les différences principales entre les bases de données de type OLTP et les entrepôts de données [Men97].

Caractéristique	Base de données	Entrepôt de données
Données	Courantes	Historiques
Usage	Support de l'opération de l'entreprise	Support de l'analyse de l'entreprise
Unité de travail	Transaction	Requête
Nombre de données accédées	Dizaines	Millions
Mode d'accès	Lecture/écriture	Lecture
Requête	Simple Prédéterminée	Complexe Ad-hoc
Type d'utilisateur	Employé	Décideur
Nombre d'utilisateurs	Mille	Cent

TAB. 1.1: Base de données Vs Entrepôt de données

1.3 Les caractéristiques des entrepôts de données

Bill Inmon dans son ouvrage de référence "Using the Data Warehouse" [Inm92] définit l'entrepôt de la façon suivante : "l'entrepôt de données est une collection de données orientées sujet, intégrées, non volatiles et historisées, organisées pour le support d'un processus d'aide à la décision". Nous en précisons les caractéristiques ci-dessous.

Les données d'un entrepôt de données possèdent les caractéristiques suivantes:

- **Orientation sujet**: Les données d'un entrepôt s'organisent par sujets ou thèmes. Pour ce qui est du domaine commercial d'une entreprise par exemple, nous aurons des données sur la production, les ventes, le marketing et la finance. Cette organisation permet de rassembler toutes les données pertinentes et nécessaires pour toutes les analyses relatives à ce thème.
- **Intégration**: Les données d'un entrepôt de données sont le résultat de l'intégration de données en provenance de plusieurs sources. Ainsi, toutes les données nécessaires pour réaliser une analyse particulière se trouvent dans l'entrepôt de données. L'intégration est le résultat d'un processus qui peut devenir très complexe compte tenu de l'hétérogénéité des sources.
- **Historique**: Les données d'un entrepôt de données représentent l'activité d'une entreprise pendant une longue période, d'où l'importance de gérer les différentes valeurs qu'une donnée prend au cours du temps. Cette caractéristique donne la possibilité de suivre une donnée dans le temps pour analyser ses variations.
- **Non-Volatilité**: Les données chargées dans l'entrepôt de données sont surtout utilisées en interrogation et ne peuvent pas être modifiées, sauf cas particulier (rafraîchissement par exemple).

1.4 Architecture d'un entrepôt

La Figure 1.3 montre l'architecture conceptuelle d'un entrepôt de données. L'entrepôt stocke des données, brutes ou modifiées provenant des sources d'informations

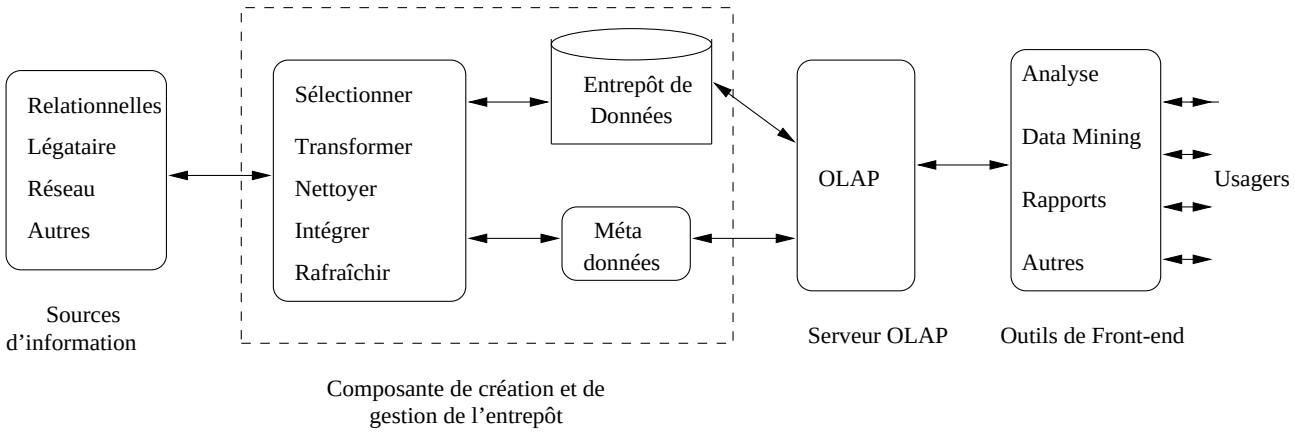


FIG. 1.3: Architecture conceptuelle d'un entrepôt de données

(qui peuvent être internes ou externes, et souvent distribuées, autonomes et hétérogènes). Avant d'être chargées dans l'entrepôt, les données sélectionnées doivent être extraites des sources et soigneusement épurées, pour éliminer les erreurs et réconcilier les différences sémantiques (ce processus est appelé nettoyage). Une fois ces données nettoyées, elles seront intégrées dans l'entrepôt, qui contient des données détaillées, des vues matérialisées et des données multidimensionnelles. Lors de changements dans les données des sources, ces derniers sont propagés vers l'entrepôt. Ce processus est connu sous le nom de rafraîchissement [WB97]. Les méta-données contiennent des informations concernant la création, la gestion, et l'usage de l'entrepôt. Les méta-données sont stockées dans un répertoire différent de celui de l'entrepôt. Elles sont considérées comme un pont entre les utilisateurs et l'entrepôt. L'entrepôt est accédé par le serveur OLAP afin de présenter les données sous forme multidimensionnelle aux clients pour des besoins d'analyses et des besoins informationnels (data mining, rapport, etc.). Le serveur OLAP interprète les requêtes des clients et les convertit en requêtes d'accès à l'entrepôt ou aux sources opérationnelles. Finalement, le serveur OLAP fournit des vues multidimensionnelles des données aux outils de front end, et ces derniers formatent les données conformément aux besoins des usagers.

1.5 Problématique abordée et résultats obtenus

Comme nous l'avons déjà mentionné, la taille des entrepôts de données peut être très grande. Par exemple, *Sagent Technology*, dont les produits sont utilisés pour construire des entrepôts de données, est régulièrement confronté à des volumes de 100 Gigabytes ou plus, et la chaîne de supermarchés *Wal*Mart* aux U.S.A. gère un entrepôt (créé en 1988) de 2,7 Tera octets sur une machine *TerraData* avec 365 processeurs d'accès [MLWZ98]. Le processus d'analyse de données s'appuie souvent sur des requêtes qui regroupent et filtrent des données de différentes formes. Compte tenu de la nature interactive des entrepôts de données, la nécessité d'avoir un temps de réponse rapide est un objectif critique [CY99a, OQ97]. Des délais trop longs sont inacceptables dans la plupart des systèmes décisionnels, puisqu'ils peuvent diminuer la productivité. L'exigence courante est un temps de réponse ne dépassant pas quelques secondes ou quelques minutes [Gup99]. Sans *technique d'optimisation de*

requêtes, l'interrogation d'un entrepôt de données serait complexe, et le traitement de ces requêtes pourrait prendre des heures, voire des jours [OQ97].

Les travaux concernant l'optimisation des performances des requêtes décisionnelles sont principalement basés sur des techniques éprouvées, et héritées des bases de données relationnelles/objets. Les éléments suivants influencent notablement les performances des requêtes (nous y reviendrons plus en détail dans le Chapitre 2):

- les vues matérialisées (sélection et maintenance),
- le calcul du cube de données,
- les techniques d'indexation (sélection et maintenance),
- la fragmentation,
- le groupement des n-uplets sur le disque (clustering).

Nous abordons dans cette thèse trois types de problèmes liés à la performance des requêtes: le premier concerne la *définition des index de jointure pour les vues matérialisées* à un niveau physique, le second est relatif à la *distribution de l'espace entre les vues matérialisées et les index*, et le dernier concerne la *fragmentation des données* d'un schéma d'un entrepôt de données à un niveau logique. Les deux premiers problèmes n'ont pas encore été traités dans la littérature, et, pour le dernier, nous proposons des solutions originales.
Nous résumons ci-dessous la problématique de recherche posée par chacun de ces problèmes.

1.5.1 Index de jointure pour les vues matérialisées

Une vue est une requête nommée. Une vue matérialisée est une table contenant les résultats d'une requête. Certaines requêtes nécessitent seulement l'accès à la vue matérialisée [GM95] et sont ainsi exécutées plus rapidement. L'entrepôt de données dans *the Mervyn's department store-chain* a 2400 vues matérialisées pour accélérer l'exécution des requêtes [Gup99]. Les vues matérialisées peuvent être utilisées pour satisfaire plusieurs besoins, comme l'amélioration de la performance des requêtes en réduisant le coût des opérations de jointure, ou la fourniture des données dupliquées. La sélection d'un ensemble de vues à matérialiser garantissant une meilleure performance des requêtes décisionnelles constitue un problème majeur. Ce problème est connu sous le nom de **problème de sélection de vues à matérialiser (PSV)**. Lorsqu'une vue est matérialisée, elle doit être mise à jour périodiquement pour tenir compte des changements pouvant survenir dans les sources de données. Ce problème est connu sous le nom de **maintenance de vues matérialisées**.

Le PSV consiste donc à sélectionner des vues de telle manière que les coûts d'évaluation des requêtes et de maintenance des vues soient les moins élevés possibles et sous une contrainte de limitation de ressources (contrainte d'espace par exemple).

Une fois les vues matérialisées sélectionnées, toutes les requêtes décisionnelles sont écrites en fonction de ces vues (voir par exemple les travaux de Divesh et al. [SDJL96]). Notons que les vues matérialisées peuvent être jointes entre elles ou avec d'autres tables du schéma. La jointure est très coûteuse, en particulier lorsque la taille des tables est importante par rapport à la mémoire centrale disponible [LR98]. De nombreuses techniques ont été proposées afin d'accélérer cette opération, en

particulier par l'intermédiaire d'index de jointure [Val87]. L'index de jointure en étoile [Sys97] a été utilisé dans les entrepôts de données modélisés par un schéma en étoile. Il consiste à installer un index pour faciliter la jointure entre une table des faits et plusieurs tables de dimensions. Il a été prouvé que cet index est la structure d'accès la plus efficace pour accélérer des jointures définies sur la table des faits et les tables de dimensions [Sys97, OQ97].

Le recours aux vues matérialisées pose un nouveau problème d'indexation, celui de savoir *comment indexer ces vues.* Une vue matérialisée peut être indexée en utilisant les techniques d'indexation des tables. Cependant, il n'existe pas de travaux concrets sur la définition et la sélection des index de jointure pour les vues matérialisées.

Notre contribution consiste à présenter une nouvelle technique d'indexation de jointure appelée *index de jointure de graphe* pour les vues matérialisées. Cet index pourra être construit en utilisant les vues matérialisées, les tables de dimensions, et la table des faits. Nous proposerons une stratégie d'exécution de requêtes en présence de ces index. Nous formulerons le problème de sélection d'index de jointure pour les vues matérialisées comme un problème d'optimisation. Nous proposerons alors trois algorithmes pour déterminer la solution optimale ou une solution quasi-optimale.

1.5.2 La répartition des ressources entre les index et les vues

La sélection des vues matérialisées se fait durant la phase de conception logique de l'entrepôt. Au niveau de la conception physique, l'administrateur de l'entrepôt doit sélectionner des index afin d'accélérer l'exécution des requêtes. Ce problème est connu sous le nom de **problème de sélection d'index (PSI)**. Comme pour les vues matérialisées, les index doivent être mis à jour périodiquement pour refléter les changements pouvant survenir dans les sources de données. Ce problème est connu sous le nom de **maintenance des index**.

Etant donné un ensemble de requêtes décisionnelles et une ressource donnée (comme l'espace disque, le coût de maintenance, et le coût de calcul), le PSI consiste à sélectionner un ensemble d'index afin de minimiser le coût d'évaluation des requêtes et de maintenance des index. Dans cette thèse nous nous intéressons à la ressource représentant l'espace disque.

De nombreux travaux ont été réalisés pour résoudre le PSV [Gup97, KR99, RSS96, TS97, YKL97] et le PSI [CN98, CN99, FST88, LQA97]. Ainsi ont été développés des algorithmes générant un ensemble de vues matérialisées et un ensemble d'index optimal ou quasi-optimal. Les vues et les index se définissent comme des structures physiques pouvant être utilisées pour accélérer les performances des requêtes. Les vues et les index présentent des similitudes. Ce sont des structures **redondantes** et **compétentes** pour la même ressource (*l'espace disque*). Tous deux peuvent entraîner des **surcharges** causées par des opérations de mise à jour, et demandent un **temps de calcul élevé**. Une vue matérialisée est stockée sous forme de table relationnelle dans le contexte ROLAP, et son indexation peut améliorer la performance des requêtes accédant à cette vue. La présence d'index peut rendre les vues matérialisées plus attractives et vice versa. Mais la plupart des travaux ne considèrent pas l'interaction entre ces deux problèmes.

La non considération de cette interaction, pose le problème de la gestion des

ressources (espace disque, coût de maintenance, etc.). Par exemple, si nous supposons que le PSV et le PSI sont contraints par l'espace, l'ignorance de l'interaction pose le problème de distribuer l'espace entre les vues et les index. La tâche de répartition de l'espace disque entre les vues matérialisées et les index dans l'objectif d'améliorer les performances des requêtes est difficile pour l'administrateur de l'entrepôt. Cette difficulté est due à plusieurs facteurs:

- La nécessité d'avoir une métrique pour décider de la distribution d'espace entre les vues et les index.
- L'interdépendance mutuelle entre les deux problèmes.
- La nécessité de tenir compte des répercussions des opérations de mise à jour sur les vues matérialisées et les index [HMA99]. Ainsi, les tailles des vues et des index peuvent augmenter ou diminuer. Il est donc nécessaire de redistribuer l'espace entre les vues et les index afin de garantir une meilleure performance.
- La nécessité de tenir compte de l'évolution des fréquences des requêtes qui peut modifier l'intérêt de certaines vues ou de certains index et par conséquent entraîner une redistribution de l'espace.

Ainsi, le problème de répartition de l'espace disque entre les vues matérialisées et les index doit être une partie intégrante du problème de sélection de vues et d'index. Ce problème a été identifié plus récemment [Bel00a] et n'a pas encore reçu de solution définitive.

Dans cette thèse, nous répondrons à la question suivante:
comment distribuer/redistribuer d'une manière automatique l'espace disque entre les vues matérialisées et les index afin de garantir une meilleure performance d'exécution d'un ensemble de requêtes décisionnelles?

1.5.3 La fragmentation

La fragmentation est une technique de conception logique introduite dans les années 80 dans les bases de données réparties [CNP82, CP84, OV91]. Étant donné une table relationnelle, la fragmentation consiste à partitionner cette table horizontalement en fonction de ses n-uplets, ou verticalement en fonction de ses attributs, de façon à réduire le nombre des accès nécessaires pour le traitement de certaines requêtes.

Dans cette thèse, nous nous intéressons seulement à la fragmentation horizontale, qui est la mieux adaptée aux entrepôts de données.

Tout d'abord, nous proposerons une méthodologie pour utiliser la fragmentation horizontale dans les entrepôts de données modélisés avec un schéma en étoile, dans le but de réduire le coût total d'exécution des requêtes. Cette méthodologie consiste à fragmenter les tables de dimensions en utilisant la fragmentation primaire (en utilisant un algorithme parmi les algorithmes disponibles dans les bases de données relationnelles), et la table des faits en utilisant la fragmentation dérivée. Durant le processus de fragmentation, nous avons remarqué que le choix des tables de dimensions utilisées pour fragmenter la table des faits a une incidence forte sur les performances. Nous développerons un algorithme "glouton" pour sélectionner les "meilleures" tables de dimensions. Puis nous proposerons un modèle de coût analy-

tique pour évaluer le coût d'exécution d'un ensemble de requêtes sur un schéma en étoile fragmenté.

Etant donné que notre objectif est d'améliorer la performance des requêtes décisionnelles, nous développerons une autre famille d'algorithmes qui comblent les lacunes des algorithmes existants. Ces algorithmes sont dirigés par un modèle de coût (cost-driven algorithms). Leur principale caractéristique est la garantie d'obtenir de bons schémas de fragmentation.

1.6 Organisation de la thèse

Cette thèse est organisée autour de six chapitres principaux.

Le chapitre 2 est un état de l'art original portant sur les problèmes d'optimisation des requêtes dans les bases de données et en particulier dans les entrepôts de données. La contribution principale de cette synthèse est de proposer un recensement et une classification de ces problèmes et de leurs algorithmes.

Le chapitre 3 explicite les limites des index de jointure existants (par exemple l'index de jointure en étoile) dans les entrepôts de données, pour indexer les vues matérialisées. Il présente également une nouvelle technique d'indexation des vues matérialisées.

Le chapitre 4 aborde le problème d'interaction entre les vues matérialisées et les index dans les entrepôts de données. Il présente deux algorithmes pour distribuer l'espace entre les vues et les index. Le premier algorithme est statique et le deuxième est dynamique.

Le chapitre 5 traite le problème de la fragmentation dans les entrepôts de données et discute des différents facteurs à prendre en compte dans les algorithmes de fragmentation. Il présente également une méthodologie pour la fragmentation dans les entrepôts de données en adaptant les algorithmes utilisés dans les bases de données relationnelles.

Le chapitre 6 décrit les principes de fragmentation dans les bases de données. Il présente les différents algorithmes de fragmentation proposés dans les bases de données relationnelles et objets et une classification de ces algorithmes. Deux nouveaux algorithmes de fragmentation sont proposés.

Le chapitre 7 présente les conclusions générales de ce travail et esquisse diverses perspectives.

Chapitre 2

État de l'art

2.1 Introduction

Depuis quelques années, les entrepôts de données ont pris une place importante dans les préoccupations des utilisateurs des bases de données. Le marché estimé a connu une croissance énorme [GGT96], et de nombreux projets ont été développés au sein des universités. Citons par exemple le projet WHIPS de l'université de Stanford, le projet H2O de l'université du Colorado en collaboration avec la Southern California University, et le projet ADMS des universités du Maryland, de Stanford et du Queensland.

L'idée sous-jacente à la mise en oeuvre d'un entrepôt est de fournir un accès permanent aux données même lorsque les bases de données individuelles sont inaccessibles, et de réduire les accès distants aux systèmes gérant les données d'origine. Les entrepôts de données sont dédiés aux applications d'analyse et de prise de décision. Le processus d'analyse est réalisé à l'aide de requêtes complexes comportant de multiples jointures et des opérations d'agrégation sur des tables volumineuses. Les performances de ces requêtes dépendent directement de l'usage qui est fait de la mémoire secondaire. En effet, chaque entrée-sortie sur disque nécessitant jusqu'à une dizaine de milli-secondes [Dar99], l'accès à la mémoire secondaire constitue de ce fait un véritable goulot d'étranglement. L'administrateur, dans le but de minimiser le coût d'exécution de ces requêtes, sélectionne un ensemble de *vues matérialisées* et un ensemble *d'index*. Cette sélection diminue le coût des requêtes, mais entraîne un autre problème: les tables, les vues matérialisées et les index occupent une place très importante, et en conséquence ils *ne peuvent pas être stockés en totalité* dans la mémoire centrale. Dans un tel environnement, le nombre des entrées-sorties peut être grand si de bonnes techniques d'optimisation ne sont pas mises en oeuvre.

Les entrepôts de données ont d'abord été abordés par les industriels, mais par la suite les chercheurs se sont également penchés sur ce concept. Afin d'atteindre un niveau de performance acceptable, les deux communautés ont développé des techniques d'optimisation des requêtes au niveau de chaque phase de la conception d'un entrepôt (phase *conceptuelle*, phase *logique*, phase *physique*). Il existe cependant beaucoup plus de travaux concernant la phase de conception logique (la sélection et la maintenance des vues matérialisées, etc.) et la phase de conception physique de l'entrepôt (la sélection des index, les techniques d'indexation, les regroupements, etc.) comme nous le verrons tout au long de ce chapitre. Ce dernier est divisé en

quatre sections:

Modèle de données pour les entrepôts: L'entrepôt de données est typiquement modélisé par des modèles multidimensionnels (encore appelés cubes de données). Ces modèles sont appropriés pour les applications OLAP [AGS97, MSRK00]. En fonction de la manière dont le cube est stocké, deux approches existent pour construire des systèmes multidimensionnels: Relational OLAP (ROLAP), and Multidimensional OLAP (MOLAP)

La section 2.2 décrit en détail ces modèles multidimensionnels et leur schéma d'implémentation.

Les vues matérialisées: Un modèle multidimensionnel de données est stocké dans l'entrepôt comme un ensemble de vues matérialisées sur les données de bases. Les vues matérialisées permettent d'offrir des réponses rapides pour des requêtes, mais introduisent le problème de leur mise à jour. Pour des raisons d'efficacité les mises à jour sont souvent effectuées en utilisant des techniques incrémentales. De nombreux travaux de recherche ont été développés pour la maintenance des vues dans les bases de données traditionnelles [BLT86, CW81, LHM+86].

La section 2.3, définit le concept de vue matérialisée, présente le problème de sélection des vues matérialisées pour satisfaire les requêtes, et discute les techniques de mise à jour.

Les index: La matérialisation des vues est une approche embarrassante du fait qu'elle nécessite une anticipation des requêtes à matérialiser. Or, les requêtes dans les environnements des entrepôts sont souvent ad-hoc et ne peuvent pas toujours être anticipées. En effet, un utilisateur peut poser une requête sur des dimensions qui ne sont pas matérialisées dans une vue. En conséquence, la stratégie de précalcul est limitée. Une traduction effective des cubes de données en schémas relationnels (schémas en étoile) exige des méthodes d'indexation compte tenu du nombre important d'opérations de sélection et de jointure. Les requêtes qui ne sont pas précalculées doivent être élaborées à *la volée*. Cela nécessite des structures d'accès rapides pour conserver l'intérêt d'une analyse on-line.

Dans la section 2.4, nous montrons que les techniques d'indexation des systèmes OLTP ne sont pas appropriées pour un système multidimensionnel. Nous présentons également le problème de sélection d'index dans les entrepôts, et son utilité dans la réduction du temps de réponse des requêtes.

La fragmentation des données: La fragmentation est une technique de conception logique utilisée dans les modèles relationnels et objet. Elle consiste à partitionner le schéma d'une base de données en plusieurs sous-schémas dans le but de réduire le temps d'exécution des requêtes.

Dans la section 2.5 nous présentons ce concept et les algorithmes associés pour les modèles relationnel et objet.

2.2 Les modèles de données pour les entrepôts

La conception d'un entrepôt est très différente de celle d'une base de données pour un système OLTP. Les concepts sont plus ouverts et plus difficiles à définir. De

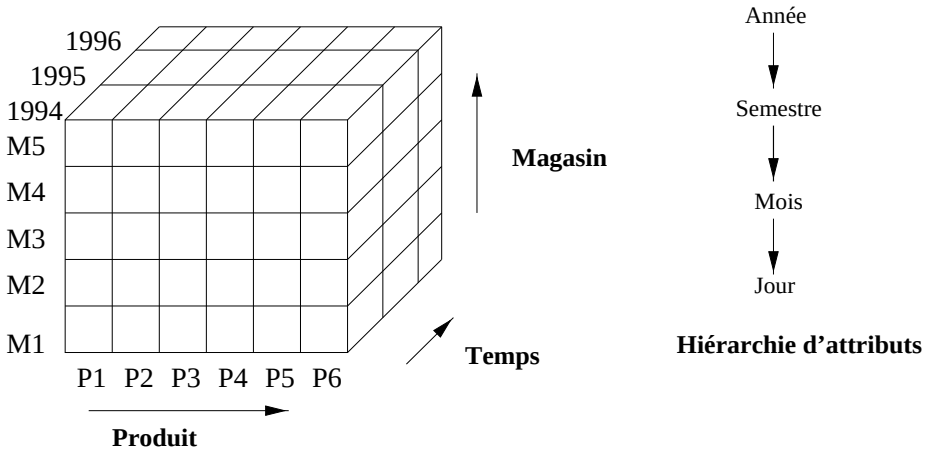


FIG. 2.1: Cube de données

plus, les besoins des utilisateurs de l'entrepôt ne sont pas aussi clairs que ceux des utilisateurs des systèmes OLTP. Les modèles de données utilisés dans la conception des systèmes transactionnels traditionnels ne sont pas adaptés aux requêtes complexes. En effet, les transactions dans les systèmes OLTP sont simples, alors que, dans les entrepôts, les requêtes utilisent beaucoup de jointures, demandent beaucoup de temps de calcul et sont de nature ad-hoc. Pour ce type d'environnement on a suggéré une nouvelle approche de modélisation: les modèles multidimensionnels.

2.2.1 Les modèles multidimensionnels

La conception des bases de données est en général basée sur le modèle Entité-Association (Entity-Relationship) (E-R). Ce modèle permet de décrire des relations entre les données élémentaires (entités) en éliminant des redondances, ce qui provoque l'introduction d'un nombre important de nouvelles entités. De ce fait, l'accès aux données devient compliqué et le diagramme généré difficile à comprendre pour un utilisateur. C'est pour cette raison que l'utilisation de la modélisation E-R pour la conception d'un entrepôt n'est pas considérée comme appropriée [Kim97].

Le modèle multidimensionnel de données, par contre, permet d'observer les données sous plusieurs perspectives. Son axe d'analyse facilite l'accès aux données. Il est plus facilement compréhensible, même pour les personnes qui ne sont pas expertes en informatique. Dans ce modèle, les lignes et les colonnes sont remplacées par des dimensions, en tant que catégories descriptives, et par des mesures qui font office de valeurs quantitatives. La modélisation multidimensionnelle part du principe que l'objectif majeur est la vision multidimensionnelle des données. Le problème de performance est inhérent au modèle. Le constructeur fondamental de ces modèles est le *cube* de données.

2.2.1.1 Le cube de données

Le *cube* de données offre une abstraction très proche de la façon dont l'analyste voit et interroge les données. Il organise les données en une ou plusieurs *dimensions* qui déterminent une *mesure* d'intérêt. Une dimension spécifie la manière dont on regarde les données pour les analyser, alors qu'une mesure est un objet d'analyse.

Chaque dimension est formée par un ensemble d'attributs et chaque attribut peut prendre différentes valeurs. Les dimensions possèdent en général des hiérarchies associées qui organisent les attributs à différents niveaux pour observer les données à différentes granularités. Une dimension peut avoir plusieurs hiérarchies associées, chacune spécifiant différentes relations d'ordre entre ses attributs.

Exemple 1 *Nous pouvons modéliser les données de ventes d'une chaîne de magasins en utilisant un cube à trois dimensions. Chaque cellule de ce cube stocke le total des ventes pour un produit particulier, sur une localisation particulière et pour un jour particulier. Dans cet exemple, l'attribut de mesure (ou fait) est le total de ventes et les attributs de dimension sont le produit, la localisation et le jour. Nous pouvons avoir la hiérarchie suivante pour la dimension "localisation* $\longrightarrow$ *ville* $\longrightarrow$ *état" qui reprécise l'attribut localisation en l'attribut ville, et l'attribut ville en l'attribut état. D'une manière similaire l'attribut jour est représenté par des hiérarchies "année* $\longrightarrow$ *semestre* $\longrightarrow$ *mois* $\longrightarrow$ *jour" (voir Figure 2.1).*

Plus formellement, la structure d'un cube de données est la suivante [AGS96]:

- chacune des d dimensions porte un nom D_i. À chaque dimension D_i est associé un domaine de valeurs Dom_i.
- Une référence de cellule est un n-uplet $< v_1, v_2, ..., v_d >$ appartenant à $Dom_{D_1} \times Dom_{D_2} \times ... \times Dom_{D_d}$. Le contenu d'une cellule d'un cube est soit la constante 0, soit la constante 1, soit un n-uplet $< v'_1, v'_2, ..., v'_k >$ appartenant à $Dom_{D'_1} \times Dom_{D'_2} \times ... \times Dom_{D'_k}$.

On dit encore que $v_1, v_2, ..., v_d$ sont les dimensions membres et que $v'_1, v'_2, ..., v'_k$ sont les dimensions mesures.

Un cube C de d dimensions est une fonction F_c associant à chaque cellule de coordonnées $< v_1, v_2, ..., v_d >$ l'un des éléments suivants:

- La constante 0 si la cellule de référence $< v_1, v_2, ..., v_d >$ n'existe pas pour C;
- La constante 1 si la cellule de référence $< v_1, v_2, ..., v_d >$ existe mais ne contient pas de mesure;
- Un n-uplet $< v'_1, v'_2, ..., v'_k >$ si la cellule de coordonnées $< v_1, v_2, ..., v_d >$ contient $< v'_1, v'_2, ..., v'_k >$.

Exemple 2 *La Figure 2.2 décrit un cube de données VENTES modélisant les ventes d'un magasin. Les dimensions sont client, produit, temps et quantité dont les domaines sont:* $dom_{produit} = \{P1, P2, P3\}$, $dom_{client} = \{C1, C2, C3\}$, $dom_{temps} = \{1999, 2000\}$, *et* $dom_{quantit} \subseteq N$.

Le cube VENTES est défini par trois dimensions pour les membres (produit, client et temps), et une dimension pour les mesures (le n-uplet <quantité> et la fonction F_{ventes} *de* $dom_{produit} \times dom_{client} \times dom_{temps}$ *vers* $dom_{quantit} \cup \{0, 1\}$*). Une cellule du cube VENTES est par exemple* $< P_2, C_2, 2000 >$.

Plusieurs opérations sont introduites pour offrir des possibilités d'animation dans la représentation du cube à l'écran. Elles consistent à faire pivoter le cube, le couper en tranches, interchanger ou combiner les coordonnées et/ou les contenus.

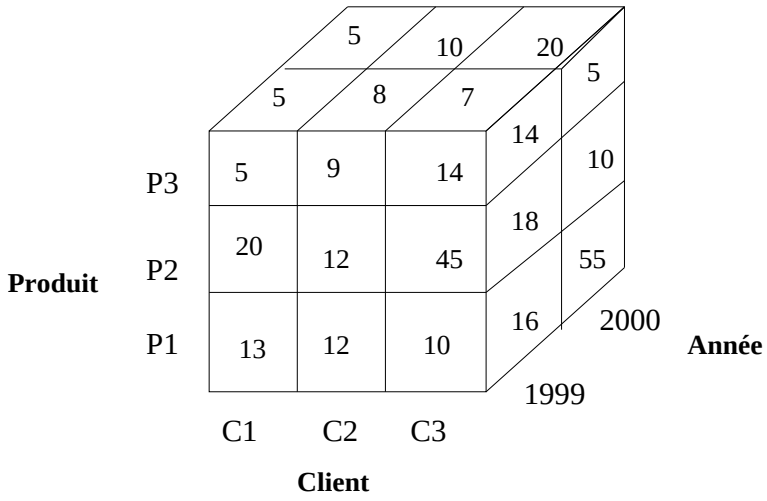


FIG. 2.2: Le cube VENTES

2.2.1.2 Opérations liées à la structure

Les opérations agissant sur cette structure multidimensionnelle de l'information sont motivées par l'aspect interactif de l'analyse en ligne de données, et le souci d'offrir des possibilités d'animation de la représentation [Mar98]. De plus, elles illustrent l'importance des liens entre la manipulation des données et la représentation du cube à l'écran.

Ces opérations sont regroupées sous le nom de restructuration. Tout cube obtenu par une opération de restructuration d'un cube initial contient tout ce qu'il faut pour régénérer le cube initial par restructuration réciproque. Ces opérations sont: *pivot, swich, split, nest, push, et pull.*

Pivot Cette opération consiste à faire effectuer à un cube une rotation autour d'un des trois axes passant par le centre de deux faces opposées, de manière à présenter un ensemble de faces différent.

Switch Cette opération consiste à interchanger la position des membres d'une dimension.

Split Elle consiste à présenter chaque tranche du cube, et à passer d'une représentation tridimensionnelle d'un cube à sa représentation sous la forme d'un ensemble de tables. D'une manière générale, cette opération permet de réduire le nombre de dimensions d'une représentation. On notera que le nombre de tables résultant d'une opération split dépend des informations contenues dans le cube de départ et n'est pas connu à l'avance.

Nest Cette opération permet d'imbriquer des membres. L'un de ses intérêt est qu'elle permet de grouper sur une même représentation bi-dimensionnelle toutes les informations (mesures et membres) d'un cube, quel que soit le nombre de ses dimensions. L'opération réciproque, "unnest", reconstitue une dimension séparée à partir des membres imbriqués.

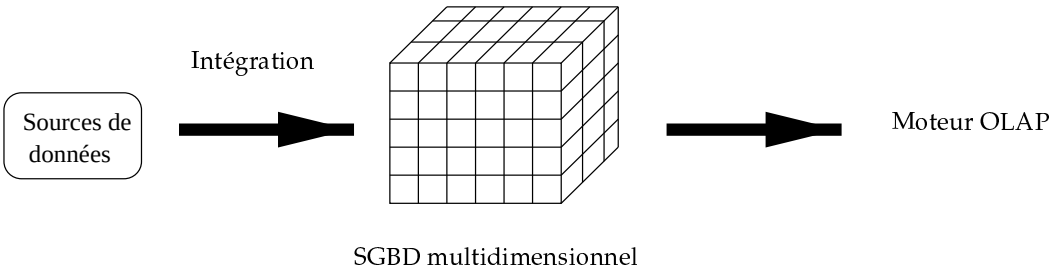


FIG. 2.3: Systèmes MOLAP

Push Cette opération consiste à combiner les membres d'une dimension aux mesures du cube, et donc de faire passer des membres comme contenus de cellules. L'opération réciproque appelée pull, permet de changer le statut de certaines mesures d'un cube en membres, et de constituer une nouvelle dimension pour la représentation du cube, à partir de ces nouveaux membres.

2.2.1.3 Opérations associées à la granularité

Le deuxième aspect de la vision de l'analyste est de hiérarchiser l'information en différents niveaux de détail appelés niveaux de granularité. Les opérations permettant la hiérarchisation sont: *roll-up et drill-down*. Ces deux opérations autorisent l'analyse de données à différents niveaux d'agrégation en utilisant des hiérarchies associées à chaque dimension.

Roll-up Cette opération effectue l'agrégation des mesures en allant d'un niveau particulier de la hiérarchie vers un niveau général. Elle est dénotée par:

$$RollUp_{niveau\_inf}^{niveau\_sup}(CUBE)$$

Drill-down Elle consiste à représenter les données d'un cube à un niveau inférieur, et donc sous une forme plus détaillée. Elle peut être vue comme l'opération réciproque du roll-up. Elle est dénotée par:

$$DrillDown_{niveau\_inf}^{niveau\_sup}(CUBE)$$

2.2.2 Les implémentations des modèles multidimensionnels

Selon la façon dont le cube de données est stocké, il existe deux approches fondamentales pour construire des systèmes basés sur un modèle multidimensionnel. L'approche MOLAP (Multidimensional OLAP) (par exemple Hyperion Essbase OLAP Server [Hyp]) implémente le cube de données dans un tableau multi-dimensionnel (multi-dimensional array). Par contre, l'approche ROLAP (Relational OLAP) (par exemple, Informix Red Brick Warehouse [Cor97b]) utilise un SGBD relationnel pour gérer et stocker le cube de données. Une comparaison détaillée des produits OLAP peut être trouvée dans [PC98].

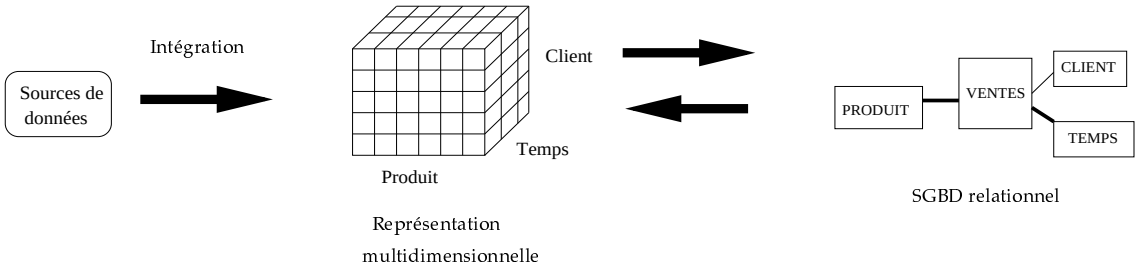


FIG. 2.4: Systèmes ROLAP

2.2.2.1 Les systèmes MOLAP

Les système de type MOLAP stockent les données dans un SGBD multi-dimensionnel sous la forme d'un tableau multi-dimensionnel (multi-dimensional array). Chaque dimension de ce tableau est associée à une dimension du cube (voir Figure 2.3). Seules les valeurs de données correspondant aux données de chaque cellule sont stockées. Ces systèmes demandent un pré-calcul de toutes les agrégations possibles. En conséquence, ils sont plus performants que les systèmes traditionnels, mais difficiles à mettre à jour et à gérer [VS99].

Les systèmes MOLAP apparaissent comme une solution acceptable pour le stockage et l'analyse d'un entrepôt lorsque la quantité estimée des données d'un entrepôt ne dépasse pas quelques gigaoctets et lorsque le modèle multidimensionnel évolue peu [GCA01]. Mais, lorsque les données sont éparses, ces systèmes sont consommateurs d'espace [CD97] et des techniques de compression doivent être utilisées. Les produits de Hyperion Essbase OLAP Server [Hyp] ont adopté cette technique de stockage.

2.2.2.2 Les systèmes ROLAP

Les systèmes de type ROLAP utilisent un SGBD relationnel pour stocker les données de l'entrepôt. Ils représentent une interface multidimensionnelle pour le SGBD relationnel. Le moteur OLAP est un élément supplémentaire qui fournit une vision multidimensionnelle de l'entrepôt, des calculs de données dérivées et des agrégations à différents niveaux. Il est aussi responsable de la génération des requêtes SQL mieux adaptées au schéma relationnel, qui profitent des vues matérialisées existantes pour exécuter efficacement ces requêtes. Les mesures (par exemple les quantités vendues) sont stockées dans une table qu'on appelle *la table des faits*. Pour chaque dimension du modèle multidimensionnel, il existe une table qu'on appelle *la table de dimension* (comme Produit, Temps, Client) avec tous les niveaux d'agrégation et les propriétés de chaque niveau (voir Figure 2.4).

Ces systèmes peuvent stocker de grands volumes de données, mais ils peuvent présenter un temps de réponse élevé. Les principaux avantages de ces systèmes sont: (1) une facilité d'intégration dans les SGBDs relationnels existants, (2) une bonne efficacité pour stocker les données multidimensionnelles. Les exemples de produits de cette famille sont *DSS Agent de MicroStrategy* [Mic98] et *MetaCube d'Informix* [Cor97b].

Mendelzon [Men97] fait une comparaison entre l'approche ROLAP et l'approche MOLAP (voir la Table 2.1).

	Avantages	Inconvénients
ROLAP	Technologie familière	Lent
	Scalable	
	Ouvert	
MOLAP	Modèle multidimensionnel	Technologie non prouvée
	Traitement de requête spécialisé	Non scalable
	Techniques d'indexation spécialisées	

TAB. 2.1: ROLAP vs. MOLAP

Trois schémas sont utilisés pour modéliser les systèmes ROLAP: (1) *le schéma en étoile*, (2) le *schéma en flocon de neige* et (3) *le schéma en constellation.*

Le schéma en étoile Dans ce type de schéma [Kim96], les mesures sont représentées par une table de faits et chaque dimension par une table de dimensions. La table des faits référence les tables de dimensions en utilisant une clé étrangère pour chacune d'elles et stocke les valeurs des mesures pour chaque combinaison de clés. Autour de cette table des faits figurent les tables de dimensions qui regroupent les caractéristiques des dimensions. La table des faits est normalisée et peut atteindre une taille importante par rapport au nombre de n-uplets. Les tables de dimension sont généralement dénormalisées afin de minimiser le nombre de jointures nécessaires pour évaluer une requête. Ce schéma est largement utilisé dans les applications industrielles (les groupes *Redbrik* [Sys97], ou encore *Informix* [Cor97b]).

Cependant, un schéma en étoile est souvent un concept centré-requête, par opposition au schéma centré-mise à jour employé par les applications de type OLTP. Les requêtes typiques de ce schéma sont appelées les requêtes de jointure en étoile (star-join queries) qui ont les caractéristiques suivantes:

1. Il y a des jointures multiples entre la table des faits et les tables de dimension.
2. Il n'y a pas de jointure entre les tables de dimensions.
3. Chaque table de dimension impliquée dans une opération de jointure a plusieurs prédicats de sélection sur ses attributs descriptifs.

La syntaxe générale de ces requêtes est la suivante:

SELECT <Liste de projection> <Liste d'agrégation>
FROM <Nom de la table des faits> <Liste de noms de tables de dimension>
WHERE <Liste de prédicats de sélection & jointure>
GROUP BY <Liste des attributs de tables de dimension>

La Figure 2.5 montre un schéma en étoile pour le cube VENTES où la table des faits VENTES stocke la quantité de produits vendus et les tables correspondant à PRODUIT, à CLIENT et à TEMPS comportent les informations pertinentes sur ces dimensions.

Comme nous le constatons dans la Figure 2.5, la table de dimension TEMPS est dénormalisée, et donc le schéma en étoile ne capture pas directement les hiérarchies (c'est-à-dire les dépendances entre les attributs).

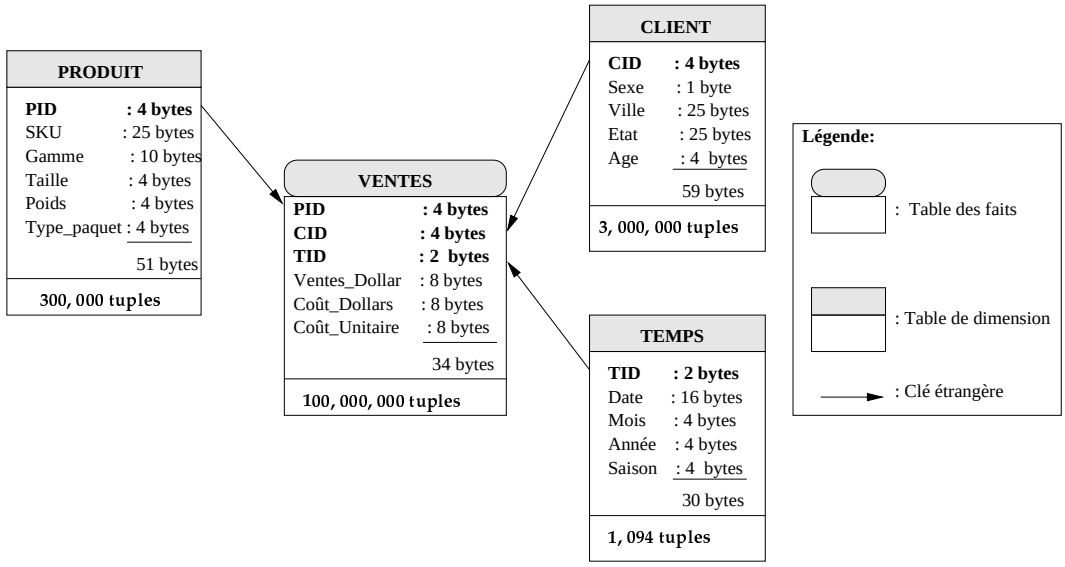


FIG. 2.5: Un exemple d'un schéma en étoile

Le schéma en flocon de neige Le schéma en étoile ne reflète pas les hiérarchies associées à une dimension [JLS99b]. Il exige que les informations complètes associées à une hiérarchie de dimension soient représentées dans une seule table, même lorsque les différents niveaux de la hiérarchie ont des propriétés différentes. Pour résoudre ce problème, le schéma en flocon de neige a été proposé. Ce dernier est une extension du schéma en étoile. Il consiste à garder la même table des faits et à éclater les tables de dimensions afin de permettre une représentation plus explicite de la hiérarchie [JLS99b]. Cet éclatement peut être vu comme une normalisation des tables de dimensions.

Contrairement au schéma en étoile, le schéma en flocon de neige capture les hiérarchies entre les attributs. La Figure 2.6 montre un schéma en flocon de neige après la normalisation de la table de dimension TEMPS.

Ce schéma a été fortement déconseillé par Kimball [Kim96] qui disait : "*ne structurez pas vos dimensions en flocons de neige même si elles sont trop grandes*", mais en même temps, conseillé par des chercheurs (comme Jagadish et al. [JLS99b]) et des industriels de AT&T Labs-Research [JLS99b].

Le schéma en constellation Il est possible d'avoir plusieurs tables des faits partageant des tables de dimensions. Dans ce cas, les tables des faits forment une famille [KS95]. Chaque membre de cette famille possède ses propres dimensions. Si les tables des faits partagent une dimension, il faut vérifier que cette dimension soit exactement la même. Le schéma résultant s'appelle constellation des faits [CD97].

Dans cette thèse nous utilisons un schéma en étoile pour élaborer nos techniques, mais ces dernières peuvent facilement être étendues aux schémas en flocon de neige et constellation.

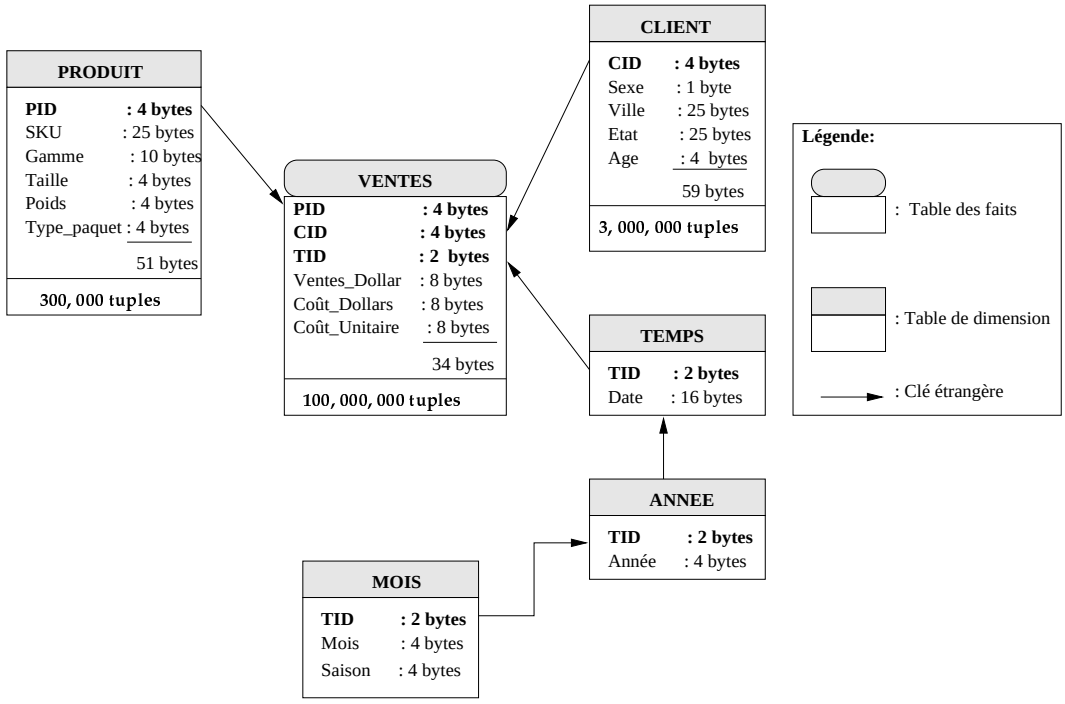


FIG. 2.6: Un exemple d'un schéma en flocon de neige

2.2.3 Les Systèmes HOLAP

Ces systèmes essayent de tirer les bénéfices des deux systèmes précédemment décrits. Dans les systèmes HOLAP, les données sont maintenues par un SGBD relationnel, tandis que les agrégations le sont dans un SGBD multidimensionnel. Le produit *Express d'Oracle Corp.* [Cor98] est un des exemples de cette famille de systèmes.

2.3 Les vues matérialisées

Une vue est une requête nommée. Une vue matérialisée est une table contenant les résultats d'une requête. Les vues améliorent l'exécution des requêtes en précalculant les opérations les plus coûteuses comme la jointure et l'agrégation, et en stockant leurs résultats dans la base. En conséquence, certaines requêtes nécessitent seulement l'accès aux vues matérialisées et sont ainsi exécutées plus rapidement. Les vues dans le contexte OLTP ont été largement utilisées pour répondre à plusieurs rôles: la sécurité, la confidentialité, l'intégrité référentielle, etc.

Les vues matérialisées peuvent être utilisées pour satisfaire plusieurs objectifs, comme l'amélioration de la performance des requêtes ou la fourniture des données dupliquées. Le concept a été largement utilisé dans l'informatique distribuée et l'informatique mobile. Les vues matérialisées dans l'informatique distribuée sont utilisées pour dupliquer des données au niveau des sites distribués. Les réplicas permettent de résoudre des requêtes uniquement par des accès locaux. L'informatique mobile a pris un essor considérable au cours des cinq dernières années grâce aux performances croissantes des matériels (utilisation des batteries, baisse de la

consommation d'énergie) et à des coûts de plus en plus faibles. Afin de pallier les déconnexions inhérentes aux environnements sans fil, les machines mobiles sont souvent contraintes de *copier localement* les données.

Deux problèmes majeurs sont liés aux vues matérialisées: (1) *le problème de sélection des vues matérialisées* et (2) *le problème de maintenance des vues matérialisées.* Nous abordons ces deux problèmes dans les sections suivantes.

2.3.1 La sélection des vues matérialisées

Dans l'environnement d'un entrepôt de données, il est généralement possible d'isoler un ensemble de requêtes à privilégier. L'ensemble des vues matérialisées doit être déterminé en fonction de cet ensemble de requêtes.

Le problème de sélection des vues matérialisées (PSV) peut être vu sous deux angles en fonction du type de modèles de données: (1) le PSV de type ROLAP et (2) le PSV de type MOLAP (voir Figure 2.7).

- Dans le PSV de type MOLAP, nous considérons le cube de données comme la structure primordiale pour sélectionner les vues matérialisées. Chaque cellule du cube est considérée comme une vue potentielle. Notons que le cube de données est un cas spécial d'entrepôt, ne contenant que les requêtes ayant des agrégations sur les relations de base.
- Dans le PSV de type ROLAP, chaque requête est représentée par un arbre algébrique [BC81]. Chaque noeud (non feuille) est considéré comme une vue potentielle. Ce type de PSV est plus général que le premier type.

Il existe trois possibilités pour sélectionner un ensemble de vues [Ull96]:

1. matérialiser toutes les vues: Dans le cube de données cette approche consiste à matérialiser la totalité du cube, tandis que dans le cas du ROLAP, elle consiste à matérialiser tous les noeuds intermédiaires des arbres algébriques représentant les requêtes. Cette approche donne le meilleur temps de réponse pour toutes les requêtes. Mais stocker et maintenir toutes les cellules/noeuds intermédiaires est impraticable pour un entrepôt important. De plus, l'espace utilisé par les vues peut influencer la sélection des index.
2. ne matérialiser aucune vue: Dans ce cas nous sommes obligés d'accéder aux données des relations de base. Cette solution ne fournit aucun avantage pour les performances des requêtes.
3. matérialiser seulement une partie du cube/des noeuds: Dans un cube, il existe une certaine dépendance entre les cellules, c'est à dire que la valeur de certaines cellules peut être calculée à partir des valeurs d'autres cellules. Dans le cas d'un système ROLAP, on trouve également cette dépendance dans les arbres algébriques. Il est alors souhaitable de matérialiser les parties partagées (cellules ou noeuds) par plusieurs requêtes. Cette approche a pour but de sélectionner les cellules ou les noeuds partagés. Cette solution semble la plus intéressante par rapport aux deux approches précédentes.

Quel que soit le type de PSV, ce dernier peut être défini de la manière suivante [Gup99, Ull96]:
Étant donné une contrainte de ressource S (capacité de stockage, par exemple),

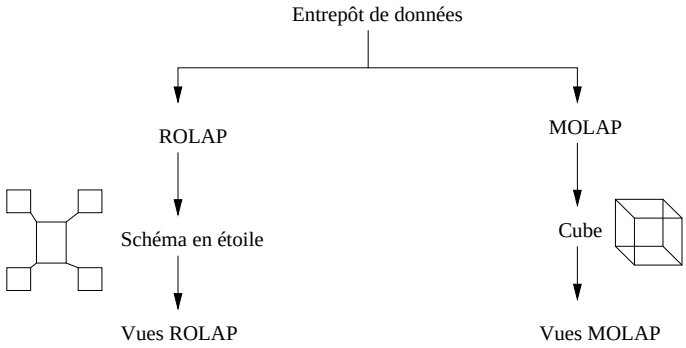


FIG. 2.7: Les types de PSV

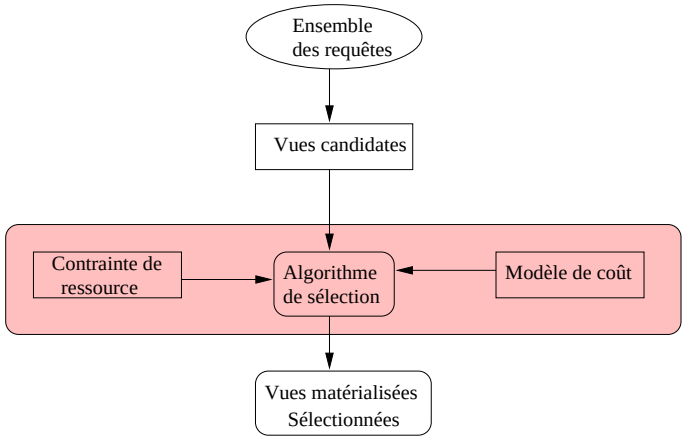


FIG. 2.8: Le processus de sélection des vues matérialisées

le PSV consiste à sélectionner un ensemble de vues $\{V_1, V_2, ..., V_k\}$ minimisant une fonction objectif (coût total d'évaluation des requêtes et/ou coût de maintenance des vues sélectionnées) et satisfaisant la contrainte (voir la Figure 2.8).

Ce problème a été largement étudié tant pour l'approche MOLAP [KR99, Ull96, DRSN98] que pour l'approche ROLAP [Gup99, Gup97, TS97, YKL97, Ked99] afin de sélectionner un ensemble de vues optimales résultant d'une énumération de toutes les vues à matérialiser (ou à précalculer). Leur nombre est très grand et a la complexité $O(2^n)$, où n est le nombre des agrégations dans le schéma. Si d est le nombre de dimensions dans ce schéma, et qu'il ne contient aucune hiérarchie, alors $n = 2^d$. De ce fait, le PSV est NP-difficile [Gup99, SDN98, Ull96].

Rappelons que le rôle principal des vues matérialisées est de réduire le coût d'évaluation de certaines requêtes $Q = \{Q_1, ..., Q_k\}$ (les plus fréquentes, par exemple) définies sur l'entrepôt. La question qui se pose concerne la connaissance préalable ou non de l'ensemble des requêtes Q, d'où la distinction de deux catégories de PSV: (1) *le PSV statique* et (2) le *PSV dynamique*.

Le PSV statique Ce problème possède les données suivantes comme entrées:

- un schéma d'un entrepôt
- un ensemble de k requêtes les plus fréquemment utilisées(les requêtes sont donc connues a priori)

– une ressource.

Le PSV statique consiste à sélectionner un ensemble de vues à matérialiser afin de minimiser le coût total d'évaluation de ces requêtes, le coût de maintenance, ou les deux, sous la contrainte de la ressource.

Le problème suppose donc que l'ensemble des requêtes n'évolue pas. Si des évolutions des requêtes sont enregistrées alors il est nécessaire de reconsidérer totalement le problème (en reconstruisant les vues à matérialiser).

Le PSV dynamique Pour combler les lacunes du PSV statique, Kotidis et al. [KR99] ont proposé un système appelé *DynaMat*, qui matérialise les vues d'une manière dynamique. *DynaMat* combine en fait les problèmes de sélection et de maintenance des vues. Ce système enregistre les évolutions des requêtes et matérialise dans chaque cas le meilleur ensemble de vues pour satisfaire ces requêtes. La contrainte à satisfaire est celle de la capacité d'espace sur mémoire secondaire. Pendant les opérations de mise à jour, DynaMat rafraîchit les vues et si la taille des vues dépasse la capacité de l'espace autorisé, il procède à certaines éliminations selon des critères de placement (par exemple les vues les moins utilisées sont éliminées).

De nombreux algorithmes ont été développés pour élaborer une solution optimale ou quasi-optimale pour le PSV. La plupart de ces algorithmes étaient destinés au cas statique comme nous allons le voir dans la section suivante.

2.3.2 Les algorithmes de sélection des vues

Les algorithmes proposés pour la sélection des vues peuvent être classés en trois catégories, en fonction du type de contrainte qu'ils utilisent: (1) *algorithmes sans aucune contrainte* [RSS96, YKL97, BPT97, TS97], (2) *algorithmes dirigés par la contrainte d'espace* [Gup97] et (3) *algorithmes dirigés par la contrainte du temps total de maintenance des vues* [GHRU97].

2.3.2.1 Les algorithmes sans aucune contrainte

Il existe plusieurs approches pour la sélection des vues sans contrainte. Nous en retenons deux, qui sont l'approche de Yang et al. [YKL97] proposée dans le contexte ROLAP, et l'approche de Baralis [BPT97] proposée dans le contexte MOLAP.

Les travaux de Yang et al. [YKL97] Les auteurs ont développé un algorithme de sélection des vues dans un contexte ROLAP statique. Les auteurs partent du principe suivant: *la principale caractéristique des requêtes décisionnelles est qu'elles utilisent souvent les résultats de certaines requêtes pour répondre à d'autres requêtes* [BPT97]. On peut tirer de cette caractéristiques que les requêtes décisionnelles partagent certaines expressions.

L'algorithme de Yang et al. procède de la façon suivante:
Chaque requête est représentée par un arbre algébrique [BC81, EN94]. Etant donné que chaque requête peut avoir plusieurs arbres algébriques, les auteurs sélectionnent l'arbre optimal (en fonction d'un modèle de coût). Une fois les arbres optimaux identifiés, l'algorithme essaye de trouver des expressions communes entre ces arbres

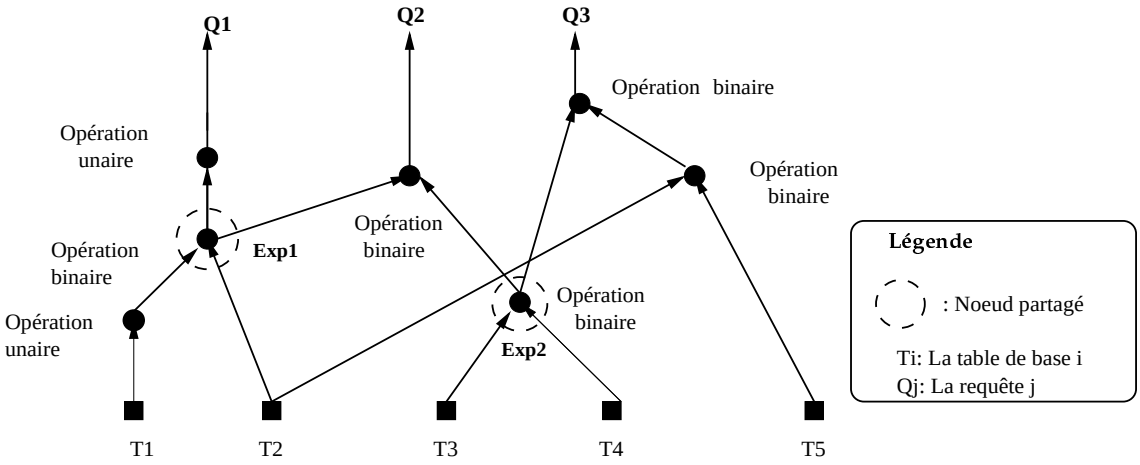


FIG. 2.9: Le principe de base de la sélection de Yang et al. [YKL97]

(ou noeud partagé). Finalement, les arbres sont fusionnés en un seul graphe, appelé *plan multiple d'exécution des vues* en utilisant les noeuds partagés identifiés.

Ce graphe a plusieurs niveaux. Les feuilles sont les tables de base de l'entrepôt et représentent le niveau 0. Dans le niveau 1, nous trouvons des noeuds représentant les résultats des opérations algébriques de sélection et de projection. Dans le niveau 2, les noeuds représentent les opérations ensemblistes comme la jointure, l'union, etc. Le dernier niveau représente les résultats de chaque requête. Chaque noeud intermédiaire de ce graphe est étiqueté par le coût de l'opération algébrique (sélection, jointure, union, etc.) et le coût de maintenance. Ce graphe est utilisé pour rechercher l'ensemble des vues dont la matérialisation minimise la somme des coûts d'évaluation des requêtes et de maintenance des vues. La solution prend en considération l'existence de plusieurs expressions possibles pour une requête. Chaque noeud intermédiaire est considéré comme une vue potentielle.

Exemple 3 *Soit un schéma d'un entrepôt ayant cinq tables et sur lequel trois requêtes sont définies. La Figure 2.9 montre que les requêtes Q_1 et Q_2 ont une expression commune (Exp_1). Ces noeuds sont de bons candidats pour la matérialisation.*

La description de cet algorithme sera détaillée dans le chapitre 3 car nous l'utilisons comme support de nos travaux.

Les travaux de Baralis et al. [BPT97] Dans le contexte MOLAP, Baralis et al. [BPT97] ont développé une heuristique pour le PSV statique. Leur technique consiste à élaborer le treillis des vues qui prend en compte les hiérarchies des attributs (par exemple jour $\longrightarrow$ semaine $\longrightarrow$ semestre). Avant de parler des treillis des vues, quelques définitions s'imposent.

Définition 1 Une relation de dépendance $\preceq$ entre les requêtes *Soient Q_i et Q_j deux requêtes (vues). On dit que $Q_i \preceq Q_j$ si et seulement si Q_i peut être évaluée en utilisant seulement les résultats de la requête Q_j.*

Définition 2 Un treillis *est construit de la façon suivante: les noeuds de ce treillis représentent les vues (agrégées sur certaines dimensions) et un arc existe entre deux noeuds V_i et V_j si elles sont dépendantes ($V_i \preceq V_j$). Le noeud V_i est un noeud ancêtre*

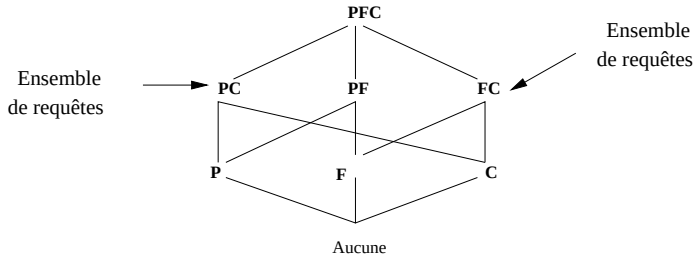


FIG. 2.10: Treillis des vues

et V_j un descendant. Les ancêtres et les descendants d'un noeud V_i du treillis sont définis comme suit:

$$ancêtre(V_i) = \{V_j \quad | V_i \preceq V_j\}$$
$$descendant(V_i) = \{V_j \quad | V_j \preceq V_i\} \quad .$$

Exemple 4 *La Figure 2.10 présente le treillis des vues possibles à partir des dimensions PIECE (représenté par P), CLIENT (représenté par C) et FOURNISSEUR (représenté par F) tirées de la base de données du benchmark TPC-D [Pag].*

Le treillis permet non seulement d'établir des dépendances entre les vues à matérialiser mais constitue également un bon support pour les algorithmes de sélection. Il permet aussi de dire dans quel ordre les vues sont matérialisées [HRU96]. On dit qu'une vue V_i du treillis est une vue candidate si une des deux conditions suivantes est satisfaite:

- V_i est associée à quelques requêtes (voir Figure 2.10).
- il existe deux vues candidates V_j et V_k, telle que V_i est la plus petite borne supérieure de V_j et V_k (le coût de mise à jour des vues V_j et V_k est supérieur à celui de V_i).

Une fois les vues candidates sélectionnées, le bénéfice pour chaque noeud descendant est recalculé et la vue de bénéfice maximal est sélectionnée.

Pour conclure sur ce type d'algorithmes, nous pouvons dire qu'étant donné l'absence de contrainte liée à ces algorithmes, il n'existe aucun moyen d'évaluer leur résultat [Gup99].

2.3.2.2 Algorithmes dirigés par la contrainte d'espace

Dans le travail initial réalisé pour la sélection des vues, Harinarayan et al. [HRU96] ont présenté un algorithme glouton pour sélectionner un ensemble de cellules (vues) dans un cube de données. L'objectif de cet algorithme est de minimiser le temps de réponse des requêtes sous la contrainte que la taille des vues sélectionnées ne dépasse pas la capacité d'espace S. Les auteurs s'intéressent seulement aux requêtes ayant des fonctions d'agrégation.

Exemple 5 *Prenons les mêmes dimensions de l'exemple 4: PIECE (P), FOURNISSEUR (F) et CLIENT (C), et la mesure représente le total des ventes. Chaque cellule (p, f, c) du cube de données stocke le total des ventes de la pièce p qui est acheté par le fournisseur f et vendue au client c. On ajoute la valeur ALL aux domaines des dimensions pour répondre aux requêtes qui ne référencent pas toutes les dimensions. Par exemple, si l'utilisateur cherche le total des ventes de la pièce*

p pour un client donné c (la représentation est (p, ALL, c)). Pour trois dimensions, le nombre de vues potentielles est de l'ordre de $2^3 = 8$.

Les auteurs ont modélisé le problème sous la forme d'un treillis de vues (voir Figure 2.10). Les auteurs ont développé un modèle de coût afin d'estimer le coût de chaque vue du treillis. Ce coût est basé sur le principe suivant:
Pour évaluer une requête Q (vue), nous choisissons une vue ancêtre de Q (disons Q_A), qui a été matérialisée. Le coût d'évaluation de la requête Q est le nombre de n-uplets présents dans la table correspondante à Q_A. Chaque vue est associée à un coût de stockage correspondant au nombre de n-uplets de cette vue.

L'algorithme de sélection des vues est décrit de la façon suivante (voir l'algorithme 1):
Soit S l'ensemble des vues matérialisées à l'étape j de l'algorithme. Pour chaque vue v, un **bénéfice** dénoté par $B(v, S)$ et défini comme suit:

- Pour chaque relation de dépendance $w \preceq v$, définir une quantité B_w par:
  1. Soit u la vue ayant le plus petit coût dans S tel que $w \preceq u$
  2. Si $C(v) < C(u)$, alors $B_w = C(u) - C(v)$, sinon, $B_w = 0$
- $B(v, S) = \sum_{w \preceq v} B_w$

En d'autres termes, le bénéfice de V est calculé en considérant sa contribution dans la réduction du coût d'évaluation des requêtes. Pour chaque vue w couvrant v, nous calculons le coût d'évaluation w en utilisant v et d'autres vues dans S offrant un coût moins élevé pour évaluer w. Si la présence de la vue v est inférieure au coût de w, alors la différence représente une partie du bénéfice de la matérialisation de la vue v. Le bénéfice total $B(v, S)$ est la somme de tous les bénéfices en utilisant v pour évaluer w, et fournissant un bénéfice positif. Les auteurs montrent que cet algorithme

Algorithme 1 Algorithme de Harinarayan et al. [HRU96]

```
1: A: la capacité d'espace
2: |v|: la taille de la vue v
3: while (A > 0) do
4:    v:= vue ayant un bénéfice maximal
5:    if (A − |v| > 0) then
6:       V := V ∪ v {V: l'ensemble des vues}
7:    end if
8: end while
```

présente des performances très proches de l'optimum. Cependant, il parcourt l'espace des solutions possibles à un niveau élevé de granularité et peut éventuellement laisser échapper de bonnes solutions [SDN98].

2.3.2.3 Algorithmes dirigés par le temps de maintenance

Ce type d'algorithmes a été étudié par Gupta [Gup99]. Avant de décrire ces algorithmes, quelques définitions sont introduites.

Définition 3 Un graphe d'une requête (ou vue) de type ET *est un graphe de requête (vue) dans lequel cette dernière possède un plan d'exécution unique.*

Définition 4 Un graphe d'une requête (ou vue) de type OU *est un graphe de requête (vue) dans lequel cette dernière possède plusieurs plans d'exécution.*

Etant donné un graphe de vues de type ET-OU et une quantité S (temps de maintenance disponible), le PSV consiste à sélectionner un ensemble de vues minimisant le temps de réponse total tel que le temps total de maintenance des vues soit inférieur à S. Les auteurs ont présenté deux heuristiques pour résoudre ce problème: (1) un algorithme "glouton" polynômial fournissant une solution quasi-optimale pour les graphes de vues de type ET et pour les graphes de vues de type OU (où chaque requête a de multiple plans d'exécution), (2) un algorithme de type A^* [Nil80] pour les graphes de vues de type ET et OU. Notons que tout algorithme de type A^* cherche la solution optimale dans un graphe ayant un nombre petit de noeuds, où chacune représente une solution potentielle.

2.3.3 Maintenance de vues matérialisées

Un entrepôt de données contient un ensemble de vues matérialisées dérivées à partir de tables qui ne résident pas dans l'entrepôt de données. Les tables de base changent et évoluent à cause des mises à jour. Cependant, si ces changements ne sont pas reportés dans les vues matérialisées, leurs contenus deviendront obsolètes et leurs objets ne représenteront plus la réalité. Par conséquent, un objet d'une vue peut continuer à exister alors que les objets à partir desquels il a été dérivé ont été supprimés ou modifiés. Afin de résoudre ce problème d'inconsistance des données, une procédure de maintenance des vues doit être mise en place.

Trois stratégies fondamentales ont été proposées. Dans la première stratégie, les vues sont mises à jour périodiquement [AL80, LHM+86]. Dans ce cas, ces vues peuvent être considérées comme des photographies (snapshots). Dans la deuxième stratégie [CW81, BLT86], les vues sont mises à jour immédiatement à la fin de chaque transaction. Dans la dernière stratégie, les modifications sont propagées d'une manière différée [SP89]. Dans ce cas, une vue est mise à jour uniquement au moment où elle est utilisée par une requête d'un utilisateur.

Quelle que soit la stratégie adoptée, la maintenance pourrait consister à simplement recalculer le contenu des vues matérialisées à partir des tables sources. Cependant, cette approche est complètement inefficace (très coûteuse). En effet, une bonne maintenance des vues est réalisée lorsque les changements (insertions, suppressions, modifications) effectués dans les tables sources peuvent être propagés aux vues *sans obligation de recalculer complètement leur contenu.*

Plusieurs techniques ont été proposées dans la littérature pour répondre à ce besoin: la maintenance incrémentale, la maintenance autonome des vues, et la maintenance des vues en batch.

2.3.3.1 Maintenance incrémentale

Une vue matérialisée peut être maintenue d'une manière incrémentale. Cette maintenance consiste à identifier le nouvel ensemble de n-uplets à ajouter à la vue

dans le cas d'une insertion ou le sous-ensemble de n-uplets à retirer de la vue dans le cas d'une suppression, sans pour autant réévaluer complètement la vue [BLT86].

Plusieurs approches traitant le problème de la maintenance incrémentale des vues matérialisées ont été développées dans les bases de données relationnelles classiques [BLT86] et dans les entrepôts de données [RSS96, Gup99]. Nous en retenons deux, les travaux de Blakeley et al. [BLT86] dans le contexte de base de données et les travaux de Ross et al. [RSS96] dans le contexte des entrepôts de données.

Blakeley et al. [BLT86] s'intéressent aux vues SPJ (sélection, projection et jointure). Par souci de clarté, nous présentons l'algorithme incrémental pour les vues de jointures [1]. Soit $V = R_1 \bowtie R_2 \bowtie ... \bowtie R_n$ une vue joignant n relations. Supposons que l'on insère les ensembles de n-uplets $\triangle R_i$ dans les relations R_i, $i = 1, ..., n$, respectivement. D'après la propagation de ces insertions au niveau de la vue V, nous obtenons $V' = (R_1 \cup \triangle R_1 \bowtie (R_2 \cup \triangle R_2) \bowtie ... \bowtie (R_n \cup \triangle R_n)$. En appliquant la propriété de distributivité de la jointure sur l'union, la mise à jour de V est égale à l'union de 2^n multi-jointures. Les auteurs ont montré que pour maintenir la vue V de façon incrémentale, il est facile d'obtenir toutes les multi-jointures de V' grâce à une table de vérité dont chaque entrée B_i $(1 \leq i \leq n)$ est associée à une relation R_i de la vue.

$$\mathbf{B_i} = \begin{cases} 1, \text{correspond aux nouveaux n-uplets (i.e., } \triangle R_i) \\ 0, \text{correspond aux anciens n-uplets} \end{cases}$$

Exemple 6 *Supposons que $n = 3$ ($V = R_1 \bowtie R_2 \bowtie R_3$). L'ensemble de toutes les multi-jointures est décrit dans la Table 2.2.*

B_1	B_2	B_3	Multi-jointure
0	0	0	$R_1 \bowtie R_2 \bowtie R_3$
0	0	1	$R_1 \bowtie R_2 \bowtie \triangle R_3$
0	1	0	$R_1 \bowtie \triangle R_2 \bowtie R_3$
0	1	1	$R_1 \bowtie \triangle R_2 \bowtie \triangle R_3$
1	0	0	$\triangle R_1 \bowtie R_2 \bowtie R_3$
1	0	1	$\triangle R_1 \bowtie R_2 \bowtie \triangle R_3$
1	1	0	$\triangle R_1 \bowtie \triangle R_2 \bowtie R_3$
1	1	1	$\triangle R_1 \bowtie \triangle R_2 \bowtie \triangle R_3$

TAB. 2.2: La description de l'algorithme de Blakeley et al. [BLT86]

Si on suppose que la relation R_3 n'est pas mise à jour, nous devons donc évaluer l'expression suivante:

$$(\triangle R_1 \bowtie R_2 \bowtie R_3) \cup (R_1 \bowtie \triangle R_2 \bowtie R_3) \cup (\triangle R_1 \bowtie R_2 \bowtie R_3)$$

La même procédure est appliquée dans le cas de suppression de n-uplets, en respectant la propriété de la distributivité de la jointure par rapport à la différence.

Ross et al. [RSS96] ont abordé le problème de la maintenance incrémentale de la manière suivante: Étant donné un ensemble de vues V, quelles sont les vues qu'on

[1] Cet algorithme s'étend naturellement aux vues de types SPJ.

peut matérialiser (et maintenir) afin de garantir une maintenance incrémentale optimale de l'ensemble V?
Il est clair que l'ajout d'un nouvel ensemble de vues augmente d'une manière considérable le besoin d'espace, mais il peut également réduire le coût de maintenance (compromis entre l'espace et le coût de maintenance). Les auteurs ont formulé le problème de détermination d'un ensemble optimal de vues additionnelles à matérialiser comme un problème d'optimisation sur l'espace de toutes les vues possibles. Ils ont proposé deux algorithmes dirigés par un modèle de coût (pour plus de détails voir [RSS96]) dont l'un est exhaustif.

2.3.3.2 Maintenance autonome des vues

Considérons un schéma d'un entrepôt de données ayant un ensemble de tables de base $T = \{T_1, T_2, ..., T_n\}$. Soit V une vue matérialisée. L'ensemble des tables T peut résider dans une ou plusieurs source de données. Un changement au niveau de la table T_i noté par $\triangle T_i$ pourra affecter la vue V.

Définition 5 *Soit V une vue matérialisée calculée à partir d'une table T_i. Supposons qu'un changement intervienne au niveau de T_i ($\triangle T_i$). La vue V est dite à maintenir d'une manière autonome si elle peut être maintenable en utilisant seulement la vue V et les changements de T_i ($\triangle T_i$).*

Plusieurs travaux ont été réalisés pour garantir la maintenance autonome des vues matérialisées. Une des approches consiste à *dupliquer* entièrement toutes les données de base dans l'entrepôt. Le processus de maintenance des vues devient alors local à l'entrepôt [GM95, GMS93]. Toutefois cette approche crée de nouveaux problèmes. Les données ajoutées au niveau de l'entrepôt augmentent la capacité d'espace pour stocker les données dupliquées et entraînent une redondance des informations pouvant causer des inconsistances.

Quass et al. [QGMW96] se sont intéressés à la situation où la vue V n'est pas auto-maintenable. Dans ce cas, les auteurs cherchent un ensemble de *vues auxiliaires* $\mathcal{V}$ défini sur les mêmes relations que la vue V tel que $\{V\} \cup \mathcal{V}$ soit auto-maintenable. Notons que l'ensemble de vues auxiliaires doit être minimal. Les auteurs ne fournissent pas de solution sur ce point.

La plupart des approches maintiennent les vues séparément en utilisant un gestionnaire de vues pour chacune d'elles. Les approches reconnaissant la possibilité pour les vues d'être maintenues ensemble en identifiant les vues en relation sont moins courantes. Cette vision de maintenance autonome de vues multiples a été proposée pour la première fois dans [Huy97].

Un problème qui reste ouvert est de trouver *l'ensemble de vues auxiliaires* le plus économique en termes d'espace et de coût de calcul pour réduire le coût de maintenance des vues matérialisées.

2.3.3.3 Maintenance de vues en batch ou en temps réel

Dans les systèmes commerciaux, la mise à jour s'effectue en batch par l'intermédiaire de *transactions de maintenance* [QW97]. Une transaction de maintenance est relativement longue et peut donc bloquer l'usage de l'entrepôt. Pour cette raison elle est exécutée pendant les périodes d'activité creuse (la nuit par exemple).

Avec Internet, cette approche n'est pas très souhaitable, l'entrepôt devant rester en ligne 24 heures sur 24. La maintenance incrémentale qui vise une mise à jour instantanée à chaque changement au niveau d'une source de données est coûteuse et n'évite pas les lectures inconsistantes durant l'opération. Dans [QW97], les auteurs ont proposé une approche pour ce problème en maintenant deux versions de chaque n-uplet de l'entrepôt simultanément afin que les transactions de lecture et la transaction de maintenance n'interférent pas.

2.3.4 La réécriture des requêtes

Les vues matérialisées sont stockées sous forme de tables relationnelles [BDD+98]. Cela permet à l'utilisateur de les interroger, de les indexer, de les partitionner pour améliorer les performances. Après la sélection des vues matérialisées, toutes les requêtes définies sur l'entrepôt doivent être réécrites en fonction des vues disponibles. Ce processus est appelé *réécriture des requêtes en fonction des vues* [SDJL96]. La réécriture des requêtes a attiré l'attention de nombreux chercheurs car elle est en relation avec plusieurs problèmes de gestion de données: l'optimisation de requêtes, l'intégration des données, la conception des entrepôts de données, etc. Le processus de réécriture des requêtes a été utilisé comme une technique d'optimisation pour réduire le coût d'évaluation d'une requête.

Plus formellement, ce processus peut se définir ainsi:
Soit une requête Q définie sur un schéma d'une base de données et un ensemble de vues $\{V_1, V_2, ..., V_n\}$ sur le même schéma. Est-il possible de répondre à la requête Q en utilisant seulement les vues? Alternativement, quel est le plan d'exécution le moins cher pour Q en supposant qu'en plus des tables de la base de données, on a aussi un ensemble de vues?

Supposons que nous ayons une requête Q exprimée en SQL dans laquelle nous trouvons un ensemble de tables de dimensions et la table des faits. La réécriture d'une requête Q en utilisant des vues est une requête Q' référençant ces vues. C'est à dire que dans la clause FROM, nous trouvons des vues, des tables de dimensions et la table des faits.

Sélectionner la meilleure réécriture pour une requête est une tâche difficile. Plusieurs travaux ont été développés dans ce sens (pour plus de détail voir l'étude de Levy [Lev]). La plupart des solutions proposées sont basées sur des modèles de coût. Dans [BDD+98], les auteurs décrivent une implémentation d'un algorithme de réécriture pour ORACLE.

2.4 Les index

Compte tenu de la complexité des requêtes décisionnelles et de la nécessité d'un temps de réponse court, plusieurs techniques d'indexation ont été développées pour accélérer l'exécution des requêtes. Dans les entrepôts de données, lorsque nous parlons des index, nous devons faire la différence entre: (1) les techniques d'indexation, et (2) la sélection des index.

2.4.1 Les techniques d'indexation

Les techniques d'indexation utilisées dans les bases de données de type (OLTP) ne sont pas bien adaptées aux environnements des entrepôts des données. En effet la plupart des transactions OLTP accèdent à un petit nombre de n-uplets, et les techniques utilisées (index B^+ par exemple) sont adaptées à ce type de situation. Les requêtes décisionnelles adressées à un entrepôt de données accèdent au contraire à un très grand nombre de n-uplets (ce type de requête est encore appelé *requêtes d'intervalle*). Réutiliser les techniques des systèmes OLTP conduirait à des index avec un grand nombre de niveaux qui ne seraient donc pas très efficaces [OQ97, Cor97b].

Un index peut être défini sur une seule colonne d'une relation, ou sur plusieurs colonnes d'une même relation. Nous appelons ce type d'index mono index. Il pourra être clustérisé ou non clustérisé. Nous pouvons également avoir des index définis sur deux relations comme les index de jointure [Val87] qui sont appelés multi-index.

Dans les entrepôts de données, les deux types d'index sont utilisés: index sur liste des valeurs, et index de projection (pour les mono index), index de jointure en étoile (star join index) pour les index multi-index. Dans la section suivante, nous allons énumérer les nouvelles techniques d'indexation et des variantes des techniques existantes efficaces pour les requêtes décisionnelles.

2.4.1.1 Index sur liste des valeurs

Un index sur liste de valeurs est constitué de deux parties. La première partie est une structure d'arbre équilibré et la deuxième est un schéma de correspondance. Ce schéma est attaché aux feuilles de l'arbre et il pointe vers les n-uplets de la table à indexer. L'arbre est généralement de type B avec une variation de pourcentage d'utilisation. Deux types différents de schéma de correspondance sont utilisés. Le premier consiste en une liste de RowID associée à chaque valeur unique de la clé de recherche. Cette liste est partitionnée en blocs disque chaînés entre eux. Le deuxième schéma est de type bitmap. Il utilise un index binaire [OQ97] représenté sous forme d'un vecteur de bits. Dans ce vecteur, chaque n-uplet d'une relation est associé à un bit qui prend la valeur 1 si le n-uplet est membre de la liste ou 0 dans le cas contraire. Un index binaire est une structure de taille réduite qui peut être gérée en mémoire, ce qui améliore les performances. De plus, il est possible d'exécuter des opérations logiques (par exemple les opérations ET, OU, XOR, NOT) de manière performante [OQ97].

Cette technique d'indexation est appropriée lorsque le nombre de valeurs possibles d'un attribut est faible (par exemple l'attribut *sexe* qui peut prendre comme valeur masculin ou féminin) (voir Figure 2.11). Évidemment, le coût de maintenance peut être élevé car tous les index doivent être actualisés à chaque nouvelle insertion d'un n-uplet. L'espace de stockage augmente en présence de dimensions de grande cardinalité, parce qu'il faut gérer une quantité importante de vecteurs qui contiennent un grand nombre de bits avec la valeur 0. Pour éviter ce problème, des techniques de compression ont été proposées, comme le "run-length encoding". Dans cette technique, une séquence de bits de la même valeur est représentée de manière compacte par une paire dont le premier élément est la valeur des bits et le deuxième le nombre de bits dans la séquence. L'utilisation de ce type de méthode dégrade les

FIG. 2.11: Index binaire

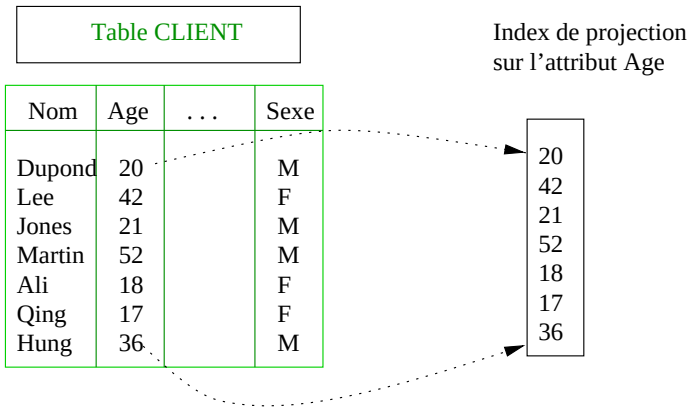


FIG. 2.12: Index de projection

performances du système décisionnel à cause des traitements de compression et de décompression des index.

2.4.1.2 Index de projection

Soit C la colonne d'une table à indexer. L'index de projection sur C est constitué d'une séquence stockée des valeurs de C, ces dernières apparaissant dans le même ordre que le numéro de n-uplet dans la table de laquelle les valeurs ont été extraites (voir Figure 2.12). Les auteurs de [OQ97] ont montré que les index de projection non seulement améliorent d'autres techniques d'indexation traitant les requêtes impliquant deux ou plusieurs colonnes, mais sont également souhaités pour les requêtes de groupement (GROUP-BY). Cette technique est implémentée dans Sybase IQ [OQ97].

2.4.1.3 Index de jointure

Les requêtes complexes définies sur une base de données relationnelle demandent fréquemment des opérations de jointure entre plusieurs tables. L'opération de jointure est fondamentale dans les bases de données, et est très coûteuse en terme de temps de calcul lorsque les tables concernées sont grandes [LR98]. Plusieurs méthodes ont été proposées pour accélérer ces opérations. Ces méthodes incluent les boucles imbriquées, le hachage, la fusion etc [Ram98].

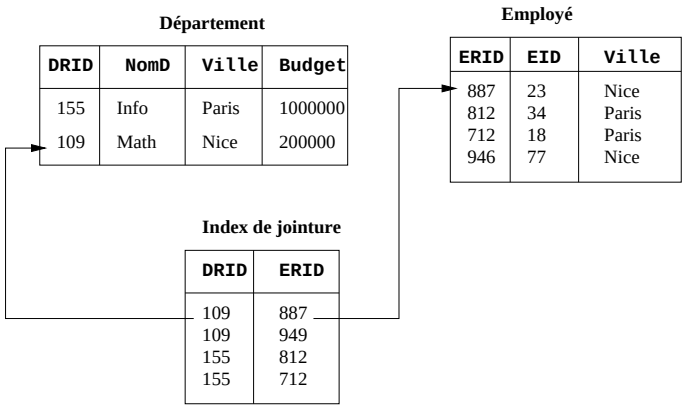


FIG. 2.13: Index de jointure pour l'équi-jointure Employé.Ville = Département.Ville

Valduriez [Val87] a proposé des index spécialisés appelés *index de jointure*, pour pré-joindre des relations. Un index de jointure matérialise les liens entre deux relations par le biais d'une table à deux colonnes, contenant les RID (identifiant de n-uplet) des n-uplets joints deux par deux (voir Figure 2.13). Cet index peut être vu comme une jointure précalculée. Créé à l'avance, il est implémenté par une *relation d'arité 2*. L'efficacité dépend du coefficient de sélectivité de jointure [Val85]. Si la jointure a une forte sélectivité, l'index de jointure sera petit et aura une grande efficacité. Ce genre d'index est souhaité pour les requêtes des systèmes OLTP car elles possèdent souvent des jointures entre deux tables [Sys97]. Par contre, pour les entrepôts de données modélisés par un schéma en étoile (schéma le plus couramment utilisé), ces index sont limités. En effet les requêtes décisionnelles définies sur un schéma en étoile possèdent plusieurs jointures (entre la table des faits et plusieurs tables de dimension). Il faut alors subdiviser la requête en fonction des jointures. Or le nombre de jointures possibles est de l'ordre de $N!$, N étant le nombre de tables à joindre (*problème d'ordonnancement de jointure* [KL90]).

Pour résoudre ce problème, Red Brick [Sys97] a proposé un nouvel index appelé index de jointure en étoile (star join index), adapté aux requêtes définies sur un schéma en étoile. Un index de jointure en étoile peut contenir toute combinaison de clés étrangères de la table des faits. Supposons par exemple que nous ayons un schéma en étoile modélisant les ventes au niveau d'un grand magasin. Ce schéma contient une table des faits VENTES et trois tables de dimensions (TEMPS, PRODUIT et CLIENT). Un index de jointure en étoile peut être n'importe quelle combinaison contenant la clé de la table de faits et une ou plusieurs clés primaires des tables de dimensions.

Ce type d'index est dit **complet** s'il est construit en joignant toutes les tables de dimensions avec la table des faits. Un index de jointure partiel est construit en joignant certaines des tables de dimensions avec la table des faits. En conséquence, l'index complet est bénéfique pour n'importe quelle requête posée sur le schéma en étoile. Il exige cependant beaucoup d'espace pour son stockage.

A ce stade de notre présentation, deux remarques s'imposent concernant l'index de jointure en étoile:

- Comme son nom l'indique, ce type d'index est exclusivement adapté aux schémas en étoile [JLS99b]. Par contre, pour d'autres schémas comme le flocon de

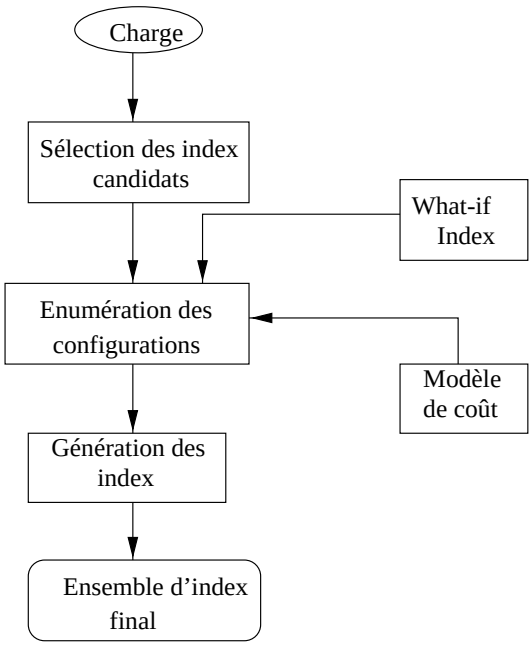


FIG. 2.14: L'architecture de l'outil de sélection d'index

neige, ces index ne sont pas bien adaptés. Notons qu'il n'existe pas d'index de jointure efficace pour tous les schémas logiques de type ROLAP.

– Dans la littérature, nous ne trouvons pas d'algorithme de sélection d'index de jointure en étoile pour un ensemble de requêtes. Ce problème est important car très souvent il n'est pas possible de construire un index de jointure en étoile complet. Il faut donc sélectionner un ou plusieurs index de jointure en étoile pour satisfaire au mieux les requêtes.

2.4.2 Le problème de sélection des index

Comme pour le PSV, le PSI consiste, à partir d'un ensemble de requêtes décisionnelles et la contrainte d'une ressource donnée (l'espace, le temps de maintenance, etc.) à sélectionner un ensemble d'index afin de minimiser le coût d'exécution des requêtes. Ce problème a été reconnu par la communauté académique et industrielle. Les chercheurs ont modélisé le PSI comme un problème d'optimisation et ont proposé des heuristiques afin de trouver des solutions optimales ou quasi-optimales.

Récemment, le groupe base de données de Microsoft a développé un outil pour sélectionner des index avec Microsoft SQL Server 7.0 [CN98]. Étant donné une charge constituée d'un ensemble de requêtes SQL, l'outil de sélection des index recommande d'une manière automatique un ensemble d'index pour cette charge. L'utilisateur peut spécifier des contraintes, par exemple une borne maximale pour l'espace alloué ou pour le nombre d'index. L'outil permet à l'utilisateur d'établir une analyse quantitative de l'impact de la recommandation proposée. Si l'utilisateur accepte les recommandations, l'outil crée (et/ou élimine) des index afin que les index recommandés soient matérialisés. L'architecture de l'outil de sélection des index proposé est illustrée dans la Figure 2.14.

L'outil prend un ensemble de requêtes définies sur un schéma de base de données.

Le traitement est itératif. Durant la première itération, il choisit les index sur une colonne (mono-index); dans la deuxième les index sur deux colonnes et ainsi de suite. L'algorithme de recherche d'index est testé en fonction de ces trois modules: (1) la sélection des index candidats, (2) l'énumération des configurations [2] et (3) la génération des multi-index.

- Le module de sélection des index candidats permet de déterminer la meilleure configuration pour chaque requête d'une manière indépendante. Finalement, il fait l'union de ces configurations.
- Le module d'énumération des configurations: s'il existe n index candidats, et que l'outil doit sélectionner k parmi n index, le module d'énumération doit énumérer toutes les configurations, et à l'aide d'un modèle de coût sélectionner le meilleur ensemble de configurations garantissant un coût minimal.

Cet algorithme de sélection des index prend une requête à un moment donné et sélectionne tous les index possibles. Cependant, l'ensemble des index utilisant cette méthodologie pourra exiger beaucoup d'espace de stockage et des coûts de maintenance élevés.

Dans le but de minimiser les coûts de stockage et de maintenance, Chaudhuri et al. [CN99] ont proposé une technique appelée *fusion d'index* (index merging). Elle prend un ensemble d'index ayant une capacité d'espace S et fournit un nouvel ensemble d'index ayant une capacité d'espace S' inférieure à celle de départ ($S' < S$). L'opération de fusion est guidée par un modèle de coût: la fusion est appliquée s'il y a une réduction dans le coût d'exécution des requêtes. La technique de fusion d'un ensemble d'index ressemble à la reconstruction des fragments verticaux d'une relation donnée. Le problème de fusion des index est formulé comme suit:
Entrées:

- Une configuration d'index initiale $C = \{I_1, I_2, ..., I_N\}$
- Un ensemble de requêtes $Q = \{Q_1, Q_2, ..., Q_p\}$
- Une borne maximale U pour le coût d'exécution des requêtes

Le but consiste à trouver une configuration minimale $C' = \{J_1, J_2, ..., J_k\}$, $k \leq N$ tel que:

- coût(Q, C') $\leq U$, (coût(Q, C') représente le coût (ou estimation) d'exécution de l'ensemble des requêtes de Q en présence de la configuration C')
- C' doit avoir un coût de stockage inférieur à celui de C.

2.4.3 Discussion

Jusqu'à présent, nous avons présenté les deux problèmes PSV et PSI ainsi que les algorithmes proposés pour les résoudre. Avant d'aller plus loin, quelques remarques s'imposent:

1. Le PSV et PSI sont réalisés durant la phase de conception logique et physique de l'entrepôt, respectivement. Ces deux problèmes sont fortement dépendants.
2. Tous les algorithmes proposés pour résoudre ces problèmes sont dirigés par un modèle de coût. Ce dernier permet non seulement de dire si une vue (ou

[2] Les auteurs utilisent le terme configuration pour signifier un ensemble d'index

index) est plus bénéfique qu'une autre vue (ou index), mais également de guider ces algorithmes dans leur sélection. En conséquence il faut prévoir un modèle de coût des requêtes pour mieux les optimiser [Naa99]. Le modèle de coût accepte en paramètre le plan d'exécution d'une requête et retourne son coût. Le coût d'un plan d'exécution est évalué en cumulant le coût des opérations élémentaires (sélection, jointure, etc.). Ces modèles de coûts contiennent, d'une part, des statistiques sur les données et, d'autre part, des formules pour évaluer le coût. Ces coûts sont mesurés en unités de temps si l'objectif est de réduire le temps de réponse des requêtes, le nombre d'entrées-sorties ou le temps de maintenance des vues et des index.

3. Ces deux problèmes sont traités d'une manière séquentielle; c'est-à-dire, d'abord la sélection des vues matérialisées et ensuite la sélection des index. Cette façon de procéder ne prend pas en compte l'interaction entre les vues et les index et pose un problème de gestion de ressources. Par exemple, considérons que ces deux problèmes soient contraints par la capacité d'espace. Il s'agit alors de savoir comment distribuer l'espace entre les vues et les index afin de garantir une meilleure performance des requêtes.

Dans la section suivante, nous présenterons quelques travaux traitant du PSV en présence des index.

2.4.4 La sélection des vues en présence des index

Tous les travaux précédents considèrent la sélection des vues à matérialiser sans index et la sélection des index sans vues matérialisées. Dans cette section, nous regardons certains travaux de sélection considérant simultanément les vues et les index. Parmi ces travaux, on trouve ceux de Labio et al. [LQA97] et Gupta et al. [GHRU97].

2.4.4.1 Les travaux de Labio et al.

Rappelons d'abord que Ross et al. [RSS96] ont abordé le problème de la maintenance des vues d'une manière incrémentale, et ont démontré la possibilité de réduire le temps total de maintenance des vues en matérialisant un ensemble de vues additionnelles.

Labio et al. [LQA97] ont repris la même formulation du problème [RSS96] (voir section 2.3.3), mais en plus de vues additionnelles, ils considèrent l'ajout des index. Soit un ensemble de vues matérialisées $\{V_1, V_2, ..., V_n\}$, *déjà sélectionnées* par un algorithme de sélection donné. Leur approche consiste à choisir un ensemble de *vues additionnelles (appelées vues de support)* et un ensemble d'index à matérialiser afin que le coût total de maintenance soit minimal.

Exemple 7 *La Figure 2.15 montre un entrepôt de données avec trois relations R, S et T provenant des sources R_{SRC}, S_{SRC} et T_{SRC} avec l'objectif de répondre à des requêtes basées sur une jointure $R \bowtie S \bowtie T$. Supposons que $R \bowtie S \bowtie T$ (RST) soit matérialisée. Cette vue est appelée la vue primaire. Dans la Figure 2.16, nous supposons qu'une autre vue (ST) est matérialisée. Elle est appelée vue de support. En matérialisant la vue ST, le coût total de maintenance des deux vues RST et ST pourra être moins élevé que le coût de maintenance de la vue primaire seule.*

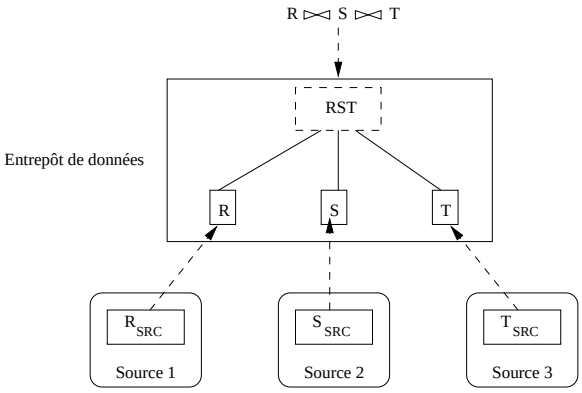


FIG. 2.15: Entrepôt avec une vue primaire

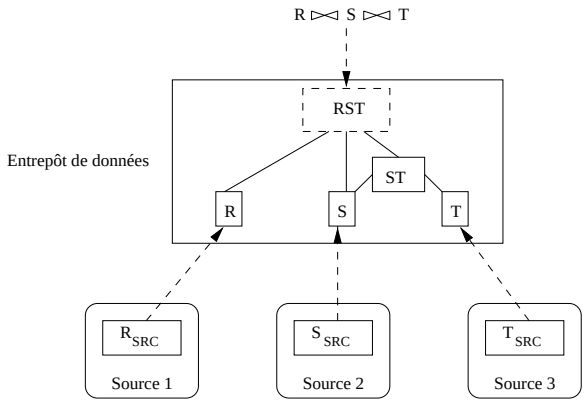


FIG. 2.16: Entrepôt avec une vue de support

Les auteurs calculent d'abord le nombre d'ES nécessaire pour maintenir les vues matérialisées et les relations de base de l'entrepôt de données. Ils ajoutent ensuite un ensemble de vues additionnelles et des index qui eux-mêmes doivent être maintenus. L'opération d'ajout peut avoir lieu si le coût total de l'évaluation des requêtes l'emporte sur le coût de maintenance des vues et des index.

Pour résoudre le problème, Labio et al. [LQA97] ont proposé un algorithme basé sur A^* [Nil80] qui donne la solution optimale. L'inconvénient majeur de cette solution est sa difficulté d'application lorsque le nombre de vues devient très grand. Pour surmonter la complexité de cet algorithme, les auteurs ont proposé des *règles de bon sens* (rules of thumb) qui peuvent aider l'administrateur de l'entrepôt dans le choix des vues de support et d'index (pour plus de détails, voir [LQA97]).

En fait, nous pouvons dire que Labio et al. [LQA97] ont traité le problème de maintenance des vues mais non pas le problème de sélection des vues et des index.

2.4.4.2 Les travaux de Gupta et al. [GHRU97]

Les auteurs ont combiné le PSV et PSI en un seul problème appelé problème de sélection des vues et des index (PSVI) dans un cube de données. L'idée de base de leur travaux est de considérer les vues et les index comme un seul ensemble. Le PSVI est formulé comme suit:

- Soit un ensemble de vues matérialisées $V = \{V_1, V_2, ..., V_n\}$, dont chaque vue V_i possède un ensemble de mono index.
- Soit un ensemble de requêtes Q les plus fréquemment utilisées (une vue et des index peuvent être utilisés pour répondre à une requête donnée).
- Soit S la quantité d'espace allouée pour stocker les vues et les index sélectionnés.

Le PSVI consiste à sélectionner un ensemble de vues et d'index qui ne devront pas occuper un espace supérieur à S, afin de minimiser le coût total d'évaluation de ces requêtes. Ce problème est NP-complet [GHRU97]. Le PSVI est formalisé sous la forme d'un graphe bipartite $G = (V \cup Q, E)$, appelé graphe de requête-vue.

- Chaque vue $V_i \in V$ est associée à un u-uplet (S_i, I_i), S_i est l'espace occupé par la vue V_i, et I_i est l'ensemble des index de cette vue.

- Un coût par défaut est associé à chaque requête, représentant le coût pour évaluer cette requête sans vue et sans index.
- Chaque arête (Q_i, V_j) est associée à un ensemble d'étiquettes (k, t_{ijk}), où t_{ijk} est le coût d'évaluation de la requête Q_i en utilisant la vue V_j et son kième index.

L'algorithme naïf pour résoudre ce problème consiste à tester tous les candidats (vues et index) et à sélectionner la meilleure solution. Cet algorithme est impraticable dans la réalité, compte tenu du nombre trop élevé de candidats. Les auteurs ont donc proposé un algorithme appelé "r-glouton" pour résoudre ce problème. L'algorithme "r-glouton" fonctionne par étapes. Au lieu de sélectionner un objet à chaque étape de l'algorithme naïf, cet algorithme sélectionne plusieurs objets à la fois. Supposons que nous soyons dans une étape où nous avons sélectionné une structure M [3]. Pendant cette étape, l'algorithme considère tous les ensembles possibles ayant au plus r structures qui sont, soit (1) une vue et quelques uns de ses index, soit (2) un seul index dont la vue a déjà été sélectionnée dans les étapes précédentes.

Parmi tous les ensembles considérés, l'ensemble présentant le bénéfice maximum au niveau de l'espace sera sélectionné. Cet algorithme garantit de bonne performances lorsque chaque structure occupe le même espace. Cela est rarement le cas en pratique.

En ce qui concerne la complexité de cet algorithme, supposons qu'il y ait v vues et i index. Pour chaque étape, l'algorithme doit alors considérer et calculer le bénéfice d'au moins $(v \times i + C^i_{r-1})$ ensembles. La borne supérieure du temps d'exécution de cet algorithme est donnée par $O(l \times m^r)$, où m est le nombre de structures dans le graphe de requête-vue, et l le nombre de structures sélectionnées par l'algorithme, respectivement.

2.5 La fragmentation

Dans la littérature, deux termes sont utilisés pour la segmentation d'une relation: *partitionnement* et *fragmentation*. Le partitionnement d'une relation se définit par la division de cette dernière en plusieurs partitions disjointes. La fragmentation consiste en la division en plusieurs fragments (sous-ensembles de la relation) qui peuvent être non disjoints. Cependant, la plupart des chercheurs utilisent les deux termes indifféremment [NKR95, OV91]. Dans cette thèse, nous ne ferons pas de différence entre les deux concepts.

Dès le début des années 80, de nombreuses études ont été menées sur la fragmentation dans les bases de données relationnelles [CNP82, CP84, CY87, NCWJ84, NR89]. Avec l'apparition des systèmes de bases de données objets, de nouvelles études ont vu le jour [KNM94, KL95, KLV96, Bel96, BSS96, BS96, EB95, BKS97, BKL98b].

Définition 6 Un schéma de fragmentation [KNM94] *est le résultat du processus de la fragmentation.*

[3] Une structure est un ensemble de vues et d'index

2.5.1 Les types de fragmentation

Comme nous l'avons déjà mentionné précédemment, deux sortes de fragmentation sont possibles: la fragmentation horizontale et la fragmentation verticale.

2.5.1.1 La fragmentation verticale

Dans la fragmentation verticale, une relation est divisée en sous relations appelées *fragments verticaux* qui sont des projections appliquées à la relation.

Soit une relation $R(K, A_1, A_2, ..., A_n)$, avec n attributs $A_1, A_2, ..., A_n$ et une clé K. Chaque attribut A_i a un domaine de valeurs que nous appelons $dom(A_i)$. La fragmentation verticale de la relation R est donnée par:
$V^1(K, A_1^1, A_2^1, ..., A_{k_1}^1), V^2(K, A_1^2, A_2^2, ..., A_{k_2}^2)$, ... $V^p(K, A_1^p, A_2^p, ..., A_{k_p}^p)$ où chaque attribut $A_j^i \in \{A_1, A_2, ..., A_n\}$. Les fragments verticaux $V_1, V_2, ..., V_p$ sont disjoints si chaque attribut A_i $(1 \leq i \leq n)$ de la relation R appartient à un et un seul fragment vertical. La clé K est dupliquée dans chaque fragment afin de faciliter la reconstruction de la relation R à partir de ces fragments qui est réalisée par la jointure de p fragments verticaux.

La fragmentation verticale favorise naturellement le traitement des requêtes de projection portant sur les attributs utilisés dans le processus de la fragmentation, en limitant le nombre de fragments à accéder. Son inconvénient est qu'elle requiert des jointures supplémentaires lorsqu'une requête accède à plusieurs fragments.

2.5.1.2 La fragmentation horizontale

Comme pour la fragmentation verticale, la fragmentation horizontale a été étudiée dans le cadre des bases de données relationnelles. Elle consiste à diviser une relation R en sous ensembles de n-uplets appelés *fragments horizontaux*, chacun étant défini par une opération de restriction appliquée à la relation. Les n-uplets de chaque fragment horizontal satisfait une clause de prédicats [4]. Le schéma de fragmentation de la relation R est donné par: $H_1 = \sigma_{cl_1}(R), H_2 = \sigma_{cl_2}(R), ..., H_q = \sigma_{cl_q}(R)$, où cl_i est une clause de prédicats. La reconstruction de la relation R à partir de ces fragments horizontaux est obtenue par l'opération d'union de ces fragments.

La fragmentation horizontale se décline en deux versions [CNP82]: les fragmentations *primaire* et *dérivée* (ou indirecte [GGT96]). La fragmentation primaire d'une relation est effectuée grâce à des prédicats de sélection définis sur la relation [OV91]. La fragmentation horizontale dérivée s'effectue avec des prédicats de sélection définis sur une autre relation.

La fragmentation horizontale primaire favorise le traitement des requêtes de restriction portant sur les attributs utilisés dans le processus de la fragmentation.
La fragmentation horizontale dérivée est utile pour le traitement des requêtes de jointure.

2.5.1.3 La fragmentation mixte

La fragmentation mixte combine les deux types de fragmentation: horizontale et verticale. Elle consiste à partitionner une relation en sous-ensembles de sous-

[4] Une clause de prédicats est une combinaison de prédicats avec les opérateurs logiques $\wedge$ et $\vee$

relations, ces dernières étant définies par la fragmentation verticale et les sous-ensembles par la fragmentation horizontale.

2.5.2 Les contextes de la fragmentation

L'idée sous-jacente à la fragmentation est de pouvoir constituer des ensembles de données partielles dont les attributs (pour la fragmentation verticale) et les n-uplets (pour la fragmentation horizontale) ont des propriétés géographiques communes. Le mot géographique peut, comme nous allons le voir, être perçu à une échelle différente suivant le contexte *centralisé*, *réparti* ou *parallèle*.

Définition 7 Une base de données répartie [GGT96, OV91] *est un ensemble de bases de données coopérantes, chacune résidant sur un site différent, cet ensemble étant vu et manipulé par l'utilisateur comme une seule base de données centralisée.*

Définition 8 Une base de données parallèle *est une base de données répartie homogène dont les sites sont les noeuds d'une machine parallèle (multi-processeurs), les noeuds communiquent par messages.*

Le contexte centralisé La fragmentation verticale dans un contexte centralisé favorise naturellement le traitement des requêtes de projection portant sur les attributs utilisés dans la définition des fragments, en limitant le nombre de fragments à accéder.
La fragmentation horizontale permet de limiter le nombre de pages accédées, ou en d'autres termes, à réduire le nombre de n-uplets accédés inutilement.

Le contexte réparti Le principe général reste bien sûr le même. Seule, l'interprétation de la notion de propriétés géographiques, communes entre attributs, change. Dans le contexte réparti, la fragmentation consiste à découper une relation par rapport à un ensemble de sites. Il faut donc une certaine adéquation entre les sites et l'utilisation des données. Dans un tel contexte, deux phases se distinguent lorsqu'on parle de la fragmentation: la phase de partitionnement en elle-même, et la phase d'allocation des fragments aux sites.

Les premiers travaux importants sur la fragmentation ont été réalisés dans le contexte de la conception d'une base de données répartie (distributed database design) [CNP82, CP84, NCWJ84, OV91, OV99].

Le contexte parallèle Dans ce contexte, la fragmentation consiste à partitionner les données d'une relation afin de pouvoir les traiter 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. Les fragmentations verticale et horizontale permettent respectivement l'augmentation du parallélisme inter-requêtes et intra-requêtes.

2.5.3 Les avantages et les inconvénients de la fragmentation

Les objectifs de la fragmentation de données sont multiples:

- limiter le nombre d'accès inutiles aux données (attributs ou n-uplets).
- limiter le transfert de données (en nombre et en volume) en les répartissant où elles sont le plus utilisées dans le cas d'une base de données répartie (par exemple dans les agences d'une banque nationale, ou encore aux guichets de réservation d'une compagnie aérienne).
- accroître les performances par la répartition de la charge de travail en plusieurs unités de traitement opérant en parallèle.
- augmenter la fiabilité en faisant effectuer le même traitement par plusieurs ordinateurs. Par exemple, dupliquer les données sur différents sites, lorsque la base de données concerne un domaine sensible où la moindre erreur peut avoir des conséquences catastrophiques (comptes bancaires, centrales nucléaires, lancement de fusées spatiales).
- étendre la disponibilité des informations, en les dupliquant sur plusieurs sites. L'exemple type est l'annuaire téléphonique national dont chaque ville importante possède un exemplaire.

La fragmentation a plusieurs avantages, mais elle a aussi des inconvénients [OV91]:

- Certains problèmes, tels que la sécurité, la synchronisation ou la coordination, sont difficiles à résoudre dans les systèmes répartis.
- Le manque de méthode de conception d'une base de données répartie. De nos jours, il n'existe pas ou peu de méthodes ou outils pour convertir une base de données centralisée en une base de données répartie.
- Qui dit fragmentation des données, dit surplus d'informations de maintenance tels que les dictionnaires globaux et locaux de données, les fonctions de partitionnement, les index locaux et globaux, etc. La gestion de ces informations et de ces mécanismes est parfois complexe et nécessite une charge supplémentaire. L'administrateur d'une base de données fragmentée doit assurer la transparence aux utilisateurs.
- La principale difficulté de la fragmentation horizontale réside dans le fait qu'une opération de mise à jour dans une relation de schéma global se traduit par plusieurs sous-opérations de mises à jour dans différents fragments. Il faut donc identifier les fragments concernés, puis décomposer l'opération initiale en un ensemble d'opérations sur ces fragments. Après l'opération de mise à jour, certains n-uplets devront changer de fragments (migration de n-uplets). Cette opération de migration peut être coûteuse.

La fragmentation semble être une bonne technique pour l'amélioration des performances dans les entrepôts de données connus pour leur taille volumineuse. Oracle dans sa version *Oracle8i* offre une option de fragmentation (partitioning option) permettant de fragmenter un schéma d'un entrepôt de données, ses vues matérialisés, et ses index.

2.5.4 Les algorithmes de fragmentation verticale

Le problème de la fragmentation verticale est de déterminer comment partitionner une relation (ou une classe) en fragments, dans le but de maximiser la performance du système. Dès les années 70, de nombreux travaux ont été proposés pour

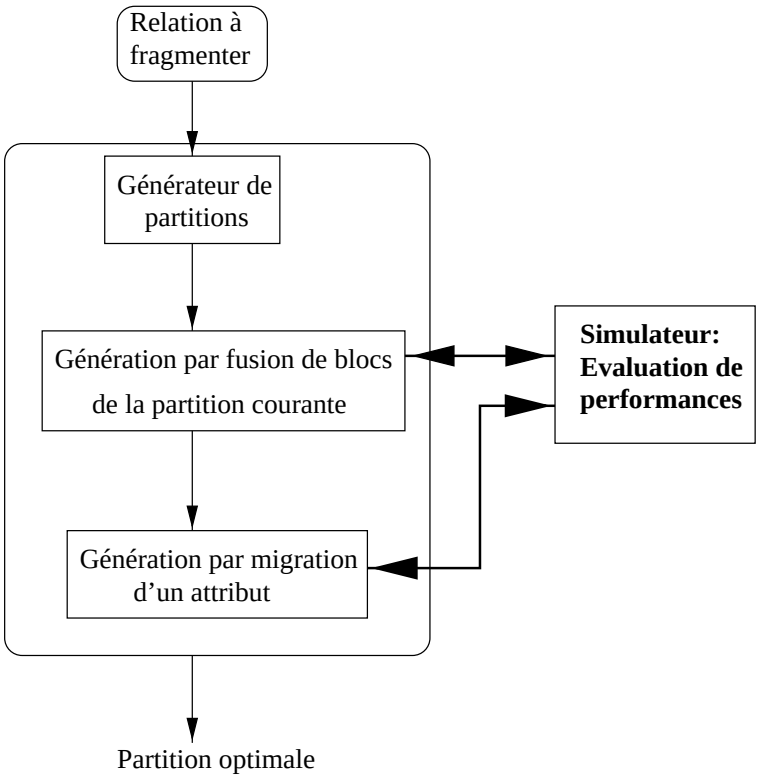


FIG. 2.17: Principe général de la méthode de Hammer et Niamir

définir les fragments verticaux d'un fichier [ES76], d'une relation [NCWJ84, CPW89, HN79, NR89], d'une classe [EB95, BSS96] ou de bases de données parallèles définies avec le modèle NF^2 (Non First Normal Form) [Nic91]. Sélectionner les fragments optimaux d'une relation est un problème difficile: en effet, une relation de m attributs peut être fragmentée en $B(m)$ différentes manières [OV91], $B(m)$ étant le nombre de Bell. Pour m grand, $B(m)$ peut atteindre m^m.

Dans cette section, nous présentons deux travaux que nous considérons de base dans la littérature de la fragmentation verticale: les travaux de Hammer et al. [HN79] et ceux de Navathe et al. [NCWJ84].

Travaux de Hammer et Niamir Les auteurs ont proposé une solution basée sur un modèle de simulation pour l'évaluation des performances présentées par le schéma après une fragmentation verticale. Les deux composantes principales de cette méthodes sont: (1) un générateur de partitions et (2) un évaluateur de performances qui attribue une valeur de mérite à chaque partition (voir Figure 2.17).

Le générateur de partitions fonctionne aussi selon un principe heuristique qui comprend deux étapes itératives. A 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 la fragmentation courante de l'itération suivante.

Les différences entre les deux étapes concernent le choix de la fragmentation initiale et la génération des variations. Dans la première étape, la fragmentation initiale est constituée de fragments à un seul attribut. La fragmentation initiale de

la deuxième étape est celle obtenue à l'issue de la 2ème étape. Dans la première étape, les variations sont générées en considérant toutes les fusions (deux à deux) possibles des blocs de la fragmentation courante. Dans la seconde, les variations sont obtenues par mouvement d'un seul attribut à la fois de son bloc actuel vers tous les autres blocs.

L'évaluation des performances se fait grâce à un estimateur de coûts de transactions qui simule le nombre de pages de la mémoire secondaire transférées pendant l'exécution des transactions. L'algorithme s'arrête dès qu'aucune opération de fusion n'est possible ou qu'aucune réduction du coût n'est constatée.

Les travaux de Navathe et al. Ces travaux ont approfondi ceux de Hoffer et al. [HS75] et Hammer et al. [HN79]. Deux approches ont été proposées par Navathe et al.: le partitionnement binaire [NCWJ84] et le partitionnement graphique [NR89].

La première approche suppose une relation ayant m attributs à fragmenter et un ensemble de n requêtes les plus fréquentes. Elle s'exécute en deux phases. Pendant la première phase, trois matrices sont construites: (i) la matrice d'usage des attributs, (ii) la matrice des affinités d'attributs, et (iii) la matrice d'affinité ordonnée.

- Dans la matrice d'usage des attributs, la valeur de l'élément de la ligne i et la colonne j a pour valeur 1 si l'attribut j est accédé par la requête i, sinon cette valeur est nulle. Cette matrice est complétée par une valeur de fréquence d'accès pour chacune des requêtes représentée dans une colonne supplémentaire.
- La matrice des affinités d'attributs a m lignes et m colonnes correspondant aux attributs de la relation à fragmenter. La valeur de l'élément de la ligne i et la colonne j reporte les valeurs d'affinité définies entre ces attributs. Cette affinité correspond à la somme des fréquences d'accès des requêtes accédant simultanément aux deux attributs.
- La matrice d'affinité ordonnée est obtenue après l'application de l'algorithme de B.E.A. [MSW72] sur la matrice des affinités d'attributs. Par permutation de lignes et de colonnes, l'algorithme B.E.A. regroupe les attributs qui sont utilisés simultanément en fournissant une matrice sous la forme d'un semi-bloc diagonal.

La deuxième phase est la recherche des fragments verticaux qui optimisent une fonction objectif. Elle consiste à effectuer un partitionnement binaire récursif de la matrice d'affinité ordonnée. Cette étape revient à définir des sous-matrices sur la diagonale principale. Chaque sous-matrice regroupe un ensemble de valeurs proches et donc d'attributs fréquemment accédés simultanément. Chaque sous-matrice contient donc les attributs d'un fragment vertical.

Pour la seconde approche, Navathe et Ra [NR89] ont présenté une méthode de regroupement des attributs basée sur les graphes. Cet algorithme part de la matrice des affinités d'attributs considérée comme un graphe complet, appelé graphe d'affinité. Les noeuds de ce graphe représentent les attributs de la relation à fragmenter, et les valeurs des arêtes représentent les valeurs d'affinités. Cet algorithme cherche à générer des cycles dont les arêtes possèdent des grandes valeurs d'affinités. Ces cycles sont appelés cycles d'affinités qui sont en fait des fragments potentiels.

Cet algorithme présente l'avantage d'avoir une complexité en $O(m^2)$ plus faible que celle de l'algorithme proposé dans [NCWJ84] ($O(m^2(logm))$, m étant le nombre d'attributs de la relation à fragmenter.

2.5.5 Les algorithmes de fragmentation horizontale

Les algorithmes de fragmentation primaire proposés dans les contextes relationnel et objet se divisent en deux catégories: (1) *les algorithmes basés sur la minimalité et la complétude des prédicats* (travaux de Ceri et al. [CNP82, CP84] et de Özsu et al. [OV91]), (2) *les algorithmes dirigés par les affinités des prédicats* (travaux de Navathe et al. [NR89]). Ces algorithmes ont été développés dans les bases de données relationnelles et ont été adaptés pour prendre en considération les caractéristiques du modèle objet (méthodes, attributs complexes, héritage, etc.) [EB95, BKS97, Rav96].

Ces deux types d'algorithmes partagent les mêmes entrées, c'est-à-dire, qu'ils partent d'un ensemble de requêtes et de leur fréquence d'accès. Ces requêtes respectent la règle de 80/20, signifiant que 20% de requêtes produisent 80% des accès aux données [NKR95].

Avant de décrire ces algorithmes, quelques définitions s'imposent:

Définition 9 Un prédicat simple [CNP82] *p est de la forme:*
p : Attribut θ Valeur; avec $\theta \in \{=, <, \leq, >, \geq, \neq\}$, Valeur $\in D_i$ et D_i est le domaine de l'attribut A_i.

Définition 10 La fréquence d'accès d'une requête [EB95] *est le nombre d'accès aux données que la requête pourra faire pendant une période spécifique. Dans notre contexte, ces données peuvent être une relation, un fragment, ou un n-uplet.*

Définition 11 Un prédicat pertinent [CNP82] *Un prédicat simple p est pertinent à un ensemble de prédicats simples P s'il existe un prédicat simple $q \in P$ tel que les fragments horizontaux définis par $(p \wedge q)$ et $(p \wedge \neg q)$ soient accédés individuellement par au moins une application.*

Définition 12 *Un ensemble de prédicats est dit minimal s'il ne contient que des prédicats qui lui sont pertinents.*

2.5.5.1 Les algorithmes dirigés par la minimalité et la complétude des prédicats

Özsu et al. [OV91] ont proposé dans le prolongement des travaux de Ceri et al. [CNP82] une méthode de fragmentation dont les étapes sont les suivantes:

1. Classification des transactions: il s'agit de regrouper, dans une même classe, toutes les transactions ayant pour origine le même noeud du réseau d'ordinateurs composant le système réparti. Notons qu'une fréquence d'accès est associéee à chaque transaction.
2. Spécification d'un ensemble complet et minimal de prédicats simples: à partir de ces transactions, on énumère tous les prédicats simples $\mathcal{P}$. La complétude garantit que chaque fragment soit accédé par toutes les applications avec la même probabilité. L'algorithme de production d'un ensemble de prédicats complet et minimal (COM-MIN) à partir d'un ensemble de prédicats simples procède

en deux étapes. La première étape initialise l'ensemble résultat avec un prédicat et sa négation qui partitionne la relation en deux fragments, chacun étant accédé par au moins une application. La seconde étape est itérative, et ajoute tout nouveau prédicat qui partitionne un fragment défini dans l'ensemble résultat en deux fragments, chacun étant accédé par au moins une application. Lors de cette étape, les prédicats qui deviennent non pertinents par rapport à l'ensemble résultat sont éliminés.

Algorithme 2 Algorithme générant des prédicats minimaux et complets

1: **Données:** Une relation R et un ensemble de prédicats simples $\mathcal{P}$ définis sur R
2: **Sortie:** Un ensemble complet et minimal de prédicats simples $\mathcal{P}'$ de $\mathcal{P}$
3: Déclaration: F : l'ensemble de fragments, f_i : un fragment défini par un minterm m_i
4: **Règle 1:** Toute relation est partitionnée en au moins deux fragments qui seront accédés différemment par au moins une application
5: Initialisation:
6: $\mathcal{P}' = \phi$
7: Chercher un prédicat q_i de $\mathcal{P}$ tel que q_i partitionne R en respectant la règle 1
8: $\mathcal{P}' = q_i$; $\mathcal{P} = \mathcal{P} - q_i$; $F = f_i$
9: **repeat**
10: Chercher un prédicat q_i de $\mathcal{P}$ tel que q_i partitionne f_k de F en respectant la règle 1
11: $\mathcal{P}' = \mathcal{P}' \cup q_i$; $\mathcal{P} = \mathcal{P} - q_i$; $F = F \cup f_i$
12: **if** il existe $q_k \in \mathcal{P}'$ qui n'est pas pertinent **then**
13: $\mathcal{P}' = \mathcal{P}' - q_k$
14: $F = F - f$
15: **end if**
16: **until** $\mathcal{P}'$ est complet

3. Construction des prédicats minterm M de $\mathcal{P} = \{q_1, ..., q_n\}$: ces prédicats sont générés de la façon suivante:
   $M = \{m_i \mid m_i = \wedge_{q_j \in \mathcal{P}} \ q_j^*\}$ avec $q_j^* = q_j$ ou $q_j^* = \neg q_j$ $(1 \leq j \leq n), (1 \leq i \leq 2^n)$.
4. Elimination des minterms non significatifs: cette élimination est réalisée par l'identification des minterms contradictoires par rapport à un ensemble d'implications (voir l'algorithme 3).

2.5.5.2 Les algorithmes dirigés par les affinités

Zhang et al. [ZO94] ont adapté l'algorithme proposé par Navathe et al. [NCWJ84] en considérant les trois matrices: la matrice d'usage des prédicats, la matrice d'affinité des prédicats et la matrice d'affinité ordonnée. Ces dernières sont construites de la même façon que celles introduites pour la fragmentation verticale.

Algorithme 3 L'algorithme de fragmentation primaire de Ozsu et al.

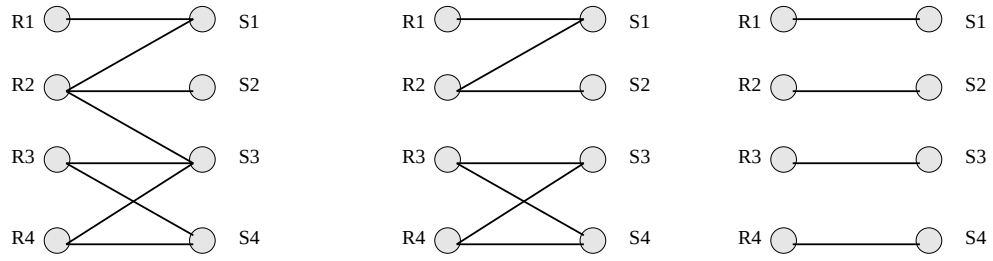
1: **données:** Une relation R, un ensemble de prédicats simples $\mathcal{P}$
2: **sortie:** Un ensemble de fragments
3: $\mathcal{P}' =$ COM_MIN(R, $\mathcal{P}$)
4: déterminer l'ensemble de minterms M_i
5: déterminer l'ensemble des implications I entre les prédicats p_i de $\mathcal{P}'$
6: **for** each minterm $m_i \in M_i$ **do**
7: **if** m_i est contradictoire en respectant I **then**
8: $M_i := M_i - m_i$
9: **end if**
10: **end for**
11: Le nombre de fragments horizontaux correspond aux nombre de minterms résultat.
12: La clause de chaque fragment horizontal représente un minterm.

- Les lignes et les colonnes de la matrice d'usage des prédicats représentent les n requêtes de départ utilisées dans le processus de la fragmentation et les m prédicats de sélection définis dans ces requêtes. La valeur de l'élément de la ligne i et la colonne j a pour valeur 1 si le prédicat j est utilisé par la requête i, sinon cette valeur est nulle.
- La matrice d'affinité des prédicats a m lignes et m colonnes correspondants aux prédicats de sélection. La valeur de l'élément de la ligne i et la colonne j reporte les valeurs d'affinité définies entre ces prédicats. Cette affinité correspond à la somme des fréquences d'accès des requêtes accédant simultanément aux deux attributs.
- La matrice d'affinité ordonnée est obtenue après l'application de l'algorithme de B.E.A. [MSW72] sur la matrice d'affinité des prédicats (voir section 2.5.4).

La dernière étape de l'algorithme proposé par Zhang et al. [ZO94] consiste à construire les fragments horizontaux à partir des semi-blocs diagonaux obtenus après l'application de l'algorithme de B.E.A. La construction d'un fragment à partir d'un semi-bloc consiste à lier tous les prédicats simples comprenant le même attribut par l'opérateur logique OU et les prédicats avec des attributs distincts avec l'opérateur logique ET. Enfin, ils définissent le fragment résiduel en construisant la négation de la disjonction des différents prédicats de fragmentation.

2.5.6 La fragmentation horizontale dérivée

Une relation S (relation membre) peut contenir une clé étrangère vers une relation R (relation propriétaire) [CP84]. La fragmentation horizontale dérivée est définie sur la relation membre S en fonction des fragments horizontaux de la relation propriétaire R.



(a) Graphe de jointure réduit (b) Graphe de jointure partitionné (c) Graphe de jointure simple

FIG. 2.18: Les différents types de graphe de jointure

Plus formellement, soit R une relation propriétaire horizontalement fragmentée en m fragments horizontaux $\{R_1, R_2, ..., R_m\}$. Soit S une relation membre. Les fragments horizontaux dérivés $\{S_1, S_2, ..., S_m\}$ de S sont définis par l'opération de jointure suivante:

$$S_i = S \ltimes R_i \tag{2.1}$$

Trois informations sont nécessaires pour effectuer une fragmentation horizontale dérivée: (1) le nom de la relation membre et de la relation propriétaire, (2) le nombre de fragments horizontaux de la relation membre, et (3) la qualification de la jointure entre ces deux relations.

La fragmentation horizontale dérivée pourra jouer un rôle important dans la réduction du coût de jointure entre deux ou plusieurs relations, qui est une des opérations les plus coûteuses dans les bases de données [LR98]. Pour mieux situer ce rôle, nous introduisons quelques définitions.

Définition 13 La jointure distribuée [CNP82] *est une jointure entre des relations horizontalement fragmentées.*

Lorsqu'une application nécessite une opération de jointure entre deux relations globales R et S, tous les n-uplets de R doivent être comparés à ceux de S, d'où la nécessité de comparer tous les fragments $\{R_1, R_2, ..., R_n\}$ de R avec tous les fragments $\{S_1, S_2, ..., S_n\}$ de S. Mais souvent certaines jointures partielles ($R_i \bowtie S_j$) sont vides. Il est donc intéressant de les identifier a priori.

Une jointure distribuée peut être représentée par un graphe appelé *graphe de jointure* dont les noeuds représentent les fragments de R et S. Une arête est introduite lorsque la jointure entre les fragments correspondants n'est pas vide (Figure 2.18).

Définition 14 *Un graphe de jointure est dit* **total** *[CNP82] lorsqu'il contient toutes les arêtes possibles entre les fragments de R et les fragments de S. Il est dit* **réduit** *sinon. Un graphe de jointure réduit est dit* **partitionné** *s'il est composé de deux ou plusieurs sous-graphes non connexes. Un graphe de jointure réduit est dit* **simple** *s'il est partitionné et si chaque sous-graphe a une seule arête.*

Déterminer si une jointure a un graphe de jointure simple est très important dans les bases de données relationnelles fragmentées [CP84], car il permettra la jointure en parallèle en joignant séparément les fragments liés par une arête et en faisant ensuite l'union des résultats.

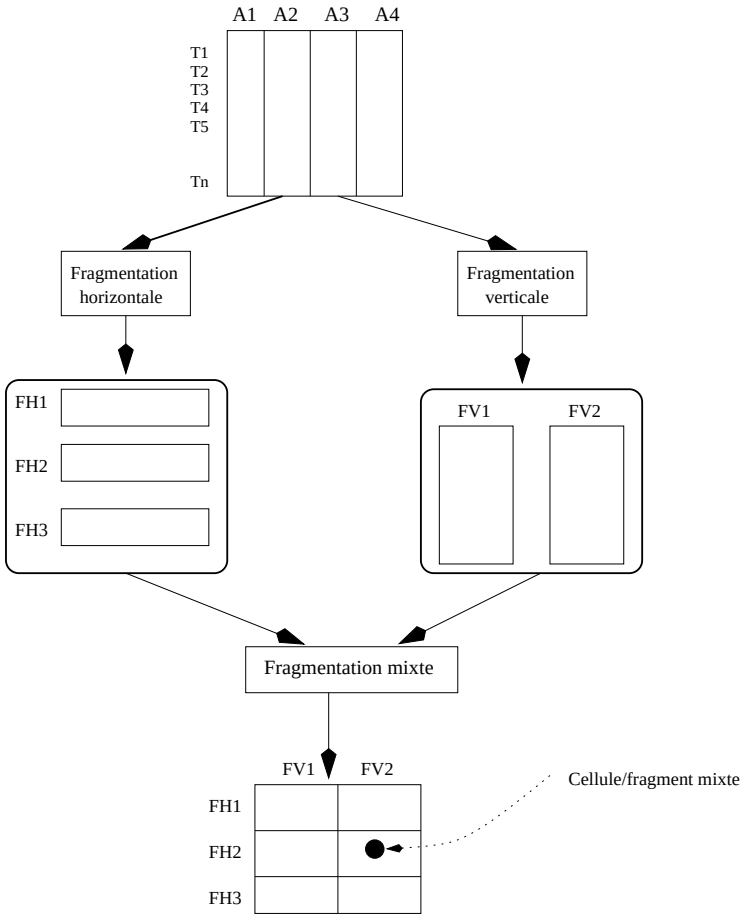


FIG. 2.19: Le principe général de la fragmentation mixte

2.5.7 Les algorithmes de fragmentation mixte

La fragmentation mixte combine la fragmentation horizontale et la fragmentation verticale [CP84, NR89, PKN91]. Une méthodologie de fragmentation mixte a été développée comme une partie du Projet (D^3T) (Distributed Database Design Tool) à l'Université de Floride [Kar96]. Tous les algorithmes de fragmentation mixte ont été étudiés dans le contexte relationnel. Deux types d'algorithmes sont distingués: la fragmentation par création de grilles [NR89] et la fragmentation par la définition de vues [PKN91].

2.5.7.1 La fragmentation par création de grilles

Cet algorithme a été proposé par Navathe et al. [NR89]. Sa première étape consiste à fragmenter horizontalement et verticalement une relation puis à combiner le résultat de ces deux fragmentations afin de construire une grille comme le montre la Figure 2.19. Chaque cellule de cette grille définie par un prédicat de sélection de la fragmentation horizontale et un prédicat de projection de la fragmentation verticale correspond à un fragment mixte [Kar96] (Figure 2.19). Puisque le nombre de cellules peut être très grand, Karlapalem [Kar96] dans sa thèse présente un algorithme d'optimisation qui consiste à trouver les cellules adjacentes afin de minimiser un coût.

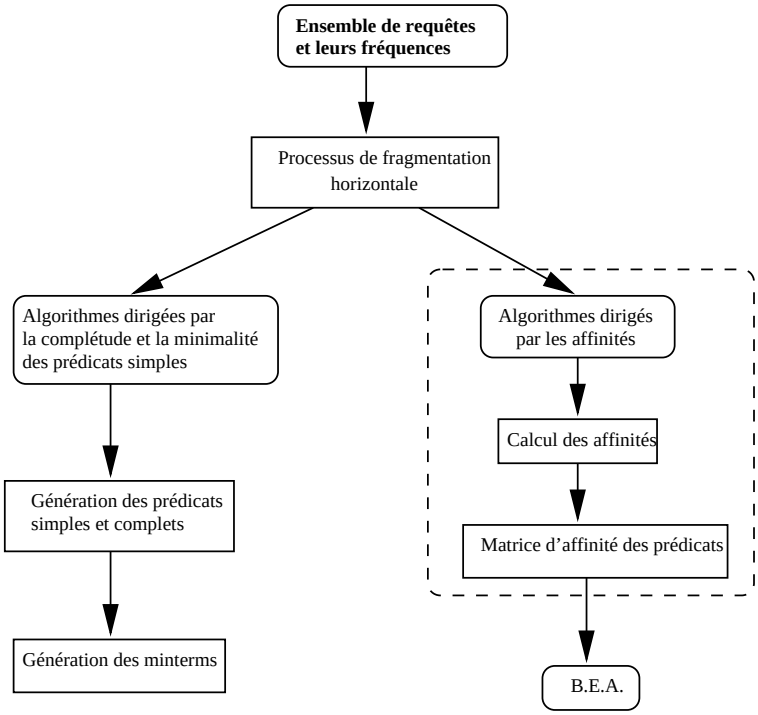


FIG. 2.20: Les différents algorithmes pour la fragmentation horizontale

2.5.7.2 La fragmentation par définition de vues

Pernul et Karlapalem [PKN91] ont considéré que les vues font partie intégrante du modèle de données d'une base de données. Les auteurs ont présenté une méthodologie de décomposition des relations du schéma d'une base de données en sous-ensembles de fragments disjoints. Un ou plusieurs fragments représentent une vue. La procédure de fragmentation se décompose en trois étapes:

1. Trouver toutes les vues ayant une intersection,
2. Pour chaque paire de vues recouvrantes, effectuer une décomposition structurée d'une vue en appliquant respectivement les processus de la fragmentation verticale, horizontale primaire et horizontale dérivée afin de construire des fragments potentiels disjoints.
3. Appliquer les processus de la fragmentation verticale, horizontale primaire et horizontale dérivée sur les vues isolées.

2.5.8 Bilan et discussion

La Figure 2.20 propose un récapitulatif des différents travaux de la fragmentation horizontale dans les bases de données. L'inconvénient majeur des algorithmes dirigés par la complétude et la minimalité des prédicats est qu'ils génèrent 2^n minterms pour n prédicats simples. Dans certaines applications comme les entrepôts de données, où les requêtes possèdent un grand nombre de prédicats de sélection [DRT99], ces algorithmes peuvent être impraticables. En plus, ils demandent un calcul combinatoire des probabilités d'accès pour déterminer la complétude des prédicats simples (voir l'étape 2 de ces algorithmes).

Les algorithmes dirigés par les affinités sont moins complexes que ceux basés sur la complétude et la minimalité. Mais l'inconvénient majeur est le même que celui défini pour la fragmentation verticale [FKL97]. La matrice d'affinité peut exprimer l'affinité *seulement* entre des paires de prédicats, mais pas entre plusieurs prédicats. D'où la nécessité de développer des algorithmes de fragmentation d'une complexité raisonnable et garantisant une meilleure performance.

Finalement, ces deux types d'algorithmes n'offrent aucune garantie sur l'optimalité de leur solution. Intuitivement, la fragmentation horizontale réduit le nombre d'accès inutiles pendant l'évaluation d'une requête. Malheureusement ces algorithmes ne possèdent pas de mesure qui permettrait d'évaluer le bénéfice apporté par l'utilisation d'un algorithme donné de fragmentation. Le principe de groupement qu'utilise ces algorithmes est basé sur une seule information, à savoir la fréquence des requêtes (Figure 2.20). Cependant, d'autres paramètres plus importants sont à considérer (par exemple, le coût d'exécution d'une requête).

Notons que tous les algorithmes de fragmentation (horizontale et verticale) sont statiques. Lors d'un changement dans les entrées de ces algorithmes, ces derniers devront être réexécutés pour définir de nouveaux fragments. Deux types de changements peuvent être distingués dans les bases de données: *les changements physiques* et *les changements logiques.*

- Les changements physiques concernent le système physique supportant la base de données. Ils incluent l'ajout de nouveaux disques, le changement de la topologie du réseau (si la base de données est répartie).
- Les changements logiques incluent la modification (ajout, suppression) des requêtes, le changement du modèle de données de la base de données, le changement de la fréquence d'accès des requêtes.

La notion de grille proposée par Karlapalem [Kar96] est un concept important qui pourra être utilisé dans le processus de re-fragmentation des relations grâce aux opérations de groupement de cellules [Kar96].

Notons que la plupart des algorithmes de fragmentation partent d'un ensemble de requêtes (les plus fréquentes) que le concepteur doit déterminer. La détermination de cet ensemble est parfois délicate. Pernul et al. [PKN91] ont suggéré d'utiliser les vues à la place des requêtes dans le processus de fragmentation, car les vues peuvent être construites en *avance* sans connaître les requêtes des utilisateurs.

2.5.9 La fragmentation dans les bases de données objet

Karlapalem et al. [KNM94] ont été les premiers à étudier les aspects de la fragmentation dans les modèles objets. Ces derniers se sont essentiellement concentrés sur les besoins des algorithmes de fragmentation et non sur les algorithmes en eux-mêmes [ODV92]. Ils définissent les concepts suivants: attribut simple, attribut complexe, méthode simple et méthode complexe.

Dans [KL95], les auteurs définissent les concepts de fragments verticaux et horizontaux de classes. Un *fragment-classe vertical* d'une classe C est un sous-ensemble non vide de l'ensemble d'attributs de C et son ensemble d'objets est identique à C. Un *fragment-classe horizontal* d'une classe C possède une définition structurelle identique à celle de C et son ensemble d'objets représente un sous-ensemble non

vide de C. Une représentation objet est proposée pour implémenter les fragments verticaux et horizontaux. Cette représentation facilite le support de la transparence de la fragmentation. Les algorithmes de partitionnement définis pour le modèle relationnel [NCWJ84, NR89] peuvent être adaptés pour les bases de données objet. Toutefois aucune précision n'est donnée à ce sujet.

Ezeife et Barker [EB95] ont suggéré des algorithmes pour les fragmentations verticale et horizontale. Se basant sur les concepts de Karlapalem et al. [KNM94], ils définissent les attributs simples (le domaine de l'attribut est atomique) et complexes (le domaine de l'attribut est une autre classe) ainsi que les notions méthodes simples (elles n'invoquent pas d'autres méthodes) et complexes (elles invoquent d'autres méthodes). Les algorithmes proposés sont basés sur les affinités entre les méthodes définies dans les requêtes. La construction des fragments suit la démarche suivante:

- Les méthodes sont réparties en groupes en utilisant le concept d'affinité entre méthodes et en reprenant les algorithmes développés dans [NCWJ84]. Cette première étape permet de construire la matrice d'usage pour chaque classe. Les lignes et les colonnes de cette matrice représentent les requêtes et les méthodes spécifiques à la classe C, respectivement.
- Les groupes de méthodes sont complétés par les attributs manipulés par ceux-ci. Si un même attribut est manipulé par plusieurs groupes, il est placé dans le groupe avec lequel il a le plus d'affinité.

Un ensemble d'algorithmes pour la fragmentation horizontale pour quatre modèles de classes est proposé: (1) classes d'attributs simples et méthodes simples, (2) classes d'attributs complexes et méthodes simples, (3) classes d'attributs complexes et méthodes complexes, (4) classes d'attributs simples et méthodes complexes. Ils reprennent les principes de l'algorithme de Özsu et al. [OV91] avec la même notion de prédicats simples. Leur complexité est exponentielle en nombre de prédicats simples.

2.5.10 La fragmentation dans les entrepôts de données

Peu de travaux ont apporté une solution à la fragmentation dans les entrepôts de données, bien que ces derniers contiennent de très grandes tables.

Wu et al. [WB97] dans leur article "research issues in data warehousing" ont recommandé l'utilisation de la fragmentation verticale et horizontale dans les entrepôts de données pour améliorer l'évaluation des requêtes en évitant le balayage des grandes tables. Il est précisé que les algorithmes développés dans les bases de données réparties doivent être adaptés et réévalués pour les entrepôts de données. Deux cas d'utilisation de la fragmentation dans les entrepôts sont considérés:

- Une table des faits peut être partitionnée d'une manière horizontale en fonction d'une ou plusieurs table(s) de dimension. En d'autres termes, la table des faits peut être fragmentée en utilisant la fragmentation dérivée.
- Une table des faits peut également être fragmentée verticalement; toutes les clés étrangères de la table des faits y sont partitionnées. Cependant la reconstruction des tables fragmentées verticalement nécessite l'opération de jointure, qui est très coûteuse.

Les auteurs [WB97] n'ont pas proposé d'algorithme de fragmentation.

Le concept de fragmentation verticale a été introduit dans la définition des index de projection dans les entrepôts par O'Neil et al. [OQ97]. Les index de projection ressemblent à un fragment vertical d'une relation.
Dernièrement, Datta et al. [DRT99] ont développé un nouvel index appelé *"Curio"* dans les entrepôts modélisés par un schéma en étoile. Basé sur la fragmentation verticale de la table des faits, il permet d'accélérer l'exécution des requêtes décisionnelles, et élimine la duplication des données en stockant l'index et non pas les colonnes indexées. Cela permet de réduire l'espace nécessaire pour ces index.

Golfarelli et al. [GMR99] ont utilisé la fragmentation verticale pour partitionner les vues matérialisées. Pour eux, le terme de fragmentation verticale signifie deux choses: d'une part *le partitionnement* des attributs d'une vue en plusieurs fragments et d'autre part *l'unification* en une seule vue de deux ou plusieurs vues ayant une clé commune. L'unification correspond à la règle de reconstruction d'une table fragmentée à partir de ses fragments verticaux [OV91].

Pour ce qui est de la fragmentation horizontale, peu de travaux ont été réalisés, excepté celui de Noaman et al. [NB99]. Ces auteurs ont proposé une technique de construction d'un entrepôt réparti en utilisant la stratégie descendante [GV90]. Cette stratégie est couramment utilisée pour la conception de bases de données réparties. Elle part du schéma conceptuel global d'un entrepôt, qu'elle répartit pour construire les schémas conceptuels locaux. Cette répartition se fait en deux étapes essentielles, à savoir, la fragmentation et l'allocation, suivies éventuellement d'une optimisation locale [CMVN94]. Les auteurs [NB99] se sont inspirés de ce qui avait été fait pour les bases de données relationnelles [OV91]. Ils ont proposé une architecture pour un entrepôt distribué, basée sur l'architecture ANSI/SPARC [TK78] et suggéré un algorithme adapté de Özsu et al. [OV91] pour fragmenter la table des faits.

2.5.11 La transparence de la fragmentation

Lorsqu'une relation est fragmentée en plusieurs fragments, le système doit assurer la transparence aux utilisateurs, c'est à dire cacher aux utilisateurs le fait que les données sont fragmentées. L'utilisateur ne manipule que des relations conceptuelles non partitionnées. Dès que l'utilisateur pose une requête, ces requêtes doivent être automatiquement transposées sur les fragments, en identifiant les fragments valides. Le même processus d'identification est réalisé lors des opérations de mise à jour. Ce problème a jusqu'à présent été peu pris en compte par les chercheurs.

2.6 Le groupement

Le groupement de données (clustering dans la terminologie anglo-saxonne) consiste à stocker les données (n-uplets ou objets) logiquement liées (c'est à dire, susceptibles d'être utilisées fréquemment ensemble) proches les unes des autres en mémoire secondaire. L'avantage de cette technique est que lorsqu'un objet est chargé depuis le disque, tous les objets liés sont chargés en même temps en mémoire primaire. En conséquence, un groupement judicieux des objets minimise en fait non seulement le nombre de pages disques à lire, mais aussi le nombre d'objets non pertinents chargés en mémoire primaire. Dans les systèmes relationnels, de telles optimisations sont

principalement destinées à accélérer les opérations de jointure, grâce à la création de clusters [Dar99]. Dans le cadre des bases de données objet, le groupement est plus complexe compte tenu des caractéristiques des modèles objet (lien de composition, hiérarchie d'héritage, etc.) [Dar99].

Dans les entrepôts, peu de travaux ont été réalisés sur le groupement. Un seul article traite le problème [JLS99a]. Il s'agit de Jagadish et al. [JLS99a] qui ont considéré le problème de regrouper les enregistrements de la table des faits, afin de minimiser le nombre entrées-sorties d'exécution d'un ensemble de requêtes. Sélectionner les clusters optimaux est un problème combinatoire compte tenu des hiérarchies définies sur les attributs des tables. Les auteurs ont proposé des heuristiques de groupement basés sur les courbes Z et les courbes de Hilbert.

Le problème de regroupement a été largement étudié dans le data mining. Le but de cette technologie est d'extraire des données pour permettre à une entreprise d'améliorer ses fonctions de soutien au marketing, aux ventes et au service client à travers une meilleure compréhension de ses clients. Parmi les techniques les plus utilisées dans le data mining nous trouvons le regroupement, qui consiste à segmenter une population hétérogène en un certain nombre de sous-groupes plus homogènes, ou clusters. Les enregistrements sont en fait regroupés en fonction d'une similitude mutuelle. L'analyse des clusters est souvent préalable à une autre forme d'exploitation des données. Par exemple, pour une étude de segmentation de marché, au lieu de chercher une règle comme "à quel type de promotion les clients répondent le mieux?", il est possible de diviser la base de données client en clusters de personnes ayant les mêmes habitudes de consommation avant de poser la question pour chaque cluster.

Après avoir défini le concept de groupement, nous avons quelques remarques à présenter:

- La fragmentation et le groupement sont orthogonaux, le premier est réalisé pendant la conception logique de l'entrepôt et le deuxième pendant la conception physique.
- Les deux concepts ont le même objectif, à savoir regrouper les objets afin de minimiser le nombre des entrées-sorties.
- Les fragments peuvent être considérés comme des unités de groupement, si tous les n-uplets d'un fragment sont stockés de façon contiguë sur le disque.

2.7 Incidences pratiques

Les problèmes et les approches considérés dans ce chapitre et dans ce travail ont des incidences pratiques importantes et ont suscité l'intérêt des éditeurs et des utilisateurs de SGBD. Dans cette section nous explicitons certaines de ces incidences, [BKM00a]. Nous indiquons en particulier comment certaines de ces approches ont été incorporées dans ORACLE 8i et Microsoft SQL server.

2.7.1 Partitionnement des données

ORACLE 8i incorpore le partitionnement horizontal pour gérer les grands objets (tables, vues matérialisées, index) et les décomposer en parties plus petites

appelées partitions. *ORACLE 8i* propose plusieurs méthodes de partitionnement : le partitionnement par intervalles, le partitionnement par hachage, le partitionnement composite, et le partitionnement orienté jointure. Dans le partitionnement par intervalles, les données dans la table (la vue ou l'index) sont partitionnées relativement à des intervalles de valeurs. Dans le partitionnement par hachage les données sont partitionnées relativement à une fonction de hachage. Dans le partitionnement composite les données sont partitionnées par intervalles et ensuite subdivisées en utilisant une fonction de hachage. Finalement dans le partitionnement orienté jointure, une opération de jointure est divisée en jointures plus petites qui sont exécutées en séquence ou en parallèle. Pour utiliser ce partitionnement les deux tables doivent être équi-partitionnées.

2.7.2 Vues matérialisées

ORACLE 8i incorpore un processus de réécriture de requête qui transforme une commande SQL de telle façon qu'elle puisse accéder aux vues matérialisées. Cet outil de réécriture permet de réduire significativement le temps de réponse pour des requêtes d'agrégation ou de jointure dans les grandes tables des entrepôts. Quand une requête cible une ou plusieurs tables de base pour calculer un agrégat (ou pour réaliser une jointure) et qu'une vue matérialisée contient les données requises, l'optimiseur d'ORACLE peut réécrire la requête d'une manière transparente pour exploiter la vue, et procurer ainsi un temps de réponse plus court. Si les données requises ne sont pas disponibles dans une unique vue matérialisée, mais dans plusieurs, l'administrateur doit définir un index de jointure couvrant les tables de base pour optimiser la requête. Il n'est pas certain que les administrateurs disposent des outils adéquats pour identifier les situations pour lesquelles ces index sont intéressants.

2.7.3 Interaction entre les index et les vues

Ce problème a reçu une grande attention de la part des industriels et des utilisateurs. Récemment le DEM (Data Management, Exploration Mining Group) de Microsoft Research a présenté des solutions pour sélectionner automatiquement un ensemble approprié de vues matérialisées et d'index pour une base de données relationnelle. Cette sélection passe par l'énumération de tous les index et vues pouvant contribuer à l'exécution d'un ensemble de requêtes. Une réduction du nombre de candidats est effectuée ensuite par l'optimiseur de requêtes (via son modèle de coût). Un ensemble final d'index et de vues matérialisées est alors proposé.

D'une manière plus générale, les outils et les techniques développés dans ce chapitre constituent un ensemble cohérent sur lequel peut s'appuyer valablement l'administrateur d'un entrepôt, non seulement pour concevoir l'entrepôt, mais aussi pour l'optimiser en tenant compte des changements dans les requêtes des utilisateurs. Un entrepôt convenablement conçu exécutera les requêtes des usagers plus rapidement tout en facilitant l'actualisation des données. Ces facilités contribueront à améliorer la compétitivité de l'entreprise et faciliteront l'incorporation des changements dans le comportement des usagers et des changements dans l'environnement de l'entreprise.

2.8 Conclusion

Dans cette partie, nous avons non seulement présenté les principales techniques d'optimisation des requêtes utilisées dans les entrepôts de données, mais également décrit les problèmes liés à chaque technique. Dans les prochaines sections, nous apportons des solutions à quelques problèmes concernant: la définition des index de jointure pour les vues matérialisées, la gestion et la distribution de l'espace entre les vues et les index, la fragmentation horizontale.

Chapitre 3

Les index de jointure pour les vues matérialisées

3.1 Introduction

À partir des données d'un entrepôt, de nombreuses analyses (e.g., data mining) peuvent être réalisées afin d'améliorer la productivité et la qualité des services offerts aux utilisateurs de l'entrepôt. Ces analyses sont effectuées en examinant des données extraites de l'entrepôt à l'aide des requêtes décisionnelles. Ces dernières sont très complexes (contenant de multiples jointures, opérations de groupement, etc.) et, par conséquent, peuvent demander un temps de calcul élevé. Les décideurs au niveau des entreprises exigent un temps de réponse raisonnable afin de prendre rapidement des décisions. Pour satisfaire cet objectif, l'entrepôt matérialise un ensemble de vues (persistantes) définies sur des tables sources, qui seront stockées et périodiquement rafraîchies pour refléter les modifications survenues au niveau des sources. La présence des vues accélère l'exécution de *certaines* requêtes. Les requêtes en question sont appelées requêtes bénéficiaires. Cette accélération est due au fait que ces requêtes n'accèdent pas aux tables sources (c'est l'avantage principal des vues matérialisées).

La principale caractéristique des requêtes bénéficiaires est que leur résultat coïncide exactement ou partiellement avec des vues matérialisées. En effet, le résultat d'une requête peut être une vue entière, une projection d'une vue, une sélection d'une vue, ou une jointure de plusieurs vues. Il faut noter que nous ne pouvons pas matérialiser toutes les vues possibles pour satisfaire toutes les requêtes (actuelles et futures) définies sur un entrepôt. Cette contrainte est due à la limitation des ressources, e.g., l'espace disque destiné au stockage des vues matérialisées, ou le coût de leur maintenance. En conséquence, certaines requêtes ne retirent aucun bénéfice des vues matérialisées. Pour les exécuter, on doit accéder aux tables sources, et cette exécution peut être très coûteuse.

La sélection des vues matérialisées dans un entrepôt joue un rôle important dans la réduction du coût d'évaluation des requêtes. Une bonne sélection doit maximiser le nombre des requêtes bénéficiaires. En effet, la sélection des vues à matérialiser représente un enjeu important pour les administrateurs des entrepôts. Cet enjeu a motivé chercheurs [BPT97, Gup99, GM95, MK00, YKL97] et industriels [BDD+98, SSN00] pour développer des algorithmes de sélection offrant de bonnes performances

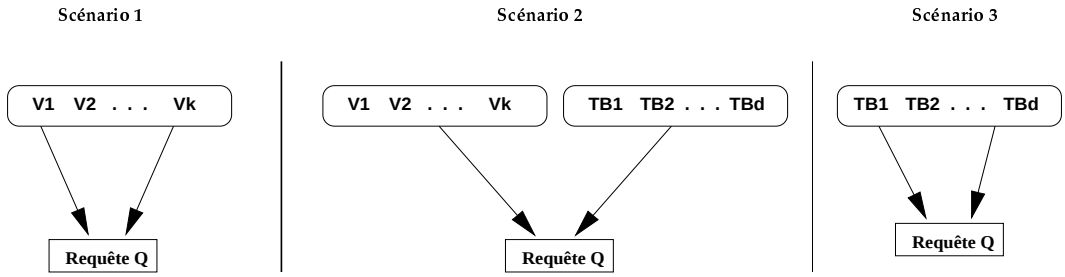


FIG. 3.1: Les trois scénarios d'exécution d'une requête

(voir Chapitre 2).

Une fois les vues sélectionnées, toutes les requêtes décisionnelles définies sur l'entrepôt doivent être réécrites en fonction de ces vues et des tables de base. Cette réécriture permet de réduire le coût des requêtes [SDJL96, BDD+98].
Plus formellement, soit Q une requête décisionnelle définie sur un schéma d'un entrepôt ayant un ensemble de d tables de base dénoté par $\mathcal{TB} = \{TB_1, TB_2, ..., TB_d\}$. Supposons que, sur ce schéma, un ensemble de vues matérialisées $V = \{V_1, V_2, ..., V_k\}$ ait été sélectionné. L'optimiseur doit prendre en compte ces vues matérialisées pour trouver le meilleur plan d'exécution de la requête Q. Trois scénarios se présentent à l'optimiseur pour réaliser sa tâche (voir Figure 3.1):

- **Scénario 1:** La requête Q est exécutée en utilisant seulement des vues matérialisées.
- **Scénario 2:** La requête Q est exécutée en utilisant des vues matérialisées et des tables de base,
- **Scénario 3:** La requête Q est exécutée en utilisant des tables de base seulement.

Si nous associons à chaque scénario un coût (calculant le nombre des entrées-sorties pendant l'évaluation de la requête Q, par exemple), c'est-à-dire le coût C_1 (pour le scénario 1), le coût C_2 (pour le scénario 2), et le coût C_3 (pour le scénario 3), nous pouvons déclarer d'une manière intuitive que:

$$C_1 << C_2 << C_3 \tag{3.1}$$

Généralement, cette inégalité n'est pas toujours satisfaite comme nous allons le voir dans la section 3.9.

Certes, la présence des vues matérialisées a un effet positif sur la réduction du coût d'exécution des requêtes, mais sans techniques d'indexation, ce coût reste toujours important [BKS00, LQA97, SSN00]. Étant donné que les vues peuvent être stockées sous forme de tables, les techniques d'indexation des tables peuvent être facilement adaptées aux vues. La définition de ces index dépend du scénario.

Les index dans le premier scénario Les index sont définis seulement sur les vues. Deux types d'index sont possibles: (1) *les index simples* et (2) *les index de jointure*.

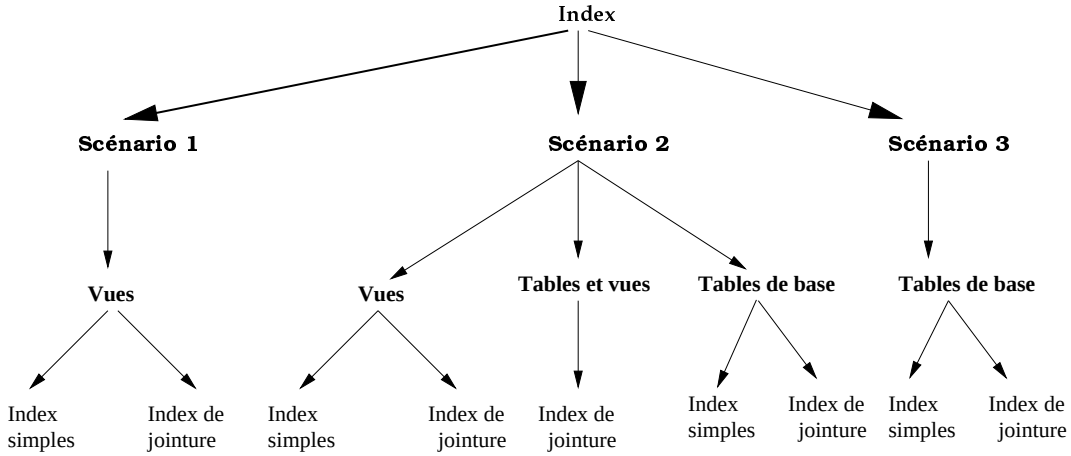


FIG. 3.2: L'indexation dans les trois scénarios

1. Un index simple peut être défini sur une ou plusieurs colonnes d'une vue. Cette indexation peut être réalisée en utilisant les mêmes techniques que celles dédiées aux tables (e.g., arbre B^+, index de projection, index binaire, etc. (voir chapitre 2)).
2. Un index de jointure peut être défini sur deux ou plusieurs vues. Si l'opération de jointure est définie sur deux vues seulement, l'index de jointure de Valduriez et al. [Val87] peut être utilisé. Par contre, si elle concerne plus de deux vues, il n'existe pas a priori d'index de jointure performant [BKL00].

Les index dans le deuxième scénario Les index simples peuvent être définis soit sur les vues soit sur les tables de base. Les index de jointure peuvent être définis entre: (1) des vues seules, pour accélérer par exemple des jointure de plusieurs vues, (2) des tables seules, ou (3) entre des vues et des tables. Les observations sur les index de jointure décrites dans le paragraphe précédent sont valides également dans ce scénario.

Les index dans le troisième scénario Ce cas ressemble au premier, mais au lieu d'indexer les vues, on indexe les tables. Les index simples sont définis sur les tables. L'application des index de jointure dépend du schéma de l'entrepôt. Si ce dernier est modélisé par un schéma en étoile, l'index de jointure en étoile défini par *Redbrick* [Sys97] peut être utilisé. Il peut réduire le coût d'exécution de toute requête en étoile [1]. Mais si l'entrepôt de données est modélisé par un autre schéma que le schéma en étoile, il n'existe pas d'index de jointure utilisable pour satisfaire les requêtes joignant plusieurs tables [JLS99b].

La Figure 3.2 résume les techniques d'indexation pour chaque scénario.

3.2 Objectifs visés

La conception d'un entrepôt de données est réalisée en trois étapes principales, à savoir: (i) *le processus de sélection des vues matérialisées*, (ii) *le processus de*

[1] Cette appellation est destinée aux requêtes définies sur un schéma en étoile (voir chapitre 2)

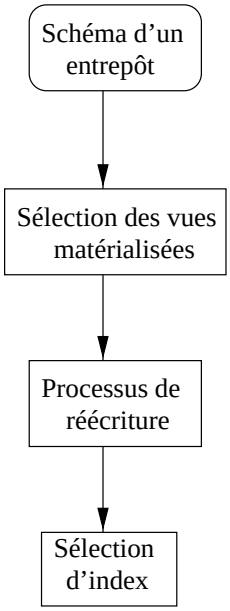


FIG. 3.3: Les étapes de conception physique de l'entrepôt

réécriture des requêtes en fonction des vues matérialisées, et (iii) *le processus de sélection d'index* (Figure 3.3). Ces étapes partagent le même objectif: *optimiser la performance des requêtes décisionnelles* [SDJL96, CN98, YKL97].

En examinant la littérature, nous constatons que la première et la deuxième étape ont été largement étudiées par les communautés académique et industrielle (voir Chapitre 2). Dans ce chapitre, nous ne développons pas de nouveaux algorithmes pour la sélection des vues à matérialiser, ou la réécriture des requêtes en fonction des vues, mais nous utilisons les algorithmes existants.

Pour la sélection des vues matérialisées, nous étendons l'algorithme de Yang et al. [YKL97]. Notre choix s'est porté sur cet algorithme pour deux raisons:

1. Il est bien adapté au contexte ROLAP et coïncide donc avec notre étude.
2. Il a été développé par le groupe de recherche avec lequel j'ai collaboré pendant deux ans (Department of Computer Science, Hong Kong University of Science and Technology).

La description de cet algorithme fera l'objet de la section 3.3.

Pour le processus de réécriture, nous utilisons l'approche adopté par [SDJL96, BDD+98]. Nous illustrons ce processus dans la section 3.4.

Dans ce chapitre, nous nous concentrons sur la troisième phase de la conception d'un entrepôt, la sélection d'index, et plus précisément sur la définition et la sélection d'index de jointure en présence des vues matérialisées. Ce type d'index n'a pas suscité beaucoup d'intérêt de la part des chercheurs.

Le reste du chapitre décrit notre approche de sélection des index en présence des vues matérialisées. La section 3 décrit en détail l'algorithme de sélection des vues matérialisées de Yang et al. [YKL97]. La section 4 présente le processus de réécriture des requêtes en fonction des vues matérialisées. À travers la section 5, nous tirons un bilan sur l'insuffisance des techniques d'optimisation des jointures existantes dans les entrepôts de données. Cette section introduit les index de graphe de jointure et

justifie leur intérêt. La section 6, présente une stratégie d'exécution des requêtes en présence des index de graphe de jointure. La section 7 décrit un modèle de coût évaluant le coût d'exécution d'un ensemble de requêtes sur un schéma indexé par les index proposés. La section 8 donne la complexité de la sélection des index de graphe de jointure. Elle formule le problème de sélection des index et propose trois algorithmes: un algorithme exhaustif et deux algorithmes gloutons pour sélectionner les index optimaux ou quasi-optimaux pour un ensemble de requêtes. Finalement la section 9 illustre l'utilité de ces index à travers des expérimentations.

3.3 La sélection des vues matérialisées

Dans cette section, nous décrivons l'algorithme de Yang et al. [YKL97], que nous appelons *sélect_vue* [2]. La description de cet algorithme nécessite le rappel de quelques concepts et définitions.

3.3.1 Concepts et définitions

3.3.1.1 Bases de données relationnelles

Les notions de base du modèle relationnel sont les suivantes: univers, attribut, schéma, domaine, relation [Cod70]. Pour modéliser une application réelle, on détermine d'abord les propriétés pertinentes auxquelles on s'intéresse. Dans la terminologie relationnelle, ces propriétés sont appelées *attributs*.

Un univers est un ensemble fini, non vide, noté $U = \{A_1, A_2, ..., A_x\}$, où chaque élément est appelé attribut. À chaque attribut A_i de U, est associé un ensemble appelé domaine de A_i, noté par $Dom(A_i)$; chaque élément de ce domaine est considéré comme une valeur possible de la propriété représentée par l'attribut.

Nous considérons des requêtes SQL de la forme suivante [SDJL96]:

```
Q:  SELECT   Sel(Q)        % Q: requête
    FROM     Table(Q)
    WHERE    Cond(Q)
    GROUPBY  Groups(Q)
```

où Sel(Q), Table(Q), Cond(Q) et Groups(Q) représentent respectivement l'ensemble des colonnes (agrégation ou non) de la clause SELECT, l'ensemble des tables à manipuler, la condition de sélection, et un sous-ensemble des colonnes de Sel(Q) servant au regroupement.

3.3.1.2 L'algèbre relationnelle

L'algèbre relationnelle a été introduite par Codd [Cod70] comme une collection d'opérations formelles agissant sur des relations et produisant des relations en résultats [Cod70]. Codd a initialement introduit des opérations, dont certaines peuvent être composées à partir d'autres. Ces opérations sont: la sélection (σ), la projection (Π), la jointure ($\bowtie$), l'union ($\cup$), la différence, le produit cartésien [Gar83, EN94].

[2] Rappelons que cet algorithme appartient à la catégorie statique de PSV: il optimise un ensemble de requêtes a priori connues (voir chapitre 2)

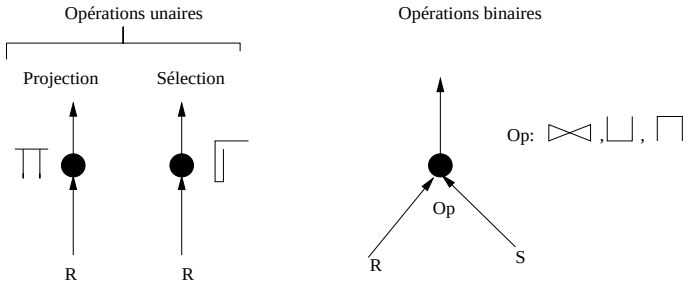


FIG. 3.4: Les opérateurs relationnels

Une requête exprimée sous forme d'une expression algébrique peut être représentée par un arbre relationnel. La figure 3.4 représente les opérateurs relationnels ainsi que leur représentation graphique.

Définition 15 (Un arbre relationnel) *Soit Q une requête exprimée sous forme algébrique E_Q. L'arbre relationnel de la requête Q est un arbre $A : (N, V)$ dont les noeuds N correspondent à:*

- *Le nom de la requête dénoté par Q et représentant le sommet de cet arbre,*
- *Les noms des relations à partir desquelles la requête Q est définie. Ces noeuds représentent les feuilles de l'arbre, et*
- *Les noms des résultats intermédiaires résultant de l'évaluation des opérations successives.*

Les arcs désignent les opérandes de chaque opérateur. Les expressions suivantes: graphe de requête, graphe de calcul, arbre algébrique, renvoient au même concept. Au cours de cette thèse, nous utilisons indifféremment ces termes.

Exemple 8 *Soit Q la requête définie en SQL sur le schéma en étoile de la Figure 2.5 de la façon suivante:*

```
SELECT   PID, SUM(V.ventes_sales)
FROM     CLIENT, VENTES
WHERE    CLIENT.Cid = VENTES.Cid     % prédicat de jointure
AND      CLIENT.Etat ='IL'           % prédicat de sélection
GROUP BY PID
```

L'arbre algébrique de cette requête est illustré par la Figure 3.5. Il contient deux feuilles représentant la table de dimension CLIENT et la table des faits VENTES, et trois noeuds intermédiaires représentant les résultats de l'opération de jointure, de sélection et de projection.

Tout graphe de requête doit respecter les règles suivantes:

1. Un sommet admet deux prédécesseurs s'il représente une relation produite par un opérateur relationnel binaire.
2. Un sommet n'admet qu'un seul prédécesseur s'il représente une relation produite par un opérateur relationnel unaire.
3. Quel que soit le type de l'opérateur, un sommet intermédiaire n'admet qu'un seul successeur.

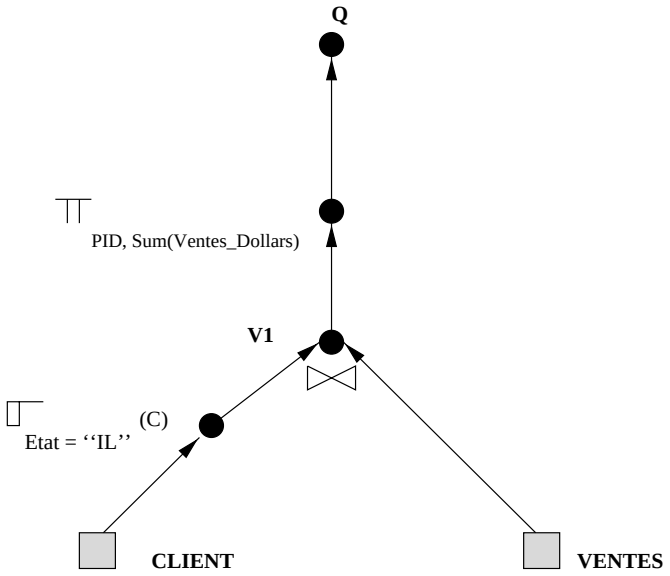


FIG. 3.5: L'arbre algébrique correspondant à la requête Q

4. Un sommet représentant une table de base n'admet pas de prédécesseur, les tables de base sont les feuilles du graphe. Les feuilles sont les seuls noeuds à admettre plusieurs successeurs car dans la même expression algébrique, on peut définir plusieurs variables sur la même relation.
5. Il existe un sommet unique dans N (l'ensemble des noeuds de graphe) n'admettant aucun successeur; il s'agit de la racine de l'arbre à laquelle est attachée le résultat et qui porte le nom de la requête.

3.3.2 Motivation de l'algorithme sélect_vue

Cette section présente un aperçu progressif des aspects clés de l'algorithme *sélect_vue*. Pour ce faire, considérons le schéma en étoile de la Figure 2.5. Supposons que nous ayons deux requêtes décisionnelles Q (définie dans l'exemple 8) et Q' dont la description est:

```
SELECT   PID, SUM(VENTES.ventes_dollar)
FROM     CLIENT, TEMPS, VENTES
WHERE    CLIENT.Cid  = VENTES.Cid
AND      TEMPS.Tid   = VENTES.Tid
AND      TEMPS.Mois  = ''Mars''
AND      CLIENT.Etat ='IL'
GROUP BY PID
```

La Figure 3.6 donne l'arbre algébrique de chaque requête. Pour atteindre un bon temps de réponse, nous pouvons matérialiser quelques noeuds intermédiaires de chaque arbre. Notons que le résultat intermédiaire V_1 de la requête Q est équivalent à celui de la requête Q'. Ce noeud est appelé *sous-expression commune* [Sel88]. Étant donné que V_1 est partagé par les deux requêtes, il sera un bon candidat pour la matérialisation. S'il est matérialisé dans les deux plans d'exécution indépendemment (cas 1), le coût des deux requêtes sera inférieur à celui obtenu par l'accès aux tables

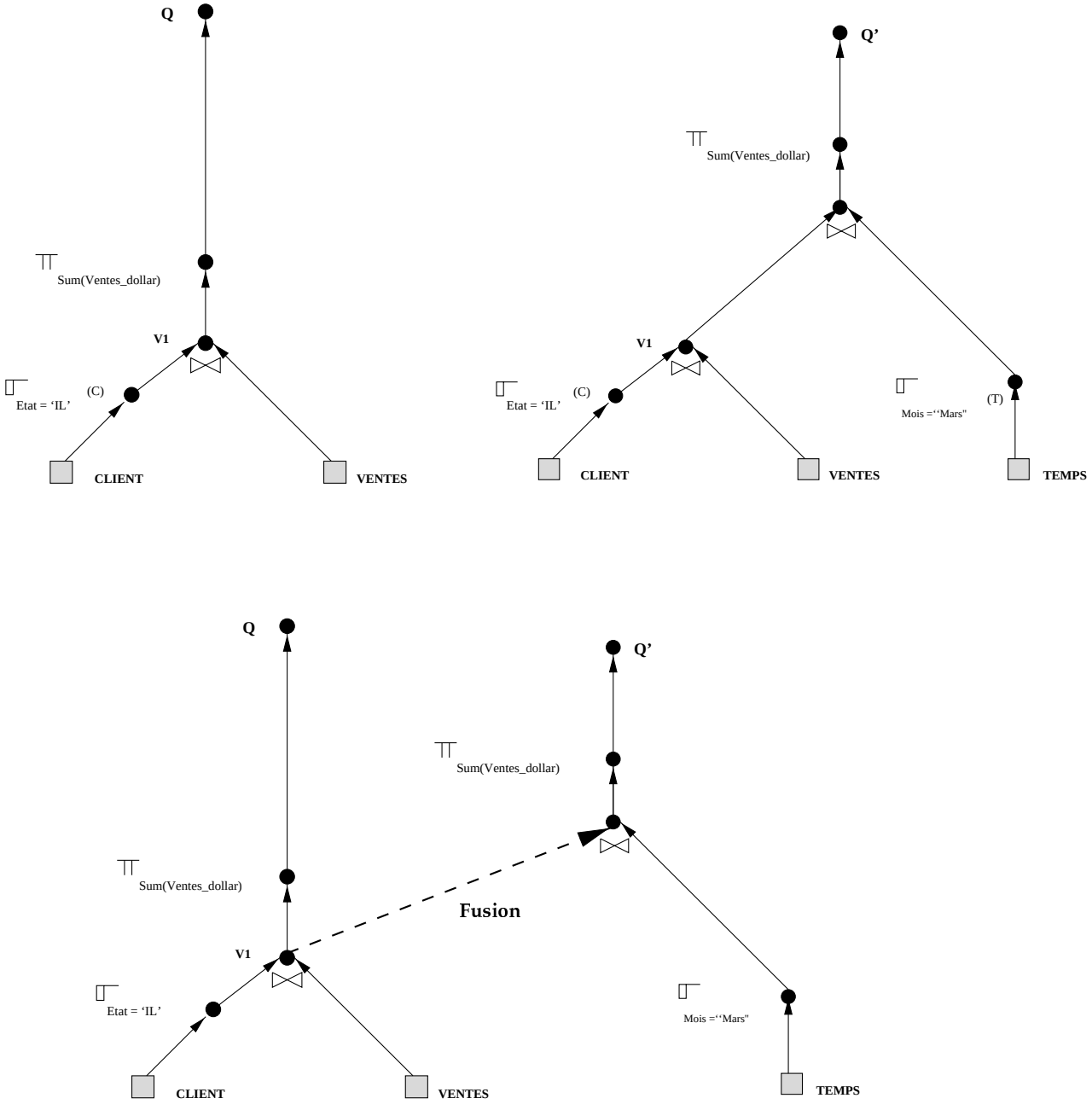


FIG. 3.6: Les arbres algébriques individuels

de base seulement (cas 0).

Maintenant supposons que les deux plans correspondants à Q et Q' soient fusionnés en utilisant la sous-expression commune V_1 en un seul graphe appelé plan multiple d'exécution des vues (PMEV) (Figure 3.6). Si le noeud V_1 est matérialisé (cas 2), nous pouvons conclure que:

1. Le coût d'exécution des requêtes Q et Q' dans le cas 2 est inférieur à celui des cas 0 et 1,
2. Lorsque les deux plans sont fusionnés, le coût de maintenance de la vue V_1 est inférieur à celui de la somme des vues non fusionnées.

L'idée principale sur laquelle repose l'algorithme *sélect_vue* est *la fusion des sous-expressions communes* trouvées dans les graphes de requêtes. Cette idée se situe dans la lignée des travaux réalisés sur l'optimisation des requêtes dans les bases de données relationnelles traditionnelles en utilisant des techniques de traitement de requêtes multiples (multiple query processing) [Sel88].

3.3.3 Les étapes principales de sélect_vue

Yang et al. [YKL97] ont formulé le PSV comme suit:

- Soit un entrepôt de données modélisé par un schéma relationnel ayant d tables de dimensions et une table des faits,
- Soit $Q = \{Q_1, Q_2, ..., Q_n\}$ un ensemble de requêtes d'interrogation (les plus fréquentes) avec leur fréquence d'accès $\{f_1, f_2, ..., f_n\}$,
- Soit $U = \{U_1, U_2, ..., U_{n'}\}$ un ensemble de requêtes de mise à jour (les plus fréquentes) avec leur fréquence d'accès $\{g_1, g_2, ..., g_{n'}\}$.

Le PSV consiste à sélectionner un ensemble de vues minimisant la somme des coûts d'évaluation de requêtes et de maintenance des vues sélectionnées.

L'algorithme *sélect_vue* se caractérise par quatre étapes, à savoir, (1) *la construction de l'arbre algébrique de chaque requête*, (2) *la génération de PMEV*, (3) *l'attribution des coûts à chaque noeud de graphe*, et (4) *la sélection des vues.*

1. La construction des arbres algébriques: Cette étape consiste à construire un arbre algébrique pour chaque requête Q_i ($1 \leq i \leq n$). Notons qu'une requête Q_i peut avoir plusieurs arbres algébriques. Parmi ces arbres, il en existe un qui est *optimal* (cette optimalité est évaluée en fonction d'un modèle de coût). L'existence de plusieurs arbres algébriques est due à la présence de propriétés sur des opérations algébriques [Ull89] (appelées règles de transformation des arbres [Gar83]). Ces règles sont au nombre de huit: (1) *la commutativité des jointures*, (2) *l'associativité des jointures*, (3) *le groupement des restrictions*, (4) *la commutativité des restrictions et des projections*, (5) *la commutativité des restrictions et des jointures*, (6) *la commutativité des restrictions et des unions, ou des restrictions et des différences*, (7) *la commutativité des projections et des jointures*, et (8) *la commutativité des projections et des unions.*

   Ces règles peuvent servir dans plusieurs cas de figure:

   (a) Démontrer qu'une expression algébrique est équivalente à une autre

   (b) Réécrire une expression sous une autre forme

ID	Formulation en SQL	Sélectivité	Freq
1	SELECT sum(V.Ventes_Dollars) FROM CLIENT C, VENTES V WHERE C.Etat = "IL" AND C.CID = V.CID GROUP BY CID	$\frac{1}{50}$	15
2	SELECT sum(V.Ventes_Dollars), sum(S.Coût_Unitaire) FROM PRODUIT, VENTES V, TEMPS T WHERE V.PID = P.PID AND V.TID = T.TID AND P.Type_paquet = "Boîte" AND T.Mois = "Mars" GROUP BY PID	$\frac{1}{25}$ $\frac{1}{36}$	5
3	SELECT sum(V.Ventes_Dollars), sum(S.Coût_Unitaire) FROM VENTES V, PRODUIT P, CLIENT C, TEMPS T WHERE V.PID = P.PID AND V.CID = C.CID AND V.TID = T.TID AND P.Type_paquet = 'Boîte" AND T.Mois = "Mars" AND C.Sexe = 'M' GROUP BY TID	$\frac{1}{25}$ $\frac{1}{36}$ $\frac{1}{2}$	5
4	SELECT sum(V.Vente_Dollars) FROM VENTES V, CLIENT C, TEMPS T WHERE V.TID = T.TID AND V.CID = C.CID AND C.Etat = "IL" AND T.Mois = "Mars" GROUP BY CID	$\frac{1}{50}$ $\frac{1}{36}$	10
5	SELECT sum(S.Vente_Dollars) FROM VENTES V, CLIENT C, PRODUIT P WHERE V.PID = P.PID AND V.CID = C.CID AND C.Sexe = 'M' AND P.Type_paquet = "Boîte" GROUP BY PID	$\frac{1}{2}$ $\frac{1}{25}$	15

FIG. 3.7: L'ensemble des requêtes utilisées dans l'évaluation de la fragmentation

(c) Générer toutes les expressions équivalentes à une autre.

Appliquées aux arbres algébriques, ces règles vont permettre leur restructuration, afin d'obtenir un plan d'exécution optimal pour chaque requête.

2. La génération du PMEV: Une fois les graphes optimaux obtenus, les sous-expressions communes sont identifiées, puis les graphes de requêtes ayant des sous-expressions communes sont fusionnés en PMEV [3].

   Un PMEV est un graphe orienté, acyclique et étiqueté, dont la structure est définie par: (N, A), où N et A représentent les ensembles de noeuds et d'arcs du graphe dont les constructions se réalisent de la façon suivante:

   - Pour chaque opération dans l'arbre algébrique de la requête Q_i $(1 \leq i \leq n)$ créer un noeud.
   - $\forall v \in N$, $T(v)$ est la table générée par le noeud v. $T(v)$ pourra être une table de base, un résultat intermédiaire en exécutant une requête, ou le résultat final d'une requête.
   - Pour tout noeud feuille v, $T(v)$ correspond à la table de base. Soit L l'ensemble de feuilles.
   - Pour tout noeud racine v (le noeud qui n'a pas d'arc sortant du noeud), $T(v)$ correspond au résultat de la requête globale. Soit R l'ensemble des noeuds racine.
   - Si la table de base ou le résultat intermédiaire $T(v)$ correspondant à un noeud v est demandé pour un autre traitement au niveau d'un noeud v', introduire un arc $v \longrightarrow v'$
   - Pour tout noeud v, soit $S(v)$ l'ensemble des noeuds origines des arcs pointant vers v. Si $v \in L; S(v) = \phi$. Soit S^*v l'ensemble des descendants de v.
   - Pour $v \in N$, $C_{Q_i}(v)$ est le coût d'exécution de la requête Q_i accédant à $T(v)$, lorsque $T(v)$ est matérialisée, $C_{U_j}(v)$ est le coût de maintenance de $T(v)$ causé par la requête de mise à jour U_j et basé sur les changements des tables de bases $S*(v) \bigcap L$.

   En fonction des types d'opérations relationnelles (sélection, projection, et opérations ensemblistes), le PMEV peut être divisé en quatre niveaux:

   (a) Le niveau 0 représente seulement les relations de base (feuilles).

   (b) Le niveau 1 contient des noeuds représentant les résultats des opérations de sélection et projection. Nous appelons les noeuds de ce niveau *noeuds de SP* (sélection et projection).

   (c) Le niveau 2 contient les résultats intermédiaires des requêtes résultantes des opérations ensemblistes (comme la jointure, union, etc.). Nous appelons ces noeuds les *noeuds ensemblistes.*

   (d) Le niveau 3 représente les résultats des requêtes.

Définition 16 Une vue de sélection *est définie par l'expression algébrique suivante:* $\sigma_{cl(Y)}(T)$*, où la condition de sélection* $cl(Y)$ *est une expression booléenne définie sur* $Y \subseteq T$*, et* T *étant une table.*

[3] Le PMEV final n'est pas forcément optimal [YKL97]

Définition 17 Une vue de projection *est définie par l'expression algébrique suivante:* $\Pi_X(T)$, *où* $Y \subseteq T$.

Définition 18 Une vue de jointure *est définie par l'expression algébrique suivante:* $T_1 \bowtie T_2 \bowtie ... \bowtie T_p$, *où les tables* T_i $(1 \leq i \leq p)$ *sont définies sur le schéma de l'entrepôt.*

Exemple 9 *Considérons le schéma en étoile de la Figure 2.5 et les cinq requêtes décisionnelles dans la Figure 3.7. Ces dernières sont représentées par cinq arbres algébriques qui seront fusionnés en PMEV (voir Figure 3.8).*

Yang et al. [YKL97] ont montré que les noeuds de niveaux 1 et 2 représentent des vues potentielles. La couche couvrant ces deux niveaux est appelée couche des vues potentielles.

3. Attribution des coûts: Avant de définir ces coûts, quelques notations sont introduites:
   - O_v dénote l'ensemble des requêtes utilisant le noeud v; $O_v = R \cap D*(v)$; avec R représentant l'ensemble des noeuds racines et $D*(v)$ l'ensemble des ascendants de v
   - I_v est l'ensemble des tables de base utilisées pour produire v, $I_v = L \cap S*(v)$; où L est l'ensemble de feuilles et $S*(v)$ est l'ensemble des descendants de v.
   - $\mathcal{V}$ est l'ensemble de vues matérialisées.
   - A chaque noeud est associé un poids statique $w(v)$ représentant le bénéfice si le noeud v est matérialisé. Ce poids est donné par la formule suivante:

$$w(v) = \sum_{Q_i \in O_v} \left(f_i \times C_{Q_i}(v) \right) - \sum_{j \in I_v} \left(g_j \times C_{U_j}(v) \right) \qquad (3.2)$$

     Ce poids est utilisé pour ordonner les noeuds du PMEV, le noeud ayant la plus grande valeur sera privilégié pour la matérialisation.
   - O'_v dénote les requêtes globales qui utilisent v, excluant celles qui peuvent utiliser les ascendants de v déjà sélectionnés. $O'_v = O_v - O_u$, avec $u \in \mathcal{V} \cap D*(v)$
   - C_s dénote le poids dynamique d'un noeud donné par:

$$C_s = \sum_{Q_i \in O'_v} \left(f_i \times (C_{Q_i}(v) - \sum_{u \in S_v \cap \mathcal{V}} C_{Q_i}(v)) \right) - \sum_{r \in I_v} \left(g_r \times C_{U_r}(v) \right). \qquad (3.3)$$

     La première partie de la formule 3.3 représente le bénéfice si v est matérialisée. La deuxième est le coût de maintenance de la vue v.
     $\sum_{u \in S_v \cap \mathcal{V}} C_{Q_i}(v)$ est un facteur dupliqué dans le cas où quelques descendants de v sont déjà choisis pour la matérialisation.

4. la sélection des vues: Cette étape consiste à ordonner les noeuds du PMEV en fonction de leur poids statique $w(v)$; ensuite on calcule leur poids dynamique. Si $C_s > 0$, v est matérialisée (voir l'algorithme 4).

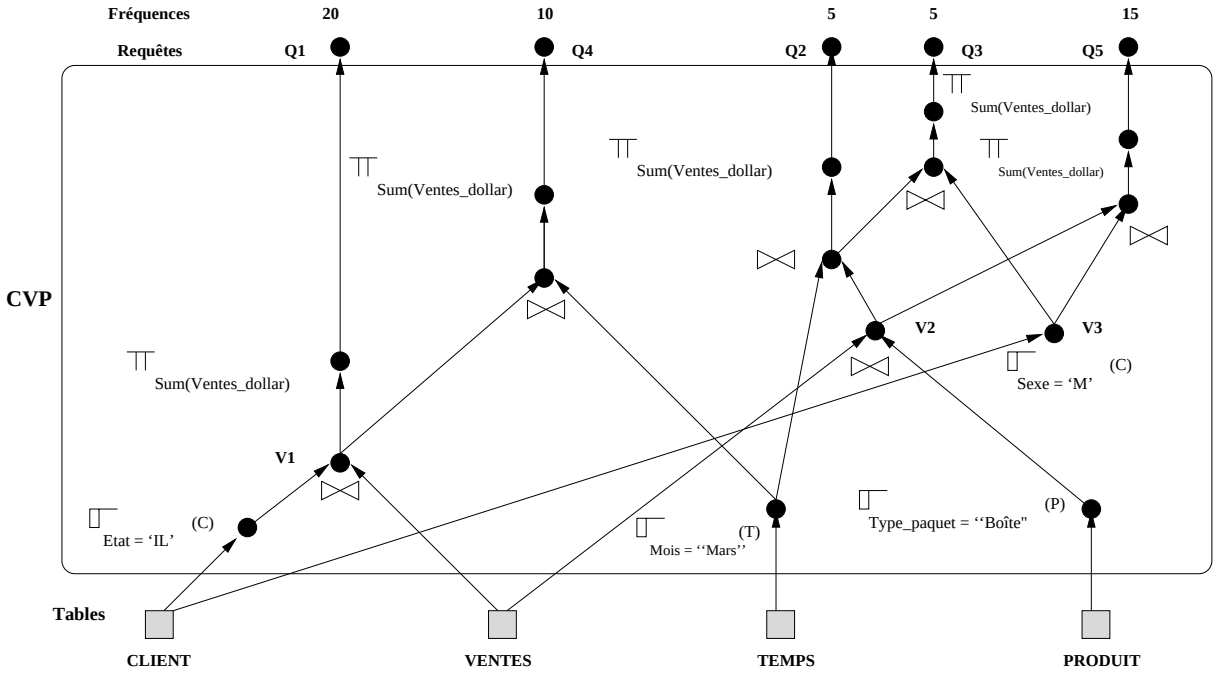


FIG. 3.8: Le PMEV correspondant aux cinq requêtes de la Figure 3.7

Algorithme 4 Les étapes de sélect_vue

1: $V := \phi$
2: Créer la liste LV de tous les noeuds ayant des poids positifs (dans l'ordre descendant)
3: Prendre le premier élément v de LV
4: Calculer O_v, I_v, et C_v
5: Calculer C_s
6: **if** $C_s > 0$ **then**
7: Insérer v dans M
8: Retirer v de LV
9: **else**
10: Retirer v et tous les noeuds listés après v dans LV et qui sont dans la même branche de v
11: **end if**
12: **repeat**
13: étape 3
14: **until** $LV = []$ % []: la liste vide
15: $\forall v \in V$, si $D(v) \subset M$, alors retirer v de V

3.4 Le processus de réécriture des requêtes

Pour que les requêtes retirent un bénéfice des vues matérialisées, elles doivent être réécrites en fonction de ces dernières.

Définition 19 La réécriture d'une requête [SDJL96] *Soient Q et V une requête et une vue matérialisée, respectivement. Supposons que la requête Q utilise la vue V. Une requête Q' est une réécriture de la requête Q si les deux conditions suivantes sont satisfaites:*

- *(i) Q et Q' donnent la même réponse, et*
- *(ii) Q' contient une ou plusieurs occurrences de V dans la clause FROM.*

La réécriture des requêtes ne garantit pas toujours une meilleure performance. Le processus d'exécution des requêtes en présence des vues matérialisées se fait de la manière suivante:
Soit un entrepôt de données modélisé par un schéma ayant k tables. Supposons que nous ayons un ensemble de vues matérialisées $\mathcal{V} = \{V_1, V_2, ..., V_l\}$. Lorsqu'une requête Q est posée sur l'entrepôt, l'optimiseur des requêtes élabore les trois processus suivants (voir Figure 3.9): (1) **la réécriture**, (2) **la génération des plans d'exécution de la requête** Q, et (3) **la sélection du meilleur plan de** Q [BDD+98].

Le processus de réécriture: Lorsqu'une requête est posée sur l'entrepôt, l'optimiseur doit trouver une réponse à la question suivante: existe t-il une réécriture de la requête Q?
Trois réponses sont possibles:

- Pas de réécriture: Cela signifie qu'il n'existe aucune possibilité de réécriture de la requête Q en fonction des vues sélectionnées. Pour exécuter Q, l'optimiseur est donc obligé d'accéder aux tables de base.
- Une réécriture presque parfaite: Dans ce cas, l'optimiseur trouve une réécriture Q' de Q dans laquelle, Q' utilise des vues et des tables de base.
- Une réécriture parfaite: Comme dans le cas précédent, l'optimiseur trouve une réécriture Q' de Q. Pour exécuter Q, l'optimiseur utilise uniquement des vues matérialisées.

La génération des plans : L'optimiseur génère deux plans d'exécution pour la requête Q: dans le premier, la requête Q est exécutée sans utilisation des vues (voir Figure 3.9), et dans le deuxième, la requête Q' est exécutée [4].

La sélection du meilleur plan: L'optimiseur dispose d'un modèle de coût calculant le coût de chaque plan. Le coût est calculé en fonction des statistiques sur les tables de base et les vues (e.g., leurs tailles). Finalement, l'optimiseur sélectionne le plan ayant un coût minimal. Si le plan sélectionné est celui de Q', la requête Q est considérée comme une requête bénéficiaire.

Exemple 10 *Soient V_1 et V_2 deux vues sélectionnées par l'algorithme* sélect_vue *dont les définitions sont:*

[4] Nous supposons qu'il existe une réécriture de la requête Q

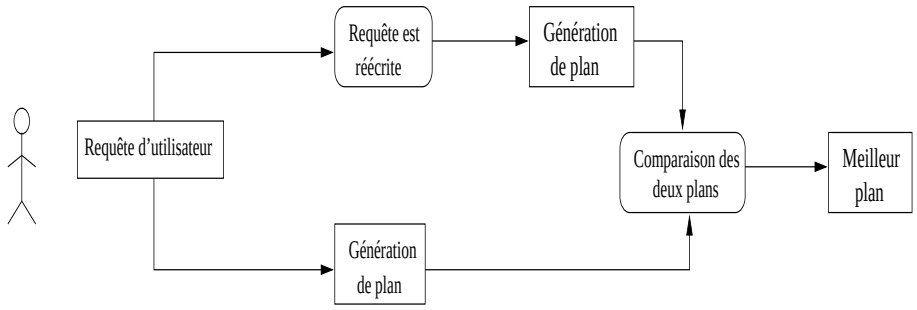


FIG. 3.9: Le processus de réécriture

```
CREATE VIEW V_1
SELECT   *
FROM     CLIENT, VENTES
WHERE    CLIENT.Cid  = VENTES.Cid
AND      CLIENT.Etat =''Illinois''

CREATE VIEW V_2
SELECT   *
FROM     PRODUIT, VENTES
WHERE    PRODUIT.Pid   = VENTES.Cid
AND      PRODUIT.type_paquet =''Boîte''
```

Les cinq requêtes de la Figure 3.7 sont réécrites en fonction de V_1 et V_2 et d'autres tables. Cette réécriture peut être réalisée en utilisant une structure de données appelée matrice de réécriture *($W = w_{ij}$). Les lignes et les colonnes de W représentent les requêtes et les vues (et tables), respectivement. Ses valeurs w_{ij} sont définies comme suit:*

$$\mathbf{w_{ij}} = \begin{cases} 1 \text{ si la requête } Q_i \text{ utilise la vue (table) } j \\ 0 \text{ sinon} \end{cases}$$

Pour exécuter les cinq requêtes, l'optimiseur utilise les deux vues V_1, V_2 et deux tables de dimensions CLIENT et TEMPS (voir la Figure 3.10). À partir de cet exemple, nous constatons qu'il existe une réécriture parfaite pour la requête Q_1 (qui utilise seulement la vue V_1) et des réécritures presque parfaites pour les autres (Q_2, Q_3, Q_4 et Q_5).

$$\begin{array}{c} \\ Q_1 \\ Q_2 \\ Q_3 \\ Q_4 \\ Q_5 \end{array} \begin{array}{c} \begin{array}{cccc} V_1 & V_2 & T & C \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \end{array}$$

FIG. 3.10: La matrice de réécriture

3.5 Le problème des jointures et les index de jointure

Après la réécriture des requêtes en fonction des vues, dans la plupart des cas, l'opération de jointure est toujours présente. Rappelons que cette dernière est une des opérations les plus coûteuses dans les bases de données [LR98]. Deux méthodes principales existent pour évaluer une opération de jointure:

1. L'utilisation des techniques ad-hoc (e.g., la jointure par hachage, la jointure par tri-fusion, la jointure imbriquée, etc.). Il faut noter que ces techniques sont efficaces lorsque le nombre de tables à joindre est égal à 2. Si ce dernier est supérieur à 2, avant de les utiliser, il faut régler le problème d'ordonnancement des jointures (voir chapitre 2).
2. L'utilisation des index de jointure. Il s'agit de structures précalculées qui garantissent généralement une meilleure performance que les techniques ad-hoc [BS95, Val87, LR98]. L'utilisation des index dans les bases de données doit être utilisée soigneusement, car elle pose le problème de leur maintenance. Mais dans les entrepôts de données ce dernier est moins important que dans les bases de données de type OLTP (voir chapitre 2).

Notons que des index de jointure ont été proposés dans les bases de données de type OLTP pour des jointures entre deux tables [Val87]. Dans les entrepôts de données, l'existence de jointures entre n tables ($n > 2$) est fréquente. Prenons l'exemple d'un entrepôt modélisé par un schéma en étoile. La principale caractéristique des requêtes en étoile définies sur ce schéma est leurs multiples (> 2) jointures [Sys97]. Ces dernières doivent être optimisées afin de respecter les délais imposés par les décideurs. Les chercheurs ont d'abord essayé de voir si les méthodes de jointure développées dans les bases de données traditionnelles pouvaient être appliquées. Le résultat n'était pas significatif à cause du problème d'ordonnancement des jointures.

Pour résoudre ce problème, *Red Brick* [Sys97] a développé un nouvel index de jointure appelé index de jointure en étoile.

Définition 20 Un index de jointure en étoile (IJE) *est le résultat de l'opération de jointure des tables de dimensions avec la table des faits en utilisant les clés primaires des tables de dimensions et les clés étrangères de la table des faits.*

Exemple 11 *Un IJE complet* [5] *sur le schéma en étoile de la Figure 2.5 peut être défini de la manière suivante:* (VID, CID, PID, TID)*, où* $VID, CID, PIDetTID$ *représentent les identifiants de la table des faits et des trois tables de dimensions, respectivement. Cet index est le résultat de la requête suivante [Cor97b]:*

```
SELECT   CID, PID, VID, TID
FROM     CLIENT, PRODUIT, VENTES, TEMPS
WHERE    CLIENT.CID = VENTES.CID
AND      PRODUIT.PID = VENTES.PID
AND      TEMPS.TID = VENTES.SID
```

[5] Un IJE complet est un IJE possédant toutes les tables de dimensions et la table des faits dans sa définition (voir chapitre 2)

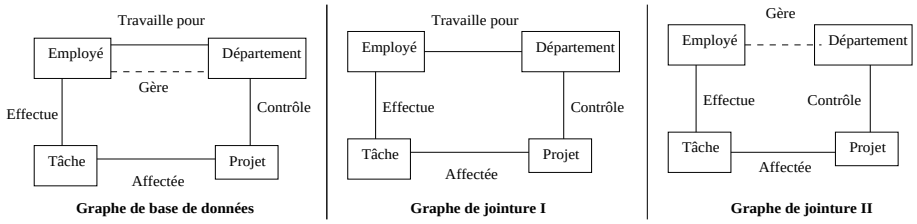


FIG. 3.11: Exemple de graphe d'une base de données

D'une manière similaire, un IJE partiel (VID, CID) [6] *peut être défini entre la table des faits VENTES et la table de dimension CLIENT. Il est le résultat de la requête ci-dessous:*

```
SELECT   CID, VID
FROM     CLIENT, VENTES
WHERE    CLIENT.CID = VENTES.CID
```

Il faut noter que l'avantage principal de cet index est qu'il peut optimiser n'importe quelle opération de jointure définie sur un schéma en étoile (voir les avantages et les inconvénients de ces index dans le chapitre 2).

En conséquence, la présence des vues matérialisées rend ces index de jointure inefficaces. Pour exécuter une requête ayant plusieurs jointures de vues, nous devons avoir des techniques d'indexation offrant une meilleure performance que les techniques développées pour les systèmes OLTP. Ces techniques feront l'objet de ce chapitre.

Avant de décrire ces techniques d'indexation, quelques concepts et définitions sont nécessaires.

Définition 21 Un graphe d'une base de données *[MW89] est un graphe connecté où chaque noeud représente une table et où une arête entre deux noeuds* T_i *et* T_j *représente une possibilité de jointure entre ces deux noeuds.*

Pour un graphe d'une base de données donné, nous pouvons avoir plus d'une arête entre deux noeuds comme le montre la Figure 3.11, où nous avons le cas de deux jointures entre les tables *Employé* et *Département* à représenter. En effet, une jointure relie un employé au département où il travaille, et l'autre relie l'employé au département où il/elle est gérant.

Dans l'environnement des entrepôts de type ROLAP (c'est-à-dire, les entrepôts modélisés par un schéma en étoile ou un schéma en flocon de neige), un graphe d'une base de données possède une et une seule arête entre deux noeuds. Dans ce cas, il est appelé un graphe d'entrepôt de données.

Définition 22 Un graphe de jointure *est un graphe d'une base de données tel qu'il existe exactement une condition de jointure entre deux tables. Le nombre de noeuds d'un graphe de jointure est appelé la cardinalité de ce graphe.*

Dans les entrepôts de données sans vues matérialisées, le graphe de jointure et le graphe d'entrepôt de données sont équivalents (c'est-à-dire, un graphe de jointure est un graphe d'entrepôt de données et vice versa).

[6] Un IJE partiel est un IJE possédant certaines tables de dimensions et la table des faits dans sa définition (voir chapitre 2)

La présence des vues matérialisées change la structure du graphe d'entrepôt de données, car les vues peuvent remplacer les tables de dimensions et la table des faits. Par conséquent, le graphe de jointure correspondant au graphe d'entrepôt de données peut être non connecté, dans le sens où il peut contenir plus d'un sous-graphe. Dans ce cas, il est appelé graphe de jointure partitionné. Cette situation est attestée dans les cas suivants:

1. Les vues n'ont pas d'attribut de jointure commun. Par exemple, une vue est définie sur la table de dimensions CLIENT et l'autre sur la table de dimension PRODUIT.
2. Les vues ont des attributs de jointure communs mais ne peuvent jamais être jointes. Supposons par exemple que nous ayons deux vues matérialisées V_1 et V_2 définissant les montants de ventes pour les clients féminins et pour les clients masculins, respectivement. Dans cette situation, le résultat de la jointure entre V_1 et V_2 est vide.

Définition 23 Index de graphe de jointure (IGJ) *Soit un graphe de jointure de cardinalité m ($m \geq 2$). Un IGJ est un sous ensemble de h ($h \leq m$) noeuds de graphe de jointure.*

Dans le cas d'un graphe de jointure partitionné, l'IGJ peut être défini sur chaque sous-graphe du graphe de jointure.

D'après ces définitions, nous pouvons conclure qu'un index de jointure en étoile est un cas particulier d'IGJ (quand les vues ne sont pas considérées).

Dans cette étude, nous utilisons un schéma en étoile ayant un ensemble de tables de dimensions, et une table des faits. Sur ce dernier, un ensemble de vues matérialisées est sélectionné. Chaque table T_i (qui peut être une table de dimensions, la table des faits ou une vue) a une clé dénotée par K_{T_i}. Un IGJ G sur m tables peut être représenté par une table ayant m attributs $(K_{T_1}, K_{T_2}, ..., K_{T_m})$. Cet index est dénoté par:

$$G : (T_1 \sim T_2 \sim ... \sim T_m)$$

Définition 24 Un IGJ simple *est un IGJ défini sur deux tables seulement.*

Définition 25 Un IGJ complet *est un IGJ couvrant tous les noeuds du graphe de jointure.*

Définition 26 Un IGJ partiel *est un IGJ couvrant certains noeuds du graphe de jointure.*

Exemple 12 *Reprenons l'exemple 10 dans lequel deux vues matérialisées V_1 et V_2 sont sélectionnées. À partir de la matrice de réécriture de la Figure 3.10, le graphe de jointure peut être construit. Il contient 4 noeuds représentant V_1, V_2, CLIENT et TEMPS et 5 arêtes (voir Figure 3.12). Notons qu'il n'existe pas une arête entre CLIENT et TEMPS car ces derniers ne peuvent pas être joints. Le seul IGJ complet correspondant à ce GJ est $(V_1 \sim V_2 \sim CLIENT \sim TEMPS)$. Plusieurs IGJs simples peuvent être identifiés, par exemple l'IGJ $(V_1 \sim V_2)$.*

Considérons l'IGJ $(CLIENT \sim TEMPS)$. Nous remarquons qu'il n'existe pas d'arête entre CLIENT et TEMPS dans le GJ, mais deux chemins $CLIENT - V_1 - TEMPS$, et $CLIENT - V_2 - TEMPS$. Ces derniers donnent deux IGJs, le premier référence tous les n-uplets de CLIENT qui peuvent être joints avec TEMPS

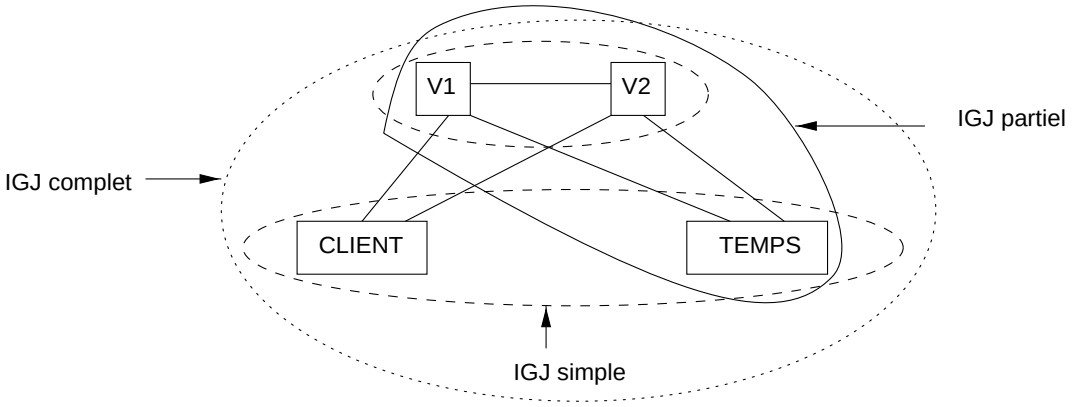


FIG. 3.12: Graphe de jointure

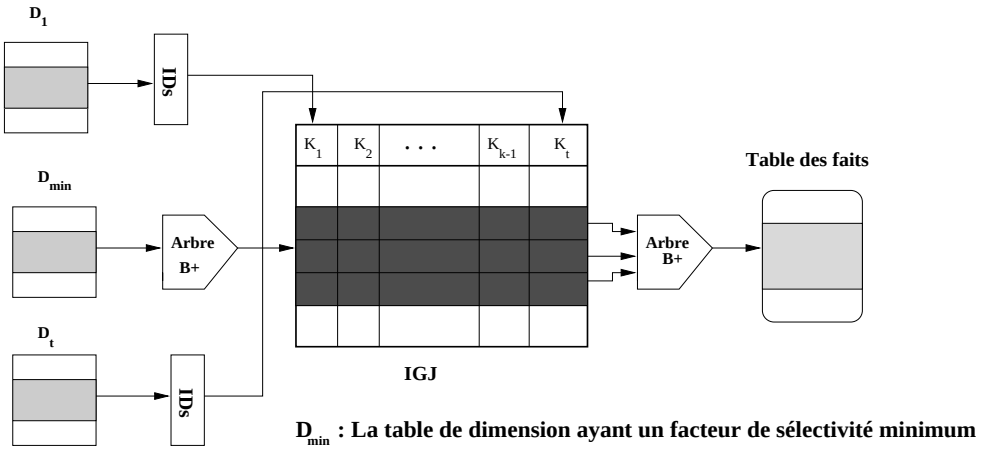


FIG. 3.13: La stratégie d'exécution d'une jointure en présence d'un IJE

en se basant sur les n-uplets communs de la vue V_1, et l'autre représente tous les n-uplets de CLIENT qui peuvent être joints avec TEMPS en se basant sur les n-uplets communs de la vue V_2. Un IGJ est utilisé en fonction des requêtes. Il est donc important de considérer la sémantique des requêtes dans la sélection des index.

3.6 Une stratégie d'exécution de requêtes

Dans cette section, une stratégie d'exécution des requêtes en présence des IGJs est décrite. Pour rendre cette présentation compréhensible, considérons un schéma en étoile SED de d tables de dimensions $\{D_1, D_2, ..., D_d\}$ et une table des faits F. Supposons que toutes ces tables sont stockées en mémoire secondaire.

Soit Q_i une requête dont la syntaxe est similaire à celle décrite en section 3.3. Cette requête est représentée par un arbre algébrique G_{Q_i} dont l'ensemble des noeuds est dénoté par N_{Q_i} (la cardinalité de N_{Q_i} est dénotée par t). Soit g un IGJ pour supporter l'exécution de la requête Q_i. Soit $Table(g)$ l'ensemble des tables sur lesquelles, g est défini.

La Figure 3.13 présente la stratégie d'exécution de Q_i en utilisant l'IGJ g. Ce dernier a une position centrale. Le parcours des tables se fait de la manière suivante:

tables de dimensions, IGJ et enfin table des faits (tables de dimensions $\longrightarrow$ IGJ $\longrightarrow$ table des faits). Cette stratégie suit exactement le principe des requêtes de jointure en étoile (la sélection d'abord et ensuite la jointure).

Pour accéder aux n-uplets de l'IGJ à partir d'une table de dimensions, nous disposons d'un arbre B^+. Nous disposons également d'un arbre B^+ pour accéder aux n-uplets de la table des faits à partir de n-uplets de l'IGJ (voir Figure 3.13).

En fonction de la définition de Q_i, quatre scénarios d'exécution de la requête Q_i sont possibles:

1. Pas de jointure: La présence de l'IGJ g n'a pas d'influence.

   Pour exécuter Q_i, des techniques d'indexation simples (comme l'index de projection, l'index binaire, etc.) peuvent être utilisées.

2. Non Correspondance (non matching): La requête Q_i possède une opération de jointure, mais l'IGJ g ne couvre aucune table de l'arbre de Q_i (c'est-à-dire: $N_{Q_i} \cap Table(g) = \phi$ et $t \geq 2$).

   Pour exécuter la requête Q_i, les techniques ad-hoc de jointure peuvent être utilisées (voir section 3.5).

3. Correspondance totale (total matching): L'IGJ couvre toutes les tables de G_{Q_i} (c'est-à-dire: $N_{Q_i} \subseteq Table(g)$).

   Pour exécuter la requête Q_i, nous effectuons les étapes suivantes:

   - Charger toutes les tables de dimensions utilisées par la requête Q_i.
   - Charger l'arbre B^+ correspondant à la table de dimension D_{min} (celle qui a le facteur de sélectivité minimale). Une fois cet arbre chargé, les n-uplets de D_{min} sont utilisés pour charger l'IGJ vers la mémoire centrale. Cette opération permet de charger seulement une partie de l'index et non sa totalité (Figure 3.13) .
   - Exécuter toutes les opérations de sélection définies sur les tables de N_{Q_i} (si elles existent bien sûr). Pour chaque table de N_{Q_i} $(1 \leq i \leq t)$, nous considérons l'ensemble des clés primaires de ses n-uplets sélectionnés. Ces derniers peuvent être utilisés pour accéder à l'IGJ [7]. En calculant l'intersection de ces ensembles, nous obtenons les n-uplets satisfaisant l'opération de jointure.
   - Charger l'arbre B^+ afin d'accéder aux n-uplets de la table des faits (voir Figure 3.13).
   - Charger les n-uplets pertinents de la table des faits.

4. Correspondance partielle (partial matching): l'IGJ g ne couvre que certaines tables de G_{Q_i} (la table des faits incluse) (c'est-à-dire $Table(g) \cap N_{Q_i} \neq \phi$). Par exemple, l'IGJ ($VENTES \sim TEMPS$) couvre partiellement la requête Q_3 (Figure 3.7).

   Pour exécuter la requête Q_i, l'IGJ est utilisé pour traiter les jointures définies sur les tables qu'il couvre (comme dans le total matching). Les résultats de ces jointures sont combinés avec les tables non couvertes pas l'IGJ en utilisant les techniques de jointure classiques (comme dans le cas de non matching).

[7] Chaque clé primaire identifie un ensemble de n-uplets de l'IGJ

3.7 Un modèle de coût

Dans cette section, un modèle de coût est présenté. Son rôle est d'estimer le coût des requêtes en présence des IGJs. Il peut également être utilisé pour calculer le coût de chaque noeud (vue) de jointure du PMEV (voir section 3.3).

3.7.1 Les composantes d'un modèle de coût

Le coût total pour évaluer une requête dans un SGBD se décompose en trois coûts principaux [LDH+84, OV91]: (1) *le coût des entrées-sorties (IO)* pour lire et écrire entre la mémoire et le disque, (2) *le coût CPU* et (3) *le coût COM de communication sur le réseau* (si les données sont réparties sur le réseau). Ce dernier est généralement exprimé en fonction de la quantité totale des données transmises [Ape88, BKL98a]. Il dépend de la topologie du réseau.

Les modèles de coût utilisés par les optimiseurs ont généralement les mêmes composantes [Gru96, Naa99]: (1) *des statistiques des données stockées dans une métabase*, et (2) *des formules* pour évaluer la cardinalité des résultats intermédiaires et des requêtes. Le modèle de coût a comme entrée un plan d'exécution d'une requête et comme sortie le coût de ce plan. Le coût d'un plan d'exécution est évalué en cumulant le coût des opérations élémentaires (la sélection, la projection, la jointure, etc.). Une fonction de coût est associée à chaque opérateur élémentaire. Pour établir avec précision une fonction de coût nous devons connaître à l'avance les nombreux paramètres dont dépend l'opérateur ainsi que le détail de l'algorithme qui l'implémente.

Exemple 13 *Supposons que nous voulions évaluer le coût d'une requête ayant une opération de jointure entre deux tables R et S. Les paramètres dont l'opérateur de jointure dépend sont: la taille des deux tables, le nombre de pages occupées par ces deux tables, la sélectivité des prédicats de jointure, les méthodes d'accès déterminant la nature de l'algorithme de jointure utilisée [Ram98] (jointure avec boucles imbriquées, jointure en utilisant un index, jointure par fusion, etc.). La prise en compte de ces paramètres rend l'estimation de l'opération de jointure entre R et S plus facile.*

3.7.2 Les statistiques et les estimations

3.7.2.1 Les statistiques de base

L'optimiseur des requêtes se base sur les statistiques sur les données stockées pour générer les plans d'exécution des requêtes. La disponibilité de statistiques pertinentes peut largement améliorer la qualité des plans choisis par l'optimiseur [CN00]. Par contre, l'absence de statistiques peut donner de mauvais plans d'exécution. Le *Groupe DMX* (Data Management Exploration) de Microsoft Research a proposé des techniques d'automatisation des statistiques [CN00]. Ces dernières sont disponibles dans le catalogue du système [Cha82, YC84]. Ces statistiques peuvent être divisées en trois catégories:

- Les statistiques décrivant les tables et les distributions des valeurs d'attribut: le nombre n-uplets et leur taille moyenne pour chaque table, le nombre de valeurs distinctes pour chacun des attributs (qui permet de calculer la sélectivité

des opérations de sélection contenant des prédicats d'égalité), les valeurs minimales et maximales des attributs de sélection (pour calculer la sélectivité des prédicats de comparaison). La représentation la plus simple est celle qui suppose l'uniformité de la distribution des valeurs et l'indépendance des attributs [SAD+79, GGT96]. Ces paramètres sont pris en compte par les systèmes commerciaux [CN00].

- Les statistiques décrivant les paramètres de la machine utilisée: taille des pages disque et mémoire, taille mémoire, coût moyen d'une entrée-sortie, etc.
- Les statistiques des méthodes d'accès: les index sont des structures persistantes, il faut donc estimer leurs tailles, le nombre de pages occupées par ces index, etc.

3.7.2.2 Estimation de la sélectivité des prédicats

La sélectivité des prédicats (de sélection et de jointure) est un paramètre primordial pour estimer le coût des requêtes [IC91].

Définition 27 *La sélectivité est un coefficient représentant le nombre d'objets sélectionnés rapporté à un nombre d'objets total d'une table. Si la sélectivité vaut 1, tous les objets sont sélectionnés. Si elle vaut 0, aucun objet n'est sélectionné.*

Nombre de travaux ont été développés pour estimer la sélectivité [IC91]. La plupart d'entre eux supposent une distribution uniforme des valeurs des attributs et une indépendance entre les attributs de chaque relation. Cette estimation se fait de la manière suivante:

Soient A_i et A_j deux attributs d'une relation R, les formules de sélectivité sont les suivantes:

$$Sel(A_i = valeur) = \frac{1}{card(\pi_{A_i}(R))}$$

$$Sel(A_i > valeur) = \frac{max(A_i) - valeur}{max(A_i) - min(A_i)}$$

$$Sel(A_i < valeur) = \frac{valeur - max(A_i)}{max(A_i) - min(A_i)}$$

$$Sel(p(A_i) \wedge p(A_j)) = Sel(p(A_i)) * Sel(p(A_j))$$

$$Sel(p(A_i) \vee p(A_j)) = Sel(p(A_i)) + Sel(p(A_j)) - (Sel(p(A_i)) * Sel(p(A_j))$$

$$Sel(A_i \in \{valeurs\}) = Sel(A_i = valeur) * card(\{valeur\})$$

La taille d'une opération de jointure entre deux tables T_1 et T_2 est estimée en utilisant la formule suivante [Ull89]:

$$||T_1 \bowtie_A T_2|| = \frac{||T_1|| \times ||T_2||}{max(card(\pi_A T_1), card(\pi_A T_2))}$$

Toutefois, certains travaux ne supposent aucune distribution particulière [Naa99]. Dans ce cas, une approche à base d'histogrammes est proposée dans [Ioa93].

3.7.2.3 Estimation de la sélectivité des prédicats complexes

Les prédicats complexes sont composés de prédicats conjonctifs (ET) et disjonctifs (OU). Grâce à la forme normale, la sélectivité des prédicats est calculée de proche en proche. La formule générale pour la conjonction est la suivante:

$$Sel(pred_1\ AND\ pred_2) = Sel(pred_1) * Sel(pred_2|pred_1)$$

$pred_2|pred_1)$ veut dire $pred_2$ sachant $pred_1$. Si les deux prédicats sont indépendants on aura $Sel(pred_2|pred_1) = Sel(pred_2)$. C'est toujours l'hypothèse envisagée [Gru96]. En effet aucun système n'est réellement capable de tenir compte de la corrélation entre les prédicats.

Pour les disjonctions, la formule générale est la suivante:

$$Sel(pred_1\ OU\ pred_2) = Sel(pred_1) + Sel(pred_2) - Sel(pred_1 ET pred_2)$$

Si $pred_1$ et $pred_2$ sont mutuellement exclusifs, on a $Sel(pred_1\ ET\ pred_2) = 0$.

Une méthode basée sur l'échantillonnage (sampling) [HS95] permet d'estimer la sélectivité des prédicats en exécutant la requête sur une portion des tables. Suivant les résultats obtenus sur ces petites portions, la sélectivité de la requête peut être estimée. Le problème dans un tel système est qu'il faut optimiser partiellement la requête pour pouvoir exécuter quelques parties. Ce système ne fonctionne que pour des requêtes simples et très coûteuses. En effet, le processus de l'échantillonnage doit être très efficace par rapport à l'exécution globale de la requête pour obtenir un gain significatif en performance.

3.7.3 Les méthodes d'accès

En l'absence d'index de jointure d'autres méthodes d'accès sont utilisées. Parmi ces méthodes, nous trouvons la jointure imbriquée, la jointure par tri, la jointure par hachage, etc. Dans ce chapitre nous utilisons la jointure par hachage. Ce choix s'est porté sur cette technique parce qu'elle est utilisée par plusieurs systèmes existants (e.g., Oracle, Informix). Les autres techniques peuvent également être utilisées dans cette thèse. La jointure par hachage se fait en deux phases [Ram98]. Dans la première phase, les deux relations sont partitionnées suivant la même fonction de hachage appliquée aux attributs participant à la jointure. Dans la deuxième phase, les partitions en correspondance sont jointes.

3.7.4 Le modèle de coût dans notre contexte

3.7.4.1 Les hypothèses

Nous avons maintenant tous les ingrédients pour présenter le modèle de coût pour évaluer un ensemble de requêtes décisionnelles. Nous nous sommes fixés comme objectif de donner une approximation du nombre des entrées-sorties et nous n'avons pas pris en compte le coût de CPU et le coût COM de communication sur le réseau. Ce choix se justifie dans la mesure où, dans les grandes bases de données, la contribution du coût CPU n'est pas aussi significative que le coût des entrées-sorties [FKL97, BKL98b, YKL97, Gup99]. Notons que notre modèle de coût peut facilement être étendu pour inclure les deux coûts non pris en compte. Nous considérons également que l'entrepôt est centralisé.

Soit un entrepôt de données modélisé par un schéma en étoile ayant d tables de dimensions et une table des faits. Sur ce schéma, un ensemble de requêtes sont définies $\mathcal{Q} = \{Q_1, Q_2, ..., Q_s\}$. Chaque requête Q_i $(1 \leq i \leq s)$ a une fréquence d'accès f_i. Soient $\mathcal{V} = \{V_1, V_2, ..., V_l\}$ et $\mathcal{IJ} = \{I_1, I_2, ..., I_m\}$ un ensemble de vues matérialisées et d'index de jointure.

Avant de décrire le modèle, deux hypothèses sont à considérer:

- Les tables de dimensions, la table des faits, les vues matérialisées, et les index sont stockés sur disque (notre modèle peut être facilement adapté lorsque les tables de dimensions, la table des faits, les vues matérialisées, ou les index sont en mémoire centrale).
- Les conditions de sélection sont toujours descendues sur l'arbre syntaxique comme dans [LQA97].

3.7.4.2 Les paramètres

Les paramètres de notre modèle de coût peuvent être classés en trois catégories: *les paramètres relatifs à l'entrepôt* (Table 3.1), *les paramètres relatifs aux requêtes* (Table 3.3) et *les paramètres relatifs aux méthodes d'accès (les index)* (Table 3.2).

Nom de paramètre	Description
$\|\|T_j\|\|$	La cardinalité de la table T_j
$\|T_j\|$	Le nombre de pages stockant la table T_j
$w(c_{ij})$	La taille, en bytes, de la colonne c_i de la table T_j
$w(K_{T_j})$	La taille, en bytes, de la clé primaire de la table T_j

TAB. 3.1: Les paramètres de l'entrepôt

Nom de paramètre	Description
$\|\|G_i\|\|$	La cardinalité de l'IGJ G_i
$\|G_i\|$	Le nombre de pages stockant l'index G_i

TAB. 3.2: Les paramètres des index

Nom de paramètre	Description
π	Taille de pointeur de la page
PS	Taille d'une page
M	Taille du tampon (en nombre de pages)

TAB. 3.3: Les paramètres physiques

3.7.4.3 L'estimation des facteurs de base du modèle de coût

Dans les bases de données relationnelles, les tables sont stockées "as-is" [DRT99]: chaque table est représenté comme une série de n-uplets partitionnés en blocs de

données liés par des *pointeurs de bloc*. Dans cette étude, nous supposons que chaque n-uplet est assez petit pour tenir dans un bloc de données. En conséquence, le nombre de pages nécessaires pour stocker une table T_j est donné par la formule suivante:

$$|T_j| = PS \times \left\lceil \frac{||T_j||}{\lfloor \frac{PS-\pi}{w(T_j)} \rfloor} \right\rceil \tag{3.4}$$

où $\lceil\ \rceil$ et $\lfloor\ \rfloor$ représentent les fonctions *ceiling* et *floor*, respectivement.
La même formule pourra être utilisée pour estimer le nombre de pages stockant une vue ou un IGJ, car ces derniers sont stockés sous la forme d'une table.

Le coût d'une jointure par hachage $Cost\_Hash$ entre deux tables T_i et T_j $(i \neq j)$ est donné par la formule suivante [Ram98]:

$$Cost\_Hash = 3 \times (|T_i| + |T_j|) \tag{3.5}$$

3.7.4.4 Les formules du modèle de coût

Dans la section 3.6, nous avons vu qu'il existe quatre scénarios pour évaluer une requête Q_i en présence d'un IGJ. Nous présentons maintenant, des formules pour chaque scénario:

Non matching: Dans ce cas, la technique de jointure par hachage est utilisée. Son coût est décrit par l'équation 3.5.

Total matching Le coût d'exécution de la requête Q_i sous ce scénario est divisé en plusieurs coûts correspondant à chaque étape d'exécution (voir section 3.6):

1. Le coût de chargement $Load\_Cost(D)$ des tables de dimensions est donné par:

$$Load\_Cost(D) = \sum_{i=1}^{t} |D_i| \tag{3.6}$$

2. Le coût de sélection de la table de dimensions D_{min} parmi toutes les tables de N_{Q_i} est nul (car toutes les tables de dimensions sont déjà chargées en mémoire centrale).
3. Le coût de chargement de l'arbre B^+ est défini par la taille de cet arbre. Comme dans [GHRU97], nous considérons que la taille de l'arbre B^+ peut être estimée à partir de la taille de toutes les feuilles de cet arbre (le nombre de feuilles est approximativement égal au nombre de n-uplets de la table sous-jacente).
4. Le coût de chargement de l'IGJ G_j est:

$$Load\_Cost(G_j) = \left| SF_{D_{min}} \times ||G_j|| \right| \tag{3.7}$$

5. Le coût de l'opération d'intersection est nul puisque les tables de dimensions et les index sont déjà chargés en mémoire centrale.
6. Le coût de chargement de l'arbre B^+ est identique à celui défini dans l'étape 3.

7. Le coût de chargement des n-uplets de la table des faits est:

$$Load\_Cost(F) = \left| SF_F \times ||F|| \right| \tag{3.8}$$

Partiel matching Le coût sous ce scénario peut être dérivé de celui de non matching et total matching.

Le coût total d'exécution de l'ensemble de toutes les requêtes Soit $C(Q_i)$ le coût d'exécution de la requête Q_i $(1 \leq i \leq s)$. Le coût total d'exécution de l'ensemble de toutes les requêtes (CT) est donné par l'équation suivante:

$$CT = \sum_{i=1}^{s} f_i \times C(Q_i) \tag{3.9}$$

3.7.5 La maintenance des IGJs

Comme nous l'avons déjà vu dans le chapitre 2, lors de changements (opérations de suppression, ou d'ajout) au niveau des tables de base, les index correspondants doivent impérativement être remis à jour. Nous présentons le coût de maintenance des IGJs pour les opérations de suppression et d'ajout causées par des requêtes de mise à jour.

Le cas de suppression Soit G un IGJ défini sur m tables [8] de la façon suivante: $G : (T_1 \sim T_2 \sim ... \sim T_m)$. Supposons qu'une opération de suppression survienne sur la table T_1. Soit ∇T_1 les n-uplets concernés par cette opération. Avant de supprimer ces n-uplets de l'index, ces derniers doivent être identifiés. L'identification de ces n-uplets est réalisée en évaluant l'opération suivante:

$$\nabla G = G \ltimes \nabla T_1 \tag{3.10}$$

où $\ltimes$ représente l'opération de semi jointure.
Une fois identifiés, ils doivent être éliminés de l'IGJ G.

Le cas d'ajout D'une manière similaire, la propagation des insertions au niveau de la table T_1 $(\triangle T_1)$ vers l'index G est égale à l'évaluation de l'opération suivante:

$$\triangle G = \triangle T_1 \bowtie T_2 \bowtie ... \bowtie T_m \tag{3.11}$$

Finalement, les n-uplets de $\triangle G$ sont insérés dans l'IGJ G.

[8] tables peuvant être tables de base ou vues

3.7.6 Le coût de stockage des IGJs

La Figure 3.12 nous donne une idée claire sur les composantes d'un IGJ: (1) les arbres B^+ qui permettent d'accéder aux n-uplets de l'IGJ à partir de chacune des tables de dimensions, (2) l'IGJ lui-même et (3) l'arbre B^+ qui permet d'accéder aux n-uplets de la table des faits à partir de l'IGJ.

La taille d'un IGJ peut être estimée en utilisant les mêmes techniques d'estimation de la taille des jointures [HS95].

Le coût de stockage d'une structure d'un IGJ défini sur m tables [9] est la somme de la taille des arbres B^+ et de la taille de l'IGJ (voir Figure 3.13).

3.8 Les algorithmes de sélection des IGJs

3.8.1 La complexité de la sélection des IGJs

Notons que pour un schéma d'un entrepôt de n tables, le nombre total de IGJs qui peut être généré est donné par:

$$\begin{pmatrix} 2^n-n-1 \\ 0 \end{pmatrix} + \begin{pmatrix} 2^n-n-1 \\ 2 \end{pmatrix} + ... + \begin{pmatrix} 2^n-n-1 \\ 2^n-n-1 \end{pmatrix} = 2^{(2^n-n-1)}$$

Si on veut sélectionner un seul IGJ, le nombre de possibilités est 2^n-n-1. Par exemple pour un entrepôt de 4 tables, $2^4-4-1=11$ IGJs peuvent être considérés, pour sélectionner le meilleur index.

Par contre, pour sélectionner plus d'un IGJ, le nombre de possibilités est $2^{(2^n-n-1)}$. Pour $n=5$, il est 67108864, et pour $n=6$, il est supérieur à 10^{17}. Sélectionner les meilleurs index pour un ensemble de requêtes reste aujourd'hui une question ouverte.

Étant donné que l'espace de recherche est très large ($2^{(2^n-n-1)}$, pour n tables), nous sommes obligés d'avoir des algorithmes de sélection pour trouver les IGJs optimaux ou quasi-optimaux afin d'évaluer un ensemble de requêtes. Ces algorithmes font l'objet de la section suivante.

Nous montrons d'abord que le problème de sélection des IGJs peut être formulé comme un problème d'optimisation. Nous suggérons ensuite des algorithmes pour résoudre ce problème.

3.8.2 La formulation du problème de sélection des IGJs

La plupart des algorithmes de sélection d'index proposés pour les bases de données [FST88, Val87, BG93, FKL98] ou les entrepôts [DRT99, CN97, GHRU97, LR98] sélectionnent un ensemble d'index optimisant une fonction objectif et satisfaisant une contrainte de ressource. Nous nous placons dans les mêmes conditions générales en choisissant comme fonction objectif le nombre d'entrées-sorties pour exécuter un ensemble de requêtes et comme contrainte de ressource l'espace disque pour stocker les index sélectionnés.

Les entrées de notre problème sont:

[9] vue, dimensions, ou faits

1. Un schéma en étoile ayant une table des faits F et d tables de dimensions $\{D_1, D_2, ..., D_d\}$.
2. Un ensemble de requêtes (les plus fréquemment utilisées) $\mathcal{Q} = \{Q_1, Q_2, ..., Q_s\}$ avec leurs fréquences d'accès $\{f_1, f_2, ..., f_s\}$.
3. Un ensemble de vues matérialisées $\mathcal{V} = \{V_1, V_2, ..., V_m\}$ sélectionné pour supporter l'exécution de l'ensemble de requêtes.
4. Une capacité de stockage S destinée au stockage des index sélectionnés.

Le problème de sélection des IGJs consiste à sélectionner un ensemble d'IGJs $\mathcal{G}'$ minimisant le coût total d'exécution de l'ensemble des requêtes (CT). L'ensemble des index sélectionnés doit satisfaire la contrainte suivante: l'espace total occupé par G' (dénoté par $S_{G'}$) doit être inférieur ou égal à S .

Plus formellement, soit $\mathcal{G}$ l'ensemble de tous les IGJs possibles. Notons par $C_{G_j}(Q_i)$ le coût d'évaluation de la requête Q_i en utilisant l'IGJ G_j. Étant donné une capacité d'espace S, l'objectif de SIGJ est de sélectionner un ensemble d'IGJs $\mathcal{G}'$ ($\mathcal{G}' \subseteq \mathcal{G}$) tel que:

$$CT = \sum_{i=1}^{s} f_i \times \{min_{\forall G_j \in \mathcal{G}'} \ C_{G_j}(Q_i)\} \tag{3.12}$$

sous la contrainte: $S_{G'} \leq S$.

3.8.3 Les algorithmes de sélection des IGJs

Trois algorithmes de sélection sont présentés, à savoir: (1) *un algorithme naïf* sélectionnant un index optimal, (2) *un algorithme glouton sélectionnant seulement un index*, et (3) *un algorithme glouton sélectionnant plusieurs index*. Ces algorithmes sont décrits dans les sections qui suivent.

3.8.3.1 L'algorithme naïf

Cet algorithme est utilisé seulement à des fins de comparaison. Il consiste à énumérer d'une manière exhaustive l'ensemble $\mathcal{G}$ de tous les IGJs possibles. Pour chaque IGJ $G_i \in \mathcal{G}$, l'algorithme naïf calcule le coût d'exécution de l'ensemble de requêtes (en utilisant cet index). Finalement, l'index offrant un coût d'exécution des requêtes minimum et satisfaisant la contrainte d'espace est sélectionné.

L'avantage majeur de cet algorithme est que l'index sélectionné est optimal. Il est à noter cependant que cet algorithme est impraticable dans les cas réels [10] (voir section 3.8.1).

3.8.3.2 L'algorithme glouton pour sélectionner un seul index

Comme nous l'avons vu, l'énumération exhaustive des IGJs est très coûteuse. Pour éviter cette énumération, un algorithme glouton sélectionnant un seul IGJ (AGS1) est présenté. Avant de présenter cet algorithme, définissons la notion de graphe de jointure étiqueté.

[10] Un entrepôt de données typique peut contenir 10 ou plus tables de dimensions [KR98]

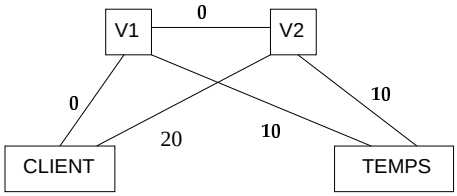


FIG. 3.14: Le graphe de jointure étiqueté

Définition 28 Un graphe de jointure étiqueté *est un GJ où chaque arête* $e(T_i, T_j)$ *est associée à un poids. Ce dernier est égal à la somme des fréquences d'accès des requêtes utilisant cette arête.*

Exemple 14 *Le graphe de jointure étiqueté correspondant au graphe de jointure de la figure 3.12 est illustré dans la figure 3.14. La représentation du poids de chaque arête peut être réalisée à l'aide de la matrice de réécriture. Par exemple, l'arête (*V_2*, CLIENT) a un poids égal à 20. Cette arête est utilisée par les requêtes* Q_3 *et* Q_5 *ayant les fréquences 5 et 15 (dont la somme est égale au poids de cette arête: 20) (voir Figure 3.14).*

L'algorithme AGS1 a le graphe de jointure étiqueté et la capacité d'espace S comme entrées. Il est décrit de la façon suivante:
Il commence par sélectionner un IGJ simple (une arête de graphe de jointure étiqueté) ayant un poids maximal. Une fois cet index sélectionné, l'algorithme essaye d'ajouter d'autres noeuds à cet index. L'objectif de cette opération est de *réduire le coût d'exécution des requêtes.*

Lorsque plus aucune opération d'ajout n'est possible, l'algorithme essaye de réduire la taille de l'index en éliminant certains noeuds. Le but de cette élimination est de *réduire le coût d'exécution des requêtes et/ou le coût de stockage de l'index.*

Les deux opérations sont dirigées par un coût (cost-driven): c'est-à-dire qu'elles sont appliquées si elles réduisent le coût d'exécution des requêtes et si le coût de stockage de l'IGJ sélectionné ne dépasse pas S.
Maintenant, nous décrivons les deux opérations ajout et élimination.

L'opération d'ajout

1. Soit l'IGJ $G_i = (u_{i_1} \sim u_{i_1} \sim ... \sim u_{i_k})$ obtenu à l'étape i de l'algorithme. Notons que cet index doit respecter la contrainte suivante: $S_G < S$, où S_G représente la capacité de stockage nécessaire pour l'index G_i.
2. Soit CT_{G_i} le coût total d'exécution de toutes les requêtes en utilisant l'IGJ G_i. CT_{G_i} est donné par l'équation suivante:

$$CT_{G_i} = \sum_{j=1}^{s} C_{G_i}(Q_j) \tag{3.13}$$

   où $C_{G_i}(Q_j)$ représente le coût d'exécution de la requête Q_j en utilisant l'index G_i.
3. Pour ajouter un noeud à l'IGJ G_i, deux alternatives sont possibles:

   (i) $G_i' = (u_{(i_1-1)} \sim u_{i_1} \sim ... \sim u_{i_k})$

(ii) $G_i'' = (u_{i_1} \sim u_{i_2} \sim ... \sim u_{(i_k+1)})$

Si $C_{G_i'} \leq C_{G_i''} \leq C_{G_i}$, alors $G_i = G_i'$ si $S_{G_i'} \leq S$.
Si $C_{G_i''} \leq C_{G_i'} \leq C_{G_i}$, alors $G_i = G_i''$ si $S_{G_i''} \leq S$.
Sinon, G_i est inchangé.

L'opération d'élimination

1. Soit un IGJ $G_i = (u_{i_1} \sim u_{i_2} \sim ... \sim u_{i_k})$ de longueur k. À partir de G_i, les IGJs suivants de longueur $k-1$ peuvent être générés:
   $(u_{i_2} \sim ... \sim u_{i_k}), (u_{i_1} \sim u_{i_3} \sim ... \sim u_{i_k}), ..., (u_{i_1} \sim u_{i_2} \sim ... \sim u_{i_{k-1}})$.
   Soit G_i' l'IGJ offrant un meilleur coût entre tous ces IGJs. Si $C_{G_i'} < C_{G_i}$ et $S_{G_i'} < S$, alors $G_i = G_i'$, sinon, nous gardons l'IGJ courant G_i.
   Un noeud peut être éliminé si les noeuds qui restent après l'élimination *peuvent être joignables.*

Avec ces deux opérations, l'AGS1 peut facilement être décrit.

Algorithme 5 Les étapes de AGS1

1: Construire le graphe de jointure étiqueté
2: Trier les arêtes de ce graphe en fonction de leurs poids. Soit L la liste triée dans un ordre descendant.
3: Sélectionner l'index $G_i = (u_i \sim u_j)$ de L satisfaisant la contrainte de stockage.
4: **Appliquer l'opération d'ajout** (G_i)
5: Répéter cette opération jusqu'à ce qu'aucune réduction dans le coût d'exécution des requêtes ne soit constatée et que la capacité de stockage ne soit pas violée.
6: **Appliquer l'opération d'élimination** (G_i)
7: Répéter cette opération jusqu'à ce qu'aucune réduction dans le coût d'exécution des requêtes ne soit constatée et que la capacité de stockage ne soit pas violée.
8: L'IGJ sélectionné est G_i

Exemple 15 *Pour illustrer cet algorithme, considérons dans un premier temps que nous n'avons aucune contrainte d'espace; ensuite une contrainte d'espace est considérée.*

- Pas de contrainte d'espace: *Considérons le GJ étiqueté de la Figure 3.14. La liste des arêtes triées est construite de la manière suivante:*
  $L = [(V_2 \sim C), (V_2 \sim T), (V_1 \sim T), (V_1 \sim V_2), (T \sim C), (V_1 \sim C)]$ [11]
  En exécutant la première étape de l'algorithme, nous sélectionnons l'IGJ suivant $G = (V_2 \sim C)$ *(l'index représenté par l'arête de poids maximum). Le coût d'exécution des cinq requêtes de la Figure 3.7 en utilisant cet index est* $C_G = 172495$ *pages (voir Table 3.6).*
  L'opération d'ajout est ensuite utilisée afin de réduire le plus possible le coût des requêtes. Le résultat de cette opération est l'IGJ G' : $(V_1 \sim V_2 \sim C)$. *Ce*

[11] Pour des raisons de simplicité, les tables CLIENT et TEMPS sont remplacées par leurs initiaux C et T

dernier réduit le coût des requêtes à 129034. L'algorithme essaye encore une fois d'ajouter un nouveau noeud à l'index G'. Finalement, nous obtenons l'IGJ complet: $G'' = (T \sim V_1 \sim V_2 \sim C)$. Le coût des requêtes en présence de cet index est de 27240 pages (voir la Figure 3.15).

Une fois l'IGJ complet atteint, l'application de l'opération d'ajout n'est plus possible. Maintenant, vient le tour de l'opération d'élimination. Cette dernière considère le dernier index obtenu à partir de l'opération d'ajout. Elle commence par exemple par éliminer la vue V_1 de l'index G entre T et V_2 (voir Figure 3.15). Cette opération ne réduit pas le coût des requêtes et toutes les autres opérations d'élimination non plus. En effet l'IGJ complet garantit la meilleure performance des requêtes et aucune contrainte n'est considérée.

Finalement, l'IGJ obtenu par cet algorithme est: $G'' = (T \sim V_1 \sim V_2 \sim C)$. Notons que L'algorithme naïf donne également le même IGJ. Dans cet exemple, l'algorithme AGS1 fournit donc une solution optimale.

– La présence d'une contrainte: *Maintenant, nous exécutons l'algorithme AGS1 en considérant une contrainte d'espace de 0.1 Gigabytes. Dans ce cas, le premier index de la liste $(V_2 \sim C)$ demande 189002678 bytes, ce qui supérieur à la capacité d'espace initial. L'algorithme devra donc sélectionner un autre élément de la liste L satisfaisant la contrainte d'espace. L'index $(V_1 \sim T)$ (81504731 bytes) est le premier élément de L ayant un espace inférieur à 0.1 Gigabytes. L'opération d'ajout est donc appliquée à cet index. Cette dernière donne l'IGJ suivant: $(V_1 \sim T \sim C)$ exigeant plus de 0.1 Gigabytes (125159899 bytes).*
La dernière opération donne l'IGJ suivant: $(V_1 \sim T \sim V_2)$ exigeant un espace de 232690614 bytes qui est supérieur à 0.1 Gigabytes. L'index $(V_1 \sim T)$ représente le meilleur index respectant la contrainte de stockage [12]. Notons au passage que l'algorithme naïf donne la même solution.

3.8.3.3 L'algorithme glouton sélectionnant K IGJs (AGSK)

Rappelons que l'algorithme précédent sélectionne un seul IGJ. Notons que cet index peut être performant pour certaines requêtes, mais pas pour toutes. En conséquence, la sélection de plusieurs IGJs doit être considérée afin d'assurer une meilleure performance de toutes les requêtes. Dans cette section, nous développons un autre algorithme glouton sélectionnant au plus $2^n - n - 1$ IGJs sous une contrainte d'espace.

AGSK commence par une solution initiale représentant un seul index; elle est obtenue par l'algorithme AGS1. Ensuite, il sélectionne une arête du graphe étiqueté ayant un poids maximum et qui ne représente pas l'index obtenu par AGS1. L'algorithme *AGSK* applique alors les opérations d'ajout (une ou plusieurs fois) et d'élimination (une ou plusieurs fois) tant qu'il y a une possibilité de réduction dans le coût d'évaluation des requêtes et que la contrainte d'espace est satisfaite.

Cela génère un second GJI. L'algorithme répéte cette procédure jusqu'à ce que les K index soient identifiés ou que la capacité de stockage soit violée.

Exemple 16 *Dans cet exemple, nous illustrons le fonctionnement de l'algorithme AGSK (sans contrainte d'espace) en utilisant les mêmes données que dans l'exemple*

[12] Dans cet exemple, l'opération d'élimination ne peut pas être appliquée à cause de la taille de l'index trouvé

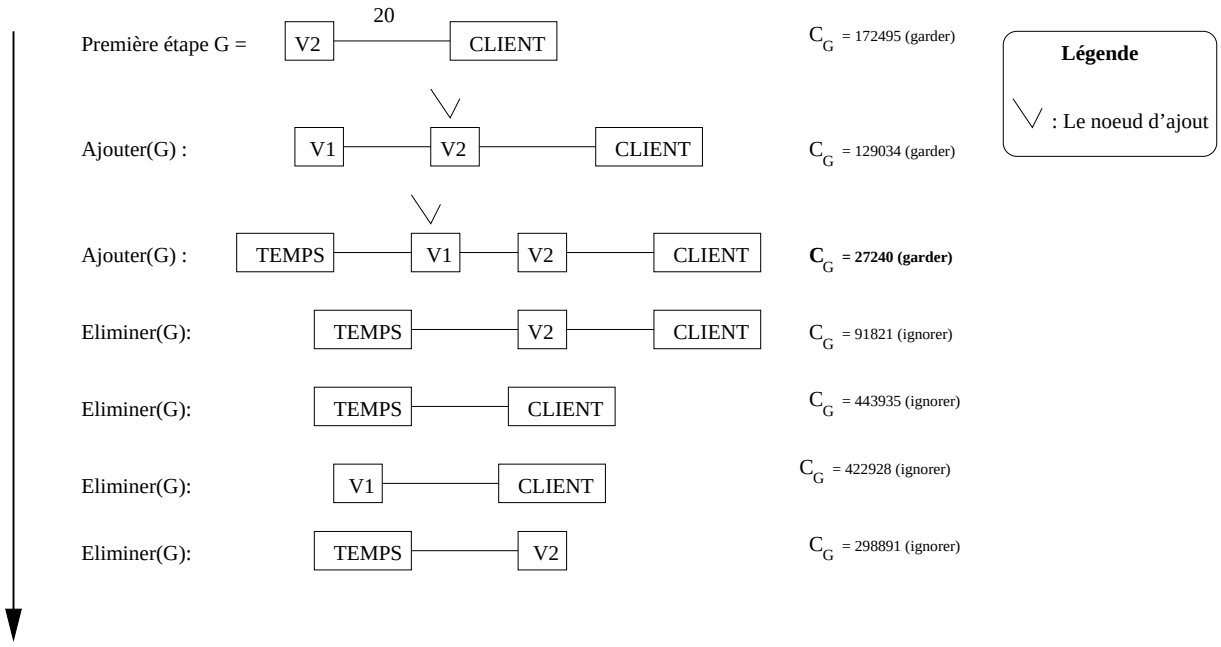


FIG. 3.15: Un exemple de déroulement de l'AGS1

Algorithme 6 Les étapes de $AGSK$

1: Soit G_i un IGJ obtenu à l'aide de l'AGS1.
2: Initialiser i à 2 ($i := 2$)
3: **while** ($i \leq 2^n - n - 1$) et $S_G < S$ **do**
4: Soit G_j l'arête suivante de L qui n'a pas été sélectionnée par l'AGS1, et $S_{G_i} + S_{G_j} < S$. Si cette arête n'existe pas alors **fin de l'algorithme**.
5: Soit $\mathcal{G} = G_i \bigcup G_j$ et $S_G = \sum_{\forall G \in \mathcal{G}}(S_G)$.
6: Appliquer l'opération ajout (G_j)
7: Appliquer l'opération d'élimination (G_j)
8: **if** aucune opération d'ajout et d'élimination n'est possible **then**
9: Incrémenter i
10: **end if**
11: **end while**
12: $\mathcal{G}$ est l'ensemble des IGJs finaux

précédent. L'algorithme AGSK commence par l'index obtenu par l'algorithme AGS1 qui est $G_1 = (T \sim V_1 \sim V_2 \sim C)$. *L'algorithme AGSK sélectionne une arête de L qui ne représente pas l'index* G_1. *L'arête sélectionnée représente l'index suivant:* $(V_1 \sim T)$ *qui réduit le coût d'exécution des requêtes. Il n'est alors plus possible d'appliquer les opérations d'ajout et d'élimination.*

Finalement, cet index est sélectionné par l'algorithme. L'ensemble final des IGJs $\mathcal{G}$ *devient:* $\mathcal{G} = \{(T \sim V_1 \sim V_2 \sim C), (V_1 \sim T)\}$.

Aucun index réduisant le coût des requêtes ne peut plus être ajouté. Les deux index sélectionnés donnent un coût de requêtes de 27213 *pages. L'algorithme naïf génère la même solution. Dans ce cas, AGSK fournit la solution optimale.*

3.9 Evaluation des index de graphe de jointure

Notre étude expérimentale utilise un schéma en étoile emprunté à Informix [Cor97b]. Comme dans la section 2, le schéma possède trois tables de dimensions *CLIENT*, *TEMPS* et *PRODUIT* et une table des faits VENTES. Les caractéristiques de cette expérience sont données dans la Table 3.4.

Paramètre	Description	Valeur
$\|\|VENTES\|\|$	Nombre de n-uplets	100000000
$\|VENTES\|$	Nombre de pages	415040
w(VENTES)	Taille d'un n-uplet de VENTES (in bytes)	34
$\|\|CLIENT\|\|$	Nombre de n-uplets	3000000
$\|CLIENT\|$	Nombre de pages	21607
w(CLIENT)	Taille d'un n-uplet CLIENT (en bytes)	59
$\|\|PRODUIT\|\|$	Nombre de n-uplets	300000
$\|PRODUIT\|$	Nombre de pages	1868
w(PRODUIT)	Taille d'un n-uplet PRODUIT (en bytes)	51
$\|\|TEMPS\|\|$	Nombre de n-uplets	1094
$\|TEMPS\|$	Nombre de pages	5
w(TEMPS)	Taille d'un n-uplet TEMPS (en bytes)	30
$\|\|Vue\_1\|\|$	Nombre de n-uplets	2000000
$\|\|Vue\_2\|\|$	Nombre de n-uplets	4000000
PS	Taille de la page (en bytes)	8192
π	Taille du pointeur de bloc (en bytes)	8

TAB. 3.4: Les paramètres utilisés dans l'expérience

Nous considérons cinq requêtes dont les facteurs de sélectivité sont décrits dans la Figure 3.7.

Afin d'accélérer l'exécution de ces requêtes, deux vues V_1 et V_2 sont sélectionnées en utilisant *sélect_vue* (voir section 3.3). Ces vues sont définies comme suit:

- $V_1 = \sigma_{Etat=\text{“}IL\text{”}}(CLIENT) \bowtie$ VENTES,
- $V_2 = \sigma_{Type\_Paquet=\text{“}Boite\text{”}}(PRODUIT) \bowtie$ VENTES.

La Figure 3.16 donne la matrice de réécriture avec la fréquence d'accès de chaque requête. Étant donné que les requêtes peuvent utiliser directement ces vues, nous

	V1	V2	T	C	FR
Q1	1	0	0	0	20
Q2	0	1	1	0	5
Q3	0	1	1	1	5
Q4	1	0	1	0	10
Q5	0	1	0	1	15

FIG. 3.16: La matrice de réécriture

commençons par évaluer l'indexation des vues et des tables de dimensions relativement à la réduction du coût d'évaluation des requêtes.

La Table 3.5 énumère les 11 IGJs possibles pour $n = 4$ noeuds. Ces index sont construits sur les tables de dimensions *TEMPS (T)* et *CLIENT (C)*, et les vues V_1 et V_2. Rappelons que les tables de dimensions *TEMPS* et *CLIENT* ont une cardinalité de 3000000 et 1094, respectivement. Les cardinalités des vues V_1 et V_2 sont 2000000 et 4000000 n-uplets, respectivement.

Index	V_1	V_2	TEMPS (T)	CLIENT (C)	N-uplets
$V_1 \sim V_2$	1	1	0	0	4000000
$V_1 \sim T$	1	0	1	0	2000000
$V_1 \sim C$	1	0	0	1	2000000
$V_1 \sim V_2 \sim T$	1	1	1	0	4000000
$V_1 \sim V_2 \sim C$	1	1	0	1	4000000
$V_1 \sim T \sim C$	1	0	1	1	2000000
$V_1 \sim V_2 \sim T \sim C$	1	1	1	1	4000000
$V_2 \sim T$	0	1	1	0	4000000
$V_2 \sim C$	0	1	0	1	4000000
$V_2 \sim T \sim C$	0	1	1	1	4000000
$T \sim C$	0	0	1	1	9846000

TAB. 3.5: L'ensemble de IGJs et leurs tailles

En regardant la Table 3.6, nous pouvons constater que l'IGJ ($V_1 \sim T$) a le coût de stockage le moins élevé avec 81504731 bytes, mais il présente le plus petit coût d'évaluation pour la Q_4 seulement. D'un autre côté, l'IGJ ($V_1 \sim V_2 \sim T \sim C$) donne le plus petit coût d'évaluation pour les requêtes Q_1, Q_2, Q_3 et Q_5, mais pas pour la requête Q_4. En fait la requête Q_4 nécessite un accès seulement à la vue V_1 et à la table de dimension TEMPS, et l'IGJ ($V_1 \sim T$) correspond exactement à cette requête. Le coût de stockage de l'IGJ ($V_1 \sim V_2 \sim T \sim C$) est de 293778358 bytes (presque 0.3 GBs). Cette expérience montre qu'il n'existe pas un unique IGJ offrant le meilleur coût d'évaluation des requêtes pour toutes les requêtes. Ceci confirme l'intérêt de l'algorithme *AGSK* (section 3.8).

La Table 3.7 donne le coût normalisé d'évaluation de chaque requête en utilisant les différents IGJs. Cette normalisation est effectuée en prenant le coût de la jointure par hachage comme référence. L'IGJ complet ($V_1 \sim V_2 \sim T \sim C$) a un coût normalisé d'évaluation des requêtes variant de **1%** (pour la requête Q_3) à 100% (pour la requête Q_1). Pour les requêtes Q_3 et Q_5, la plupart des IGJs (à l'exception de $V_1 \sim T$) donne un coût bien moindre que la jointure par hachage. Cela est dû au

Index	**Stockage**	Q_1	Q_2	Q_3	Q_4	Q_5	**CT**
$V_1 \sim V_2$	197751734	21729	41522	106343	21729	106328	297652
$V_1 \sim T$	**81504731**	21729	118683	183504	**636**	151248	475800
$V_1 \sim C$	107653595	21729	118656	141254	21729	119560	422928
$V_1 \sim V_2 \sim T$	232690614	21729	1158	104431	663	106328	234310
$V_1 \sim V_2 \sim C$	258839478	21729	41522	41522	21729	2531	129034
$V_1 \sim T \sim C$	125159899	21729	118683	118683	636	119560	379292
$V_1 \sim V_2 \sim T \sim C$	293778358	21729	1158	1158	663	2531	27240
$V_2 \sim T$	162853814	21729	1158	104431	65244	106328	298891
$V_2 \sim C$	189002678	21729	41522	41522	65190	2531	172495
$V_2 \sim T \sim C$	223941558	21729	1158	1158	65244	2531	91821
$T \sim C$	97060186	21729	118697	118697	65231	119580	443935
IJE complet	4967170048	23144	31595	31595	23147	45491	154973
Jointure par hachage	0	21729	118656	153846	65190	151248	510669

TAB. 3.6: Le coût de stockage des IGJs et le coût d'évaluation des requêtes

fait que les requêtes Q_3 et Q_5 sont réécrites en fonction de la vue V_2 et la table de dimension CLIENT, qui ont une taille plus importante que les autres tables (voir Table 3.5, Figure 3.16). Notons que le résultat de la requête Q_1 coïncide exactement avec la vue V_1 (la présence d'IGJ n'a pas d'influence (voir section 3.6).

Index	Q_1	Q_2	Q_3	Q_4	Q_5	**CT**
$V_1 \sim V_2$	0.00	0.35	0.69	0.33	0.70	0.58
$V_1 \sim T$	0.00	1.00	1.19	0.01	1.00	0.93
$V_1 \sim C$	0.00	1.00	0.92	0.33	0.79	0.82
$V_1 \sim V_2 \sim T$	0.00	0.01	0.68	0.01	0.70	0.45
$V_1 \sim V_2 \sim C$	0.00	0.35	0.27	0.33	0.02	0.25
$V_1 \sim T \sim C$	0.00	1.00	0.77	0.01	0.79	0.74
$V_1 \sim V_2 \sim T \sim C$	0.00	0.01	0.01	0.01	0.02	0.05
$V_2 \sim T$	0.00	0.01	0.68	1.00	0.70	0.58
$V_2 \sim C$	0.00	0.35	0.27	1.00	0.02	0.33
$V_2 \sim T \sim C$	0.00	0.01	0.01	1.00	0.02	0.17
$T \sim C$	0.00	1.00	0.77	1.00	0.79	0.86

TAB. 3.7: Coût normalisé d'évaluation des requêtes

Les résultats illustrent bien l'utilité des IGJs dans l'exécution des requêtes. Ces index permettent d'établir un bon compromis entre le coût d'exécution et le coût de stockage. De plus ils peuvent être uniformément appliqués aux tables de dimensions, à la table des faits et aux vues matérialisées. Ils sont significativement plus efficaces en présence des vues matérialisées. Par exemple en utilisant les deux vues V_1 et V_2, l'IGJ complet ($V_1 \sim V_2 \sim T \sim C$) procure un coût d'exécution de 27240 IOs pour un stockage de 0.3 GBs. Sans vues matérialisées, l'index de jointure en étoile complet ($S \sim P \sim T \sim C$) conduit à un coût d'exécution 6 fois plus grand (154973 IOs) pour un espace de stockage 17 fois plus grand (5 GB). Il est donc important de ne pas considérer la sélection des index de jointure indépendamment de la sélection des

vues matérialisées.

Chapitre 4

Interaction entre les vues matérialisées et les index

4.1 Introduction

Nombre de travaux récents concernant la sélection des vues matérialisées et la sélection des index ont été réalisés dans les contextes ROLAP et MOLAP (voir chapitre 2). De nombreux systèmes commerciaux proposent la création et l'utilisation des vues matérialisées et des index [Ora99, CN97, CN98].

Conceptuellement, les vues et les index se définissent comme des structures physiques pouvant être utilisées pour accélérer les performances des requêtes. Les vues et les index présentent certaines similitudes. En effet, il s'agit de structures redondantes qui se partagent la même ressource (*l'espace disque*). Toutes deux peuvent entraîner des surcharges (overhead) causées par des opérations de mise à jour, et demandent un temps de calcul élevé. Une vue matérialisée est stockée sous forme d'une table relationnelle dans le contexte ROLAP, et son indexation peut toujours améliorer la performance des requêtes accédant à cette vue. La présence d'index peut rendre les vues matérialisées plus attractives et vice versa. Cependant, un outil de conception physique d'un entrepôt de données doit prendre en considération *l'interaction* entre les vues matérialisées et les index.

Dans la littérature disponible, la sélection des vues et la sélection des index sont deux tâches **séquentielles**: *la sélection des vues passe avant la sélection des index* (voir Figure 4.1). La sélection séquentielle présente deux inconvénients majeurs:

- *L'impossibilité de considérer plusieurs algorithmes de sélection:* Nous avons vu que les algorithmes de sélection des vues à matérialiser et des index sont contraints par une contrainte donnée (parmi les trois citées dans les chapitres précédents: l'espace disque, le coût de maintenance, le coût de calcul). Rappelons que les PSV et PSI sont formulés de la manière suivante:
  Le PSV consiste à sélectionner un ensemble de vues $\{V_1, V_2, ..., V_k\}$ minimisant/maximisant une fonction "objectif" et satisfaisant une contrainte A. Le PSI sélectionne un ensemble d'index $\{I_1, I_2, ..., I_{k'}\}$ minimisant/maximisant une fonction "objectif" et satisfaisant une contrainte B. Étant donné que les deux contraintes A et B peuvent être semblables ou différentes, la sélection

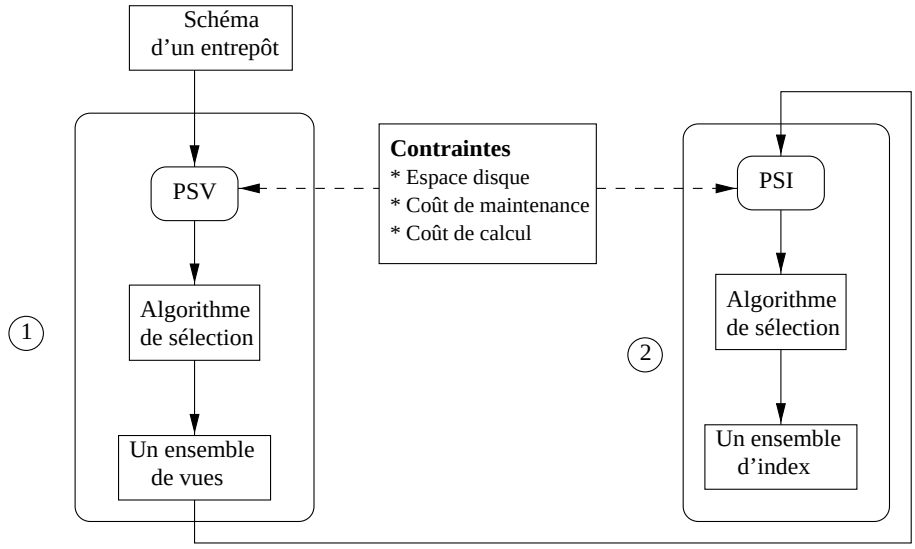


FIG. 4.1: Le processus de sélection des vues et des index

séquentielle peut entraîner la considération de 9 problèmes de sélection correspondant à la combinaison de trois contraintes (voir Figure 4.2). Nous pouvons trouver, par exemple, un algorithme de sélection de vues contraint par l'espace disque, et un algorithme de sélection des index contraint par le coût de maintenance, et ainsi de suite.

- *La non prise en considération de l'interaction entre les vues et les index:* Dans le cas où les deux problèmes ont la même contrainte (c'est-à-dire, $A = B$), trois formulations séquentielles de PSV et PVI sont possibles (voir Figure 4.3). Ignorer l'interaction entre les vues et les index entraîne *un problème de gestion des ressources (représentant les contraintes) entre les vues et les index*. Pour éclairer ce point, supposons que l'administrateur dispose d'une capacité de stockage S destinée aux vues et aux index. Étant donné que les index et les vues sont deux structures en compétition sur la même ressource (l'espace disque), la question qui se pose est de savoir comment cette capacité est distribuée d'une manière automatique entre eux afin de garantir une meilleure performance pour les requêtes.
  Une mauvaise distribution peut compromettre la qualité des solutions obtenues par les algorithmes de sélection des vues et des index.

Dans ce chapitre, nous nous attachons au problème de la gestion de la ressource représentant l'espace disque. Autrementdit le problème de distribution de l'espace disque entre les vues et les index, avec pour objectif l'amélioration de la performance des requêtes.

4.2 Motivations

Le problème de distribution de l'espace entre les vues et les index représente un enjeu important pour les chercheurs et les industriels. En effet, il peut avoir

	Contraintes		
Problème	Espace	Coût de maintenance	Temps de calcul
PSV	√		
PSI	√		
PSV	√		
PSI		√	
PSV	√		
PSI			√
PSV		√	
PSI	√		
PSV		√	
PSI		√	
PSV		√	
PSI			√
PSV			√
PSI	√		
PSV			√
PSI		√	
PSV			√
PSI			√

FIG. 4.2: La formulation séquentielle de PSV et PSI: le cas de $A \neq B$

un effet crucial sur la performance des requêtes et poser des problèmes énormes à l'administrateur dans les cas statique et dynamique, comme nous allons le voir dans les sections qui suivent.

4.2.1 Le cas statique

Pour illustrer ces difficultés, considérons le scénario suivant:
Soit S l'espace disque global que l'administrateur possède pour sélectionner un ensemble de vues V et un ensemble d'index I. Pour répartir S entre V et I, l'administrateur a trois possibilités:

	Contraintes		
Problème	Espace	Coût de maintenance	Temps de calcul
PSV	√		
PSI	√		
PSV		√	
PSI		√	
PSV			√
PSI			√

FIG. 4.3: La formulation séquentielle de PSV et PSI: le cas $A = B$

1. Attribuer la totalité de l'espace disque S aux vues matérialisées seules. Toutes les requêtes décisionnelles sont alors exprimées en fonction des vues et des relations de base. Cette attribution n'est pas efficace. En effet, plusieurs travaux ont démontré que des vues matérialisées seules sont insuffisantes pour accélérer toutes les requêtes décisionnelles [BKL99, LQA97, SSN00]. Cette possibilité est de ce fait écartée.
2. Attribuer la totalité de S aux index seuls. Toutes les requêtes décisionnelles sont maintenant traitées en fonction des index et des relations de base. Or, interroger un entrepôt de données sans vues matérialisées ne garantit aucune performance [KR98, Gup99, YKL97, CY99b]. En conséquence, les index seuls sont insuffisants pour garantir une bonne performance des requêtes décisionnelles. Cette possibilité est donc également écartée.
3. La dernière possibilité consiste en un compromis entre les deux premières possibilités. Ce compromis devrait garantir une bonne performance et de ce fait est l'objet de notre étude.

   Plus formellement, l'administrateur doit déterminer une fraction f $(0 \leq f \leq 1)$ tel que le quota d'espace définie par $f \times S$ soit réservé aux vues matérialisées (ou aux index) et le quota d'espace défini par $(1 - f) \times S$ soit réservé aux index (ou aux vues).

   Trois possibilités de distribution sont possibles: (i) une distribution *uniforme*, (ii) une distribution *aléatoire* et (iii) une distribution *calculée*.

   Définition 29 *La distribution uniforme présente une répartition équitable de S entre les vues et les index ($f = 0.5$).*

   Définition 30 *La distribution aléatoire présente une répartition quelconque de S.*

   Définition 31 *La distribution calculée consiste en l'attribution de l'espace aux vues et aux index, en respectant certains critères.*

   Certes, les distributions uniforme et aléatoire sont faciles à obtenir, mais elles présentent plusieurs inconvénients:

   - Elles ne prennent pas en compte l'interdépendance mutuelle entre les vues et les index (voir Figure 4.1).
   - La distribution aléatoire peut allouer plus d'espace que nécessaire aux vues (ou aux index). Cette allocation se fait au détriment de l'espace des index et vice versa. Notons que pour une exécution efficace des requêtes, il est parfois préférable d'avoir plus d'espace pour les vues que pour les index (ou vice-versa).
   - Ces deux distributions ne possèdent aucune métrique garantissant la réduction du coût de l'ensemble de requêtes.

4.2.2 Le cas dynamique

Étant donné que l'entrepôt de données est un environnement dynamique, le problème de redistribution d'espace entre les vues et les index doit être considéré après

les opérations de mise à jour. Ces dernières interviennent au niveau des tables des sources de l'entrepôt, et les changements correspondants doivent être répercutés sur les vues matérialisées et les index [HMA99]. Ainsi les tailles des vues et des index peuvent augmenter ou diminuer. Il est donc nécessaire de revoir la distribution initiale de l'espace global entre les vues et les index afin de garantir une meilleure performance. Un autre problème pouvant entraîner une redistribution de l'espace est le changement des requêtes de départ (ajout/suppression de requêtes, changement de fréquence d'accès des requêtes, etc). Cela est dû au fait que la plupart des algorithmes de sélection des vues et des index sont basés sur des requêtes connues a priori (voir Chapitre 2).

Dans ce chapitre, nous proposons des solutions au problème de distribution de l'espace entre les vues et les index dans les contextes *statique* et *dynamique*. Les contributions principales de ce chapitre sont: (1) la présentation d'une méthodologie itérative de distribution efficace de l'espace entre les vues et les index, (2) l'évaluation de cette méthodologie et l'illustration de son utilité.

4.3 L'intuition de notre approche de distribution

L'objectif principal à satisfaire pour la sélection des vues matérialisées et la sélection d'index est *la minimisation du coût total d'évaluation des requêtes décisionnelles* [BKS00, YKL97, Gup99, CN97]. Notre solution de distribution de l'espace entre les vues et les index *doit également satisfaire cet objectif.*

L'intuition sous-jacente de notre approche est la suivante:
Initialement, l'administrateur estime les deux quotas d'espace S^V et S^I pour les vues matérialisées et les index respectivement, ($S = S^V + S^I$). Cette estimation peut être établie en utilisant une distribution uniforme ou aléatoire. En fonction du quota réservé aux vues S^V, l'administrateur sélectionne un ensemble de vues $V = \{V_1, V_2, ..., V_s\}$ en utilisant l'un des algorithmes de sélection de vues dirigés par la contrainte d'espace (voir Chapitre 2).

Une fois les vues sélectionnées, un ensemble d'index I est construit $I = \{I_1, ..., I_r\}$ en utilisant le quota réservé aux index (S^I). Rappelons que deux types d'index sont possibles: (i) les index simples utilisés séparément sur des tables de base ou des vues matérialisées, (ii) les index de jointure définis conjointement sur des tables de base et des vues matérialisées. Pour les index simples, nous pouvons utiliser les techniques d'indexation décrites dans le chapitre 2, comme l'index de projection, l'index binaire, etc. Pour les index de jointure, nous pouvons utiliser les algorithmes de sélection des index de jointure introduits dans le chapitre 3.

En conséquence, nous obtenons une solution initiale pour le problème de distribution statique, qui consiste en un ensemble de vues V et un ensemble d'index I. Toutes les requêtes de départ sont donc exécutées en utilisant les deux ensembles (V et I). À l'aide d'un modèle de coût, le coût d'évaluation de l'ensemble des requêtes est calculé.

Le principe de notre approche est de *reconsidérer itérativement* cette solution initiale dans le but de réduire le plus possible le coût d'évaluation des requêtes. Elle

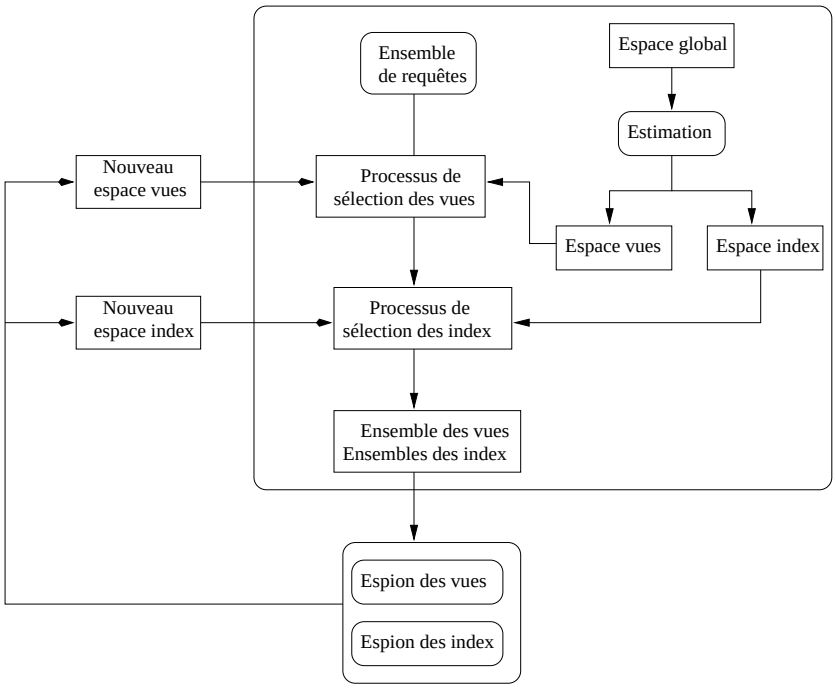


FIG. 4.4: L'intuition de notre approche itérative

s'appuie sur deux agents, **l'espion des index** et **l'espion des vues** dont les rôles sont les suivants:

- L'espion des index a pour tâche de voler de l'espace réservé aux vues. L'espace ainsi récupéré sera utilisé pour créer d'autres index et les vues correspondantes seront supprimées. L'opération est validée si le coût total d'exécution des requêtes diminue.
- D'une manière similaire, l'espion des vues a pour tâche de dérober de l'espace réservé aux index afin de créer d'autre vues dans le but de minimiser le coût total d'exécution des requêtes.

Notre algorithme organise donc une sorte de "combat" entre ces deux espions. Ce combat est pour la bonne cause: *la meilleure utilisation de l'espace disque disponible dans l'objectif de minimiser le coût d'évaluation des requêtes.*

L'algorithme s'achève lorsque les deux espions ne peuvent plus réduire le coût total d'exécution des requêtes. À la fin de cet algorithme, nous obtenons deux résultats principaux:

1. Un ensemble de vues matérialisées $V' = \{V'_1, V'_2, ..., V'_{s'}\}$ et un ensemble d'index $I' = \{I'_1, I'_2, ..., I'_{r'}\}$ garantissant un coût minimal d'exécution des requêtes. Ces deux ensembles peuvent être différents des deux ensembles de départ.
2. De nouveaux quotas d'espace pour les vues matérialisées $S^{V'}$ et les index $S^{I'}$ tels que: $S^{V'} \neq S^V$ et $S^{I'} \neq S^I$.

Un exemple d'illustration

Nous illustrons maintenant notre algorithme par un exemple. Prenons le schéma en étoile de la Figure 2.5 et supposons que nous ayons cinq requêtes fréquentes dont

les graphes sont représentés en Figure 5.1. Les cinq graphes de requêtes représentant les cinq requêtes sont fusionnés en un seul graphe appelé PMEV (voir chapitre 3) [Sel88, YKL97]. Supposons que l'administrateur dispose de 800 Mégabytes et ait affecté **600 Mégabytes** aux vues matérialisées et **200 Mégabytes** aux index.

Premièrement, nous exécutons l'algorithme *sélect_vue* pour sélectionner des vues en utilisant les 600 Mégabytes comme capacité de stockage. Les vues sélectionnées sont V_1, V_2 et V_3 comme le montre la Table 4.1. Le coût total d'évaluation de l'ensemble de requêtes (en nombre des entrées-sorties) est égal à **4054825**.

Vues	Définition de vues	Tailles (bytes)
V_1	VENTES $\bowtie \sigma_{Etat='IL'}$ CLIENT	178 094 080
V_2	VENTES $\bowtie \sigma_{Type\_paquet='Bote'}$ PRODUIT	324 435 968
V_3	$\sigma_{Sexe='M'}$ CLIENT	89 047 040
Total		591 577 088

TAB. 4.1: Les vues matérialisées sélectionnées et leurs tailles

Nous construisons ensuite les index (en utilisant les 200 Mégabytes) afin de réduire encore le coût obtenu par la présence des trois vues matérialisées. Les cinq requêtes sont réécrites en fonction de V_1, V_2, V_3, et des deux tables de dimensions CLIENT et TEMPS. En exécutant l'algorithme de sélection des index proposé dans le chapitre 3, un index de jointure entre la vue V_2 et la table de dimension CLIENT ($V_2 \sim CLIENT$) est sélectionné. La quantité d'espace nécessaire pour cet index est 181821440 bytes (voir chapitre 3). La présence de cet index réduit le coût total d'évaluation des requêtes de **4054825** à **3263405**.

Maintenant, l'espion des index tente de dérober de l'espace attribué aux vues matérialisées afin d'ajouter des index. Puisque la vue V_3 est de petite taille et qu'elle est moins utilisée que les autres vues (V_1 et V_2), son espace (89047040 bytes) est convoité par l'espion des index. L'espace attribué aux index sera donc de 289047040 bytes au total. Après l'élimination de la vue V_3, il ne reste plus que V_1, V_2, CLIENT et TEMPS pour exécuter les cinq requêtes. En utilisant un nouvel espace, l'espion des index sélectionne un nouvel index de jointure ($V_1 \sim V_2 \sim TEMPS \sim CLIENT$). Cet index réduit le coût total d'exécution des requêtes de **3263405** à **1955305**.

Finalement, aucun des deux espions ne peut plus obtenir de réduction, d'où la fin de l'algorithme. L'ensemble des vues matérialisées et l'ensemble des index deviennent respectivement $V = \{V_1, V_2\}$ et $I = \{(V_1 \sim V_2 \sim TEMPS \sim CLIENT)\}$. 500 Mégabytes sont attribués aux vues matérialisées et 300 Mégabytes aux index. Le coût total d'exécution de l'ensemble de requêtes est de 1955305 opérations d'entrées-sorties. Le déroulement de cet algorithme est résumé par la Table 4.2.

4.4 Le problème de la distribution statique

Dans cette section, nous considérons le problème de distribution d'espace dans le cas statique. Tous les paramètres de l'entrepôt (les vues, les index, les tables de bases, etc.) sont connus a priori et restent inchangés. Pour décrire notre approche, nous

Etape 1	Etape 2	Etape 3	Etape 4	Etape 5
Sans vues & index	**Avec vues**	**Avec vues et index**	**Espion des index**	**Espion des vues**
68514928	4054825	3263405	1955305	1955305 Algorithme terminé

TAB. 4.2: Réduction du coût dû à l'espion des index

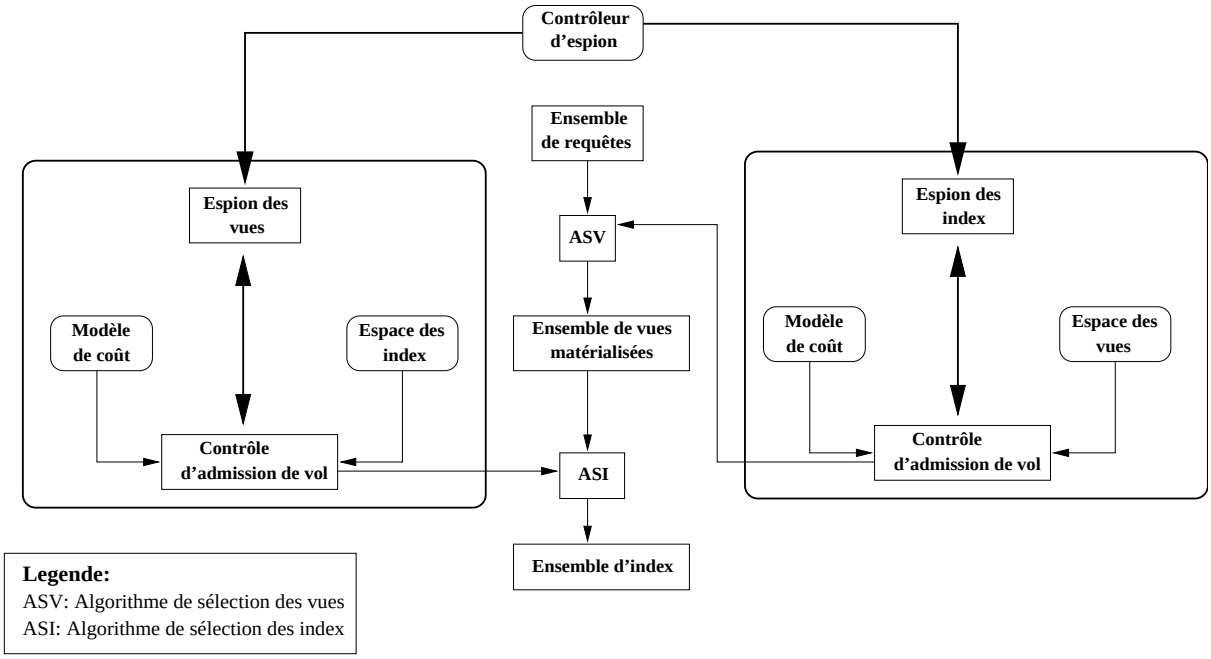


FIG. 4.5: Les éléments de notre algorithme statique et leurs interactions

considérons un entrepôt modélisé par un schéma en étoile. Il est à noter cependant que les solutions proposées peuvent être appliquées à n'importe quel schéma (de type ROLAP ou MOLAP).

4.4.1 La formulation du problème

Le problème de distribution de l'espace entre les vues matérialisées et les index est formulé de la manière suivante:

- Soit un entrepôt de données modélisé par un schéma en étoile.
- Soit $Q = \{Q_1, Q_2, ..., Q_n\}$ un ensemble de requêtes d'interrogation (les plus fréquentes) avec leurs fréquences d'accès $F = \{f_1, f_2, ..., f_n\}$
- Soit S l'espace disque destiné à stocker les vues matérialisées V et les index I, pour supporter l'ensemble des requêtes Q.

L'objectif du problème de distribution de l'espace est de répartir l'espace global S entre les vues (V) et les index (I) afin de minimiser le coût d'évaluation de l'ensemble des requêtes.

4.4.2 L'architecture de notre approche

L'architecture de notre approche de distribution de l'espace est présentée dans la Figure 4.4. Les composantes de cette architecture sont: (1) un espion des vues, (2) un espion des index, (3) un contrôle d'admission de vol, (4) un contrôleur d'espion, (5) un algorithme de sélection des vues, (6) un algorithme de sélection des index, et (7) un modèle de coût. Chaque composante a sa propre tâche:

1. L'espion des vues et l'espion des index: Comme nous l'avons déjà mentionné, les espions des index et des vues ont pour tâche de dérober l'espace réservé aux vues matérialisées (aux index) afin de créer d'autre(s) index (vues) (voir section 4.3).
2. Le contrôle d'admission de vol: Il vérifie que chaque tentative de vol d'un espion (des index ou des vues) contribue à la réduction du coût des requêtes. Cette décision est basée sur un modèle de coût. Pour clarifier le rôle de contrôle d'admission de vol, soient V et I les ensembles des vues et des index déjà sélectionnés. Soit CT_p le coût d'évaluation de l'ensemble de requêtes en présence de $\mathcal{V}$ et de $\mathcal{I}$ (avant la tentative de vol). Supposons que l'espion des vues tente l'opération de vol et que d'autres vues soient créées. Soient $\mathcal{V}'$ le nouvel ensemble de vues et $\mathcal{I}'$ le nouvel ensemble des index. Le contrôle d'admission de vol évalue alors le nouveau coût d'exécution des requêtes en présence de $\mathcal{V}'$ et de $\mathcal{I}'$ qui est noté par CT_c. Si ce dernier est inférieur à CT_p, la tentative de vol est validée par le contrôle d'admission de vol, sinon elle ne l'est pas.
3. Le contrôleur d'espion: Son rôle est d'assurer qu'à chaque moment de l'algorithme, un seul espion est actif.
4. Un algorithme de sélection des vues: Il représente l'algorithme de sélection des vues matérialisées utilisé. Dans cette étude, nous utilisons l'algorithme *sélect_vue* présenté dans le chapitre 3.
5. Un algorithme de sélection des index: Il représente l'algorithme de sélection des index utilisé. Dans cette étude, nous utilisons les techniques d'indexation présentées dans le chapitre 3.
6. Le modèle de coût: Il constitue le coeur de notre approche. Il est utilisé par chacune des composantes et quantifie la solution obtenue. Le modèle de coût décrit dans le chapitre 3 peut être utilisé.

4.4.3 Les étapes de notre algorithme

Rappelons que l'algorithme de distribution est basé sur une approche itérative dont les étapes principales sont les suivantes: (1) la détermination d'une solution initiale, (2) le mécanisme d'amélioration de la solution initiale et (3) le test d'arrêt. Dans les sections suivantes, nous décrivons plus en détail chaque étape.

4.4.4 La solution initiale

Pour amorcer notre algorithme, nous suivons la sélection séquentielle des vues et des index. Nous estimons (d'une manière uniforme ou aléatoire) les deux quotas

de l'espace des vues matérialisées S^V et l'espace des index S^I. Tout d'abord, nous exécutons un algorithme de sélection des vues à matérialiser dirigé par l'espace S^V [YKL97]. Comme résultat, nous obtenons un ensemble de vues $V = \{V_1, V_2, ..., V_m\}$. Ensuite, en utilisant le quota de l'espace réservé aux index, nous sélectionnons un ensemble d'index $I = \{I_1, I_2, ..., I_l\}$ (voir Chapitre 3). Finalement, nous calculons le coût d'évaluation de l'ensemble des requêtes en présence des ensembles de vues (V) et des index (I) noté par $CTER_i$.

4.4.5 Le processus d'amélioration de la solution initiale

Pour réaliser cette amélioration, l'algorithme utilise les deux espions décrits précédemment. La tâche des ces espions est contrôlée par les règles suivantes:

Règle 1: un espion doit être capable d'utiliser l'espace volé, c'est-à-dire qu'il doit trouver une vue (pour l'espion d'index) ou un index (pour l'espion des vues) à ranger dans l'espace volé.

Règle 2: l'espion ne peut voler que si le coût global d'exécution des requêtes est réduit après l'utilisation de l'espace volé.

Règle 3: un seul espion est actif à la fois.

4.4.5.1 Les structures utilisées dans notre algorithme

Les structures de données nécessaires à l'algorithme itératif sont deux matrices appelées: *la matrice de réécriture* et *la matrice requête-index*.

La matrice de réécriture W a déjà été définie dans le chapitre précédent (rappelons que cette dernière est une représentation du graphe de jointure). Nous ajoutons une autre colonne à cette matrice représentant la fréquence de chaque requête (voir Figure 4.6). Cette matrice sert à identifier les relations de base ou les vues utilisées pour une requête donnée.

Nous attribuons à chaque vue V_i une *fréquence d'accès* dénotée par f_{V_i} obtenue par la somme des fréquences des requêtes accédant à cette vue (voir équation 4.1):

$$f_{V_i} = \sum_{j=1}^{n} w_{ij} * f_j \tag{4.1}$$

Les victimes potentielles de l'espion des index sont les vues figurant dans cette matrice.

La deuxième matrice, que nous appelons requête-index $RI = \{ri_{lk} \mid 1 \leq l \leq n, 1 \leq k \leq m\}$, est telle que :

$$\mathbf{ri_{lk}} = \begin{cases} 1, \text{ si la requête } Q_l \text{ utilise l'index } I_k \\ 0, \text{ sinon} \end{cases}$$

D'une manière similaire, nous ajoutons une nouvelle colonne pour les fréquences et nous attribuons à chaque index I_k une *fréquence d'accès* dénotée par f_{I_k}, obtenue par la sommation des fréquences des requêtes (voir équation 4.2):

$$f_{I_k} = \sum_{j=1}^{n} ri_{kj} * f_j \tag{4.2}$$

Les victimes potentielles de l'espion des vues sont les index figurant dans cette matrice.

Notre algorithme utilise deux procédures: espion-index-vol-espace-vue et espion-vue-vol-espace-index, que nous décrivons dans les sections qui suivent.

Procédure espion-index-vol-espace-vue Comme son nom l'indique, cette procédure décrit la tâche de l'espion des index. Elle présente en entrée les éléments suivants:

- Un ensemble de requêtes d'interrogation;
- Un ensemble de vues matérialisées $V = \{V_1, V_2, ..., V_l\}$ et un ensemble d'index $I = \{I_1, I_2, ..., I_m\}$, obtenus par la solution initiale;
- Les deux quotas de stockage S^V (pour les vues) et S^I (pour les index).

Comme sortie, cette procédure génère un ensemble d'index $I' = \{I'_1, I'_2, ..., I'_{m'}\}$ qui peut être soit identique à l'ensemble initial I soit différent et plus intéressant en termes de réduction du coût des requêtes.

Pour identifier les victimes des vues matérialisées, l'espion des index doit avoir une politique de sélection de vue à éliminer pour créer de nouveaux index en garantissant une bonne performance. Nous suggérons deux politiques: la Vue la Moins Fréquemment Utilisée (VMFU) et la Vue de plus Petite Taille (VPPT).

- **La VMFU:** l'espion des index doit sélectionner la vue la moins fréquemment utilisée. Cette vue est celle qui contribue le moins à l'amélioration du coût d'exécution des requêtes.
- **La VPPT:** l'espion des index doit sélectionner la vue ayant la petite taille. En effet, la taille des vues étant en moyenne plus grande que la taille des index, une petite vue pourra être remplacée par un index.

Une fois la vue victime éliminée, les requêtes de départ doivent être réécrites en fonction des vues restantes. D'où la mise à jour de la matrice de réécriture.

Exemple 17 *Cet exemple illustre le fonctionnement de l'espion des index. Supposons que nous ayons trois vues matérialisées V_1, V_2 et V_3 comme dans l'exemple d'illustration occupant 600 Mégabytes et un index ($V_2 \sim CLIENT$). À partir de ces vues et index, nous construisons la matrice de réécriture W (voir Figure 4.6). La fréquence d'accès de chaque vue V_i $(1 \leq i \leq 3)$ est calculée: $f_{V_1} = 30$, $f_{V_2} = 25$ et $f_{V_3} = 20$.*

Étant donné que l'espion des index élimine la vue V_3 qui est la moins fréquente, la matrice de réécriture doit être modifiée de la manière suivante: la vue V_3 est remplacée par la table de dimension CLIENT comme le montre la Figure 4.7.

4.4.5.2 Algorithme de l'espion des index

La description détaillée de la tâche de l'espion des index est donnée dans l'algorithme 7.

	V_1	V_2	V_3	T	FQ
Q_1	1	0	0	0	20
Q_2	0	1	0	1	5
Q_3	0	1	1	1	5
Q_4	1	0	0	1	10
Q_5	0	1	1	0	15

FIG. 4.6: La matrice de réécriture initiale

	V_1	V_2	C	T	FQ
Q_1	1	0	0	0	20
Q_2	0	1	0	1	5
Q_3	0	1	1	1	5
Q_4	1	0	0	1	10
Q_5	0	1	1	0	15

FIG. 4.7: La matrice de réécriture après l'exécution de l'espion des index

Algorithme 7 La procédure espion-index-vol-espace-vue

1: Calculer le coût global G_i d'évaluation des requêtes en utilisant les vues et les index
2: Sélectionner une vue V_i en utilisant une stratégie parmi VMFU ou VPT
3: $S^I = S^I + espace(V_i)$ + l'espace inutilisé
4: $S^V = S^V - espace(V_i)$
5: Éliminer la vue V_i
6: Mettre à jour la matrice de réécriture W
7: Exécuter ASI (on obtient un nouvel ensemble d'index)
8: Calculer le coût global G_c d'évaluation des requêtes en utilisant le nouvel ensemble des index et les vues matérialisées restantes
9: **if** $(G_c < G_i)$ **then**
10: Mettre à jour la matrice requête-index RI
11: **else**
12: Restaurer la vue éliminée
13: Rétablir S^V, S^I et G_c à leur valeurs précédentes
14: Rétablir la matrice W à son état précédent
15: **end if**

Procédure espion-vues-vol-espace-index Cette procédure fonctionne comme celle de l'espion des index. Elle possède les mêmes entrées que la procédure espion-index-vol-espace-vue. Comme sortie, elle génère un ensemble de vues $\mathcal{V}' = \{V'_1, V'_2, ..., V'_{l'}\}$ qui peut être soit identique à l'ensemble initial V soit plus intéressant par rapport à V en termes de réduction de coût des requêtes.

Pour sélectionner les victimes, l'espion des vues possède deux stratégies: **l'Index le Moins Fréquemment Utilisé (IMFU)** et **le Plus Grand Index (PGI)**.

- **L'IMFU:** l'espion des vues doit sélectionner l'index le moins fréquemment utilisé. Cet index est celui qui contribue le moins à la réduction du coût d'exécution des requêtes.
- **La PGI:** Le plus grand index est sélectionné parce que la taille des vues est en moyenne plus grande que la taille des index. En effet, un index de petite taille ne présente pas en général suffisamment d'espace pour stocker une nouvelle vue.

4.4.5.3 Algorithme de l'espion des vues

Les étapes principales de l'espion des vues sont illustrées par l'algorithme 8.

Algorithme 8 La procédure espion-vues-vol-espace-index

1: Calculer le coût global G_i d'évaluation des requêtes en utilisant les vues et les index
2: Sélectionner un index I_i en utilisant une stratégie parmi IMFU ou PGI
3: $S^V = S^V + espace(I_i)$ + espace inutilisé
4: $S^I = S^I - space(I_i)$
5: Éliminer I_i
6: Mettre à jour la matrice RI
7: Exécuter Select_Vues (il fournit un nouvel ensemble de vues matérialisées)
8: Exécuter l'algorithme ASI
9: Calculer le coût global G_c d'évaluation des requêtes en utilisant le nouvel ensemble des index et les vues matérialisées
10: **if** $(G_c < G_i)$ **then**
11: Mettre à jour les deux matrices W et RI
12: **else**
13: Restaurer l'index éliminé
14: Rétablir S^V, S^I et G_c à leur valeurs précédentes
15: Rétablir la matrice RI à son état précédent
16: **end if**

4.4.6 Le test d'arrêt

Nous exécutons ces deux procédures jusqu'à ce qu'aucune réduction du coût global d'évaluation de l'ensemble de requêtes ne soit constatée.

4.4.7 La détermination de l'espion privilégié

L'objectif principal de notre algorithme est d'améliorer la solution initiale obtenue par l'estimation de l'espace. Notons que nous possédons deux espions, chacun ayant le droit de commencer son vol. Cependant, ils ne peuvent pas commencer simultanément (voir la règle 3). Un espion doit donc recevoir le privilège de commencer le vol. Reste à identifier cet espion, car cette identification peut avoir des effets importants sur la qualité de la solution obtenue par notre algorithme.

Pour réaliser la tâche d'identification, nous suggérons un critère de sélection basé sur la notion de *contribution de chaque espion* définie par la suite dans cette section.

Soient CT_s, CT_v et CT_i le coût total d'évaluation de l'ensemble des requêtes de départ sans les vues et les index, en présence seulement des vues, ou en présence seulement des index.
Nous définissons la **contribution de l'espion des vues** $cont(V)$ par l'équation 4.3:

$$cont(V) = CT_s - CT_v \tag{4.3}$$

Elle nous donne la réduction du coût d'évaluation des requêtes obtenue par la sélection des vues matérialisées. D'une manière similaire, nous définissons **la contribution de l'espion des index** $cont(I)$ par l'équation 4.4:

$$cont(I) = CT_s - CT_i \tag{4.4}$$

Cette contribution nous offre une réduction du coût d'évaluation des requêtes obtenue par la sélection des index.

En utilisant ces deux contributions, nous définissons l'espion à privilégier comme l'espion ayant la plus grande contribution. Plus formellement, il est défini de la manière suivante:

$$\text{l'espion à privilégier} = \begin{cases} \text{l'espion des vues si } cont(V) < cont(I) \\ \text{l'espion des index, sinon} \end{cases}$$

En cas d'égalité des contributions, nous sélectionnons un espion au hasard.

4.4.8 Description de l'algorithme

Maintenant, nous avons tous les ingrédients pour définir l'algorithme itératif distribuant l'espace entre les vues et les index. Il commence par la recherche d'une solution initiale. Une fois cette solution trouvée, il construit les deux matrices W et RI. En fonction de la contribution de chaque espion, l'algorithme active l'espion privilégié suivi par l'autre espion. Le processus d'activation des espions est répété

jusqu'à ce qu'aucune réduction au niveau du coût total d'évaluation des requêtes ne soit plus constatée.

Les étapes principales de notre approche sont décrites dans l'algorithme 9.

Algorithme 9 Algorithme statique

1: Initialisation: les deux matrice (W et RI) sont initialisées à zéro
2: Exécuter Select_Vues en respectant la contrainte d'espace S^V.
3: Construire la matrice W en fonction de l'ensemble des vues sélectionnées
4: Exécuter ASI en respectant la contrainte d'espace S^I
5: Construire la matrice RI en fonction de l'ensemble des index sélectionnés
6: Activer espion-index-vol-espace-vues et espion-vues-vol-espace-index (le choix dépend de leurs contributions respectives)
7: Répéter l'étape 6 jusqu'à ce qu'aucune réduction du coût total d'évaluation des requêtes ne soit constatée

Notre algorithme se termine parce qu'il est contrôlé par un modèle de coût.

4.4.9 Les caractéristiques de notre approche

Dans un entrepôt de données, le processus de traitement des requêtes possède plusieurs modules en relation; citons par exemple, des algorithmes de sélection des vues et des index, la réécriture des requêtes en fonction des vues, des algorithmes de maintenance des vues et des index, etc. L'algorithme que nous proposons ne remet pas en cause les modules existants d'un entrepôt, mais il peut les utiliser comme des primitives. Pour éclairer ce point, soient A et B deux algorithmes, le premier dédié à la sélection des vues et le deuxième à la sélection des index. Notre algorithme de distribution C se présente d'abord comme une "procédure maître" pouvant utiliser ces deux algorithmes (Figure 4.4).
De plus, notre méthodologie donne une certaine flexibilité à l'administrateur dans le choix d'un espion (des index ou des vues) pour la conception de son entrepôt. Dans cette optique, l'administrateur, selon son besoin, pourra attribuer plus d'espace aux index ou aux vues matérialisées.

4.5 La problème de la distribution dynamique

Dans la section précédente, nous avons étudié le problème de distribution de l'espace dans le contexte statique. Dans cette section, nous traitons ce problème dans le cas dynamique en adaptant l'algorithme statique.

4.5.1 Position du problème

Rappelons que les entrepôts sont construits en collectant les informations de plusieurs sources et en les intégrant dans une seule base de données. Les entrepôts

sont volumineux pour des applications fonctionnant dans des environnements de grande échelle composées de multiples sources hétérogènes, tel que le Web. De tels environnements sont souvent marqués par des changements dynamiques au niveau des sources d'informations, modifiant non seulement les contenus des données, mais également leurs schémas et leurs capacités de requêtes. Ces changements peuvent provoquer des modifications des vues et des index définis sur l'entrepôt.

Les changements dynamiques sur les vues peuvent être classés en trois catégories:

1. Des changements au niveau des relations de base (ajout/suppression de nouveaux n-uplets) qui seront ensuite répercutés sur les vues matérialisées (cf. le problème de maintenance des vues: chapitre 2).
2. Des changements au niveau de la définition des vues: Notons que les exigences des utilisateurs de l'entrepôt changent dans le temps; elles peuvent entraîner des modifications au niveau de la définition des vues d'une manière dynamique. Dans ces applications, les utilisateurs changent les définitions des vues d'une manière dynamique et veulent voir ces changements se refléter dans le résultat de leurs requêtes.

   Un autre cas entraînant un changement dans la définition des vues est celui où une relation d'un système distribué est fragmentée et/ou la localisation physique des données change; la définition des vues doit changer pour fournir l'indépendance logique des données.

   Les changements de définition d'une vue peuvent être très coûteux, surtout si la vue implique des données de sources multiples. Plusieurs travaux ont considéré le problème d'adaptation des vues existantes pour répercuter les changements au niveau des vues [Bel98].
3. Changements dus à l'évolution du schéma
4. Changements au niveau des requêtes: Certaines requêtes peuvent être supprimées, d'autres ajoutées. Les fréquences peuvent changer. Puisque le PSV est basé sur un ensemble donné de requêtes, ces changements entraînent la sélection d'autres vues.

Quelle que soit la nature du changement, il peut avoir une forte incidence sur l'ensemble des vues et des index. Une vue après la maintenance peut occuper plus ou moins d'espace par rapport à son espace initial. Ceci est valable également pour les index. La contrainte d'espace risque de ne plus être respectée. Si l'espace total occupé par les vues et les index est plus grand (respectivement plus petit) que l'espace demandé, il est nécessaire de **déterminer** quelle vue ou quel index doit être éliminé (ou ajouté) [1]. Notre approche itérative peut facilement être adaptée pour traiter ce problème en ajustant les espaces réservés aux vues et aux index. Dans la section suivante, nous décrivons les étapes de l'algorithme que nous appelons *répartition_dynamique*.

[1] Notre approche ne prend pas en considération le coût de création et de suppression d'une vue et d'un index. Car notre objectif principal est de satisfaire la contrainte de l'espace. Il serait cependant intéressant de combiner la contrainte de l'espace et la contrainte du coût de calcul pour le problème de redistribution de l'espace entre les vues et les index.

4.5.2 L'algorithme répartition_dynamique

Soient S_i^V et S_i^I les espaces occupés par les vues et les index avant les opérations de mise à jour. Après ces dernières, l'ensemble des vues matérialisées V et l'ensemble des index I occupent de nouveaux quotas d'espace nommés S_c^V et S_c^I, respectivement. Soit S_i $(= S_i^V + S_i^I)$ et S_c $(= S_c^V + S_c^I)$ les espaces totaux avant et après les opérations de mises à jour. Notre algorithme exécute les étapes suivantes pour redistribuer et gérer les nouveaux quotas d'espace:

1. Calculer les nouveaux quotas de l'espace disque occupé par les vues (S_c^V) et les index (S_c^I)
2. Si l'espace total S_c diminue (en cas d'opération de suppression par exemple), nous aurons de l'espace non utilisé qui pourrait être utilisé par les vues, les index ou les deux. Dans le cas où cet espace est utilisé pour la sélection d'autres vues, l'algorithme cherche des vues appartenant à la couche des vues potentielles (voir chapitre 3) et ayant un bénéfice supérieur à α (voir Section 3.3). Dans le cas où cet espace est utilisé pour sélectionner d'autres index, l'algorithme cherche à les sélectionner d'une manière incrémentale en utilisant l'algorithme $AGSK$ (voir chapitre 3).
3. Si l'espace global S_c augmente (en cas de création par exemple) et devient plus grand que l'espace initial S_i, il faut libérer de l'espace. On peut considérer différents cas:
   - Augmentation de la taille des vues seulement: Si l'espace des vues S_c^V dépasse son quota initial (S_i^V), nous devons donc éliminer certaine(s) vues afin de satisfaire la contrainte d'espace. Mais avant d'entamer le processus d'élimination, nous devons tester si l'espion des vues peut voler de l'espace réservé aux index et garder ces vues (si ces vues contribuent fortement à réduction du coût total des requêtes). Si la tentative de vol de l'espion des vues échoue, nous éliminons certaines vues en utilisant une politique parmi VMFU et VPPT.
   - Augmentation de la taille des index seulement: Si l'espace des index S_c^I dépasse son quota initial (S_i^I), comme dans le cas précédent, nous testons si l'espion des index peut voler de l'espace réservé aux vues et garder ces index. Si cette tentative échoue, nous éliminons les index en utilisant une politique parmi IMFU et PGI.
   - Augmentation de la taille des vues et les index: Si les deux espaces sont plus grands que leurs espaces d'origine, nous éliminons des vues et des index en utilisant les politiques des vues et des index jusqu'à ce que la contrainte d'espace soit satisfaite.

4.6 Evaluation

Notre expérimentation utilise le schéma en étoile de la Figure 2.5. Nous supposons que la taille de la page est de 8192 bytes et que la taille du pointeur de bloc est de

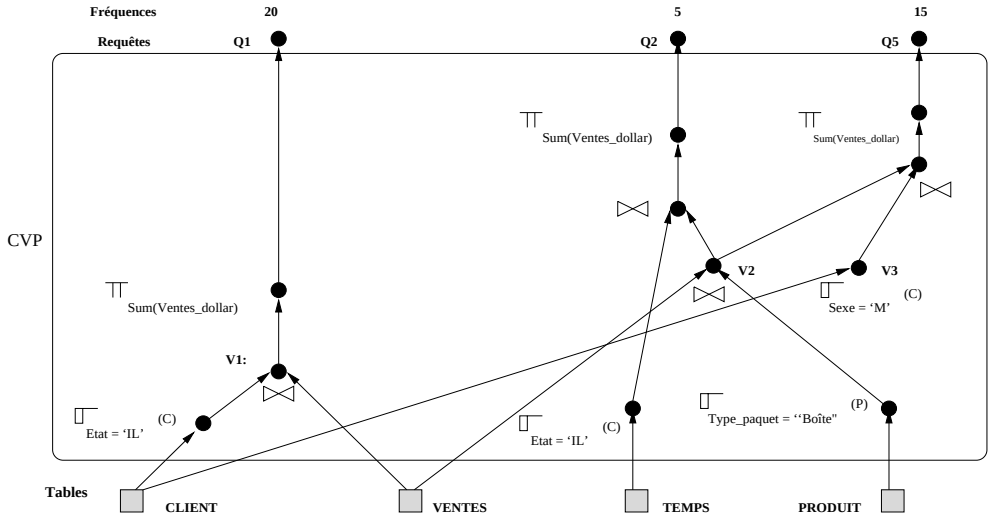


FIG. 4.8: Le PMEV pour les requêtes: Q_1, Q_2 and Q_3

8 bytes comme dans [DRT99]. À partir de ces informations et des cardinalités de chaque table, nous pouvons facilement calculer la taille de chaque table. Pour évaluer notre algorithme, nous utilisons deux ensembles de requêtes: le premier est l'ensemble des cinq requêtes $\{Q_1, Q_2, Q_3, Q_4, Q_5\}$ (Figure 5.1) et le deuxième est l'ensemble des trois premières requêtes $\{Q_1, Q_2, Q_3\}$. Les PMEVs, pour le premier et le deuxième ensemble de requêtes sont illustrés par les Figures 3.8 et 4.8, respectivement. Pour le premier ensemble, la capacité de stockage $S^V = 600$ Mégabytes pour les vues matérialisées et $S^I = 200$ Mégabytes pour les index. Pour le deuxième ensemble, ces valeurs sont: $S^V = 200$ Mégabytes and $S^I = 600$ Mégabytes, respectivement.

Solution	**Vue & Index**	**CT**	S^V	S^I
Solution initiale	Sans vue & index	68514928	0	0
Avec vues	V_1: VENTES $\bowtie \sigma_{Etat='IL'}$ CLIENT V_2: VENTES $\bowtie \sigma_{Type\_paquet='Bote'}$ PRODUIT V_3: $\sigma_{Sexe='M'}$ CLIENT	4054825	600	0
avec vues et index	V_1, V_2, V_3 & I_1 ($V_2 \sim V_3$)	3263405	600	200
Espion des vues		3263405	600	200
Espion des index	$V_1 \sim V_2 \sim TEMPS \sim CLIENT$	1955305	500	300

TAB. 4.3: La description des solutions pour les cinq requêtes

Solution	**Vues et index**	**CT**	S^V	S^I
Solution initiale	Sans vue et index	37371780	0	0
Avec vue	V_1: VENTES $\bowtie \sigma_{Etat='IL'}$ CLIENT	12892060	178	0
Avec vues et index		12892060	178	600
Espion des vues	V_2: VENTES $\bowtie \sigma_{Mois="Mars"}$ TEMPS	2017640	345	433
Espion des vues	V_3: VENTES $\bowtie \sigma_{Type\_paquet='Bote'}$ PRODUIT	1274237	667	109

TAB. 4.4: La description des solutions pour les trois requêtes

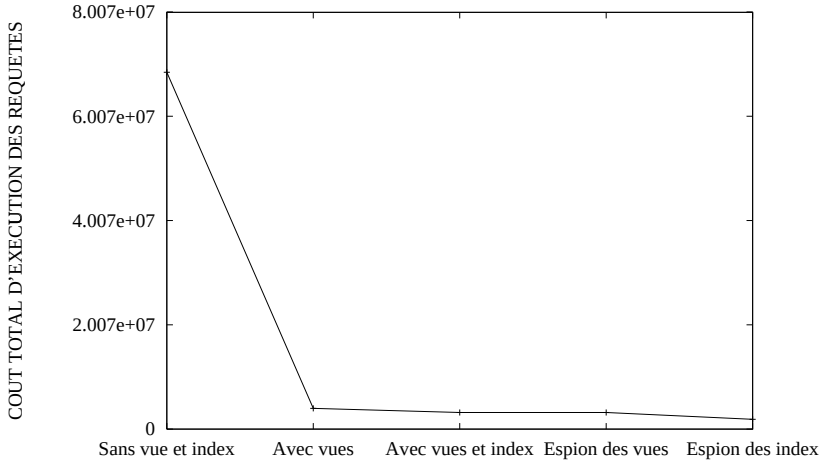


FIG. 4.9: Les effets des espions dans le cas des cinq requêtes

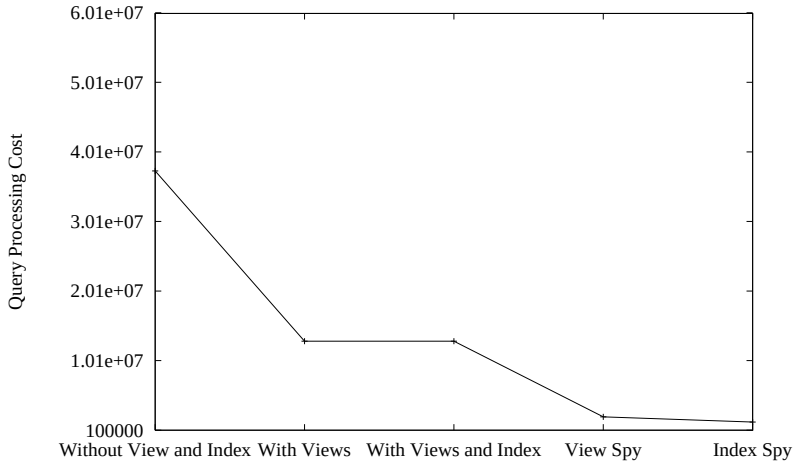


FIG. 4.10: Les effets des espions dans le cas des trois requêtes

Pour les deux ensembles de requêtes, le coût d'évaluation des requêtes est d'abord calculé sans considérer les vues et les index (avec la technique de hachage). Nous sélectionnons un ensemble de vues en utilisant l'algorithme *sélect_vue*. Ces vues réduisent d'une manière significative le coût initial obtenu par la technique de hachage (Figure 4.9). Ensuite, nous sélectionnons des index réduisant le coût. Une fois la solution initiale obtenue, nous activons l'espion des vues car il présente une plus grande contribution que l'espion des index. Cet espion échoue dans la réduction du coût des requêtes (Figure 4.9). L'espion des index est alors activé et réduit le coût des requêtes. Les vues sélectionnées sont: V_1 et V_2, et l'index sélectionné est $(V_1 \sim V_2 \sim TEMPS \sim CLIENT)$ (voir Table 4.3).

Dans la Table 4.4, nous présentons les résultats de notre algorithme pour le deuxième ensemble de requêtes. Nous constatons que le quota réservé aux index (600MB) n'est pas suffisant pour sélectionner des index. En conséquence, cet espace est utilisé par l'espion des vues qui sélectionne deux autres vues.

Dans cet exemple, nous pouvons faire l'observation suivante: l'approche espion

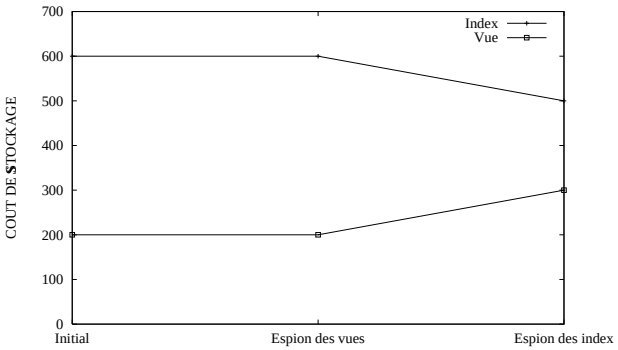


FIG. 4.11: La distribution d'espace pour les cinq requêtes

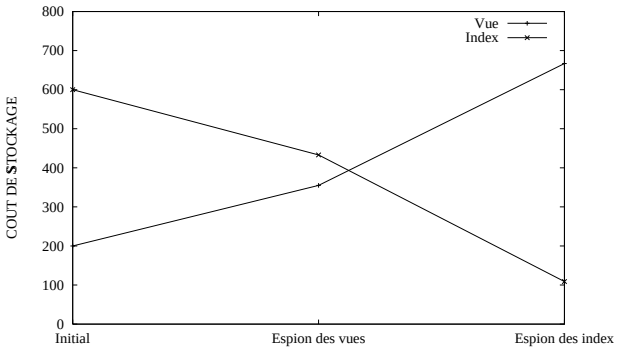


FIG. 4.12: La distribution d'espace pour les trois requêtes

contre espion distribue l'espace avec l'objectif de minimiser le coût d'évaluation des requêtes. Dans les deux cas, nous constatons une réduction significative du coût des requêtes. Nous notons que dans cette évaluation, le coût total d'évaluation des requêtes en utilisant la solution initiale (c'est-à-dire, sans l'application de notre approche) est trois fois plus coûteuse dans le cas des cinq requêtes et dix fois plus coûteuse dans le cas des trois requêtes. On constate aussi que pour les cinq requêtes l'espion des index **réduit** davantage le coût que l'espion des vues. Dans ce cas, l'espion des index vole 100 Mégabytes sur les 600 alloués aux vues matérialisées (Figure 4.11). Mais pour les trois requêtes, l'espion des vues **réduit** davantage le coût que l'espion des index. Dans ce cas, l'espion des vues vole 491 Mégabytes d'espace des 600 Mégabytes alloués aux index (Figure 4.12). On peut donc considérer que notre approche contribue effectivement à une meilleure distribution de l'espace disponible entre les vues et les index afin de réduire le coût total d'évaluation des requêtes.

Nous évaluons maintenant notre approche itérative dans le cas dynamique. Supposons que 500 000 nouveaux n-uplets soient ajoutés à la table des faits. Ils doivent être propagés dans les vues matérialisées et les index obtenus par l'algorithme statique (Table 4.3). Nous supposons que 40% de ces n-uplets sont propagés dans la vue V_1 (VENTES $\bowtie \sigma_{Etat='IL'}$ CLIENT) et la vue V_2 (VENTES $\bowtie \sigma_{Type\_paquet='Bote'}$ PRODUIT). Après ces mises à jour, l'espace total occupé par les vues et les index est de 860 Mégabytes et la contrainte de stockage est violée. Il est nécessaire de redistribuer l'espace entre les vues et les index. Puisque les deux vues matérialisées contribuent le plus à la réduction du coût d'évaluation des cinq requêtes, l'espion des vues essaye d'abord de voler de l'espace aux index pour garder ces vues. Il n'existe qu'un seul index défini sur les vues V_1 and V_2 et les deux tables de dimension CLIENT et TEMPS. L'espion des vues s'attribue donc l'espace alloué à cet index et sélectionne une nouvelle vue V_3 ($\sigma_{Sexe='M'}$ CLIENT). Ces trois vues V_1, V_2, V_3 occupent maintenant 620 Mégabytes et il reste 180 Mégabytes. L'espion des index intervient donc en éliminant la vue V_3 et en récupérant l'espace disponible. Il provoque la création de l'index: ($V_2 \sim$ TEMPS $\sim$ CLIENT) qui réduit d'une manière significative le coût total des requêtes.

Solution	Vues et index	CT	S^V	S^I
Solution statique	V_1, V_2 & ($V_1 \sim V_2 \sim TEMPS \sim CLIENT$)	1955305	500	300
Après mise à jour	V_1, V_2 & ($V_1 \sim V_2 \sim TEMPS \sim CLIENT$)	2022680	540	320
Espion des vues	V_1, V_2 and V_3	4848146	800	0
Espion des index	V_1, V_2 & ($V_2 \sim$ TEMPS $\sim$ CLIENT)	2478150	550	250

TAB. 4.5: Les vues sélectionnées et leurs tailles pour les cinq requêtes après mises à jour

4.7 Conclusion et perspectives

La conception physique des entrepôts de données est très importante pour l'efficacité des requêtes. Deux techniques ont été proposées dans la littérature pour assurer cette efficacité: les vues matérialisées et les index. Ces deux structures sont interdépendantes et contribuent au même objectif: réduire le coût d'évaluation des requêtes. Pour exploiter ces deux techniques, le concepteur de l'entrepôt doit successivement (1) réserver un espace aux vues et (2) réserver un espace aux index. Cette approche ignore complètement les effets d'interaction entre les vues et les index. Dans la littérature cette réservation n'est pas automatisée.

Dans ce chapitre, nous avons développé une nouvelle méthodologie de la distribution automatique de l'espace entre les vues et les index dans les contextes statique et dynamique en prenant en considération l'interaction entre les vues et les index. Cette méthodologie est basée sur une approche itérative. Elle commence par une sélection initiale des vues et des index, puis essaye d'améliorer cette dernière en utilisant deux algorithmes gloutons (espion des index et espion des vues).

Finalement, nous avons évalué notre approche en utilisant un schéma en étoile emprunté à Informix [Cor97b]. Les résultats ont montré que notre approche réduit d'une manière significative le coût total d'évaluation des requêtes. Dans certains cas, c'est l'espion des vues qui contribue le plus et dans d'autres, c'est l'espion des index. Cela démontre l'intérêt d'opposer les deux espions. Cependant, l'administrateur de l'entrepôt peut privilégier les vues matérialisées en exécutant seulement l'espion des vues, ou privilégier les index en exécutant seulement l'espion des index. Ainsi, cette approche peut conduire à plusieurs stratégies de conception d'un entrepôt.

Pour identifier les victimes (vues ou index à éliminer) notre algorithme utilise quatre politiques différentes basées sur la fréquence (la vue la moins fréquemment utilisée, l'index le moins fréquemment utilisé) et la taille (la vue de plus petite taille, l'index de plus grande taille). Il serait intéressant de considérer d'autres politiques en se basant sur des critères de coût (en éliminant les vues ou les index qui contribuent le moins dans la réduction du coût d'évaluation des requêtes).

Chapitre 5

La fragmentation dans les entrepôts de données

L'une des caractéristiques principales des entrepôts de données est leur grande taille. Cette dernière est à l'origine de contre-performance dans l'exécution des requêtes. Par exemple, Jagadish et al. [JMS95] citent le cas d'une grande compagnie de télécommunication qui collecte 75 GB de données d'appels téléphoniques chaque jour (ce qui représente 27 TB par an)! Nous avons vu dans les chapitres précédents que les vues matérialisées et les index jouent un rôle très important dans la réduction du temps d'exécution des requêtes décisionnelles. Leurs tailles peuvent être importantes et leur stockage sur mémoire secondaire et leur maintenance posent des problèmes délicats. Pour exécuter des requêtes décisionnelles, les vues et les index doivent être chargés en mémoire centrale; opération qui peut être *très coûteuse*.

La fragmentation horizontale semble être une réponse à ce problème. En effet elle a été introduite dans les bases de données réparties dans le but de minimiser le nombre d'entrées-sorties (ou le coût de transfert de données) pendant l'exécution des requêtes [KNM94, OV91, NR89]. Malheureusement, elle n'a pas reçu une grande attention de la part des chercheurs dans le contexte des entrepôts des données. Récemment, *Oracle8i* a introduit le concept de fragmentation horizontale de la table des faits, des vues matérialisées et des index [Ora99] et a démontré son intérêt. Notons au passage que le benchmark de Transaction Processing Concil (TPC-H), dans ses règles d'implémentation, n'autorise pas l'utilisation de la fragmentation verticale, mais accepte l'utilisation de la fragmentation horizontale.

Le partitionnement peut être utilisé à différents niveaux dans les entrepôts de données ou les très grandes bases de données pour permettre diverses améliorations dans les performances:

- On peut définir des vues matérialisées et des index sur des partitions et non pas sur la totalité de la base de données ou de l'entrepôt; chaque partition est gérée indépendamment des autres,
- On peut éliminer les partitions non valides pour réduire le temps de maintenance et d'interrogation. Admettons par exemple que la table des faits VENTES de la Figure 2.5 soit partitionnée en mois (12 partitions). Pour exécuter une requête demandant des informations sur les trois derniers mois, il

est possible d'éliminer 9 partitions,

- On peut améliorer les performances en utilisant la technique du "striping" [WLDH96]. Cette technique consiste à répartir les fragments de données sur des disques différents, ce qui permet de réduire les déplacements des têtes, et de réaliser plusieurs opérations de lecture ou d'écriture en parallèle. On peut, par exemple utiliser 5 disques de 200 MB pour obtenir un volume de 1 GB. Les performances du système à 5 disques seront, on le comprend aisément, supérieures à celles d'un disque monobloc. Par contre, la fiabilité de l'ensemble n'est pas améliorée puisqu'il faut que l'ensemble des 5 disques soit opérationnel en permanence. La défaillance d'un seul disque met le système en panne,
- Il est possible de tirer partie de la fragmentation pour concevoir des architectures efficaces d'entrepôts répartis [NB99]. Les fragments peuvent être répartis judicieusement sur des sites différents pour réduire le nombre d'accès aux données non pertinentes et les transferts de données [KNM94],
- Un outil de conception des data marts à partir d'un entrepôt de données.

Dans ce chapitre, nous répondons à la question suivante: *que peut apporter la fragmentation horizontale aux entrepôts de données en terme de réduction du temps d'exécution des requêtes et à la conception des data marts?*

5.1 Position du problème

La fragmentation contribue considérablement à la performance des entrepôts de données en rendant plus efficace *les vues matérialisées, les index, les jointures, le parallélisme*. Nous explicitons ci-après ces différents points.

- Les vues matérialisées: étant donné que les vues matérialisées sont stockées sous forme de relations de grandes tailles, il est intéressant de les fragmenter pour garantir les caractéristiques suivantes:
  1. Faciliter le processus d'adaptation des vues (maintenir un fragment d'une vue est plus facile que maintenir la totalité de la vue [BM00]).
  2. Réduire le coût d'exécution des requêtes en minimisant le nombre d'entrées-sorties.
  3. Rendre les vues auto-maintenables [BM00].

  La fragmentation des vues matérialisées peut être réalisée de deux façons différentes: *directe* et *indirecte*.

  1. La fragmentation directe: Dans ce cas, ce ne sont pas les tables du schéma de l'entrepôt qui sont fragmentées, mais les vues matérialisées définies sur ce schéma. Cette approche a été développée par Bellahsène et al. [BM00] dans l'objectif de rendre les vues auto-maintenables en cas de modification des données.
  2. La fragmentation indirecte: Dans ce cas, les tables du schéma sont supposées fragmentées. En conséquence, on a plusieurs sous-schémas sur chacun

des sous-schémas, des vues matérialisées peuvent être sélectionnées (en utilisant les mêmes algorithmes décrits dans le chapitre 2).

- Les index: même en utilisant des index appropriés, le nombre de données non pertinentes explorées lors de l'exécution d'une requête reste très grand. Construire, par exemple, des index de jointure sur la totalité du schéma de l'entrepôt pose un problème de performance, car, à chaque exécution de requête, l'index doit être transféré en mémoire. Par contre, si l'entrepôt est partitionné, on peut construire des index pour chaque partition. Ces index, plus petits, sont plus faciles à maintenir et à charger.
- Les jointures: étant donné que les requêtes décisionnelles requièrent presque systématiquement des jointures entre les tables de dimensions et la table de faits, la fragmentation horizontale dérivée [CNP82] peut être utilisée pour accélérer ces jointures en éliminant les jointures inutiles.
- Le parallélisme: d'une manière générale le parallélisme est une bonne technique pour accélérer l'exécution des requêtes. Avec l'utilisation de la fragmentation horizontale, nous pouvons affecter les fragments horizontaux à différents processeurs et exécuter les requêtes en parallèle. Par exemple, MCI Telecommunications'IT (le Colorado Springs) a déployé un entrepôt de 2TB appelé *warehouseMCI* sur une machine parallèle de 104 noeuds (de type IBM RS/6000 SP). La base de données est en croissance de 100 GB à 200 GB par mois [D.96].

La difficulté majeure de la fragmentation horizontale dans les bases de données est sa contre-performance due aux opérations de mise à jour. Lorsqu'une opération de mise à jour intervient sur une relation (fragmentée) du schéma global, elle se traduit par plusieurs sous-opérations de mise à jour dans différents fragments. Pour exécuter cette opération, il faut d'abord identifier les fragments concernés par cette opération, puis l'exécuter sur les fragments identifiés. Après l'opération de mise à jour, certains n-uplets doivent changer de fragment (ce que nous appelons la migration de n-uplets). Cette migration de n-uplets peut être une opération coûteuse.

Pour l'illustrer, prenons l'exemple d'une relation EMPLOYE qui est partitionnée en fonction de l'attribut Salaire, en deux fragments: Emp_1 et Emp_2, caractérisés par:

$$\text{Emp\_1: } \sigma_{Salaire \leq 7000} \text{ (EMPLOYE)}$$
$$\text{Emp\_2: } \sigma_{Salaire > 7000} \text{ (EMPLOYE)}$$

Considérons un employé touchant un salaire de 6 500 FF (il appartiendrait donc au fragment Emp_1). Après une augmentation de 10% des salaires, le salaire de cet employé s'élève à 7 150 FF. Il doit donc changer de fragment.

Cependant, les problèmes de performance de mise à jour sont moins importants dans les entrepôts que dans les bases de données, pour les raisons suivantes:

1. Dans les entrepôts, la table des faits reflète l'aspect dynamique de l'entrepôt [Kim97], et la plupart des opérations de mises à jour se font au niveau de cette table. Les opérations de mise à jour dans les entrepôts sont de type *ajout* (append) de n-uplets. Si nous nous intéressons à une table des faits VENTES d'un magasin comme Carrefour, par exemple, cette table est mise à jour après

chaque achat.
L'ajout est donc l'opération la plus fréquente. La modification et la suppression peuvent intervenir également essentiellement au niveau des tables de dimensions [HMA99]: départ d'un client, changement d'adresse d'un client, arrivée d'un nouveau produit.

2. Même si les opérations de mise à jour sont nombreuses, elles sont effectuées lorsque l'entrepôt est off-line [OQ97].

Dans ce chapitre, nous proposons une méthodologie pour la fragmentation horizontale dans les entrepôts de données modélisés par un schéma en étoile. Cette méthodologie va nous permettre de dégager les problèmes liés à cette technique. Nous aborderons ensuite la partie algorithmique. L'objectif est d'élaborer un algorithme de fragmentation qui pourra fournir un bon schéma de fragmentation.

Ce chapitre est organisé comme suit:

- Dans la section 5.2, nous montrons par un exemple le rôle de la fragmentation horizontale dans la réduction du coût d'exécution des requêtes. Cet exemple, va nous permettre de dégager les principes du processus de fragmentation dans les entrepôts de données.
- Dans la section 5.3, nous décrivons notre méthodologie de fragmentation des tables de dimensions et de la table des faits.
- Dans la section 5.4, nous montrons le rôle des tables de dimensions dans la fragmentation de la table des faits. Nous développons un algorithme glouton afin de sélectionner les *meilleures tables de dimensions* pour la fragmentation de la table des faits.
- Dans la section 5.5, nous développons une stratégie d'exécution des requêtes sur un schéma en étoile fragmenté.
- Dans la section 5.6, un modèle de coût est décrit pour évaluer le coût d'un ensemble de requêtes sur un un schéma en étoile fragmenté.
- Dans la section 5.7, nous montrons l'utilité de la fragmentation dans les entrepôts à travers une évaluation.
- Finalement, dans la section 5.8, nous discutons quelques perspectives de la fragmentation.

5.2 Exemple d'illustration: service de ventes d'un magasin

Prenons l'exemple d'une compagnie qui s'intéresse à l'analyse de ses ventes. Pour rendre compréhensible les tables de dimensions et la table des faits qui modélisent cette compagnie, supposons que cette dernière compte des produits achetés par des clients à un moment donné. Les dirigeants de cette compagnie s'interrogent quotidiennement à la fin de chaque jour ouvrable pour récupérer les informations sur les ventes par produits et par clients. Le schéma présentant les opérations de cette compagnie contient quatre tables: trois tables de dimensions: CLIENT, PRODUIT

et TEMPS et la table des faits VENTES. Les attributs, leurs caractéristiques et la taille de chaque table sont donnés en Figure 2.5.

Considérons maintenant un exemple d'illustration afin de montrer l'intérêt de la fragmentation dans l'optimisation des requêtes. Supposons que les requêtes les plus fréquemment utilisées soient les six requêtes de jointure en étoile Q_1 à Q_6 de la Figure 5.1.

Supposons que seule la table de dimensions CLIENT soit décomposée en deux fragments horizontaux Client_1 et Client_2, résultant de deux sélections disjointes sur l'attribut Sexe :

$$Client\_1 = \sigma_{Sexe='M'}(CLIENT) \tag{5.1}$$

$$Client\_2 = \sigma_{Sexe='F'}(CLIENT) \tag{5.2}$$

La table des faits VENTES peut alors être décomposée en utilisant la fragmentation horizontale dérivée, en deux fragments horizontaux Ventes_1 et Ventes_2 tels que :

$$Ventes\_1 = VENTES \ltimes Client\_1 \tag{5.3}$$

$$Ventes\_2 = VENTES \ltimes Client\_2 \tag{5.4}$$

où $\ltimes$ représente l'opération de semi-jointure.
Le schéma initial $SED : (VENTES, CLIENT, PRODUIT, TEMPS)$ est la juxtaposition de deux sous-schémas en étoile SED_1 et SED_2 tels que:

SED_1 : (Ventes_1, Client_1, PRODUIT, TEMPS) (ventes des clients masculins)
SED_2 : (Ventes_2, Client_2, PRODUIT, TEMPS) (ventes des clients féminins).

A partir de la définition des six requêtes de la Figure 5.1 nous remarquons que:

- Les requêtes Q_1, Q_2 et Q_6 accèdent à la globalité du schéma SED, c'est à dire à $SED_1 \cup SED_2$.
- La requête Q_4 accède seulement au sous-schéma SED_2.
- Les requêtes Q_3 et Q_5 accèdent seulement au sous-schéma SED_1.

Ainsi, la moitié des requêtes (Q_3, Q_4 et Q_5) accèdent seulement à une partie du schéma en étoile S. La fragmentation pour ces requêtes pourra apporter de meilleures performances. Pour l'autre moitié des requêtes (Q_1, Q_2 et Q_6) qui accèdent à la totalité du schéma, nous pourrons les exécuter en parallèle.

5.3 Les principes de notre méthode de fragmentation

5.3.1 Les alternatives

La fragmentation horizontale dans les entrepôts représente un enjeu plus important que dans un contexte de base de données relationnelles ou objet. Cette importance est due au choix des tables (de dimensions ou des faits) à fragmenter. Les choix possibles sont les suivants:

ID	Formulation en SQL	Caractéristiques	Freq
1	SELECT sum(V.ventes_Dollars) FROM CLIENT C, VENTES V AND C.CID = V.CID GROUP BY CID	1 jointure	15
2	SELECT sum(V.Ventes_Dollar), sum(S.Coût_Unitaire) FROM PRODUIT, VENTES V, TEMPS T WHERE V.PID = P.PID AND V.TID = T.TID AND P.Type_paquet = "Boîte" AND T.Saison = "Eté" GROUP BY PID	2 jointures 2 prédicats	20
3	SELECT sum(V.Ventes_Dollar), sum(S.Coût_Unitaire) FROM VENTES V, PRODUIT P, CLIENT C, TEMPS T WHERE V.PID = P.PID AND V.CID = C.CID AND V.TID = T.IID AND P.Type_paquet = 'Papier" AND T.Saison = "Hiver" AND C.Sexe = 'M' GROUP BY TID	3 jointures 3 prédicats	20
4	SELECT sum(V.Ventes_Dollar) FROM VENTES V, CLIENT C, TEMPS T WHERE V.TID = T.TID AND V.CID = C.CID AND C.Sexe = 'F' AND T.Saison = "Eté" GROUP BY CID	2 jointures 2 prédicats	15
5	SELECT sum(S.Ventes_Dollar) FROM VENTES V, CLIENT C, PRODUIT P WHERE V.PID = P.PID AND V.CID = C.CID AND C.Sexe = 'M' AND P.Type_paquet = "Boîte" GROUP BY PID	2 jointures 2 prédicats	20
6	SELECT max(S.Ventes_Dollar) FROM VENTES V, CLIENT C WHERE V.CID = C.CID GROUP BY CID	1 jointure 0 Prédicat	10

FIG. 5.1: L'ensemble des requêtes utilisées dans l'évaluation de la fragmentation

1. Fragmenter seulement quelques tables de dimensions en utilisant la fragmentation horizontale primaire. Ce choix n'est pas souhaitable pour les requêtes décisionnelles, pour deux raisons: (i) les tailles des tables de dimensions sont généralement petites, (ii) les requêtes décisionnelles accèdent à la table des faits qui est très volumineuse. En conséquence, toute fragmentation ne prenant pas en considération la table des faits est à exclure.

2. Partitionner seulement la table des faits en utilisant la fragmentation primaire. La table des faits est composée des clés étrangères des tables de dimensions et des données brutes. Ces dernières représentent des mesures numériques, comme le montant des ventes, le nombre d'articles soldés, etc. Généralement, dans une requête décisionnelle, nous trouvons rarement des prédicats de sélection définis sur la table des faits. De plus, une requête décisionnelle typique commence par la sélection des critères selon lesquels s'effectuera l'analyse sur les tables de dimensions, puis s'orientera sur la valeur des indicateurs pour la sélection effectuée [NB99]. Ce choix est donc moins souhaitable pour le processus de la fragmentation.

3. Fragmenter totalement ou partiellement les tables de dimensions en utilisant la fragmentation primaire et utiliser leurs schémas de fragmentation pour partitionner la table des faits [1]. Ce choix est bien adapté aux entrepôts, car il prend en considération les exigences des requêtes décisionnelles, ainsi que les relations entre les tables de dimensions et la table des faits. Dans notre proposition, nous optons donc pour ce choix.

5.3.2 Une méthodologie de fragmentation

Dans cette section nous présentons une méthodologie pour fragmenter un schéma en étoile SED ayant d tables de dimensions $\{D_1, D_2, ..., D_d\}$ et une table des faits F. L'idée de base de cette méthodologie est de fragmenter *certaines ou toutes* les tables de dimensions en utilisant les algorithmes de fragmentation horizontale primaire (voir chapitre 2). Finalement, la table des faits sera fragmentée en utilisant la fragmentation horizontale dérivée.

Rappelons que tout algorithme de fragmentation nécessite des informations sur les applications qui utilisent la base de données [CNP82, OV91]. Deux types d'informations sont à distinguer: *qualitative* et *quantitative* Les informations qualitatives concernent les tables de dimensions et sont représentées par les prédicats de sélection simples utilisés dans les requêtes. Les informations quantitatives concernent la sélectivité de ces prédicats et les fréquences d'accès des requêtes.

L'algorithme que nous proposons a en entrée un ensemble de n requêtes de jointure $Q = \{Q_1, Q_2, ..., Q_n\}$ les plus fréquentes avec leur fréquence d'accès. Cet algorithme comprend six étapes principales [BKM00b, Bel00b]: (1) *l'énumération des prédicats de sélection*, (2) *l'attribution des prédicats aux tables*, (3) *l'identification des tables de dimensions à fragmenter*, (4) *la vérification de la règle de la*

[1] La table des faits est alors fragmentée en utilisant la fragmentation dérivée

complétude et de la minimalité, (5) *la fragmentation des tables de dimensions* et (6) *la fragmentation de la table des faits.*

Par la suite, nous décrivons chaque étape en détail.

1. **L'énumération de prédicats de sélection:** Cette étape consiste à énumérer tous les prédicats de sélection simples utilisés par les n requêtes de départ.
2. **L'attribution des prédicats aux tables**: Les prédicats de sélection trouvés dans l'étape précédente sont définis a priori sur l'ensemble des tables de dimensions. L'objectif de cette phase est d'attribuer à chaque table de dimensions D_i $(1 \leq i \leq d)$ un *ensemble de prédicats simples* EPS^{D_i} qui lui est propre.
3. **L'dentification des tables de dimensions à fragmenter:** Nous avons vu dans la section 5.3, que certaines tables de dimensions seront fragmentées. Cette étape consiste à les identifier. Pour ce faire, toute table de dimensions D_i ayant $(EPS^{D_i} = \phi)$, ne sera pas prise en compte dans le processus de fragmentation, car ce dernier est basé sur les prédicats de sélection. Par conséquent, toute table D_i ayant $EPS^{D_i} = \phi$, est accédée en totalité par les requêtes.
   Soit $D_{candidat}$ l'ensemble de toutes les tables de dimensions ayant leur ensemble de prédicats simples non vide $(EPS^{D_i} \neq \phi)$. Ces tables seront candidates pour le processus de fragmentation horizontale. Soit g la cardinalité de l'ensemble $D_{candidat}$ $(g \leq d)$.
4. **La vérification de la règle de la complétude et de la minimalité:** L'objectif de cette étape est de s'assurer que si une table de dimensions est fragmentée en au moins deux fragments, elle sera accédée différemment par au moins une application [OV91]. Pour réaliser cette tâche, nous appliquons l'algorithme COM_MIN [OV91] à chaque ensemble de prédicats simples d'une table de dimensions D_i. Cet algorithme fournit en sortie une forme complète et minimale de ces ensembles (voir Chapitre 2).
5. **La fragmentation des tables de dimensions:** Chaque table de dimensions de $D_{candidat}$ est partitionnée en utilisant un algorithme parmi ceux que nous avons présentés dans le Chapitre 2 (algorithmes basés sur la complétude et la minimalité des prédicats [CP84, OV91]).
   Comme résultat, chaque table de dimensions D_i de $D_{candidat}$ a m_i fragments $\{D_{i1}, D_{i2}, ..., D_{im_i}\}$, où chaque fragment D_{ij} est défini par:
   $$D_{ij} = \sigma_{cl^i_j}(D_i) \tag{5.5}$$
   avec cl^i_j $(1 \leq i \leq g, 1 \leq j \leq m_i)$ représentant une clause de prédicats simples.
6. **La fragmentation de la table des faits:** La table des faits est alors fragmentée en fonction de tous les schémas de fragmentation des tables de dimensions de l'ensemble $D_{candidat}$. Ainsi, chaque fragment horizontal F_i de la table des faits est défini de la manière suivant:
   $$F_i = F \ltimes D_{1j} \ltimes D_{2k} \ltimes ... \ltimes D_{gl}$$
   avec $\ltimes$: représente l'opération de semi-jointure.

La fragmentation de la table des faits entraîne la division du schéma en étoile initial SED en plusieurs sous-schémas en étoile $\{SED_1, SED_2, ..., SED_N\}$, où N représente le nombre de fragments horizontaux de la table des faits.

Exemple 18 *Reprenons le schéma en étoile de la Figure 2.5 avec 3 tables de dimensions: CLIENT, PRODUIT, et TEMPS et une table de faits VENTES. Sur ce schéma six requêtes décisionnelles sont exécutées, chaque requête ayant une fréquence d'accès donnée (voir Table 5.1).*

À partir de ces requêtes, nous énumérons (étape 1 de l'algorithme) tous les prédicats de sélection comme le montre la Table 5.1.

Tables	Prédicats simples
CLIENT	p_1: Sexe = 'M' p_2 : Sexe = 'F'
PRODUIT	p_3 : Type_paquet = "Boîte" p_4 : Type_paquet = "Papier"
TEMPS	p_5 : Saison = "Eté" p_6 : Saison = "Hiver"

TAB. 5.1: Les tables et leurs prédicats respectifs

Les ensembles de prédicats simples correspondants aux trois tables de dimensions sont (étape 2 de l'algorithme) :

$$EPS^{CLIENT} = \{p_1, p_2\},$$
$$EPS^{PRODUIT} = \{p_3, p_4\}$$
$$EPS^{TEMPS} = \{p_5, p_6\}.$$

Dans cet exemple, toutes les tables de dimensions sont candidates pour la fragmentation.

Pour chacun de ces ensembles, nous générons l'ensemble de prédicats simples minimal et complet ($EPS^{D_i}_{Min-Com}$) en appliquant l'algorithme de Özsu et al. [OV91] (étape 4 de l'algorithme). Nous obtenons ainsi les mêmes ensembles à savoir:

$$EPS^{CLIENT}_{Min-Com} = \{p_1, p_2\},$$

$$EPS^{PRODUIT}_{Min-Com} = \{p_3, p_4\}$$

$$EPS^{TEMPS}_{Min-Com} = \{p_5, p_6\}.$$

Maintenant, nous fragmentons chaque table de dimension en utilisant l'algorithme de Özsu [OV91] qui est basé sur la complétude et la minimalité des prédicats. Les fragments obtenus pour chaque table de dimension sont:

- deux fragments pour la table CLIENT*:*

  Client_1: ayant la clause cl_1 : Sexe = 'M',

  Client_2: ayant la clause cl_2 : Sexe = 'F' .

- deux fragments pour la table PRODUIT*:*

  Prod_1: ayant la clause cl'_1: Type_paquet = "Boîte",

  Prod_2: ayant la clause cl'_2: Type_paquet = "Papier".

- deux fragments pour la table TEMPS*:*

  Temps_1: ayant la clause cl''_1: Saison = "Hiver",

Temps_2: ayant la clause cl''_2: Saison = "Eté" .

La table des faits est alors fragmentée en 8 fragments horizontaux dont la description est illustrée dans la Table 5.2.

Fragments	Clause
Ventes_1	VENTES ⋉ Client_1 ⋉ Temps_1 ⋉ Prod_1
Ventes_2	VENTES ⋉ Client_1 ⋉ Temps_1 ⋉ Prod_2
Ventes_3	VENTES ⋉ Client_1 ⋉ Temps_2 ⋉ Prod_1
Ventes_4	VENTES ⋉ Client_1 ⋉ Temps_2 ⋉ Prod_2
Ventes_5	VENTES ⋉ Client_2 ⋉ Temps_1 ⋉ Prod_1
Ventes_6	VENTES ⋉ Client_2 ⋉ Temps_1 ⋉ Prod_2
Ventes_7	VENTES ⋉ Client_2 ⋉ Temps_2 ⋉ Prod_1
Ventes_8	VENTES ⋉ Client_2 ⋉ Temps_2 ⋉ Prod_2

TAB. 5.2: Les fragments de la table des faits

5.3.3 Complexité de la fragmentation

Soit SED un schéma en étoile de d tables de dimensions et une table des faits. Soit g ($g \leq d$) le nombre de tables de dimensions horizontalement fragmentées. Le nombre de fragments horizontaux (dénoté par N) de la table des faits est donné par l'équation suivante:

$$\mathbf{N} = \prod_{\mathbf{i=1}}^{\mathbf{g}} \mathbf{m_i} \tag{5.6}$$

où m_i représente le nombre de fragments de la table de dimensions D_i.

D'après l'équation 5.6, nous constatons que notre approche de fragmentation peut générer un très grand nombre de fragments pour la table des faits et en conséquence beaucoup de sous-schémas en étoile.

Exemple 19 *Supposons par exemple que les trois tables de dimensions du schéma de l'entrepôt de la Figure 2.5 soient partitionnées comme suit:*

- CLIENT *en 50 fragments en utilisant l'attribut "Etat"* [2],
- TEMPS *en 12 fragments en utilisant l'attribut "Mois", et*
- PRODUIT *en deux fragments en utilisant l'attribut "Type_Paquet".*

La table des faits est donc fragmentée en 1200 (= 50 × 12 × 2) fragments et le schéma en étoile en 1200 sous-schémas. Il devient difficile pour l'administrateur de l'entrepôt de gérer tous ces fragments.

A travers l'exemple précédent, il apparaît indispensable de *réduire le nombre de fragments de la table des faits* pour une meilleure gestion. Nous abordons ce problème dans la section suivante.

[2] cas de 50 états aux U.S.A.

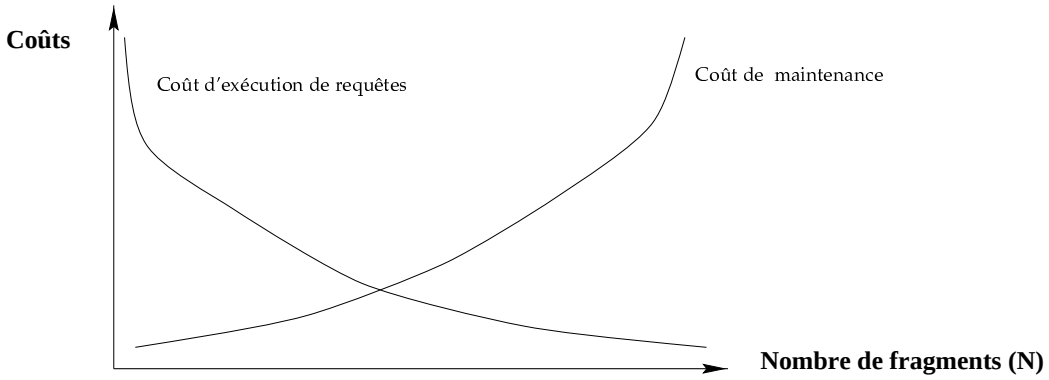


FIG. 5.2: Evolution des coûts en fonction du nombre de fragments

5.4 Sélection des tables de dimensions

5.4.1 Principes

Le nombre N de fragments de la table des faits augmente avec le nombre de tables de dimensions fragmentées (g) et le nombre de fragments de chaque table fragmentée.

Il est donc nécessaire d'introduire des méthodes de sélection des tables de dimensions à fragmenter pour :

- Éviter l'explosion du nombre de fragments de la table des faits.
- Garantir la meilleure performance d'exécution des requêtes.

Pour atteindre le premier objectif, nous donnons à l'administrateur de l'entrepôt la possibilité de choisir le nombre maximum L de fragments de la table des faits. Ce nombre est fixé en fonction du *coût de maintenance* de l'entrepôt fragmenté. Pour satisfaire le deuxième objectif, nous devons avoir la possibilité d'augmenter le nombre de fragments tant que la performance globale peut être améliorée. Le problème est donc de trouver un compromis entre le coût de maintenance et le coût d'exécution des requêtes, comme le montre la Figure 5.2.

Le problème de la sélection des tables de dimensions peut être formulé de la manière suivante:

- Soit $D_{candidat} = \{D_1, D_2, ..., D_g\}$ l'ensemble de tables de dimensions candidates au processus de la fragmentation.
- Soit L le nombre de fragments de la table des faits que l'administrateur estime pouvoir maintenir.
- Soit un ensemble de requêtes $Q = \{Q_1, Q_2, ..., Q_n\}$ et leurs fréquences d'accès $\{f_1, f_2, ..., f_n\}$.

L'objectif du problème est de sélectionner un ensemble de tables de dimensions réduisant au mieux les coûts d'évaluation de l'ensemble des requêtes et de maintenance des fragments.

5.4.2 Un algorithme "glouton"

Pour résoudre le problème de sélection des tables de dimensions à fragmenter, un algorithme "glouton" est développé. Rappelons que nous considérons seulement les tables de dimensions dont leurs ensembles de prédicats simples sont non vides.

Étant donné le nombre L, cet algorithme commence par sélectionner une table de dimensions de manière aléatoire. Cette table est alors fragmentée ainsi que la table des faits par la fragmentation dérivée. Le nombre de fragments de la table des faits peut être calculé (voir l'équation 5.6). Le coût d'évaluation de l'ensemble des n requêtes est alors calculé [3].

Si le nombre de fragments N est inférieur à L et qu'il y a une amélioration dans le coût, l'algorithme sélectionne une autre table de dimensions et réitère le même processus tant que les deux conditions (amélioration de coût et $N \leq L$) sont satisfaites (cf. Figure 5.3). Sinon l'algorithme dé-sélectionne cette table et considère une autre table. À la fin de l'algorithme, nous obtenons un entrepôt fragmenté offrant un bon compromis entre le coût de traitement des requêtes et le coût de maintenance.

La performance du résultat dépend bien sûr de la table de dimensions qui a été sélectionnée au départ. Pour éviter la sélection aléatoire, nous proposons trois méthodes: (1) *la table de dimensions la plus fréquemment utilisée*, (2) *la table de dimensions ayant le plus petit nombre de fragments*, et (3) *la table de dimensions ayant des attributs de prédicats de cardinalité faible*.

- La table de dimensions la plus fréquemment utilisée Pour déterminer cette table, dénotée par D_{Fmax}, nous définissons la métrique suivante: *la fréquence d'une table de dimensions* D_i de $D_{candidat}$ notée par F_{D_i} est obtenue par la sommation des fréquence d'accès des requêtes accédant à la table D_i.

- La table de dimensions ayant le plus petit nombre de fragments Avec ce choix, nous pouvons éviter l'explosion du nombre de fragments de la table des faits.

- La table de dimensions ayant des attributs de prédicats de cardinalité faible Cette approche consiste à sélectionner les dimensions ayant des prédicats simples définis sur des attributs ayant un domaine de faible cardinalité. Par exemple l'attribut "Sexe" de dimensions CLIENT a un domaine ayant deux valeurs: mâle et femelle. Cette solution garantit un petit nombre de fragments et permet l'utilisation d'un index binaire (bitmap index [OQ97]) sur les fragments de la table de faits. Ces index pourront améliorer la performance de certaines opérations de sélection et minimiser le coût de stockage des index.

[3] Nous supposons que nous disposons d'un modèle de coût calculant le nombre d'entrées-sorties. Un tel modèle est décrit dans la section suivante

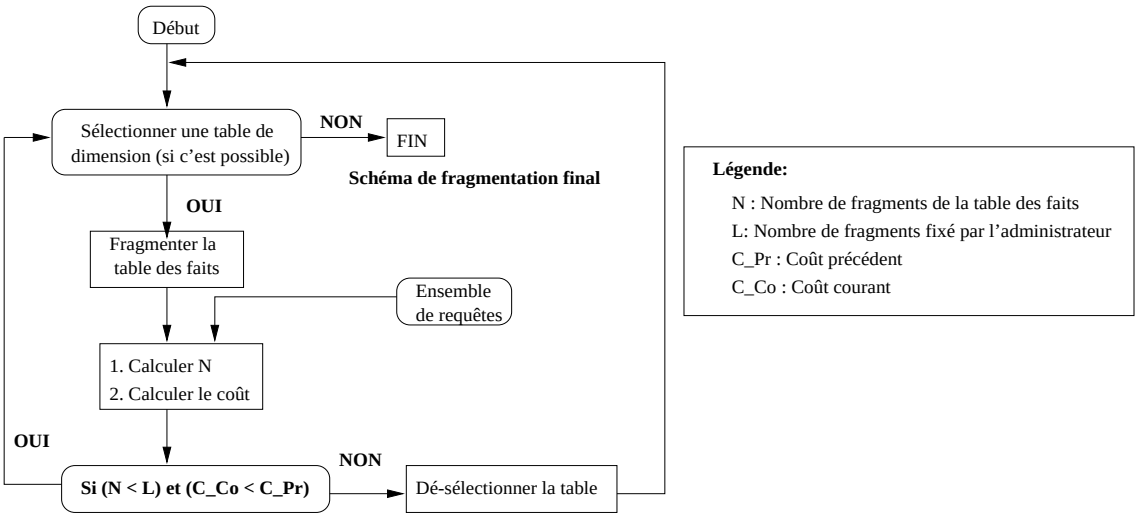


FIG. 5.3: L'algorithme glouton

5.5 Les propriétés de la fragmentation

5.5.1 Graphe de jointure simple

Lorsqu'une relation S est partitionnée en fonction d'un schéma de fragmentation d'une relation R, deux cas de jointure peuvent intervenir: *la jointure simple* et *la jointure partitionnée* (voir Chapitre 2).

Dans le contexte des entrepôts, lorsqu'une table des faits est fragmentée en fonction des tables de dimensions, nous *ne pouvons jamais avoir une jointure partitionnée* (c'est à dire un cas où un fragment de la table des faits pourrait être joint avec plus d'un fragment d'une table de dimensions), mais seulement une jointure simple comme le montre la Figure 5.4. Le théorème suivant précise cette propriété.

Théorème 1 *Soient F une table des faits et D_i une table de dimensions d'un schéma en étoile. Si la table de dimensions D_i est fragmentée horizontalement en un ensemble disjoint de fragments horizontaux $\{D_{i1}, D_{i2}, , D_{im_i}\}$, où chaque fragment est défini par une clause de prédicats : $D_{ij} = \sigma cl_j\ (D_i)$, et si la table des faits est fragmentée en utilisant le schéma de fragmentation de D_i, alors, la jointure distribuée entre D_i et F est toujours représentée par un graphe de jointure simple [Bel00b]*

Preuve 1 *Nous allons prouver ce théorème par contradiction. Soit F_p un fragment horizontal de la table des faits F défini par : $F_p = F \ltimes D_{ij}$. Supposons que F_p puisse être joint avec deux fragments horizontaux D_{ij} et D_{il} $(l \neq j)$ de la table de dimensions D_i. En conséquence:*

$$F_p \ltimes D_{ij} \neq \emptyset \tag{5.7}$$

$$F_p \ltimes D_{il} \neq \emptyset \tag{5.8}$$

Notons que les fragments de D_i sont disjoints, et le fragment F_p est généré en utilisant la semi-jointure entre la table des faits F et la table de dimensions D_i dont

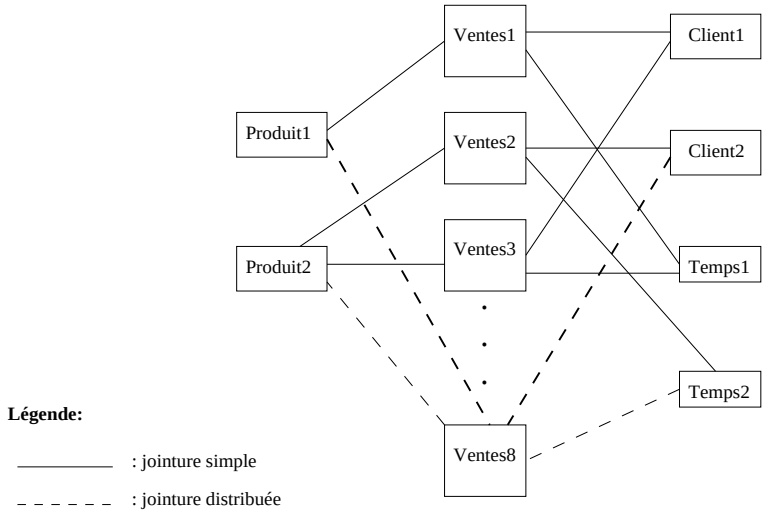


FIG. 5.4: La jointure simple dans un schéma en étoile

les attributs de jointure sont la clé étrangère de F et la clé primaire de D_i. En conséquence, seule une condition de semi-jointure parmi les deux définies en (4.6) et (4.7) doit être satisfaite, d'où $D_{ij} = D_{il}$. Puisque F_p peut être tout fragment de F, et qu'il peut toujours être joint avec exactement un fragment de D_i, il en résulte que le graphe de jointure est toujours simple [4].

Il résulte de cette propriété que la fragmentation horizontale dérivée de la table des faits apporte beaucoup dans la réduction du temps d'éxécution des opérations de jointure entre la table des faits et les tables de dimensions.

5.5.2 Complétude, reconstruction, et disjonction

Tout algorithme de fragmentation dans les bases de données ou les entrepôts doit respecter trois propriétés [CP84, OV91]: la *complétude*, la *reconstruction* et la *disjonction*.

- La complétude assure que tous les n-uplets d'une relation sont associés à au moins un fragment. Dans notre cas, il faut assurer la complétude pour les tables de dimensions et la table des faits. Pour les tables de dimensions, la complétude est garantie par l'algorithme que nous utilisons (voir Chapitre 2). La complétude pour la table des faits est garantie à travers l'intégrité référentielle entre les tables de dimensions et la table des faits du modèle d'entrepôt que nous utilisons.
- La reconstruction assure qu'une relation peut être reconstruite à partir de ses fragments [OV91]. Dans notre cas, la reconstruction de la table des faits et des tables de dimensions est obtenue par l'opération d'union, c'est à dire $F = \cup_{i=1}^{N} F_i$ et $D_i = \cup_{j=1}^{m_i} D_{ij}$.

[4] Le théorème est vrai lorsque nous avons une jointure distribuée entre la table des faits et plusieurs tables de dimensions.

- La disjonction assure que les fragments d'une relation sont disjoints deux à deux. Dans notre cas, cette propriété pour les tables de dimensions résulte de l'algorithme de fragmentation utilisé (voir chapitre 2). La disjonction des fragments de la table des faits est garantie par le théorème 1.

5.5.3 Identification des fragments participants à une requête

Dans un schéma en étoile fragmenté nous devons assurer l'accès transparent à l'utilisateur de l'entrepôt. Dans ce but, la tâche de l'optimiseur des requêtes (interrogation ou mise à jour) est de transformer les requêtes globales (c'est à dire les requêtes définies sur le schéma en étoile non fragmenté) en requêtes locales (définies sur le schéma en étoile fragmenté). Avant d'exécuter une requête sur un schéma fragmenté, l'optimiseur doit *identifier* le(s) sous-schéma(s) en étoile satisfaisant cette requête. Ces derniers sont appelés les schémas valides. Une fois ces sous-schémas sélectionnés, l'unité d'exécution de cette requête devient un sous-schéma au lieu du schéma global (Figure 5.5).

Définition 32 Un attribut de prédicat pertinent (APP) *est un attribut qui participe à un prédicat définissant la fragmentation. Tout attribut qui ne participe pas à la définition d'un prédicat de fragmentation est appelé attribut de prédicat non pertinent.*

Supposons que nous ayons un schéma en étoile SED fragmenté en N sous-schémas en étoile $\{SED_1, SED_2, ..., SED_N\}$. Les informations concernant les conditions de fragmentation de chaque table de dimensions et de la table des faits peuvent être représentées par une table que nous appelons *table de spécification de la fragmentation du schéma* (Table 5.3). Cette table a trois colonnes: la première contient la liste des noms des tables de dimensions fragmentées et de la table des faits, la deuxième fournit les fragments de chaque table de dimensions et la troisième spécifie les conditions de la fragmentation de chaque table (table de dimensions ou table des faits).

Exemple 20 *Nous reprenons l'exemple de la section 2 dans lequel la table de dimensions CLIENT est fragmentée en Client_1 et Client_2, et la table des faits VENTES en Vente_1 et Vente_2. La table de spécification de la fragmentation pour ce cas est illustrée par la Table 5.3.*

A partir de cette table, nous dégageons la propriété suivante:

Propriété 1 *Tout attribut appartenant à la table de spécification est un APP. Dans notre exemple, l'attribut "Sexe" est le seul APP.*

Dimension	Fragments	Condition de Fragmentation
CLIENT	Client_1	Sexe = 'M'
	Client_2	Sexe = 'F'
VENTES	ventes_1	VENTES $\ltimes$ Client_1
	Ventes_2	VENTES $\ltimes$ Client_2

TAB. 5.3: La table de spécification de la fragmentation

Par la suite, nous appellerons $EAPP(SED)$ l'ensemble des attributs de prédicats pertinents d'un schéma fragmenté SED.

Soit une requête décisionnelle Q avec p prédicats de sélection, exécutée sur un schéma en étoile fragmenté $SED = \{SED_1, SED_2, ..., SED_N\}$. Le but de cette section est d'identifier les sous-schémas de SED qui participeront à l'exécution de la requête Q (Figure 5.5). Pour réaliser cette identification, nous suggérons de procéder comme suit:

- Pour chaque prédicat de sélection PS_i $(1 \leq i \leq p)$, nous définissons une fonction **attr(PS_i)** qui donne le nom de l'attribut utilisé par PS_i. Soit $EAPR(Q)$ l'union de tous les attributs des prédicats de la requête Q.
- En fonction de $EAPR(Q)$ et $EAPP(SED)$, nous distinguons quatre scénarios:
  1. $EAPR(Q) = \emptyset$, (la requête Q ne contient pas de prédicat de sélection, c'est le cas de la requête 6 de la Table 5.1). Dans ce cas, deux approches existent pour exécuter la requête Q:
     (a) Faire d'abord l'union de tous les schémas en étoile et ensuite exécuter la requête, comme dans le cas non partitionné.
     (b) Exécuter la requête sur chaque sous-schéma SED_i $(1 \leq i \leq N)$ et ensuite faire l'union des résultats.
  2. $(EAPR(Q) \neq \emptyset)$ et $(EAPR(Q) \cap EAPP(SED) = \emptyset)$ (la requête Q a quelques prédicats de sélection sur des tables de dimensions non fragmentées, ou bien les attributs de prédicats de la requête Q ne coïncident pas avec les prédicats pertinents). Pour exécuter la requête Q nous utilisons l'une des approches parmi celles présentées en (1.a) et (1.b).
  3. $(EAPR(Q) \cap EAPP(SED) \neq \phi)$ (quelques attributs de la requête dans les prédicats de sélection coïncident avec certains APP). Dans ce cas, nous pouvons facilement déterminer (en utilisant les prédicats de sélection de la requête Q) les sous-schémas en étoile concernés par Q. Une fois les sous-schémas identifiés, la requête Q est exécutée sur chacun. Pour obtenir le résultat final de la requête, nous faisons l'union des résultats locaux (obtenus dans chaque sous-schéma).
  4. $(EAPR(Q) \subseteq EAPP(SED))$ (*tous* les attributs des prédicats de sélection de la requête Q sont des APP). Notons que l'ensemble des prédicats simples de la fragmentation est minimal et complet d'où la garantie que l'entrepôt est fragmenté en au moins deux fragments. Le fait qu'un APP appartienne à $EAPP(SED)$, signifie que la requête Q n'accède pas à la globalité de la table d'où cet attribut est issu. Dans ce cas, la requête est exécutée en utilisant la même approche que le troisième scénario.

Pour conclure, nous faisons les observations suivantes:

- La fragmentation détériore la performance d'exécution des requêtes satisfaisant les cas 1 et 2,
- Elle est favorable pour le cas 3,
- Elle est la plus avantageuse pour le cas 4.

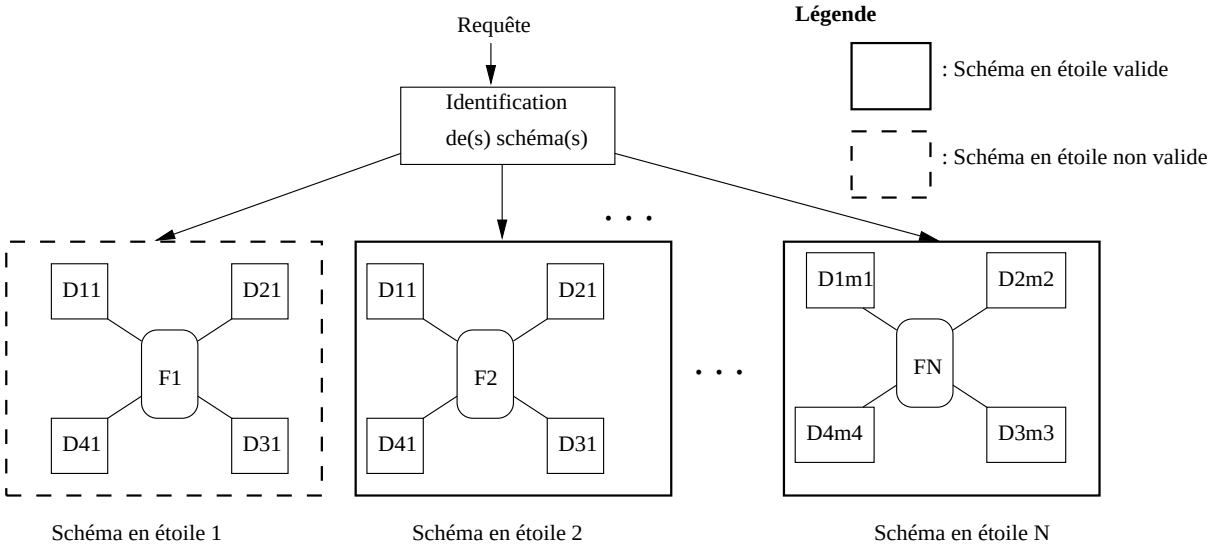


FIG. 5.5: Identification de(s) sous-schéma(s) en étoile

5.6 Modèle de coût pour l'exécution des requêtes

Pour valider les observations décrites précédemment, nous développons un modèle de coût (calculant le nombre d'entrées-sorties lors de l'évaluation d'une requête) qui peut être considéré comme une mesure de performance de la fragmentation dans les entrepôts.

Soit $SED : (D_1, D_2, ..., D_d, F)$ un schéma en étoile d'un entrepôt ayant d tables de dimensions et une table des faits F. En suivant la stratégie de fragmentation pour laquelle nous avons optée, les tables de dimensions sont fragmentées en utilisant les algorithmes de fragmentation primaire, et la table des faits est partitionnée en utilisant la fragmentation horizontale dérivée.

Supposons donc que certaines tables de dimensions soient fragmentées. Le résultat de la fragmentation de la table des faits en fonction des fragments des tables de dimensions entraîne la division du schéma global de l'entrepôt SED en N sous-schémas en étoile (cf. section 5.3). Chaque sous-schéma possède les mêmes caractéristiques que le schéma global, la seule différence étant que chaque sous-schéma contient les n-uplets satisfaisant une clause donnée.

Sur chaque sous-schéma, nous pouvons définir des vues matérialisées, des index de jointure en utilisant les mêmes algorithmes de sélection présentés dans le Chapitre 2.

Pour quantifier le bénéfice apporté par chaque schéma de fragmentation, nous proposons d'adapter le modèle de coût présenté dans le Chapitre 3 qui considère les index de jointure et les vues matérialisées. Quelques retouches sont nécessaires:

- Pour exécuter une requête Q sur un schéma d'un entrepôt de données fragmenté en m sous-schémas de l'entrepôt $\{SED_1, SED_2, ..., SED_m\}$, nous devons identifier le(s) sous-schéma(s) valide(s) avant de l'exécuter. Pour réaliser cette identification, une variable booléenne dénotée par $valide(Q, SED_i)$ est définie:

$$valide(Q, SED_i) = \begin{cases} 1 & \text{si le sous-schéma } SED_i \text{ est valide pour la requête } Q \\ 0 & \text{sinon} \end{cases}$$

- La conséquence du processus d'identification est le changement de l'unité de chargement des données de disque vers la mémoire centrale. C'est-à-dire dans un entrepôt non fragmenté, l'unité de chargement est la table, par contre dans le cas fragmenté, l'unité de chargement est le fragment.
- Il faut charger uniquement les index correspondant aux sous-schémas valides.

Le nombre d'ES va dépendre de la façon dont peuvent se dérouler les échanges avec la mémoire centrale. Nous considérons deux hypothèses.

1. *Hypothèse de Mémoire Large (HML)*: Toutes les tables de dimensions peuvent être chargées intégralement dans la mémoire centrale, car leur taille est petite [Cor97a]. La table des faits et les index de jointure sont chargés par parties et à la demande en mémoire centrale. Les résultats intermédiaires restent en mémoire centrale. Cette hypothèse devient réaliste compte tenu de l'évolution des matériels et de leur coût. Sous cette hypothèse, le coût sera le nombre d'ES réalisé en chargeant en mémoire la table des faits et les index. Dans ce cas, les caractéristiques des tables dans les formules du Chapitre 3 sont remplacées par celles des fragments valides.
2. *Hypothèse de Mémoire Moyenne (HMM)*: Elle diffère de la précédente dans la mesure où les résultats intermédiaires ne peuvent pas être stockés intégralement en mémoire centrale. Il faudra donc les transférer sur disque et les recharger par parties en mémoire centrale pour achever l'exécution de la requête. Donc pour exécuter une jointure multiple $M : D_1 \bowtie F \bowtie D_2 \bowtie ... \bowtie D_j$ sur un entrepôt non fragmenté, l'optimiseur exécute la première jointure $(D_1 \bowtie F)$, et vérifie si son résultat peut être stocké dans la mémoire centrale. Si c'est le cas, il joint ce résultat avec D_2 et il réitère le processus jusqu'à la dernière table de dimensions D_j. Sinon, il doit transférer ce résultat sur disque puis le recharger en mémoire pour continuer l'opération M. Dans le cas où l'entrepôt est fragmenté, la jointure multiple sera une jointure distribuée, et donc les résultats intermédiaires seront moins volumineux par rapport à ceux issus d'une jointure normale (cas non fragmenté).

5.7 Evaluation d'un entrepôt fragmenté

Afin de démontrer l'importance de la fragmentation dans les entrepôts, nous effectuons quelques expérimentations, en utilisant le schéma en étoile de la Figure 2.5. Les caractéristiques principales de ces expérimentations sont présentées dans la Table 3.4.

Pour caractériser l'amélioration des performances d'exécution des requêtes décisionnelles en utilisant la fragmentation, nous introduisons le facteur ES normalisé qui est défini comme suit:

$$\textbf{ES normalisé} = \frac{\textbf{NOMBRE d'ESs dans un SCHEMA EN ETOILE FRAGMENTE}}{\textbf{NOMBRE d'ESs dans un SCHEMA en ETOILE non FRAGMENTE}}$$

Si la valeur de ES normalisé est inférieure à 1.0, la présence de la fragmentation dans les entrepôts est bénéfique, sinon elle ne l'est pas.

Supposons que les tables de dimensions soient fragmentées comme dans l'exemple 18. Pour situer l'impact du nombre de tables de dimensions fragmentées, nous considérons quatre cas:

- **Cas 0 :** Aucune table n'est fragmentée.
- **Cas 1 :** Une seule table de dimensions est fragmentée, par exemple CLIENT [5]. La table de spécification correspondante est donnée en Table 5.3.
- **Cas 2 :** Deux tables de dimensions sont fragmentées, par exemple CLIENT et PRODUIT. La table de spécification correspondante est donnée en Table 5.4.
- **Cas 3 :** Toutes les tables de dimensions sont fragmentées. La table de spécification est présentée dans la Table 5.5.

Dimensions	Fragments	Condition de Fragmentation
ClIENT	Client_1	Sexe = 'M'
	Client_2	Sexe = 'F'
PRODUIT	Prod_1	Type_paquet = "Boîte"
	Prod_2	Type_paquet = "Papier"
VENTES	Ventes_1	VENTES $\ltimes$ Client_1 $\ltimes$ Prod_1
	Ventes_2	VENTES $\ltimes$ Client_1 $\ltimes$ Prod_2
	Ventes_3	VENTES $\ltimes$ Client_2 $\ltimes$ Prod_1
	Ventes_4	VENTES $\ltimes$ Client_2 $\ltimes$ Prod_2

TAB. 5.4: La spécification de la fragmentation (cas 2)

Dimension	Fragments	Condition de Fragmentation
CLIENT	Client_1	Sexe = 'M'
	Client_2	Sexe = 'F'
TEMPS	Temps_1	Saison = "Hiver"
	Temps_2	Saison = "Eté"
PRODUIT	Prod_1	Type_paquet = "Boîte"
	Prod_2	Type_paquet = "Papier"
VENTES	Ventes_1	VENTES $\ltimes$ Client_1 $\ltimes$ Temps_1 $\ltimes$ Prod_1
	Ventes_2	VENTES $\ltimes$ Client_1 $\ltimes$ Temps_1 $\ltimes$ Prod_2
	...	...
	Ventes_8	VENTES $\ltimes$ Client_2 $\ltimes$ Temps_2 $\ltimes$ Prod_2

TAB. 5.5: La spécification de la fragmentation (cas 3)

[5]La table CLIENT est la table la plus fréquemment utilisée et ayant des attributs de prédicats de cardinalité faible (Sexe)

Pour toutes ces expérimentations nous avons supposé pour simplifier que la taille des fragments des tables de dimensions et de la table des faits était la même.

Dans les Tables 5.6 et 5.7 nous donnons les résultats obtenus pour chacun des cas considérés et pour les six requêtes décisionnelles. Ces résultats appellent les commentaires suivants:

- En comparaison avec le cas non fragmenté (cas 0), la fragmentation donne souvent de meilleurs résultats. Dans tous les cas, la requête ayant bénéficié de la fragmentation est $Q3$. En effet elle possède des prédicats de sélection utilisés dans l'algorithme de fragmentation (voir le point 4 de la section 5.5.3).
- La fragmentation peut dégrader la performance d'exécution de certaines requêtes décisionnelles. Par exemple, tous les cas de fragmentation ne sont pas bénéfiques pour les requêtes $Q1$ et $Q6$ pour la raisons suivante: (1) $Q1$ et $Q6$ n'ont aucun prédicat de sélection, ainsi afin de l'évaluer, il faut accéder à tous les fragments de la table des faits et des tables de dimensions. Cela confirme nos propos de la section 5.5.3.
- Le nombre de tables de dimensions fragmentées a un rôle significatif dans la performance. Comme nous le voyons dans les Tables 5.6 et 5.7, la fragmentation donne de meilleurs résultats dans le cas 3 (pour $Q2$, $Q3$, $Q4$, $Q5$), pour lequel toutes les tables de dimensions de l'entrepôt sont fragmentées.
- Le total ES normalisé pour chaque cas (ESN_i) est toujours inférieur à 1.0. Cela signifie que la fragmentation donne de bons résultats si nous nous intéressons à la performance globale de toutes les requêtes décisionnelles. Par exemple la fragmentation n'est pas bénéfique pour les requêtes $Q1$, $Q2$ et $Q6$ (cas 1), mais elle est bénéfique pour les requêtes $Q3$, $Q4$ et $Q5$. La contre performance des requêtes non bénéficiaires de la fragmentation est compensée par la bonne performance des requêtes bénéficiaires.
- La fragmentation est toujours meilleure par rapport au cas non fragmenté (Table 5.7). Cela provient du fait que les résultats intermédiaires de la jointure distribuée peuvent être stockés dans la mémoire centrale.

Q	**ES**	**ES_1**	**ESN_1**	**ES_2**	**ESN_2**	**ES_3**	**ESN_3**
Cas	Cas 0	Cas 1	Cas 1	Cas 2	Cas 2	Cas 3	Cas 3
Q1	175826	175856	1.0001	175948	1.0006	176124	1.002
Q2	5856575	5856683	1.0	2197003	**0.38**	1464232	**0.25**
Q3	**5856585**	**2197003**	**0.37**	**1464193**	**0.25**	**732127**	**0.125**
Q4	4392432	2197003	0.5	2197003	0.5	1464232	**0.25**
Q5	5856575	2197003	**0.38**	1464193	**0.25**	1464232	**0.25**
Q6	5856545	5856582	1.0001	5856643	1.0002	5856761	1.001
Total	27994538	18480130	0.66	13354983	0.47	11157708	0.39

TAB. 5.6: Le coût d'exécution des requêtes sous l'hypothèse HML

Nous avons aussi étudié l'effet de la variation de la taille de la table des faits (dans la réalité cette table évolue sans cesse). Nous supposons que nous avons un entrepôt

Q	**ES**	**ES_1**	**ESN_1**	**ES_2**	**ESN_2**	**ES_3**	**ESN_3**
Cas	Cas 0	Cas 1	Cas 1	Cas 2	Cas 2	Cas 3	Cas 3
Q1	175826	175856	1.0001	175948	1.0006	176124	1.002
Q2	10004935	5856683	**0.58**	2197003	**0.22**	1464232	**0.15**
Q3	**12079125**	**2197003**	**0.18**	**1464193**	**0.13**	732127	**0.06**
Q4	7503702	2197003	0.3	2197003	0.3	1464232	**0.2**
Q5	10004935	2197003	**0.21**	1464193	**0.15**	1464232	**0.24**
Q6	5856545	5856582	1.0001	5856643	1.0002	5856761	1.001
Total	45625068	18840130	0.41	13354983	0.29	11157708	0.24

TAB. 5.7: Le coût d'exécution des requêtes sous hypothèse HMM

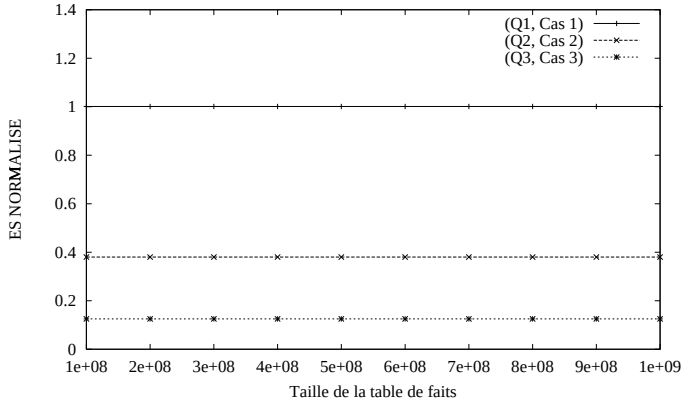


FIG. 5.6: L'effet de variation de la taille de la table des faits

fragmenté en 2, 4 et 8 fragments et nous faisons varier la taille de la table des faits de 10^8 à 10^9. Nous remarquons que le facteur ES normalisé reste constant comme le montre la Figure 5.6. Cela nous montre que ES normalisé est indépendant de la cardinalité de la table des faits.

5.8 Problèmes ouverts

Plusieurs questions sont toujours ouvertes sur le rôle de la fragmentation (en général) dans l'amélioration de la performance des entrepôts de données. Parmi ces questions, nous pouvons citer: la fragmentation et les data marts, la fragmentation et l'adaptation des vues matérialisées.

5.8.1 Principes des data marts

Dans cette section, nous abordons l'un des points évoqués précédemment, c'est-à-dire, le rôle de la fragmentation horizontale dans la conception des data marts [BKM00b].

Pourquoi les data marts? Mettre en oeuvre un entrepôt de données pour une entreprise intégrée est une tâche difficile qui nécessite la collecte des données

de différents départements, leur nettoyage, et leur intégration. Certaines entreprises ont donc préféré développer des *data marts*. Rappelons qu'un data mart est une base de données orientée sujet à la disposition des utilisateurs dans un contexte décisionnel décentralisé [Fra97]. Les caractéristiques principales d'un data mart sont les suivantes:

- Il se concentre seulement sur un sujet et un groupe d'utilisateurs. Il présente un sous-ensemble de l'entrepôt aux utilisateurs ayant un besoin fonctionnel bien spécifique.
- une entreprise peut avoir un seul entrepôt, mais plusieurs data marts.
- les data marts contiennent moins d'informations que les entrepôts; ils sont donc plus facilement gérables et interrogeables.
- les data marts nécessitent une durée réduite de mise en oeuvre, inférieure à celle des entrepôts.

Deux approches existent pour concevoir un data mart [Fir97]: (1) une approche ascendante et (2) une approche descendante.

Dans l'approche ascendante, le data mart est conçu en intégrant des données à partir des sources. Dans cette approche, les différents data marts dont a besoin une entreprise sont mis en oeuvre indépendamment (Figure 5.7).

Dans l'approche descendante, le data mart est conçu en extrayant les données de l'entrepôt, en fonction des besoins (voir Figure 5.8). Dans cette approche, le rôle des data marts est de présenter des sous-ensembles de l'entrepôt aux utilisateurs ayant un besoin fonctionnel bien déterminé. La relation entre les data marts et l'entrepôt est unidirectionnelle [Fir97]. Les data marts sont dérivés de l'entrepôt et leur contenu est strictement limité par celui de l'entrepôt. Tout changement au niveau de l'entrepôt, doit être répercuté sur les data marts.

Pour concevoir des data marts en respectant cette approche, nous pouvons penser à la duplication de l'entrepôt dans chaque data mart. L'inconvénient majeur de cette solution est l'introduction d'une grande redondance. Pour pallier ce problème, la fragmentation pourrait être un bon moyen. Elle peut être utilisée comme un outil de division de l'entrepôt en fonction des besoins de chaque département ou du groupe d'utilisateurs. En conséquence, un data mart peut être vu comme un mini entrepôt (ou fragment d'un entrepôt). Dans un tel scénario, les tables de l'entrepôt (lorsque l'entrepôt est modélisé par un schéma relationnel) seront partitionnées et chaque data mart aura donc ses propres données.

5.8.1.1 Analogies entre la conception de bases de données réparties et la conception de data marts

Les entrepôts de données sont très souvent répartis, soit en raison de leur taille, soit en raison de la variété des sources qu'ils approvisionnent. Comme l'ont mentionné Garcia Molina et al. [MLJ98], même lorsque les données de l'entrepôt sont centralisées sur un même site, compte tenu de leur taille, elles sont souvent stockées sur plusieurs disques et peuvent de ce fait être considérées comme réparties.

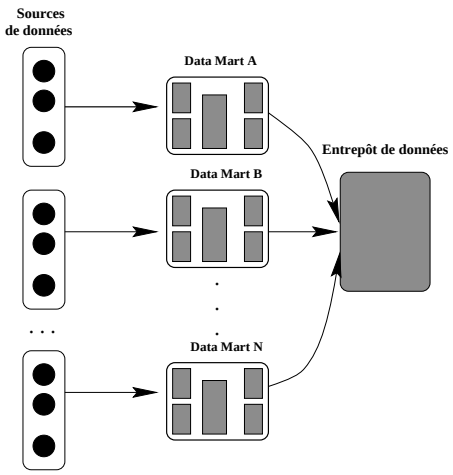


FIG. 5.7: L'approche ascendante

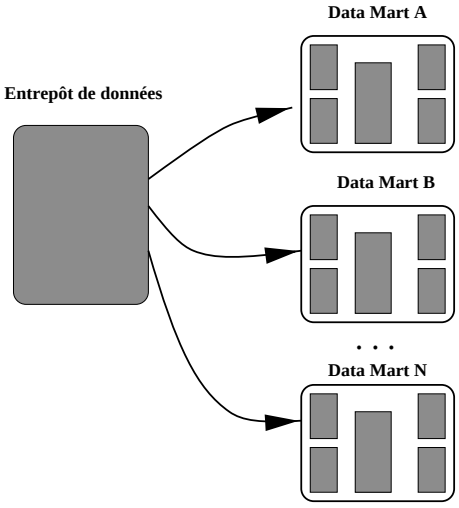


FIG. 5.8: L'approche descendante

Pour concevoir une base de données répartie, Ceri et al. [CNP82] ont proposé deux stratégies ascendante et descendante.

- *La stratégie ascendante* est une approche selon laquelle la tâche de conception consiste à intégrer dans un schéma unique les descriptions d'un certain nombre de bases de données existantes. Très souvent les bases de données à intégrer sont hétérogènes. Cette approche est essentiellement utilisée dans la conception des base des données fédérées [GV90].
- *La stratégie descendante* est utilisée pour la création d'une nouvelle base de données ou lors de la répartition d'une base de données centralisée. Elle consiste à diviser une base de données en un ensemble de bases de données locales, chacune étant gérée par le même SGBD. On est alors dans une situation où les bases de données sont homogènes. Cette répartition se réalise en deux étapes essentielles, à savoir, la fragmentation des données de la base et l'allocation, suivie éventuellement d'une optimisation locale [CMVN94]. Chaque phase peut être résolue par plusieurs approches rendant la tâche de la conception d'une base de données répartie difficile (voir Figure 5.9).

5.8.1.2 Intérêt de la fragmentation

Dans un contexte distribué, la fragmentation consiste à partitionner les tables du schéma conceptuel global de la base de données en plusieurs fragments. Cette fragmentation est faite en fonction des *besoins de chaque site du système distribué*. Les fragments issus du processus de partitionnement seront ensuite alloués de façon optimale sur les sites du système réparti (voir Figure 5.9). L'optimisation est guidée par une fonction objectif (dans la plupart des cas, on cherche à minimiser les coûts de transfert des données lors de l'exécution d'un ensemble de requêtes [Ape88, KP97]). Une fois les fragments alloués aux sites, des optimisations locales peuvent être réalisées au niveau de chaque site (comme par exemple, la création d'index propres aux fragments de chaque site).

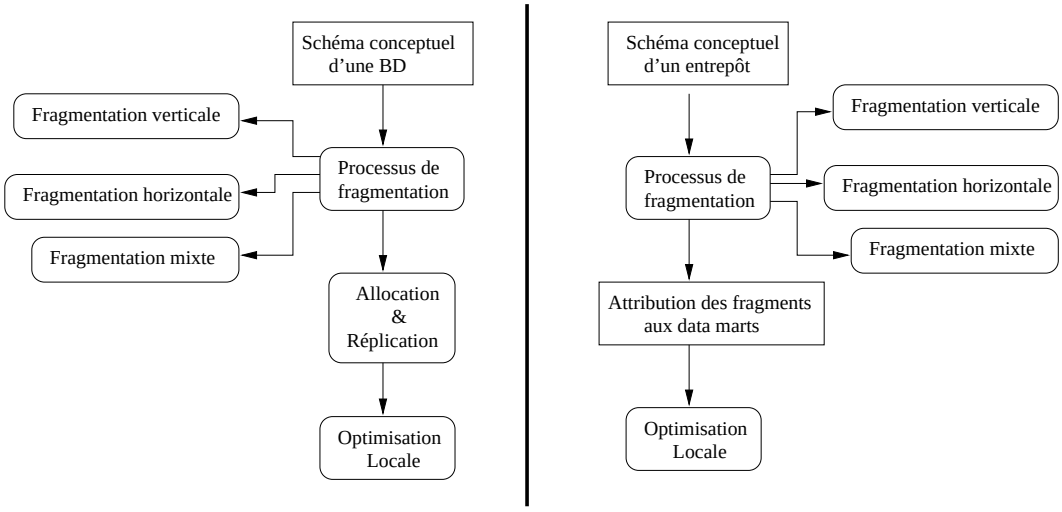


FIG. 5.9: Analogie entre la conception des base de données réparties et la conception de data marts

Par contre, la fragmentation dans les data marts consiste, en fonction des *besoins de chaque data mart*, à diviser l'entrepôt en sous-ensembles d'entrepôts. Ces derniers seront ensuite attribués (ou alloués) aux data marts. La fragmentation dans ce contexte, consiste donc à extraire les données de l'entrepôt nécessaires pour chaque data mart.

Le problème d'attribution des fragments aux data marts peut être vu comme un problème d'allocation classique dans les bases de données (relationnelles ou objet) [Ape88, BHY91, BKL98a], où les sites correspondraient aux data marts. Une fois les data marts construits, les mêmes techniques d'optimisation utilisées dans les entrepôts de données peuvent être utilisées.

Le problème d'attribution des fragments aux data marts reste de ce fait une question de recherche ouverte [BKM00b].

5.8.2 La fragmentation et l'adaptation des vues

Bellahsène et al. [BM00] travaillent maintenant sur l'utilisation de la fragmentation dans le processus d'adaptation des vues. Leurs résultats préliminaires ont montré que la fragmentation contribue d'une manière significative à l'adaptation soit après redéfinition soit lorsque les schémas des sources de données changent.

5.9 Conclusion

Dans ce chapitre nous avons présenté une méthodologie complète pour la fragmentation horizontale dans les entrepôts de données modélisés par des schémas en étoile. Nous avons d'abord développé un algorithme de fragmentation pour les tables de dimensions et la table des faits, en adaptant les techniques utilisées dans les bases de données relationnelles. Par la suite, nous avons présenté les problèmes liés à la fragmentation de la table des faits en fonction des tables de dimensions et nous

avons suggéré des solutions. Un modèle de coût a été développé afin de mesurer le résultat de la fragmentation. Finalement des expériences ont été conduites et ont montré qu'avec une fragmentation adéquate des tables de dimensions, il est possible d'améliorer les performances d'exécution des requêtes décisionnelles.

Nos perspectives se situent dans la considération des deux problèmes cités dans la section 5.8.

Chapitre 6

Nouveaux algorithmes pour la fragmentation horizontale

6.1 Introduction

Nous avons vu précédemment que la fragmentation horizontale permet de réduire le temps d'exécution des requêtes en minimisant le nombre d'accès (non nécessaires) aux données. Cependant, pour atteindre cet objectif, il faut disposer d'algorithmes de fragmentation: (i) ayant une complexité raisonnable, et (2) offrant un moyen d'évaluer la réduction du temps d'exécution des requêtes.

1. En nous penchant sur la littérature concernant la fragmentation horizontale, force est de constater que la plupart des algorithmes proposés (dans le modèle relationnel ou objet) sont d'une grande complexité (par exemple l'algorithme dirigé par la complétude et la minimalité des prédicats [CP84, OV91] génère 2^n fragments, n représentant le nombre de prédicats simples). Pour pallier cet inconvénient, Navathe et al. [NKR95] ont montré que l'algorithme qu'ils ont proposé pour la fragmentation verticale, d'une complexité moindre, pouvait être adapté à la fragmentation horizontale [NR89]. Néanmoins, les auteurs ont simplement décrit les principes de cette adaptation, mais n'ont pas élaboré un algorithme spécifique.
2. Tout algorithme de fragmentation (verticale ou horizontale) doit permettre de quantifier sa contribution à la réduction du temps d'exécution des requêtes. Il existe des moyens pour quantifier les algorithmes de la **fragmentation verticale** comme l'évaluateur de partition proposé par Chakravarthy [CMVN94] ou le modèle de coût développé dans [FKL97]. Malheureusement, cette quantification n'est pas fournie pour les algorithmes de fragmentation horizontale actuellement disponibles [CP84, OV91, BKS97, EB95, NKR95]. Il n'est donc pas possible de comparer leurs performances.

En conclusion, en l'absence d'une mesure de quantification, personne ne peut assurer que tel algorithme est meilleur qu'un autre du point de vue de la rapidité de l'exécution des requêtes. Le reproche que nous pouvons faire aux algorithmes de fragmentation horizontale existants est l'utilisation de la fréquence d'accès des requêtes comme seul *critère de regroupement* des n-uplets en fragments horizontaux.

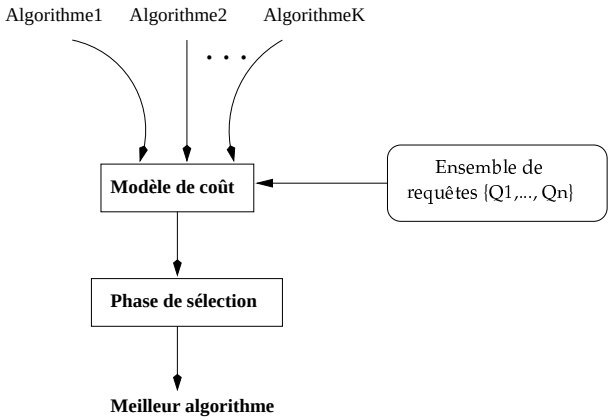


FIG. 6.1: Le modèle de coût comme outil de sélection

Or, dans les applications basées sur les entrepôts de données, d'autres paramètres sont aussi importants dans la définition des schémas de fragmentation comme: *la sélectivité des prédicats, la taille des relations à fragmenter, la taille de la mémoire centrale, la taille de la page mémoire, la taille des vues matérialisées et des index, etc.*

Ce chapitre est divisé en deux parties principales. La première est consacrée à l'adaptation de l'algorithme de Navathe et al. [NR89] à la fragmentation horizontale primaire. Rappelons que cet algorithme fait partie des algorithmes basés sur les affinités des prédicats. Dans la deuxième partie, un modèle de coût calculant le nombre d'entrées-sorties lors de l'exécution d'un ensemble des requêtes sur un schéma fragmenté est introduit. Ce modèle de coût peut avoir deux rôles différents:

- Il peut être utilisé comme outil de mesure pour quantifier le bénéfice de chaque algorithme de fragmentation proposé dans la littérature. Il est alors possible de sélectionner le meilleur algorithme pour un ensemble requêtes (Figure 6.1).
- Il peut également être utilisé pour définir de nouveaux types d'algorithmes de fragmentation horizontale. Pour éclairer ce point, considérons la situation suivante: soit T une table d'un entrepôt de données [1] que nous souhaitons fragmenter. Supposons que cette table soit accédée par un ensemble de requêtes (les plus fréquentes). À partir des prédicats de sélection définis sur cette table, nous pouvons énumérer tous les schémas de fragmentation possibles de T. À l'aide de ce modèle, le coût d'exécution de l'ensemble de requêtes sur chaque schéma est calculé. Finalement, le schéma satisfaisant un coût minimal est sélectionné.

Dans cette optique, nous introduisons un nouveau type d'algorithmes pour la fragmentation horizontale primaire, à savoir: **les algorithmes dirigés par un modèle de coût**. Deux algorithmes seront proposés: (1) *un algorithme exhaustif*, et (2) *un algorithme approximatif* [BKB98].

[1] L'idée de l'utilisation d'un modèle de coût dans la définition de nouveaux algorithmes de fragmentation horizontale a été, de prime abord, développée dans le contexte des bases de données orientées-objet [BKB98, BKL98b].

6.2 Un algorithme dirigé par les affinités

Dans cette section, nous adaptons l'algorithme de Navathe et al. [NKR95] (que nous appelons *algo_vertical*) développé pour la fragmentation verticale à la fragmentation horizontale. Rappelons que l'algo_vertical possède cinq étapes principales (voir Chapitre 2): (i) *l'énumération des attributs utilisés par les requêtes*, (ii) *la construction de la matrice d'usage des attributs*, (iii) *la construction de la matrice d'affinité des attributs*, (vi) *le regroupement des attributs*, et (v) *la construction des fragments verticaux*. Cet algorithme a une complexité polynômiale ($O(n^2)$, où n est le nombre des attributs de la table à fragmenter).

On peut envisager de reprendre les mêmes principes pour regrouper les n-uplets au lieu des attributs et obtenir ainsi une fragmentation horizontale. Par analogie les cinq étapes d'un algorithme de fragmentation horizontale seraient donc: (i) *l'énumération des prédicats utilisés par les requêtes*, (ii) *la construction de la matrice d'usage des prédicats*, (iii) *la construction de la matrice d'affinité des prédicats*, (vi) *le regroupement des prédicats*, et (v) *la construction des fragments horizontaux*.

Cependant les prédicats n'ont pas les mêmes caractéristiques que les attributs, et cette substitution des attributs par des prédicats dans *algo_vertical* n'est pas suffisante pour obtenir les fragments horizontaux. Un algorithme des affinités pour la fragmentation horizontale (que nous appelons *algo_affi_horiz*) doit en plus tenir compte des implications entre les prédicats qui peuvent influencer l'étape de regroupement.

Nous décrivons d'abord les principales étapes de *algo_affi_horiz* avant de discuter de sa complexité et de sa correction.

6.2.1 Les phases de l'algorithme des affinités

Comme tous les algorithmes basés sur les affinités, *algo_affi_horiz* part d'une table $T(K, A_1, A_2, ..., A_l)$ (appartenant à un entrepôt de données (de type ROLAP) ou à une base de données), d'un ensemble de requêtes $Q = \{Q_1, Q_2, ..., Q_n\}$ (les plus fréquemment utilisées) et de leurs fréquences d'accès $F = \{f_1, f_2, ..., f_n\}$. L'ensemble des requêtes et leurs fréquences sont déterminées par l'administrateur [Kar96].

L'algorithme *algo_affi_horiz* se caractérise par huit étapes, à savoir [BKA00, BKS97]: (1) *l'énumération des prédicats de sélection*, (2) *la construction de la matrice d'usage des prédicats*, (3) *la construction de la matrice d'affinité des prédicats*, (4) *le regroupement des prédicats*, (5) *l'optimisation des composantes*, (6) *la formation des minterms*, (7) *la formation des fragments horizontaux*, et (8) *la génération des fragments disjoints*.

Pour faciliter la compréhension de chaque étape, nous illustrons leur fonctionnement par un exemple.

1. **L'énumération des prédicats de sélection:** généralement une requête possède des prédicats de sélection et des prédicats de jointure (figurant dans la clause WHERE de la syntaxe SQL de cette requête). Rappelons que tout algorithme de fragmentation horizontale utilise *seulement* les prédicats de sélection définis dans les requêtes de départ. À partir des n requêtes de départ, nous

énumérons l'ensemble de tous les prédicats de sélection simples (dénoté par PSS) définis sur la table T. Soit $ATTR$ l'ensemble des attributs de la table T utilisés par les prédicats de PSS. Notons par N et r ($r \leq l+1$) les cardinalités de PSS et $ATTR$, respectivement [2].

Exemple 21 *Considérons la table de dimension CLIENT du schéma en étoile de la Figure 2.5. Cette table possède cinq attributs: l'identifiant du client (CID) (la clé primaire de CLIENT), l'Age, le Sexe, la Ville, et la Région. Supposons que quatre requêtes $\{Q_1, Q_2, Q_3, Q_4\}$ accèdent à cette table avec les prédicats de sélection sont les suivants:*

- Q_1 - (p_1) : *Age* ≤ 30, (p_5): *Sexe = 'M'*
- Q_2 - (p_2) : *Age* ≤ 40, (p_5) : *Sexe = 'M'*
- Q_3 - (p_3) : *Age = 18*, (p_6) : *Sexe = 'F'*, (p_7) : *Ville = "Grenoble"*
- Q_4 - (p_4) : $50 \leq$ *Age* ≤ 60, (p_6) : *Sexe = 'F'*

En examinant ces prédicats, nous constatons que seulement trois attributs parmi les cinq sont utilisés, à savoir: l'Age, le Sexe et la Ville, que nous renommons: A_1, A_2 et A_3.

	Prédicats							
Requêtes	p_1	p_2	p_3	p_4	p_5	p_6	p_7	Fréquence
Q_1	1	0	0	0	1	0	0	20
Q_2	0	1	0	0	1	0	0	35
Q_3	0	0	1	0	0	1	1	30
Q_4	0	0	0	1	0	1	0	15

FIG. 6.2: La matrice d'usage des prédicats

2. **La construction de la matrice d'usage des prédicats:** comme son nom l'indique, cette matrice représente l'utilisation des prédicats dans les requêtes. Les lignes et les colonnes de cette matrice sont associés aux n requêtes de départ et les N prédicats obtenus à l'étape 1, respectivement.
Les éléments $mup_{i,j}$ ($1 \leq i \leq n, 1 \leq j \leq N$) de cette matrice sont binaires (0 ou 1) et sont définis comme suit:
$$mup_{i,j} = \begin{cases} 1, & \text{si la requête } Q_i \text{ utilise le prédicat } p_j \\ 0, & \text{sinon} \end{cases}$$

   Exemple 22 *(suite) La matrice d'usage des prédicats de notre exemple est montrée dans la Figure 6.2. Elle contient 4 lignes (pour les requêtes) et 7 colonnes (pour les prédicats). Nous ajoutons à cette matrice une colonne représentant la fréquence d'accès de chaque requête.*

3. **La construction de la matrice d'affinité des prédicats:** cette matrice est générée de la même manière que la matrice d'affinité des attributs de la fragmentation verticale. C'est une matrice carrée ($N \times N$) et symétrique dont

[2] l représente le nombre d'attributs à l'exception de la clé de la table T

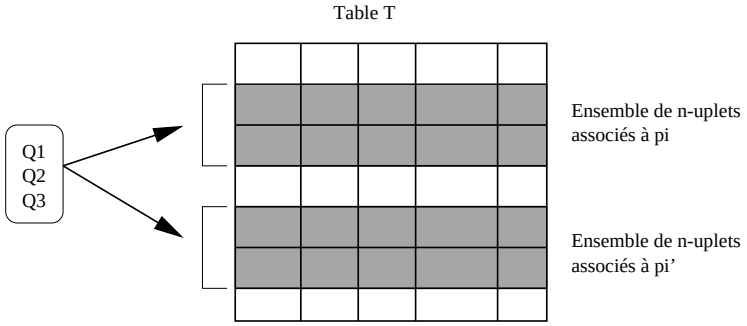


FIG. 6.3: La signification de la valeur numérique

les lignes et les colonnes représentent les prédicats obtenus dans l'étape 1. Elle est construite à partir de la matrice d'usage des prédicats. Chaque élément $aff(p_i, p_{i'})$ $(1 \leq i, i' \leq N)$ de cette matrice peut avoir l'une des trois valeurs suivantes (contrairement à une matrice d'affinité des attributs qui possède seulement des valeurs numériques (voir chapitre 2)):

(a) **Numérique:** qui donne la somme des fréquences d'accès (que nous notons par λ) de toutes les requêtes utilisant *simultanément* les deux prédicats p_i et $p_{i'}$. Si nous raisonons en fonction des n-uplets de la table T, cela signifie que les n-uplets satisfaisants les prédicats p_i et $p_{i'}$ sont accédés simultanément par une ou plusieurs requêtes ayant une fréquence égale à λ (voir Figure 6.3).

(b) **Non numérique:** qui peut être soit " $\Rightarrow$ ", " $\Leftarrow$ " ou "*".

- " $\Rightarrow$ " ou " $\Leftarrow$ " signifient que le prédicat p_i implique le prédicat $p_{i'}$ ou vice versa. Les n-uplets accédés par ces deux prédicats sont obligatoirement non disjoints (voir Figure 6.4).
- La valeur " * " indique la "similarité" entre deux prédicats p_i et p_j. Deux prédicats p_i et $p_{i'}$ sont similaires si les conditions suivantes sont satisfaites:
  i. Les deux prédicats p_i et $p_{i'}$ doivent être définis sur le même attribut,
  ii. Ils doivent être utilisés avec un prédicat commun $p_{i''}$, et
  iii. Le prédicat $p_{i''}$ doit être défini sur un attribut différent de celui utilisé par p_i et $p_{i'}$.

Exemple 23 *(suite) À partir de la matrice d'usages des prédicats de la Figure 6.2, la matrice d'affinité des prédicats est construite (voir Figure 6.5). Cette matrice contient 7 lignes et 7 colonnes autant que de prédicats. La valeur* $aff(p_1, p_5)$ *est égale à 20, la fréquence de* Q_1 *car* Q_1*, est la seule requête utilisée simultanément par* p_1 *et* p_5*. Cette matrice contient trois implications entre prédicats à savoir:* $p_1 \Rightarrow p_2$*,* $p_3 \Rightarrow p_1$*,* $p_3 \Rightarrow p_2$*. Notons que* p_1 *et* p_2 *sont similaires, et* p_3 *et* p_4 *le sont également.*

4. **Le regroupement des prédicats:** dans cette étape, nous adaptons l'algorithme de regroupement graphique développé pour la fragmentation verticale

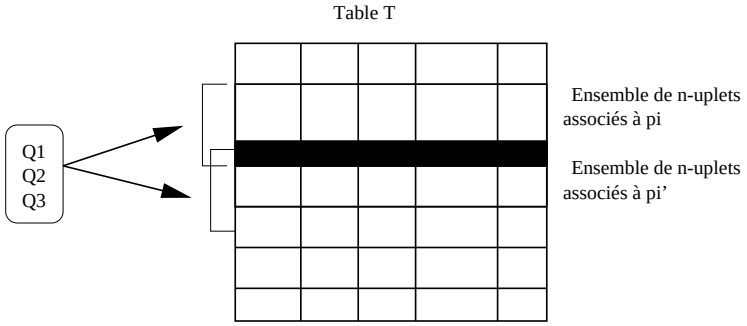


FIG. 6.4: La signification de la valeur des implications

$$
\begin{array}{c}
 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7
\end{array}
\begin{array}{c}
\begin{array}{ccccccc} p_1 & p_2 & p_3 & p_4 & p_5 & p_6 & p_7 \end{array} \\
\left(\begin{array}{ccccccc}
20 & \Rightarrow & \Leftarrow & 0 & 20 & 0 & 0 \\
\Leftarrow & 35 & \Leftarrow & 0 & 35 & 0 & 0 \\
\Rightarrow & \Rightarrow & 30 & * & 0 & 30 & 30 \\
0 & 0 & * & 15 & 0 & 15 & 0 \\
20 & 35 & 0 & 0 & 55 & 0 & 0 \\
0 & 0 & 30 & 15 & 0 & 45 & 30 \\
0 & 0 & 30 & 0 & 0 & 30 & 30
\end{array}\right)
\end{array}
$$

FIG. 6.5: La matrice d'affinité des prédicats

par Navathe et al. [NR89]. La structure de données principale utilisée est un graphe complet et étiqueté, appelé "graphe d'affinité des attributs". Les noeuds de ce graphe représentent les attributs, et une arête entre deux attributs représente la valeur d'affinité (qui est numérique) (voir le chapitre 2 pour le fonctionnement de cet algorithme). L'objectif principal est la formation des cycles qui généreront par la suite des fragments verticaux.

L'adaptation de cet algorithme dans le contexte de la fragmentation horizontale, est la prise en compte des valeurs non numériques et leurs effets sur le processus de regroupement. Pour ce faire, trois règles sont introduites:

(a) Une valeur numérique (sauf 0) a une plus grande priorité que les valeurs non numériques ("$\Rightarrow$", "$\Leftarrow$" et "*") lorsque nous explorons le graphe. En effet, le but de cet algorithme est le regroupement des prédicats ayant une grande affinité représentée par la fréquence d'accès.

(b) "$\Rightarrow$" et "$\Leftarrow$" ont une plus grande affinité que "*", puisque ce dernier représente seulement une connectivité logique entre deux prédicats à travers leur usage avec un prédicat commun.

(c) Si $p_i \Rightarrow p_j$ alors p_j a une plus grande priorité que p_i et vice versa.

L'algorithme adapté part d'une matrice d'affinité des prédicats e t construit un graphe complet et étiqueté appelé "graphe d'affinité des prédicats" $GAP = (V, E)$. L'ensemble V représente les prédicats de la matrice d'affinité des prédicats. Chaque arête $e(p_i, p_j)$ est étiquetée par un poids représentant une valeur d'affinité entre le prédicat p_i et le prédicat p_j (qui peut être, soit numérique, soit non numérique). Une fois le graphe d'affinité construit, les cycles peuvent

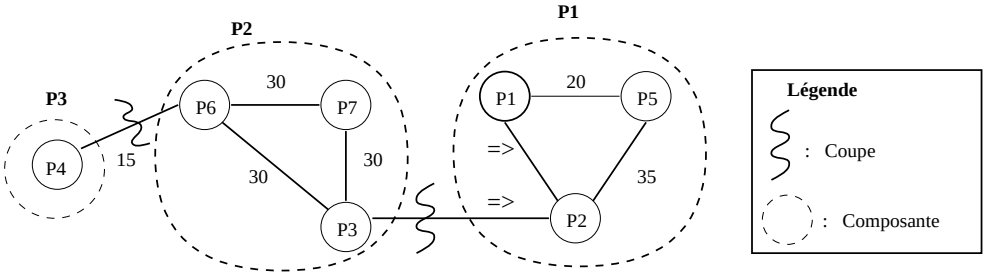


FIG. 6.6: Les ensembles de prédicats générés par l'approche graphique

être formés comme décrits ci-après.
L'algorithme commence par sélectionner un noeud p_i du graphe d'affinité d'une manière aléatoire. Ce noeud représentera le noeud de départ du cycle C_1 (c'est-à-dire, $C_1 = \{p_i\}$). Ensuite il essaye d'étendre le cycle C_1 en ajoutant d'autres noeuds. Pendant la phase d'extension d'un cycle, les arêtes ayant les valeurs d'affinités "$\Rightarrow$", "$\Leftarrow$", et "*" sont ignorées; en effet ces valeurs représentent des relations logiques implicites entre les prédicats et non pas l'affinité entre ces prédicats. Ces relations implicites sont utilisées pour réduire le nombre de fragments horizontaux.

Une fois le cycle formé, ses noeuds sont écartés, et nous réitérons la même procédure sur les autres noeuds afin de former d'autres cycles. À la fin de l'algorithme de regroupement, un ensemble de cycles $\mathcal{C} = \{C_1, C_2, ..., C_\gamma\}$ est obtenu. Notons par *une composante* P_i l'ensemble des noeuds du cycle C_i.

Le fonctionnement de cet algorithme de regroupement est présenté à travers l'exemple qui suit.

Exemple 24 *(suite) le graphe d'affinité est construit à l'aide de la matrice d'affinité des prédicats (Figure 6.5). Il possède sept noeuds, et chaque arête est étiquetée par sa valeur d'affinité. Par exemple, l'arête entre p_1 et p_5 est libellée par 20 (voir Figure 6.6). Les arêtes ayant un poids nul ne sont pas représentées dans ce graphe pour des raisons de simplicité.*

Le processus de regroupement commence par le noeud 1 qui représentera le noeud de départ du premier cycle C_1. En examinant la matrice d'affinité, nous ne trouvons qu'un seul prédicat p_5 ayant une grande affinité (numérique) avec p_1. Le cycle C_1 est donc augmenté par l'ajout de p_5. L'algorithme cherche toujours à étendre C_1, selon le même principe et le prédicat p_2 est ensuite sélectionné. À partir du prédicat p_2, nous ne pouvons pas inclure de nouveaux noeuds ayant une grande affinité (numérique), mais un cycle a été formé car le prédicat p_1 implique le prédicat p_2. Le cycle obtenu est: $C_1 = p_1, p_5, p_2, p_1$, dont sa composante est égale à $P_1 = \{p_1, p_2, p_5\}$. Le cycle C_1 est donc écarté en coupant ses éléments du reste des noeuds du graphe (voir Figure 6.6). L'algorithme réitère le processus de formation de cycle, en utilisant les noeuds restants du graphe.

Finalement, trois cycles C_1, C_2 et C_3 sont obtenus (voir Figure 6.6) dont les composantes sont: $P_1 = \{p_1, p_2, p_5\}$, $P_2 = \{p_3, p_6, p_7\}$, et $P_3 = \{p_4\}$.

5. **L'optimisation des composantes:** dans cette étape, nous optimisons chaque composante de cycle C_i $(1 \leq i \leq \gamma)$ (obtenu dans l'étape 4) en utilisant les implications entre ses prédicats (figurant dans la matrice d'affinité). S'il existe dans une composante, deux prédicats p_i et p_j tel que: $p_i \Rightarrow p_j$, alors le prédicat p_i sera éliminé. Cette procédure pourrait réduire le nombre de prédicats de chaque composante P_i.

   Tous les prédicats figurant dans les composantes sont appelés **les prédicats de fragmentation**. Les attributs figurant dans tous les prédicats de fragmentation sont appelés **attributs de fragmentation**. Soient M et r le nombre de prédicats et d'attributs de fragmentation. Nous notons par M_i le nombre de prédicats de fragmentation définis sur l'attribut A_i $(1 \leq i \leq r)$.

   Exemple 25 *(suite) Les trois composantes obtenues dans l'étape 4 sont optimisées de la manière suivante:*

   - *la composante P_1 contient initialement trois prédicats: p_1, p_2, et p_5. Notons que $p_1 \Rightarrow p_2$ (voir Figure 6.5), en conséquence, P_1 devient $P'_1 = \{p_2, p_5\}$.*
   - *la composante P_2 contient trois prédicats chacun étant défini sur un attribut différent. Donc, cette composante reste inchangée.*
   - *la dernière composante P_3 contient un seul prédicat (p_4), donc elle est inchangée.*

   L'ensemble de prédicats de fragmentation est: $\{p_2, p_3, p_4, p_5, p_6, p_7\}$. Les attributs de fragmentation correspondants aux trois composantes sont: l'Age, le Sexe et la Ville.

6. **La formation des minterms:** Notons que les composantes résultant de l'étape 5 peuvent ne pas couvrir tous les attributs de fragmentation. Par exemple, la composante P_3 de l'exemple 25 contient seulement un prédicat p_4 défini sur l'attribut de fragmentation Age, mais aucun prédicat défini sur les deux autres attributs de fragmentation (le Sexe et la Ville). Cette situation peut donner des fragments non disjoints et de grosse granularité (coarse grain). Pour aborder ce problème, la notion de complétude d'une composante est introduite.

   Définition 33 *Une composante est dite complète si elle couvre tous les attributs de fragmentation. Elle est dite incomplète si elle couvre seulement une partie des attributs de fragmentation.*

   Le but de cette étape de l'algorithme *algo_affi_horiz* est de garantir que toute composante de cycle soit complète.

   Dans le cas où toutes les composantes issues de l'étape 5 sont complètes, nous passons directement à l'étape suivante qui va produire les fragments horizontaux.

   Dans le cas contraire, nous devons identifier les composantes incomplètes afin de les rendre complètes. Pour ce faire, pour chaque composante P_i, nous sélectionnons l'ensemble des attributs de fragmentation (T_i) *qui ne sont pas utilisés* par cette dernière.

Pour réaliser cette sélection, une nouvelle matrice que nous appelons *matrice d'usage des attribute* est introduite. Les lignes et les colonnes de cette matrice représentent les composantes de cycles et les attributs de fragmentation, respectivement. Les valeurs des éléments de cette matrice sont binaires (1 ou 0). Chaque élément de cette matrice est défini de la manière suivante:

$$(P_i, A_j) = \begin{cases} 1 \text{ si l'attribut de fragmentation } A_j \text{ appartient à la composante } P_i \\ 0 \text{ sinon} \end{cases}$$

Avec cette matrice, l'identification de l'ensemble T_i est réalisée comme suit:

$$T_i = \{A_j \mid 1 \leq j \leq r,\ (P_i, A_j) = 0\}$$

Soit Φ l'ensemble de toutes les composantes incomplètes.

Une fois la phase d'identification achevée, nous exécutons la procédure nommée *étendre une composante*. Cette procédure part d'une composante incomplète P_i de Φ. Pour chaque attribut $A_j \in T_i$, elle consiste à éclater cette composante en M_j [3] nouvelles composantes. Chaque nouvelle composante contient le(s) prédicat(s) actuels de P_i **plus** l'un des prédicats définis sur A_j.

Les étapes principales de cette procédure sont décrites dans l'algorithme 10.

Algorithme 10 étendre une composante incomplète

1: **Entrées:** une composante P_i de Φ
2: **for** Tout attribut $A_j \in T_i$ **do**
3: Étendre P_i en M_j nouvelles composantes, où chacune contient le(s) prédicat(s) existants de P_i **plus** un prédicat défini sur A_j.
4: **end for**

Exemple 26 *(suite) nous illustrons cette étape en reprenant les trois composantes résultants de l'étape 5: P_1, P_2, et P_3. Elles possèdent trois attributs de fragmentation: l'Age (A_1), le Sexe (A_2) et la Ville (A_3). La MUA contient alors trois lignes et trois colonnes, comme le montre la Figure 6.7. À partir de cette matrice, nous identifions facilement l'ensemble T_i correspondant à chaque composante:*

- *La composante P_1 contient tous les attributs de fragmentation, à l'exception de l'attribut A_3. L'ensemble T_1 est égal à $\{A_3\}$.*
- *La composante P_2 contient tous les attributs de fragmentation. En conséquence, l'ensemble T_2 est égal à ϕ.*
- *La composante P_3 contient un seul attribut de fragmentation (A_1). L'ensemble T_3 est égal à $\{A_2, A_3\}$.*

Nous remarquons qu'il existe deux composantes incomplètes (correspondant à T_1 et T_3) et une complète (elle ne sera pas concernée par cette phase).

De ce fait, la procédure étendre *est exécutée deux fois: une première fois pour la composante P_1 et une seconde pour la composante P_3. Commençons par la*

[3] M_j étant la cardinalité de l'ensemble de prédicats définis sur l'attribut A_j.

$$\begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \end{array} \begin{array}{c} \begin{array}{ccc} A_1 & A_2 & A_3 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \end{pmatrix} \end{array}$$

FIG. 6.7: La matrice d'usage des attributs

composante P_1, qui n'utilise pas l'attribut Ville. Sur ce dernier, un seul prédicat est défini, à savoir p_7 : Ville = "Grenoble". Nous étendons P_1 en ajoutant ce prédicat. Une nouvelle composante P_{11} est obtenue. Elle possède donc trois prédicats couvrant tous les attributs de fragmentation: $P_{11} = \{p_2, p_5, p_7\}$.
De manière similaire, la procédure étendre *est appliquée sur la composante P_3. Deux attributs ne figurent pas dans cette composante: Sexe et Ville. Sur l'attribut Sexe, deux prédicats sont définis (p_5 : Sexe = "M" et p_6 : Sexe = "F"); de ce fait la composante P_3 est éclatée en deux nouvelles composantes P_{31} et P_{32}, telles que: $P_{31} = \{p_4, p_5\}$, et $P_{32} = \{p_4, p_6\}$. Ces deux dernières ne couvrent pas non plus l'attribut Ville. Elles sont donc éclatées en ajoutant le prédicat p_7 : Ville = "Grenoble". Nous obtenons donc deux composantes complètes P_{31} et P_{32} telles que:*
$P_{31} = \{p_4, p_5, p_7\}$, $P_{32} = \{p_4, p_6, p_7\}$.
Finalement, nous obtenons quatre composantes complètes P_1, P_2, P_3, et P_4 définies comme suit:
$P_1 = \{p_2, p_5, p_7\}, P_2 = \{p_3, p_6, p_7\}, P_3 = \{p_4, p_5, p_7\}$ *et* $P_4 = \{p_4, p_6, p_7\}$.

7. **La formation des fragments horizontaux:** Soit $\Psi = \{P_1, P_2, ..., P_\beta\}$ l'ensemble des composantes obtenues dans l'étape 6. Chaque composante génère un fragment. La clause de prédicat associé à un fragment est défini comme suit:
   Si les prédicats d'une composante P_i $(1 \leq i \leq \beta)$ référencent le même attribut, nous lions ces prédicats par le connecteur logique OU, sinon par le connecteur logique ET.
   De plus un fragment supplémentaire, que nous appelons ELSE est généré. Il représente la négation de la disjonction de toutes les clauses de prédicats définissant les β fragments.

   Exemple 27 *(suite) L'ensemble Ψ correspondant à notre exemple est égal à $\{P_1, P_2, P_3, P_4\}$. Étant donné que tous les prédicats de chaque composante sont définis sur des attributs différents, les prédicats sont liés par le connecteur logique ET. En conséquence, les fragments horizontaux générés par notre algorithme sont définis par les clauses de prédicats suivantes:*
   - *Client_1 avec la clause cl_1 : (Age $\leq$ 40) $\wedge$ (Sexe = 'M') $\wedge$ (Ville = "Grenoble")*
   - *Client_2 avec la clause cl_2 : (Age = 18) $\wedge$ (Sexe = 'F') $\wedge$ (Ville ="Grenoble")*
   - *Client_3 avec la clause cl_3 : (50 $\leq$ Age $\leq$ 60) $\wedge$ (Sexe = 'M') $\wedge$ (Ville = "Grenoble")*

- *Client_4 avec la clause* cl_4 : $(50 \leq Age \leq 60) \wedge (Sexe = \text{'F'}) \wedge (Ville = \text{"Grenoble"})$
- *Client_5 avec la clause* $cl_5 : \neg(cl_1 \bigvee cl_2 \bigvee cl_3 \bigvee cl_4)$.

Le fragment horizontal Client_5 représente la négation de la disjonction de toutes les clauses définies ci-dessus.

8. **La génération des fragments disjoints:** les fragments obtenus pourraient être non disjoints, car il n'y a aucune condition sur les prédicats de sélection définis dans les requêtes de départ. Les fragments horizontaux sont disjoints si leurs clauses sont mutuellement exclusives [KNM94, OV91]. L'objectif de cette étape est de raffiner les fragments non disjoints afin d'obtenir des fragments disjoints. Les fragments non disjoints sont donc combinés afin de former un seul fragment couvrant tous les n-uplets de ces fragments.

6.2.2 Complexité de l'algorithme des affinités

Notre algorithme étant constitué de huit étapes, nous décrivons la complexité de chacune d'elles:

- La complexité de la phase de construction de la matrice d'usage des prédicats est $O(n \times N)$, où n et N représentent le nombre des requêtes et des prédicats simples utilisés par ces dernières, respectivement.
- Notons que la matrice d'affinité des prédicats est carrée et symétrique. La complexité de sa construction est dans le pire des cas $O(N^2)$.
- L'algorithme de regroupement a une complexité de $O(N^2)$ [NR89].
- La phase d'optimisation teste chaque prédicat d'une composante, et vérifie l'existence d'une implication avec d'autres prédicats. Puisque nous avons γ composantes, et N prédicats, le pire des cas est $O(\gamma \times (N-1))$.
- La phase de formation des minterms est basée sur la matrice d'usage des attributs qui a γ lignes et r colonnes. Sa complexité est $O(r \times \gamma)$.
- La complexité de la phase de formation et de génération des fragments disjoints est $O(\beta)$.

La complexité totale de cet algorithme est: $O(n \times N + N^2)$. Rappelons que les algorithmes basés sur la complétude et la minimalité (voir Chapitre 2) ont une complexité de $O(2^n)$.

6.2.3 Les règles de correction

L'algorithme que nous avons proposé vérifie les trois règles de correction: la complétude, la reconstruction, et la disjonction.

- La complétude est assurée par l'ajout du fragment ELSE qui contient tous les n-uplets ne satisfaisant pas les prédicats des autres fragments.
- La reconstruction est évidente, elle est réalisée par l'opération d'union.
- La disjonction est assurée par l'étape 8 de l'algorithme (génération des fragments disjoints).

6.2.4 Bilan de cette section

L'algorithme algo_aff_horiz que nous venons de présenter a une complexité polynômiale, ce qui est raisonnable par rapport aux algorithmes basés sur la complétude et la minimalité des prédicats de sélection (voir Chapitre 2). Notons que le seul critère de regroupement utilisé par cet algorithme est "la fréquence d'accès des requêtes". Ce genre d'algorithmes a été développé dans le contexte des bases de données réparties [CP84, OV91, NR89, NKR95, BKS97] dans lequel la fréquence d'accès des requêtes sur des sites est considérée comme un *paramètre essentiel*. Le contexte réparti n'est pas le seul domaine d'application de la fragmentation, qui peut être également utilisée dans un contexte centralisé pour minimiser le coût des requêtes [Ora99].

Qui dit minimisation du coût d'exécution des requêtes, dit prise en considération de tous les paramètres physiques utilisés par l'optimiseur de requêtes. Ces paramètres contribuent fortement à l'amélioration ou à la dégradation des performances des requêtes. Par exemple, dans l'environnement des entrepôts de données, des paramètres comme la taille des vues matérialisées, la taille des index (de jointure), la taille de la mémoire centrale, etc. ont une incidence importante sur l'accélération des requêtes. Dans certaines applications, nous pouvons avoir des requête moins fréquentes, mais dont les coûts d'exécution sont très élevés.

Tout algorithme de fragmentation (horizontale, verticale et mixte) devrait prendre en considération des paramètres physiques, ce qui n'est malheureusement pas le cas des algorithmes de la fragmentation horizontale basés sur les affinités, la complétude et la minimalité des prédicats.

Dans la section suivante, nous nous attachons donc à développer un *nouveau type d'algorithme* prenant en compte les paramètres physiques de l'entrepôt.

6.3 Les algorithmes dirigés par un modèle de coût

Dans cette section, un nouveau type d'algorithme basé sur un modèle de coût est présenté. Grâce à ce modèle de coût, il est possible d'évaluer la qualité de la solution fournie par les algorithmes de la fragmentation horizontale basés sur les affinités, la complétude et la minimalité des prédicats. Il peut également être utilisé pour définir d'autres algorithmes de fragmentation. Deux algorithmes de ce type sont développés: (1) *un algorithme exhaustif* et (2) *un algorithme approximatif*.

6.3.1 Motivations

Un modèle de coût est la composante principale d'un optimiseur physique d'une base de données [Gru96]. Recenser les modèles de coût n'est pas une chose aisée [Gru96], car il n'existe pas un modèle unique mais une multitude de modèles adaptés à des systèmes très particuliers. Les modèles de coût peuvent avoir plusieurs rôles:

- *Choisir le meilleur plan d'exécution pour une requête donnée:* L'optimisation des requêtes se déroule en deux phases principales pour traduire la requête initiale dans un langage de plus en plus proche du système d'exécution [Naa99].

Ces deux phases sont l'optimisation logique et l'optimisation physique. La première phase traduit la requête dans l'algèbre de l'optimiseur pour la représenter sous la forme d'un plan d'exécution, arrange la requête sous une forme normalisée en employant des règles de normalisation, puis applique plusieurs transformations algébriques au moyen de règles de transformation pour produire l'espace de recherche correspondant à la requête. La deuxième phase, génère le plan d'exécution optimal parmi les nombreux plans possibles pour exécuter la requête. La génération de ce plan est réalisée en s'appuyant sur un modèle de coût.
Ce processus d'optimisation est complexe et a fait l'objet de nombreux travaux [Gra93].

- *Outil pour l'optimisation dynamique (adaptative) des requêtes:* Dès qu'il y a des changements au niveau des paramètres sur lesquels l'optimiseur sélectionne les plans optimaux d'une requête, le modèle de coût peut être utilisé pour trouver de nouveaux plans. Une technique utilisée pour répondre à ces changements est *la méthode de seuil* dans laquelle la reformulation du plan de requête intervient seulement lorsque le **coût** de la requête dépasse un certain seuil fixé par l'administrateur [Naa99].
- *Guider les algorithmes de recherche et de sélection:* dans les bases de données ou les entrepôts de données, nous trouvons plusieurs problèmes d'optimisation ou de sélection difficiles à résoudre. Deux bons exemples sont les problèmes de sélection des vues et des index (voir Chapitre 2 et 3). Pour les résoudre, nous disposons d'heuristiques qui sont guidées par des modèles de coût (voir chapitre 2).

Comme nous le constatons, les modèles de coût jouent un rôle essentiel dans l'optimisation des requêtes, ce qui nous amène à les utiliser pour *définir* de bons algorithmes de fragmentation horizontale.

Les modèles de coût reposent généralement sur trois composantes (voir Chapitre 3): (1) des statistiques des données stockées dans la métabase, (2) des formules pour évaluer la cardinalité des résultats intermédiaires, (3) et des formules pour évaluer le coût des requêtes.

6.3.2 Le modèle de coût et les algorithmes de fragmentation

Pour introduire notre approche, nous considérons un schéma en étoile d'un entrepôt de données [4]. Soit $T(K, A_1, A_2, ..., A_l)$ la table de dimension d'un entrepôt à fragmenter, et Q l'ensemble des requêtes les plus fréquemment utilisées.

Les composantes principales de ce type d'algorithmes sont: (i) *un générateur de schémas de fragmentation*, (ii) *un modèle de coût*, et (iii) *la sélection du schéma de fragmentation optimal* (voir la Figure 6.8).

[4] Les algorithmes que nous proposons peuvent être appliqués sur tout schéma de base de données (relationnelle ou objet), ou des schémas d'entrepôts de données autre que ROLAP (schéma en flocon de neige, schéma en constellation, etc.)

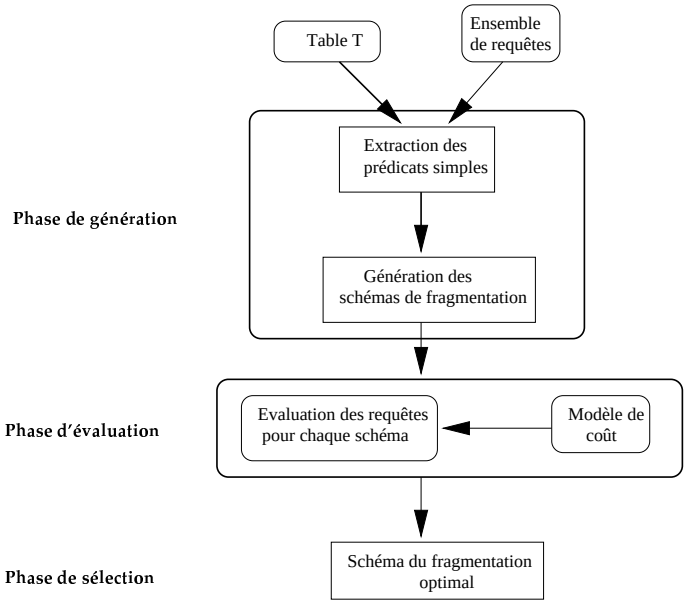


FIG. 6.8: Les principales composantes des algorithmes dirigés par un modèle de coût

- **Le générateur de schémas de fragmentation:** Les fragments horizontaux sont obtenus à partir des prédicats de sélection des requêtes définies sur la table à fragmenter (voir section 6.2).

  Le générateur de schémas de fragmentation consiste à générer tous les schémas de fragmentation possibles de la table T en utilisant les minterms obtenus à partir des prédicats de sélection définis sur T.
- **Le modèle de coût:** Il constitue le noyau de ce type d'algorithmes. Le modèle de coût peut être vu comme une fonction d'une seule variable représentant le schéma de fragmentation S_i et attribuant le coût d'exécution d'un ensemble de requêtes à ce schéma $C(S_i)$. L'unité de mesure du coût dépend de l'objectif de l'optimisation. Dans notre contexte, l'unité de mesure est le nombre d'entrées-sorties. L'optimalité et la qualité des solutions obtenues par ces algorithmes sont liées à ce modèle de coût. En conclusion, le modèle de coût doit être précis et prendre en compte les paramètres importants déterminés par l'administrateur de la base.
- **La sélection du schéma de fragmentation optimal:** Elle consiste à déterminer le schéma de fragmentation garantissant la meilleure performance parmi tous les schémas possibles.

Dans les sections suivantes, nous allons voir en détail les fonctionnalités de chaque composante.

6.3.3 Le générateur de schémas

Rappelons qu'un schéma de fragmentation représente le résultat du processus de fragmentation. Un schéma de fragmentation S d'une table T ayant m fragments horizontaux $\{F_1, F_2, ..., F_m\}$ est dénoté par: $S : \Big((F_1 : cl_1), (F_2 : cl_2), ..., (F_m :$

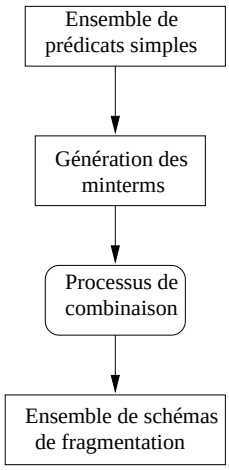


FIG. 6.9: Les étapes du processus de génération des schémas de fragmentation

$cl_m)$ $\Big)$, où chaque fragment F_i $(1 \leq i \leq m)$ est défini par une clause de prédicats cl_i.

Le processus de génération de tous les schémas de fragmentation possibles est réalisé en trois étapes principales (voir Figure 6.9): (1) *l'énumération de l'ensemble des prédicats simples*, (2) *la génération de l'ensemble des minterms*, et (3) *la combinaison de minterms.*

1. **L'énumération de l'ensemble des prédicats simples:** Comme dans l'étape 1 de l'algorithme des affinités de la section 6.2, nous énumérons l'ensemble de tous les prédicats de sélection définis sur la table T, à savoir $\Pi = \{p_1, p_2, ..., p_N\}$. Ces prédicats doivent être minimaux et mutuellement exclusifs [OV91].

2. **La génération des minterms:** À partir des prédicats de l'ensemble Π, l'ensemble de tous les minterms M est généré de la manière suivante:
$$M = \{m_i \mid m_i = \wedge_{p_k \in \Pi} \ p_k^*\}, \text{ avec } p_k^* = p_k \text{ ou } p_k^* = \neg p_k \quad .$$

3. **La combinaison des minterms:** Si nous faisons l'analogie entre les minterms et les fragments d'un schéma de fragmentation, nous pouvons dire qu'un minterm est une clause d'un fragment horizontal (voir l'étape 6 de l'algorithme de la section 6.2), d'où la propriété suivante.

   Propriété 2 *Un minterm forme un fragment horizontal potentiel.*

   La combinaison des minterms se fait à l'aide de la propriété suivante:

   Propriété 3 *Une combinaison de minterms obtenue par l'opérateur de disjonction (OU dénotée par $\vee$) forme un fragment horizontal potentiel.*

   En conséquence, le nombre de tous les schémas de fragmentation possibles d'une table donnée est le nombre de *toutes les combinaisons possibles* de minterms.

Exemple 28 *Pour illustrer les trois étapes du processus de génération de schémas de fragmentation, considérons l'exemple suivant:*

Supposons que nous ayons six prédicats simples et disjoints $\{p_1, p_2, ..., p_6\}$ *définis sur la table de dimension CLIENT (voir Table 6.1). Les minterms de prédicats générés*

Prédicats	**Description**
p_1	Age ≤ 40
p_2	Age > 40
p_3	Sexe = 'M'
p_4	Sexe = 'F'
p_5	Ville = "Grenoble"
p_6	Ville = "Paris"

TAB. 6.1: Les six prédicats

à partir de ces prédicats sont:

- m_1 : *(Age* ≤ 40*)* $\wedge$ *(Sexe = 'M')* $\wedge$ *(Ville ="Grenoble"),*
- m_2 : *(Age* ≤ 40*)* $\wedge$ *(Sexe = 'M')* $\wedge$ *(Ville = "Paris"),*
- m_3 : *(Age* ≤ 40*)* $\wedge$ *(Sexe = 'F')* $\wedge$ *(Ville = "Grenoble"),*
- m_4 : *(Age* ≤ 40*)* $\wedge$ *(Sexe = 'F')* $\wedge$ *(Ville = "Paris"),*
- m_5 : *(Age* > 40*)* $\wedge$ *(Sexe = 'M')* $\wedge$ *(Ville ="Grenoble"),*
- m_6 : *(Age* > 40*)* $\wedge$ *(Sexe = 'M')* $\wedge$ *(Ville = "Paris"),*
- m_7 : *(Age* > 40*)* $\wedge$ *(Sexe = 'F')* $\wedge$ *(Ville = "Grenoble"),*
- m_8 : *(Age* > 40*)* $\wedge$ *(Sexe = 'F')* $\wedge$ *(Ville = "Paris"),*

Parmi les 2^6 *minterms possibles, nous avons éliminé ceux qui sont contradictoires, par exemple:* $p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5 \wedge p_6$*. Les minterms résultats sont appelés les* minterms primitifs.

Chaque minterm primitif m_i $(1 \leq i \leq 8)$ *définit un fragment possible. Si c'est le cas, nous aurons un schéma de fragmentation de huit fragments (ce qui représente le même schéma obtenu par l'algorithme dirigé par la complétude et la minimalité des prédicats [OV91]). Si nous combinons par exemple, deux minterms primitifs* $m_1 : (p_1 \wedge p_3 \wedge p_5)$ *et* $m_2 : (p_1 \wedge p_3 \wedge p_6)$ *par l'opérateur logique OU, nous obtenons le minterm suivant:*
$(p_1 \wedge p_3 \wedge p_5) \vee (p_1 \wedge p_3 \wedge p_6) = (p_1 \wedge p_3) \wedge (p_5 \vee p_6) = (p_1 \wedge p_3)$ *qui définit un autre fragment potentiel. Ce minterm avec les six autres minterms (*m_3*,* m_4*,* m_5*,* m_6*,* m_7*,* m_8*) forment un schéma de fragmentation de sept fragments.*

Notre objectif est de générer tous les schémas de fragmentation possibles d'une table donnée T. On peut s'interroger sur leur nombre. Nous donnons ci-après une approche pour leur dénombrement et donc aussi leur génération.

Soient N le nombre de prédicats simples définis dans les n requêtes de départ, Z $(Z \leq 2^N)$ le nombre de minterms primitifs générés à partir des N prédicats. D'après les propriétés 2 et 3, le nombre total de tous les schémas de fragmentation possibles obtenus à partir des Z minterms primitifs est donné par le théorème suivant:

Théorème 2 *Soit Z le nombre de minterms générés à partir des N prédicats. Alors le nombre L de tous les schémas possibles pour une fragmentation horizontale de la table T est donné par:*

$$L = \sum_{k=1}^{Z} L_Z^k \tag{6.1}$$

où : L_Z^k représente le nombre de tous les schémas de fragmentation définis à partir de Z minterms et ayant k fragments.

$$L_Z^k = k * L_{Z-1}^k + L_{Z-1}^{k-1} \tag{6.2}$$

Preuve 2 *On a $L_0^1 = 0$ et $L_1^1 = 1$. Les équations 6.1 et 6.2 sont vraies pour $Z = 1$ et $k = 1$ à condition de supposer que $L_0^0 = 1$. On remarque également que $L_Z^1 = 1$, $\forall Z \in \mathbb{N}$ et que $L_Z^k = 0$, $\forall Z \in \mathbb{N}$ et $k > Z$. Les équations 6.1 et 6.2 sont donc vraies pour $k = 1$ $\forall Z \in \mathbb{N}$.*

Nous allons démontrer ces équations par récurrence. Nous supposons que 6.1 est vraie pour $Z \leq j$ et 6.2 est vraie pour tout $k \leq j$. Nous allons montrer qu'elles sont vraies pour $Z = j + 1$.

Les schémas de fragmentation à $j + 1$ minterms et k partitions peuvent être obtenus de deux façons différentes à partir des schémas à j minterms:

- *Ajouter un minterm à chacun des L_j^k schémas de k partitions avec j minterms. Cela revient à ajouter un minterm à l'une des partitions de chacun de ces schémas (le nombre de partitions doit rester le même). Il y a $k \times L_j^k$ façons différentes de le faire.*
- *Ajouter une partition construite sur l'unique minterm supplémentaire à chacun des L_j^{k-1} schémas de $k - 1$ partitions avec j minterms. Il y a L_j^{k-1} façons différentes de le faire.*
  On a donc: $L_j^{k+1} = k * L_j^k + L_j^{k-1}$

Enfin les deux équations sont vraies pour $Z = j + 1$ et $k > j + 1$, ce qui achève la démonstration.

Exemple 29 *Soient m_1 et m_2 deux minterms, à partir desquels, deux schémas de fragmentation S_1 et S_2 peuvent être générés. Ils sont définis comme suit:*
$S_1 : (F_1 : m_1 \vee m_2)$, et $S_2 : \Big((F_1' : m_1), (F_2' : m_2) \Big)$, tels que: $L_2^1 = 1$ et $L_2^2 = 1$.

Maintenant notre but est de générer les schémas de fragmentation pour les trois minterms m_1, m_2 et m_3 à partir des schémas précédents.

D'après le théorème, nous devons calculer L_3^1, L_3^2, et L_3^3. L_3^1 représente des schémas avec un seul fragment. Il est obtenu à partir du schéma S_1, en incluant le minterm m_3 à S_1, en conséquence nous aurons S_1' : $(F_{11} : m_1 \vee m_2 \vee m_3)$. Le schéma de fragmentation avec trois fragments L_3^3 est généré en considérant le mimterm m_3 comme un seul fragment qui sera regroupé avec S_2, nous aurons donc $S_2' : \Big((F_{21} : m_1), (F_{22} : m_2), (F_{23} : m_3) \Big)$.

Les schémas ayant deux fragments (L_3^2) sont générés de la manière suivante: premièrement à partir de S_2, nous pouvons générer deux schémas, en incluant m_3 dans le premier fragment de S_2 (m_1) et nous aurons $S_3' : (F_{31} : m_1 \vee m_3), (F_{32} : m_2)$, ou en incluant m_3 dans la deuxième fragment de S_2, et nous aurons le schéma S_4' défini comme suit: $S_4' : \Big((F_{41} : m_1), (F_{42} : m_2 \vee m_3) \Big)$.
Cela correspond au premier terme de l'équation 6.2. Le dernier schéma est généré en considérant le minterm m_3 comme un fragment que nous regroupons avec S_1, nous aurons donc $S_5' : \Big((F_{51} : m_1 \vee m_2), (F_{52} : m_3) \Big)$ *qui correspond au second terme de la partie droite l'équation 6.2.*
Finalement, le nombre de schémas construits à partir de trois minterms est 5.

À ce stade, tous les schémas de fragmentation possibles d'une table donnée peuvent être générés. Maintenant, il nous faut évaluer le meilleur schéma de fragmentation. Cette évaluation est assurée par le modèle de coût.

6.3.4 Le modèle de coût

Le modèle de coût décrit dans le chapitre précédent pour un entrepôt fragmenté peut être complétement réutilisé ici.

Nous avons maintenant tous les ingrédients pour décrire deux algorithmes dirigés par un modèle de coût, à savoir: (i) **l'algorithme exhaustif** et (ii) **l'algorithme approximatif**.

6.3.5 L'algorithme exhaustif

6.3.5.1 Description de l'algorithme

Soit D_i une table de dimension du schéma en étoile SED à fragmenter. Rappelons que sur cette table, nous considérons un ensemble de requêtes ($Q = \{Q_1, Q_2, ..., Q_n\}$), où chaque requête Q_j possède une fréquence d'accès fr_j.

Cet algorithme consiste **à énumérer** d'une manière exhaustive tous les schémas de fragmentation possibles de D_i (voir section 6.3.3). À l'aide du modèle de coût, un coût d'exécution pour l'ensemble de requêtes Q est attribué à chaque schéma de fragmentation S_i ($1 \leq i \leq L$). Ce coût est dénoté par $CT(S_i)$. Finalement, le schéma de fragmentation offrant le coût minimum est sélectionné (voir l'algorithme de la Figure 11). Ce schéma est noté par S_{min}, et satisfait la condition suivante:

$$CT(S_{min}) = min\{CT(S_i) \quad | \quad S_i \in S_{D_i}\}$$

6.3.5.2 La correction de l'algorithme exhaustif

Etant donné que les prédicats de sélection utilisés dans cet algorithme sont complets, les fragments résultants le sont également [OV91]. La reconstruction est réalisée par l'opération d'union. Finalement, la disjonction est assurée par le fait que les prédicats sont mutuellement exclusifs.

Algorithme 11 La description de l'algorithme exhaustif

```
1: coût_min:= +∞ {ce coût représente le coût de schéma optimal}
2: for i := 1 to L {L: le nombre de schémas de fragmentation possibles} do
3:    CT(S_i): = 0 ;
4: end for
5: for i:= 1 to L do
6:    for j := 1 to n {cette boucle parcours toutes les requêtes} do
7:       CT(S_i) := CT(S_i) + CT(Q_j) * fr_j;
8:    end for
9:    if CT(S_i) < coût_min then
10:      coût_min:= CT(S_i) ;
11:      min := i ;
12:   end if
13: end for
14: Schéma_optimal:= S_min {le schéma ayant le rang égal à "min"}
```

6.3.5.3 Avantages et inconvénients de l'algorithme exhaustif

De par sa nature exhaustive, cet algorithme fournit un schéma de fragmentation optimal (optimalité relative au modèle de coût).

La complexité de cet algorithme est tirée de l'étape 5 de l'algorithme 11 qui consiste à parcourir tous les schémas de fragmentation et à calculer pour chaque schéma le coût d'exécution de l'ensemble des requêtes. En conséquence la complexité de cet algorithme est $O(n \times L)$, où n et L représentent le nombre de requêtes et le nombre de tous les schémas de fragmentation possibles, respectivement.

Le nombre de schémas de fragmentation L est très élevé. Pour un petit nombre de minterms, ce nombre est raisonnable, mais pour un grand nombre de minterms, il devient très important. Par exemple, pour 15 minterms, le nombre de schémas de fragmentation est 1 382 958 545 (voir la Table 6.2). Notons que 15 minterms peuvent être facilement obtenus à partir de 4 prédicats simples, ce qui est peu pour une application réelle. Dans les entrepôts de données, les requêtes décisionnelles possèdent beaucoup de prédicats de sélection [DRT99]; cet algorithme n'est donc pas praticable dans les cas réels.

Dans la section suivante, nous présentons un autre algorithme appelé approximatif qui surmonte les difficultés de ce premier algorithme.

Minterms	Nombre de schémas	Borne inférieure des schémas	Borne supérieure des schémas
1	1	0.5	1
2	2	1.167	2
3	5	4	12
4	15	11.336	32
5	52	49.5	607.5
6	203	148.5	2187
7	877	586.7	57344
8	4140	2347.01	262144
9	21147	8646.88	8.78906e+06
10	115975	43234.4	4.88281e+07
.	.	.	.
.	.	.	.
.	.	.	.
15	13682958545	7.09592e+07	2.63883e+14

TAB. 6.2: La complexité du nombre de schémas

6.4 L'algorithme approximatif

6.4.1 Les étapes de l'algorithme

Pour éviter l'énumération exhaustive de tous les schémas possibles de fragmentation d'une table T, nous développons un algorithme, que nous appelons *approximatif*. Approximatif signifie que la solution obtenue par l'heuristique n'est pas forcément optimale, mais plutôt quasi-optimale.

Nous rappelons que si le nombre de minterms générés par les prédicats simples des requêtes est faible, alors l'algorithme exhaustif est utilisé, sinon, l'algorithme approximatif est utilisé. Cet algorithme est basé sur la technique de l'escalade (hill-climbing dans l'appellation anglo-saxonne) et un modèle de coût. Pour amorcer ce type d'algorithme, il faut une solution initiale. Ensuite l'algorithme effectue des changements graduels sur cette solution pour se rapprocher d'un *but*. La solution initiale doit avoir une complexité raisonnable. Le but est déterminé par une fonction objectif que nous désirons maximiser ou minimiser.

Toute technique de hill-climbing nécessite trois éléments principaux [Jai91]: (1) *une solution initiale*, (2) *un mécanisme d'amélioration*, et (3) *un test d'arrêt*. Plus précisément un algorithme de type hill-climbing se déroule selon le schéma suivant [Jai91]:

1. Générer une solution initiale
2. Calculer le bénéfice de la solution initiale
3. Modifier la solution initiale
4. Calculer le bénéfice de la nouvelle solution
5. Si le bénéfice de la nouvelle solution est supérieur alors en faire la solution initiale et reprendre l'étape 2;

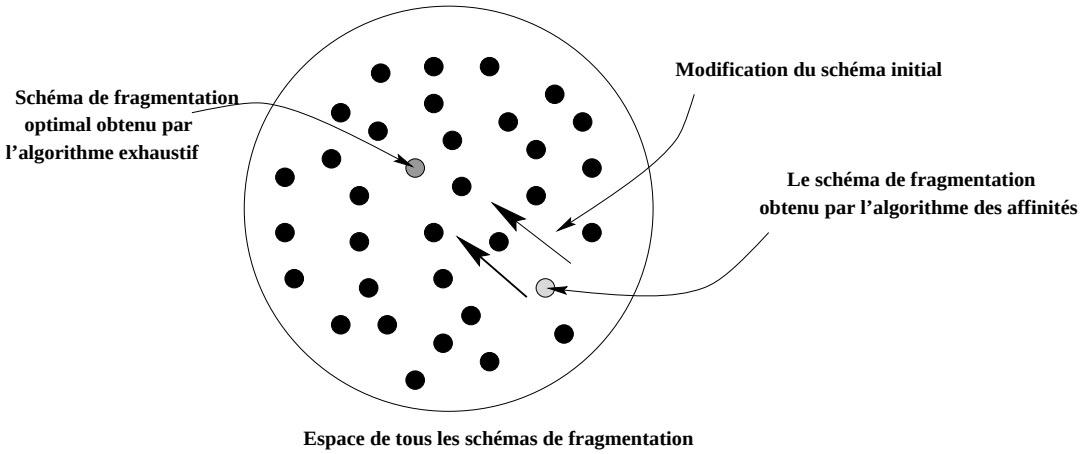


FIG. 6.10: Le mécanisme d'amélioration du schéma initial

6. Sinon, arrêt.

Dans les sections suivantes, nous allons expliciter chacune de ces étapes.

6.4.2 La détermination de la solution initiale

Le choix de la solution initiale contribue fortement à la qualité de la solution finale obtenue par l'algorithme approximatif. Dans notre cas, la solution intiale est le résultat de l'exécution de l'algorithme basé sur les affinités présenté dans la section 6.2. Nous optons pour ce choix car la complexité de cet algorithme est polynômiale. Soit S_{affi} le schéma de fragmentation obtenu par cet algorithme. Notons que les fragments de S_{affi} sont disjoints. Le schéma S_{affi} est un élément de l'ensemble de tous les schémas de fragmentation possibles de la table T (voir Figure 6.10). Le schéma S_{affi} pourrait être la solution optimale obtenue par l'algorithme exhaustif, mais cela est peu probable. L'objectif de l'algorithme approximatif est de rapprocher le schéma S_{affi} autant que possible du schéma optimal. Ce rapprochement est décrit dans la section suivante.

6.4.3 Un mécanisme d'amélioration de la solution initiale

La phase d'amélioration prend la solution initiale et essaye de la rapprocher le plus possible du schéma optimal (voir Figure 6.10) en effectuant des opérations sur la solution initiale. Dans notre cas, deux opérations sont définies sur les fragments du schéma initial S_{affi} qui sont: (1) *assembler deux fragments*, et (2) *éclater un fragment*.

La première opération consiste à fusionner deux fragments en un seul. Elle ressemble à l'opération migration de tous les n-uplets d'un fragment vers l'autre lors des mises à jour. Ce genre d'opérations a été largement utilisé dans le partitionnement des graphes [KL70] et les algorithmes d'allocation des fragments dans un système distribué [BKL98c, KP97].

La deuxième opération est l'inverse de la première: elle éclate un fragment en plusieurs. Cette dernière augmente le nombre de fragments et en conséquence diminue

la taille de chaque fragment.

L'intuition qui est derrière ces deux opérations provient de la propriété 3, c'est à dire, la combinaison de deux ou plusieurs minterms représentant un fragment.

Nous connaissons le principe de base de ces deux opérations, mais la question qui reste posée est de savoir *quels fragments sont des candidats pour ces opérations?* Pour répondre à cette question, nous devons trouver une technique de représentation des fragments. Cette technique fera l'objet de la section suivante.

6.4.3.1 Une représentation des fragments horizontaux

Les prédicats simples que les algorithmes de fragmentation utilisaient jusqu'à présent sont de la forme suivante [CP84, OV91, BKB98, BKA00, BKL98b]:

$$\text{Attribut } \theta \text{ Valeur, avec } \theta \in \{=, <, \leq, >, \geq, \neq\} \quad .$$

Nous considérons un autre type de prédicats appelés *prédicats de domaine* utilisés dans le système OSIRIS [SS94, SS89].

Définition 34 *Un prédicat sur le domaine d'un attribut attr ou* prédicat de domaine*, est un prédicat de la forme :*

$$P(attr) : attr \in Domaine \quad ,$$

où attr et Domaine représentent l'attribut sur lequel le prédicat est défini et un intervalle ou un ensemble de valeurs énumérées, respectivement.

Remarque : Les *prédicats simples* que les algorithmes de fragmentation utilisent se présentent comme un cas particulier de prédicats de domaine. Le prédicat Sexe = 'M' peut aussi s'écrire en un prédicat de domaine : Sexe $\in \{'M'\}$.

Définition 35 *Une partition d'un ensemble non vide E est une collection de sous-ensembles disjoints non vides de A tel que leur union est l'ensemble A lui-même [SM77]. Un élément d'une partition est appelé un* bloc.

L'analyse des clauses représentant les fragments horizontaux permet d'effectuer une partition du domaine des attributs de fragmentation en sous-domaines appelés sous-domaines stables [SS94]. Les éléments de cette partition sont déterminés par les prédicats de domaine. Ils sont désignés par cette appellation, compte tenu de la "stabilité" de la valeur de vérité de tous les prédicats de domaine d'un attribut lorsque cet attribut change de valeur en restant dans le même bloc de la partition.

Exemple 30 *Pour montrer comment les prédicats de fragmentation définissent des partitions de chaque domaine des attributs de fragmentation, reprenons les fragments horizontaux de la table de dimension CLIENT générés par l'algorithme des affinités. Supposons que les domaines des attributs de fragmentation (qui sont l'Age, le Sexe, et Ville) sont:*

- $Dom(Age) =]0, 120]$
- $Dom(Sexe) = \{`M', `F'\}$,
- $Dom(Ville) = \{\text{“}Grenoble\text{”}, \text{“}Paris\text{”}\}$.

Les partitions des trois attributs de fragmentation engendrés par la clause de chaque fragment sont obtenues comme suit:

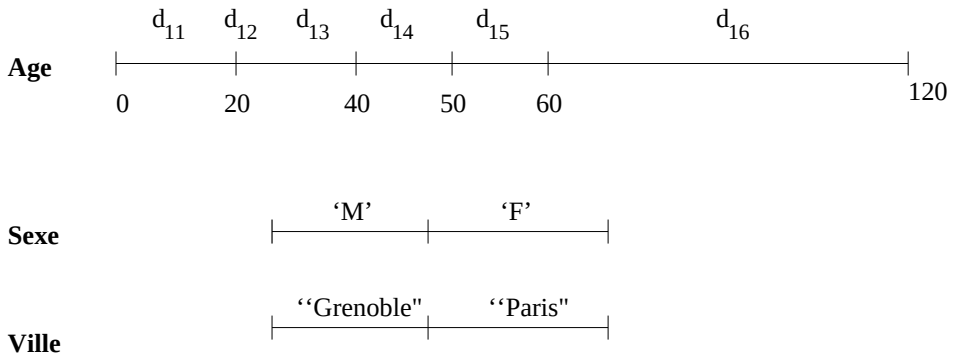


FIG. 6.11: Les SDSs des attributs de fragmentation

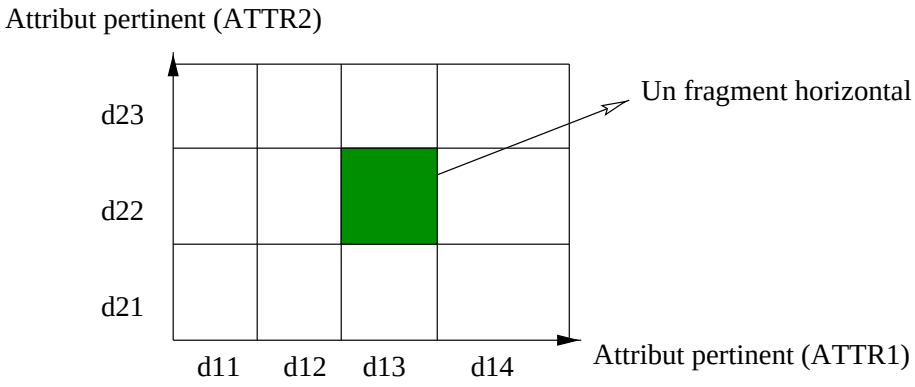


FIG. 6.12: Un exemple de représentation de fragments dans le cas de deux attributs

- *Sur l'attribut Age, trois prédicats simples sont définis:* p_1 : $Age \leq 40$, p_2 : $Age = 20$, *et* p_3 : $50 \leq Age \leq 60$. *Ces prédicats peuvent facilement être transformés en prédicats de domaine:* $p_1 : Age \in]0, 40]$, $p_2 : Age \in [20, 20]$, *et* $p_3 : Age \in [50, 60]$.
  Le domaine de définition de l'attribut Age ($]0, , 120]$) *est donc partitionné en six SDSs par les trois prédicats* (p_1, p_2, *et* p_3) *de la manière suivante:*
  $Dom(Age) = d_{11} \cup d_{12} \cup d_{13} \cup d_{14} \cup d_{15} \cup d_{16}$, *avec* $d_{11} =]0, 20[$, $d_{12} = [20, 20]$, $d_{13} =]20, 40[$, $d_{14} = [40, 50[$, $d_{15} = [50, 60]$, $d_{16} =]60, 120]$,
- *D'une manière similaire, le domaine de l'attribut Sexe est partitionné en deux SDSs:* $Dom(Sexe) = d_{21} \cup d_{22}$, *avec* $d_{21} = \{`M'\}$, $d_{22} = \{`F'\}$
- *Finalement, l'attribut Ville est partitionné en deux SDSs:* $Dom(Ville) = d_{31} \cup d_{32}$, *avec* $d_{31} = \{$*"Grenoble"*$\}$ *et* $d_{32} = \{$*"Paris"*$\}$.

Les différents SDSs de chaque attribut de fragmentation sont représentés par la Figure 6.11.

Maintenant, nous allons voir l'utilité des SDSs dans la représentation des fragments horizontaux.
Soit $T(K, A_1, A_2, ..., A_l)$ une table de l attributs dont r sont des attributs de fragmentation ($r \leq l$). Ces derniers forment un n-uplet $(attr_1, attr_2, ..., attr_r)$, où chaque $attr_i$ possède n_i SDSs dénoté par le n-uplet $(d_{i1}, d_{i2}, ..., d_{in_i})$.

Chaque fragment horizontal est valide dans un ou plusieurs SDSs d'un attribut de

fragmentation. Les SDSs valides et invalides sont marqués par 1 et 0, respectivement. En conséquence chaque fragment F_j peut être représenté par un vecteur (que nous appelons *vecteur de fragmentation*) de la manière suivante:

$$F_i : \Big((d_{11}, d_{12}, ..., d_{in_1}); (d_{21}, d_{22}, ..., d_{2n_2}); (d_{N1}, d_{N2}, ..., d_{N_{n_N}}) \Big).$$

Les petites parenthèses représentent les SDSs de chaque attribut de fragmentation (A_k), que nous appelons *éléments du sous-vecteur F_{jk} de F_j*. Le point virgule (;) est utilisé pour séparer les éléments. Cette représentation ressemble à un index binaire.

Exemple 31 *Les cinq fragments de la table de dimension CLIENT obtenus par l'algorithme des affinités (voir section 6.2) sont représentés de la façon suivante:*

$$\textit{Client\_1:} \Big((1,1,1,0,0,0); (1,0); (1,0) \Big)$$

$$\textit{Client\_2:} \Big((0,1,0,0,0,0); (0,1); (1,0) \Big)$$

$$\textit{Client\_3:} \Big((0,0,0,0,1,0); (1,0); (1,0) \Big)$$

$$\textit{Client\_4:} \Big((0,0,0,0,1,0); (0,1); (1,0) \Big)$$

$$\textit{Client\_5:} \Big((1,0,0,0,0,1); (1,0); (0,1) \Big)$$

Chaque fragment est représenté par un vecteur de 3 éléments (chacun correspondant à un attribut de fragmentation). Le premier correspond à l'attribut Age, le deuxième à l'attribut Sexe et le dernier à l'attribut Ville. Nous définissons la cardinalité de chaque élément par le nombre de SDSs correspondant à l'élément.

Graphiquement, les fragments peuvent être représentés par un hypercube dont les dimensions correspondent aux attributs de fragmentation. Considérons par exemple, deux attributs de fragmentation $attr_1$ et $attr_2$ possédant 4 et 3 SDSs, respectivement. Les fragments engendrés par ces SDSs sont représentés dans un espace bidimensionnel (plan) comme le montre la Figure 6.12. Si nous avons trois attributs de fragmentation $attr_1$, $attr_2$ et $attr_3$, les fragments sont représentés dans un espace tridimensionnel (cube) comme le montre la Figure 6.13.

Définition 36 *La distance de Hamming $dist(X, Y)$ entre deux vecteurs binaires: $X = (x_1, x_2, ..., x_k)$ et $Y = (y_1, y_2, ..., y_k)$ représente le nombre de positions dans lesquelles X et Y sont différents [PH89].*

Étant donné que les fragments horizontaux sont représentés par des vecteurs binaires, la distance de Hamming entre fragments peut être facilement calculée. Par exemple, la distance de Hamming entre le fragment Client_1 et le fragment Client_2 est égale à 4. Elle est notée par:

$$\text{dist(Client\_1, Client\_2)} = 4$$

Dans la section suivante nous montrons l'utilité de cette distance dans l'application des deux opérations assembler et éclater.

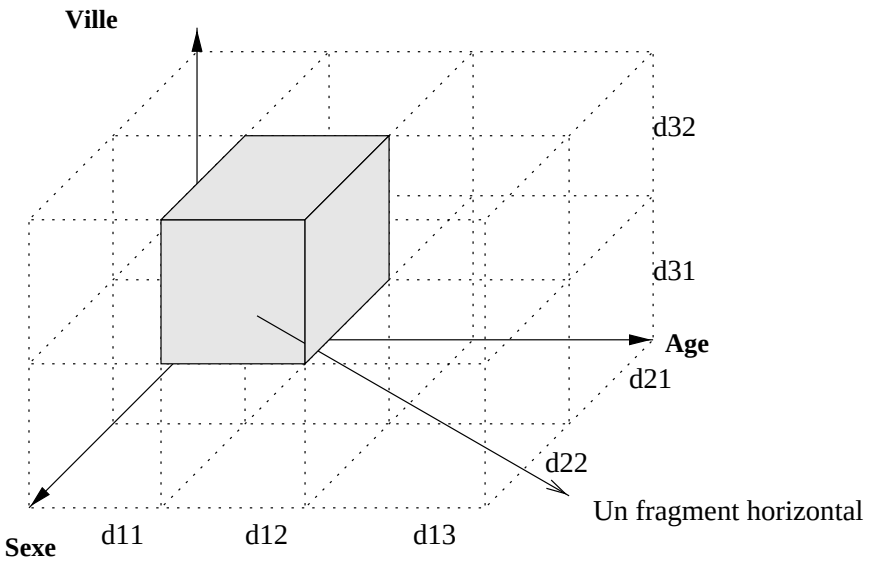


FIG. 6.13: Un exemple de représentation de fragments dans le cas de trois attributs

Assembler: Soit F_i et F_j deux fragments horizontaux définis par les clauses cl_i et cl_j. L'opération *assembler* deux fragments est appliquée si les conditions suivantes sont satisfaites:

1. Les clauses des deux fragments cl_i et cl_j diffèrent au niveau d'un seul prédicat (p_k). Pour capturer cette différence, nous utilisons la distance de Hamming entre les deux vecteurs de fragmentation représentant les deux fragments. Cette opération est appliquée, si la distance de Hamming entre les couples d'éléments correspondants de chaque vecteur est égale à 0 sauf pour un couple.
2. Les deux fragments F_i et F_j sont accèdés *simultanément* par des requêtes n'ayant pas le prédicat p_k.

L'intérêt de cette opération est le suivant: Si les deux fragments sont accédés simultanément par un ensemble de requêtes, nous aurons intérêt à les regrouper pour éviter d'avoir à faire ensuite l'union des résultats correspondants aux différents fragments.

Exemple 32 *Prenons les fragments Client_3 et Client_4 générés par l'algorithme des affinités et représentés comme suit:*

$$\textit{Client\_3:} \left((0,0,0,0,1,0);(1,0);(1,0) \right) \textit{ et}$$

$$\textit{Client\_4:} \left((0,0,0,0,1,0);(0,1);(1,0) \right).$$

La distance de Hamming pour chacun des couples d"eléments de leurs vecteurs de fragmentation est égale à 0, à l'exception des éléments situés en 2ème position. Ces deux fragments peuvent être donc assemblés en un seul fragment représenté par le vecteur suivant: $\left((0,0,0,0,1,0);(1,0) \right)$.

Le premier et le deuxième élément de ce vecteur représentent les attributs Age et Ville, respectivement. L'attribut Sexe a été éliminé.

Soit m le nombre de fragments horizontaux, le nombre maximum de toutes les opérations *assembler* est $\frac{m \times (m-1)}{2}$.

Eclater: Soit F un fragment horizontal défini par une clause ayant f prédicats simples, et Q une requête accédant à la table T et ayant g prédicats de sélection ($g > f$). Le fragment F_j peut être éclaté en incluant les prédicats qui ne sont pas définis dans la clause de F mais définis dans celle de la requête. Le nombre de fragments générés par cette opération dépend des SDSs de chaque attribut défini dans la requête et non dans le fragment.

Cette opération ressemble à l'étape 6 de l'algorithme des affinités dans la section 6.2. Cette opération augmente le nombre de fragments et diminue la taille de chacun.

Exemple 33 *Dans la section 6.2, nous avons introduit trois attributs de fragmentation sur la table CLIENT: Age, Sexe et Ville. Soit Client_1 un fragment défini par la clause de deux prédicats:* $(Age \leq 40) \wedge (Ville = \text{“}Paris\text{”})$.
Soit Q une requête ayant les trois prédicats de sélection suivants: $(Age \leq 30) \wedge (Sexe = \text{‘}M\text{’}) \wedge (Ville = \text{“}Paris\text{”})$.

Le seul attribut figurant dans la clause de la requête et pas dans la clause du fragment est Sexe. Notons que le domaine de ce dernier possède deux valeurs “M” et “F”. En conséquence, le fragment Client_1 peut être éclaté en deux fragments horizontaux: Client_2 et Client_3 ayant les clauses suivantes:
Client_2 ayant la clause $(Age \leq 40) \wedge (Sexe = \text{‘}M\text{’}) \wedge (Ville = \text{“}Paris\text{”})$.
Client_3 ayant la clause $(Age \leq 40) \wedge (Sexe = \text{‘}F\text{’}) \wedge (Ville = \text{“}Paris\text{”})$.
Dans cet exemple, nous avons éclaté le fragment en deux, car l'attribut Sexe possède deux SDSs.

Soit m le nombre de fragments horizontaux, le nombre maximum de toutes les opérations *éclater* est $\frac{m \times (m-1)}{2}$.

6.4.3.2 Identification des fragments nécessaires à une requête

Dans cette section, nous montrons l'intérêt de la distance de Hamming dans le processus d'identification des fragments nécessaires à l'exécution d'une requête. Notons qu'une requête typique, projette un ensemble d'attributs d'une table et sélectionne les n-uplets de cette table en se basant sur quelques conditions sur ses attributs de la table définis par des prédicats. Dans le cas où la requête accède à plusieurs tables (en cas de jointure ou union, etc), ces opérations sont exécutées sur les n-uplets des relations satisfaisants les prédicats de sélection. En conclusion, l'opération de jointure n'a aucun effet sur la fragmentation horizontale [Kar96].

Soit Q_u une requête définie sur une table fragmentée T ayant l attributs de fragmentation $A_1, ..., A_l$. Q_u peut être représenté par un n-uplet $(p_{u1}, p_{u2}, ..., p_{ul})$ [AGEA93], où p_{ui} est le prédicat de sélection défini sur l'attribut de fragmentation A_i (la valeur “*” sera utilisée si Q n'utilise pas A_i). Etant donné que le domaine de chaque attribut de fragmentation A_k est partitionné en n_k SDSs, la requête Q_u peut être représentée par un vecteur (vecteur de la requête) comme les fragments. Si la requête Q_u ne contient aucun prédicat de sélection, elle sera représentée par le vecteur $Q_u : (*, *, ..., *)$.

Après la création de fragments horizontaux, les requêtes définies sur le schéma global (non partitionné) doivent être transformées sur le schéma partitionné. Pour ce faire, nous devons identifier les fragments nécessaires pour chaque requête. Étant

donné que la requête peut avoir des prédicats de sélection définis sur des attributs qui ne participent pas au processus de la fragmentation, ces derniers ne seront pas pris en considération. En conséquence, la requête est représentée seulement par des attributs de fragmentation. Le vecteur de requête doit être représenté dans le même ordre que le vecteur de fragmentation.

Exemple 34 *Soit Q une requête définie sur le schéma en étoile (voir Figure 2.5) dont la description est la suivante:*

```
SELECT   SUM(V.ventes_dollar)
FROM     CLIENT, VENTES
WHERE    CLIENT.Cid = VENTES.Cid
AND      CLIENT.Sexe ='M'
AND      Région = ''Rhône''
GROUP BY PID
```

Cette requête contient deux prédicats de sélection (C.Sexe ='M' et Région = "Rhône). Parmi ces prédicats, un seul est un prédicat de fragmentation (celui défini sur l'attribut Sexe). Le prédicat défini sur l'attribut Région est donc écarté. En conséquence, le vecteur de la requête Q est $Q : \left(*; (1,0); * \right)$.

La distance de Hamming peut être également utilisée pour définir une distance $dist(Q_u, F_j)$ entre les vecteurs d'un fragment F_j et celui d'une requête Q_u (par le fait qu'ils ont la même représentation).

Propriété 4 Contribution d'un fragment à une requête
Soit n_i le nombre de SDSs de l'attribut A_i

- *Si $dist(Q_u, F_j) = 0$ alors tous les n-uplets de F_j participent à la requête Q_u.*
- *S'il existe un Q_{ui} et F_{ji} tel que: $dist(Q_{ui}, F_{ji}) = n_i$ alors aucun n-uplet de F_j ne participe à la requête Q_u.*
- *Si $0 < dist(Q_{ui}, F_{ji}) < n_i$ pour tout Q_{ui}, et F_{ji}, alors certains n-uplets de F_j participent à la requête Q_u.*

À partir de la représentation binaire des requêtes et des fragments, nous pouvons facilement identifier le nom des fragments accédés par une requête donnée. Une fois ces fragments identifiés, nous définissons une nouvelle matrice que nous appelons requête-fragment, dans laquelle, les lignes représentent les requêtes et les colonnes les fragments. Si un fragment est accèdé par une requête, la valeur correspondante de cette matrice est égale à 1, sinon elle est égale à 0. La fréquence d'accès de chaque requête à un fragment peut être ajoutée dans cette matrice. En fonction de cette matrice et de la représentation binaire des fragments, les deux opérations peuvent être facilement mises en oeuvre.

Dans l'algorithme approximatif, nous commençons par appliquer l'opération éclater et ensuite l'opération assembler. Ce choix est dicté par la remarque suivante: l'opération éclater génère plusieurs fragments horizontaux (de petite taille), et il est fort possible que les requêtes accèdent seulement à une partie de ces fragments, d'où la réduction de coût d'exécution de ces requêtes. Mais il faut noter que ce choix

est contre-balancé par la nécessité de faire des opérations d'union pour former les résultats des requêtes.

Comme chaque itération de cet algorithme ne provoque qu'un changement local du schéma initial, l'algorithme converge vers un minimum local.

6.4.4 Le test d'arrêt

L'algorithme approximatif s'achève lorsqu'il n'est plus possible d'appliquer l'une des deux opérations ou lorsque aucune réduction dans le coût d'évaluation des requêtes n'est constatée.

6.4.5 Description de l'algorithme

Maintenant, nous possédons tous les éléments pour décrire l'algorithme approximatif. Il débute par un schéma initial obtenu par l'algorithme des affinités décrit dans la section 6.2. Les fragments de ce schéma sont représentés sous la forme binaire. À l'aide du modèle de coût, le coût d'exécution des requêtes sur ce schéma ($CT_{initial}$) est calculé.

Ensuite, l'algorithme cherche une possibilité d'appliquer l'opération éclater au schéma initial. Si c'est le cas, nous obtenons un nouveau schéma de fragmentation, sur lequel, nous calculons le coût d'exécution des requêtes ($CT(courant)$). Si ce dernier est inférieur à celui obtenu pour le schéma initial ($CT_{initial}$), nous remettons à jour le schéma initial obtenu après l'opération éclater et le coût initial sera le coût courant. Sinon, l'algorithme essaye de voir s'il y a une autre possibilité d'appliquer l'opération éclater. Cette procédure est répétée tant qu'une réduction du coût est constatée.

Ensuite, nous passons à l'opération assembler. Elle prend le schéma courant obtenu par l'application de l'opération éclater. Sa tâche est similaire à celle de la première opération.

Nous itérons ces deux opérations tant qu'une réduction dans le coût est constatée (voir l'algorithme 12).

6.5 Evaluation des trois algorithmes

Nous avons implémenté les trois algorithmes (exhaustif, approximatif et basé sur les affinités) en C sur une plate-forme Unix afin de comparer leurs performances en termes de coût des entrées-sorties nécessaires pour exécuter un ensemble de requêtes. Ces algorithmes ont été testés sur un schéma d'une base de données orientée objet [BKB98]. Nous nous sommes intéressés à la fragmentation d'une seule classe de ce schéma (la classe Projet). Le modèle de coût utilisé par les algorithmes exhaustif et approximatif est différent de celui utilisé dans ce mémoire, mais il calcule le coût des entrées-sorties lors de l'exécution d'un ensemble de requêtes. La seule différence est que ce modèle ne prend pas en considération des index. Ce modèle est décrit dans [BKS98, BKL98b].

Algorithme 12 L'algorithme approximatif

1: **Entrées:** (1) la table T à fragmenter, et (2) un ensemble de requêtes (avec leur fréquences d'accès)
2: **Sortie:** Un schéma de fragmentation
3: K: représente le nombre maximum de l'opération éclater
4: $i := 1$
5: schéma_initial: = $algo\_affi\_horiz$
6: $CT_{initial}$:= coût(schéma_initial)
7: **while** $(i \leq K)$ **do**
8: nouveau_schéma: = éclater(schéma_initial)
9: $CT_{courant}$:= coût(nouveau_schéma)
10: **if** $(CT_{courant} < CT_{initial})$ **then**
11: schéma_initial : = nouveau_schéma
12: $CT_{initial} := CT_{courant}$
13: $i := i + 1$
14: **end if**
15: **end while**
16: NN:= Le nombre de fragments horizontaux résultant des opérations assembler
17: $j := 1$
18: **while** $(j \leq NN)$ **do**
19: nouveau_schéma: = assembler(schéma_initial)
20: $CT_{courant}$:= coût(nouveau_schéma)
21: **if** $(CT_{courant} < CT_{initial})$ **then**
22: schéma_initial : = nouveau_schéma
23: $CT_{initial} := CT_{courant}$
24: $j := j + 1$
25: **end if**
26: **end while**

Étant donné que le nombre de schémas de fragmentation générés par l'algorithme exhaustif est très grand (Table 6.2), nous avons effectué deux expérimentations. La première considère un nombre de minterms raisonnable afin de tester ensemble les trois algorithmes, et la deuxième compare les performances des deux algorithmes approchés, c'est-à-dire l'algorithme approximatif et l'algorithme basé sur les affinités.

6.5.1 La première expérimentation

Nous avons considéré cinq prédicats de sélection définis sur une classe Projet. À partir de ces prédicats, nous identifions 8 minterms. Premièrement, l'algorithme exhaustif est exécuté. Il génère à partir des minterms obtenus **4140** schémas de fragmentation de la classe Projet. Chaque schéma est associé à un coût d'exécution d'un ensemble de requêtes (voir Figure 6.14). Par conséquent, la sélection du meilleur schéma peut être facilement effectuée. Le schéma minimal a un coût de 2190 IOs.

Deuxièment, l'algorithme des affinités est exécuté. Il a comme entrée les cinq prédicats de sélection et l'ensemble des requêtes avec leur fréquence d'accès. Le coût d'exécution de l'ensemble des requêtes sous le schéma sélectionné par cet algorithme est de 2830 IOs (Figure 6.14).

Finalement, nous exécutons l'algorithme approximatif, qui prend le schéma obtenu par l'algorithme des affinités et essaye de l'améliorer en appliquant les deux opération *éclater* et *assembler*. Le coût d'exécution de requêtes correspondant au schéma sélectionné est de 2250 IOs.

À partir de cette expérimentation, nous voyons clairement l'avantage d'utiliser l'algorithme approximatif, qui fournit une solution ayant un coût presque identique à celui de la solution obtenue par l'algorithme des affinités. La solution de ce dernier est 1,3 fois moins performante que la solution optimale fournie par l'algorithme exhaustif.

6.5.2 La deuxième expérimentation

Nous avons considéré un nombre important de prédicats, et en conséquence nous avons comparé seulement les deux algorithmes approchés.

Les résultats ont montré que l'algorithme approximatif donne des schémas de fragmentation présentant un coût d'exécution des requêtes nettement supérieur à celui obtenu par les schémas obtenus par l'algorithme des affinités. La non prise en considération des paramètres physiques utilisés par le modèle de coût de l'algorithme des affinités explique cette situation.

6.5.3 Discussion

Les résultats obtenus peuvent être analysés en fonction de deux critères: *le temps de calcul* et *le gain du coût d'entrées-sorties des requêtes*.

- Le temps de calcul: L'algorithme exhaustif nécessite un coût de calcul très élevé pour un grand nombre de minterms. L'algorithme basé sur les affinités est très efficace en termes de temps de calcul. Finalement, le coût de calcul de

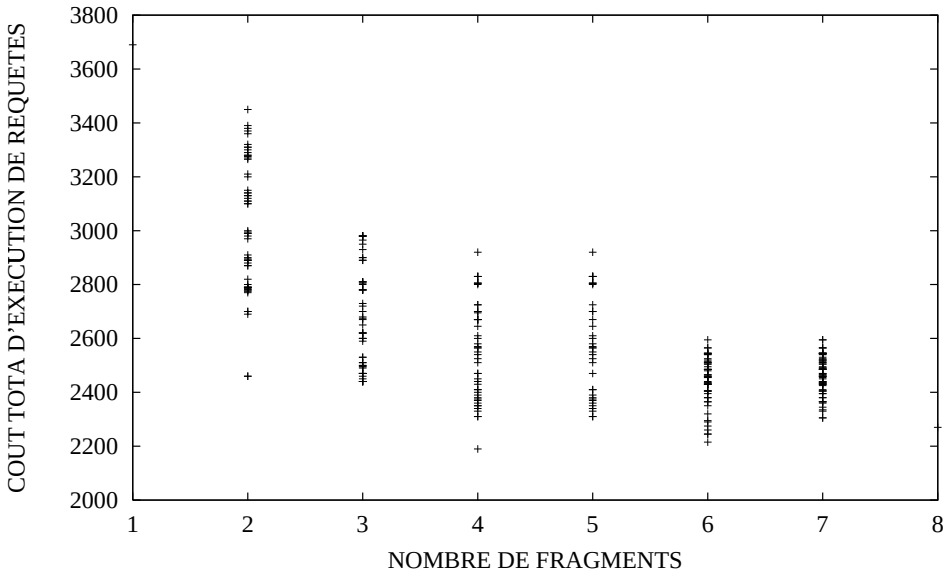


FIG. 6.14: Nombre de fragments vs. coût total d'exécution de requêtes

l'algorithme approximatif se situe celui de l'algorithmes exhaustif et celui des affinités. La performance de cet algorithme dépend du nombre d'opérations "éclater" et "assembler" réalisées au cours de son exécution. Si la solution initiale est proche de l'optimum, l'algorithme approximatif peut sélectionner le schéma de fragmentation optimal dans un délai bref.

- Le gain du coût des entrées-sorties: Nous avons conduit une expérience dans laquelle nous calculons le coût normalisé d'exécution d'un ensemble de requêtes sur une classe (1) non fragmentée, (2) fragmentée en utilisant l'algorithme exhaustif, (3) fragmentée en utilisant l'algorithme approximatif, et (4) fragmentée en utilisant l'algorithme des affinités.

  Les résultats ont montré que le coût de la solution obtenue par l'algorithme approximatif est presque identique à celui de la solution obtenue par l'algorithme exhaustif. Les deux algorithmes donnent un schéma de fragmentation ayant un coût supérieur à celui du schéma non fragmenté [BKL98b]. Le coût de la solution fournie par l'algorithme des affinités est supérieur à ceux des solutions des deux autres algorithmes, mais il est inférieur à celui du schéma non fragmenté [BKS98].

  Pour conclure, nous pouvons dire que l'utilisation d'un bon algorithme de fragmentation réduit considérablement le coût d'exécution de requêtes.

Chapitre 7

Conclusion et Perspectives

7.1 Conclusion

Cette thèse a concerné diverses techniques pour l'amélioration des performances des bases de données et des entrepôts de données. Dans cette section nous résumons les principaux apports de nos travaux.

7.1.1 Les index de jointure en présence des vues matérialisées

Le problème de la sélection d'index est sans doute le plus important dans la conception physique d'un entrepôt de données. Nous avons introduit les index de jointure qui peuvent être appliqués aux tables sources et aux vues matérialisées, ensemble ou séparément. Une stratégie d'exécution des requêtes en présence de ce type d'index a été décrite. Nous avons aussi proposé un modèle de coût pour évaluer le nombre d'entrées sorties nécessaires à l'exécution d'un ensemble de requêtes. Le problème de sélection des index de jointure a alors été formulé comme un problème d'optimisation minimisant cette fonction de coût en présence d'une contrainte d'espace. La complexité de ce problème est exponentielle. Nous avons suggéré trois algorithmes pour sélectionner des index de jointure optimaux ou quasi optimaux. Ces algorithmes consistent à réaliser un parcours du graphe de jointure de la requête. Les résultats de nos expérimentations ont montré qu'un index de jointure entre plusieurs tables peut contribuer significativement à la réduction du temps d'exécution d'un ensemble de requêtes. De plus la présence de vues appropriées peut renforcer l'efficacité d'un tel index.

7.1.2 Interaction entre les problèmes de sélection des vues et de sélection d'index

Les index et les vues matérialisées se partagent une même ressource : l'espace disque. Le problème de sélectionner simultanément, et non pas indépendamment, les vues et les index se pose donc. Nous avons suggéré une approche modulaire pour résoudre ce problème et avons proposé deux algorithmes approchés pour dis-

tribuer l'espace entre les vues et les index dans les cas statique et dynamique. Ces algorithmes sont basés sur une approche itérative. Nos expérimentations sur un cas réaliste ont montré que la prise en compte de l'interaction entre les vues et les index est effectivement indispensable pour atteindre de bons niveaux de performance.

7.1.3 La fragmentation horizontale

Même avec une utilisation appropriée des vues et des index, le coût d'exécution des requêtes peut rester important. Pour le réduire encore il est possible de recourir à la fragmentation horizontale. De nombreuses solutions pour fragmenter les tables dans un entrepôt de données ont été proposées. Nous nous sommes attachés à élaborer une méthodologie de fragmentation horizontale de la table des faits et des tables de dimensions. Nous avons illustré son déroulement à travers un exemple. Il est apparu qu'il ne suffit pas de fragmenter la table des faits (en général la plus volumineuse). Très souvent il y a lieu également de fragmenter (par fragmentation dérivée) certaines des tables de dimensions pour obtenir les meilleures performances. Nous avons suggéré un algorithme pour sélectionner les tables de dimensions à fragmenter. Les performances de la fragmentation sont évaluées par l'intermédiaire d'un modèle de coût calculant le nombre d'entrées sorties nécessaire à l'exécution d'un ensemble de requêtes.

Quelques expérimentations ont montré que la fragmentation horizontale réduit généralement le coût total d'exécution des requêtes.

7.1.4 De nouveaux algorithmes de fragmentation horizontale

Les algorithmes de fragmentation horizontale existants ne garantissent pas une réduction du temps d'exécution d'un ensemble de requêtes. Nous avons développé de nouveaux algorithmes, l'un dirigé par les affinités, le deuxième dirigé par un modèle de coût pour atteindre cet objectif. L'idée d'un algorithme de fragmentation horizontale dirigé par les affinités a été introduite par Navathe pour les systèmes relationnels mais n'avait pas été exploré complètement. L'avantage principal de cet algorithme est sa complexité qui est polynômiale. Mais il utilise la fréquence d'accès des requêtes comme seul critère de regroupement. D'autres paramètres peuvent aussi jouer un rôle important (sélectivité des prédicats, taille des tables à fragmenter, etc.). Nous avons donc étudié des aménagements pour tenir compte de ces paramètres et proposé deux nouveaux algorithmes basés sur un modèle de coût. Le premier consiste à énumérer d'une façon exhaustive tous les schémas de fragmentation pouvant être générés à partir des prédicats simples définis dans les requêtes. Le schéma optimal fournissant le coût d'exécution minimum est alors sélectionné. Cet algorithme est cependant très coûteux en temps de calcul. Le deuxième algorithme est approximatif. Il est basé sur le principe du "hill-climbing". Il considère une solution initiale obtenue par l'algorithme des affinités, et essaie ensuite d'améliorer cette solution par des opérations successives sur les fragments.

7.2 Perspectives

De nombreuses perspectives tant à caractère théorique que pratique peuvent être envisagées. Dans cette section nous présentons succinctement celles qui nous paraissent être les plus intéressantes.

7.2.1 Autres types d'algorithmes

Tous les points que nous avons abordés sont en fait des problèmes d'optimisation que nous avons résolus par des algorithmes approchés de type glouton ou "hill-climbing". Il serait intéressant de considérer d'autres types d'algorithmes comme les algorithmes de recuit simulé ou les algorithmes génétiques.

7.2.2 Utilisation d'une représentation binaire des index

Les index de jointure tels que nous les avons exploités demandent beaucoup d'espace en mémoire secondaire pour leur représentation. Leur gestion introduit des temps d'overhead importants. Il serait intéressant d'étudier comment les techniques de représentation binaire pourraient être utilisées pour réduire ces temps.

7.2.3 La sélection incrémentale des vues et des index

Après les opérations de mise à jour des données de l'entrepôt, il est nécessaire de reconsidérer la sélection des vues et des index. Nous avons abordé ce type de problèmes dans le chapitre 5 pour la sélection simultanée des index et des vues. Des études complémentaires s'imposent pour privilégier systématiquement la sélection incrémentale des vues et des index.

7.2.4 La prise en considération du coût de maintenance

Dans cette thèse nous avons considéré que la seule contrainte à respecter était la contrainte d'espace disque. D'autres contraintes peuvent être envisagées comme par exemple le coût de maintenance des index et des vues.

En effet, supposer que les mises à jour de l'entrepôt de données se font hors ligne n'est pas une hypothèse systématiquement valide car certaines applications nécessitent une mise à jour immédiate. Il serait donc intéressant d'étudier si les algorithmes que nous avons suggérés peuvent être aménagés pour tenir compte de la contrainte de coût de maintenance des vues et des index.

7.2.5 La fragmentation dynamique

La plupart des algorithmes de fragmentation existants sont de nature statique : ils fragmentent une table en se basant sur un ensemble de requêtes définies préalablement. Or les applications exploitant un entrepôt de données peuvent évoluer. Il serait donc intéressant de développer ou d'aménager des algorithmes pour tenir compte de l'évolution des requêtes tant dans leurs structures que dans leurs fréquences.

7.2.6 La fragmentation d'un schéma en flocon de neige

Dans les bases de données orientées objet, nous avons montré l'intérêt de la fragmentation horizontale dérivée dans la réduction du coût d'exécution des requêtes [BKL98b]. La fragmentation dérivée est performante lorsqu'elle est utilisée sur tout le chemin d'une hiérarchie d'un schéma. Considérons, par exemple la hiérarchie Employé - Département - Projet. Si les classes Employé et Département sont horizontalement fragmentées en fonction des fragments de la classe Projet, alors la réduction du coût d'exécution de requêtes accédant à ces trois classes est très significative.

Dans les entrepôts de données, la notion de hiérarchie est bien présente dans le cas des schémas en flocon de neige (voir le schéma de TPC-H [Pag]). Il semble que la fragmentation dérivée peut être facilement utilisée dans ce type de schéma. Il serait important de se pencher sur son utilisation et son rôle dans la réduction du coût des requêtes.

7.2.7 Evaluation des algorithmes par benchmarks

Nous avons proposé d'évaluer les performances de nos algorithmes par des modèles de coût. Même si ces modèles incorporent effectivement les paramètres les plus influents, il serait important de confirmer les résultats obtenus par des expérimentations systématiques avec des benchmarks (TPC-H par exemple [Pag]).

Bibliographie

[AGEA93] K. A. S. Abdel-Ghaffar and A. El Abbadi. Optimal disk allocation for partial match queries. *ACM Transactions on Database Systems*, 18(1):132–156, March 1993.

[AGS96] R. Agrawal, A. Gupta, and S. Sarawagi. Modeling multidimensional databases. *Research Report: IBM Almaden Research Center, San Jose, CA*, 1996.

[AGS97] A. Agrawal, A. Gupta, and S. Sarawagi. Modeling multidimensional databases. Technical report research, IBM, 1997.

[AL80] M. Adiba and B. G. Lindsay. Database snapshots. *Proceedings of the International Conference on Very Large Databases*, pages 86–91, October 1980.

[Ape88] P. M. G. Apers. Data allocation in distributed database systems. *ACM Transactions on database systems*, 13(3):263–304, 1988.

[BC81] P. A. Bernstein and D-M. W. Chiu. Using semi-joins to solve relational queries. *Journal of the ACM*, 28(1):25–40, January 1981.

[BDD+98] R. G. Bello, K. Dias, A. Downing, Feenan Jr. J., W. D. Norcott, H. Sun, A. Witkowski, and M. Ziauddin. Materialized views in oracle. *Proceedings of the International Conference on Very Large Databases*, pages 659–664, August 1998.

[Bel96] L. Bellatreche. An algorithm for vertical fragmentation in distributed object database systems. *in Proceeding of the Second International Baltic Workshop on Databases and Information Systems BALT'96, Tallin, June 1996.*, pages 65–74, June 1996.

[Bel98] Z. Bellahsene. View adaptation in data warehousing systems. *Proceedings of the 9th International Conference on Database and Expert Systems Applications (DEXA)*, pages 300–309, August 1998.

[Bel00a] L. Bellatreche. Logical and physical design in data warehousing environments. *in proceedings of International Conference on Extending Database Technology (EDBT) PhD Workshop*, March 2000.

[Bel00b] L. Bellatreche. Une méthodologie pour la fragmentation dans les entrepôts des données. *in 18th Congrès d'Informatique Des Organisations et*

Systèmes d'Information et de Décision (INFORSID'2000), Lyon, pages 245–260, May 2000.

[BG93] E. Bertino and C. Guglielmina. Path-index: An approach to the efficient execution of object-oriented queries. *Data & Knowledge Engineering*, 10:1–27, 1993.

[BHY91] R. Blankinship, A. R. Hevner, and S. B Yao. An iterative method for distributed database design. *17th International Conference on Very Large Data Bases, VLDB'91*, pages 389–400, September 1991.

[BKA00] L. Bellatreche, K. Karlapalem, and Simonet A. Algorithms and support for horizontal class partitioning in object-oriented databases. *in the Distributed and Parallel Databases Journal*, 8(2):155–179, April 2000.

[BKB98] L. Bellatreche, K. Karlapalem, and G. B. Basak. Query-driven horizontal class partitioning in object-oriented databases. *in 9th International Conference on Database and Expert Systems Applications (DEXA'98), Lecture Notes in Computer Science 1460*, pages 692–701, August 1998.

[BKL98a] L. Bellatreche, K. Karlapalem, and Q. Li. Complex methods and class allocation in distributed object-oriented databases. *in the 5th International Conference on Object Oriented Information Systems (OOIS'98)*, pages 239–256, September 1998.

[BKL98b] L. Bellatreche, K. Karlapalem, and Q. Li. Derived horizontal class partitioning in oodbss: Design strategy, analytical model and evaluation. *in the 17th International Conference on the Entity Relationship Approach (ER'98)*, pages 465–479, November 1998.

[BKL98c] L. Bellatreche, K. Karlapalem, and Q. Li. An iterative approach for rules and data allocation in distributed deductive database systems. *in the 7th International Conference on Information and Knowledge Management (CIKM'98)*, pages 356–363, November 1998.

[BKL99] L. Bellatreche, K. Karlapalem, and Q. Li. Algorithms for graph join index problem in data warehousing environments. Technical Report HKUST-CS99-07, Hong Kong University of Science & Technology, March 1999.

[BKL00] L. Bellatreche, K. Karlapalem, and Q. Li. Evaluation of indexing materialized views in data warehousing environments. *Proceeding of the International Conference on Data Warehousing and Knowledge Discovery (DAWAK'2000)*, pages 57–66, September 2000.

[BKM00a] L. Bellatreche, K. Karlapalem, and M. Mohania. Some issues in design of data warehousing systems. In *To appear in Developing Quality Complex Data Bases Systems: Practices, Techniques, and Technologies, Edited by Dr. Shirley A. Becker*. Idea Group Publishing, 2000.

[BKM00b] L. Bellatreche, K. Karlapalem, and M. Mohania. What can partitioning do for your data warehouses and data marts. *Proceedings of the International Database Engineering and Application Symposium (IDEAS'2000)*, pages 437–445, September 2000.

[BKS97] L. Bellatreche, K. Karlapalem, and A. Simonet. Horizontal class partitioning in object-oriented databases. *in 8th International Conference on Database and Expert Systems Applications (DEXA'97), Toulouse, Lecture Notes in Computer Science 1308*, pages 58–67, September 1997.

[BKS98] L. Bellatreche, K. Karlapalem, and A. Simonet. Query optimization using horizontal class partitioning in object-oriented databases. *in 16th Congrès d'Informatique Des Organisations et Systèmes d'Information et de Décision (INFORSID'98), Montpellier*, pages 405–422, May 1998.

[BKS00] L. Bellatreche, K. Karlapalem, and M. Schneider. On efficient storage space distribution among materialized views and indices in data warehousing environments. *Proceedings of the International Conference on Information and Knowledge Management (ACM CIKM'2000)*, November 2000.

[BLT86] J. A. Blakeley, P. Larson, and F. W. Tompa. Efficiently updating materialized views. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 61–71, June 1986.

[BM00] Z. Bellahsene and P. Marot. Materializing a set of views: Dynamic strategies and performance evaluation. *Proceedings of the International Database Engineering and Application Symposium (IDEAS'2000)*, pages 424–428, September 2000.

[BPT97] E. Baralis, S. Paraboschi, and E Teniente. Materialized view selection in a multidimensional database. *Proceedings of the International Conference on Very Large Databases*, pages 156–165, August 1997.

[BS95] J. C. Bontempo and C. M. Saracco. *Database Management: Principles and Products.* Prentice Hall, 1995.

[BS96] L. Bellatreche and A. Simonet. A horizontal fragmentation algorithm in object database design. *in Third International Conference of the Austrian Center for Parallel Computation (ACPC'96), with Special Emphasis on Parallel Databases and Parallel I/O, Lecture Note in Computer Science (LNCS), 1127*, pages 223–226, September 1996.

[BSS96] L. Bellatreche, A. Simonet, and M. Simonet. An algorithm for vertical fragmentation in distributed object database systems with complex attributes and methods. *in International Workshop on Database and Expert Systems Applications (DEXA'96), Zurich*, pages 15–21, September 1996.

[CCS93] E. F. Codd, S. B. Codd, and C. T. Salley. Providing olap (on-line analytical processing) to user-analysts: An it mandate. *White paper, http://www.arborsoft.com/essbase/wht_paper/coddTOC.html*, 1993.

[CD97] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *Sigmod Record*, 26(1):65–74, March 1997.

[Cha82] J. Chang. A heuristic approach to distributed query processing. *Proceedings of the International Conference on Very Large Databases*, pages 54–61, September 1982.

[CMVN94] S. Chakravarthy, J. Muthuraj, R. Varadarajan, and S. B. Navathe. An objective function for vertically partitioning relations in distributed databases and its analysis. *in Distributed and Parallel Databases Journal*, 2(2):183–207, April 1994.

[CN97] S. Chaudhuri and V. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. *Proceedings of the International Conference on Very Large Databases*, pages 146–155, August 1997.

[CN98] S. Chaudhuri and V. Narasayya. Autoadmin 'what-if' index analysis utility. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 367–378, June 1998.

[CN99] S. Chaudhuri and V. Narasayya. Index merging. *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 296–303, March 1999.

[CN00] S. Chaudhuri and V. Narasayya. Automating statistics management for query optimizers. *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 339–348, March 2000.

[CNP82] S. Ceri, M. Negri, and G. Pelagatti. Horizontal data partitioning in database design. *Proceedings of the ACM SIGMOD International Conference on Management of Data. SIGPLAN Notices*, pages 128–136, 1982.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[Cor97a] Oracle Corp. Star queries in oracle8. *White Paper*, June 1997.

[Cor97b] Informix Corporation. Informix-online extended parallel server and informix-universal server: A new generation of decision-support indexing for enterprise data warehouses. *White Paper*, 1997.

[Cor98] Oracle Corp. Oracle warehouse: Unleash the power of information. *White Paper: http://www.oracle.com/tools/datawarehouse*, November 1998.

[CP84] S. Ceri and G. Pelagatti. *Distributed Databases: Principles & Systems*. McGraw-Hill International Editions, 1984.

[CPW89] S. Ceri, B. Pernici, and G. Wiederhold. Optimization problems and solution methods in the design of data distribution. *Information Systems*, 14(3):261–272, 1989.

[CW81] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. *Proceedings of the International Conference on Very Large Databases*, pages 577–589, September 1981.

[CY87] D. Cornell and P. S. Yu. A vertical partitioning algorithm for relational databases. *Proceedings of the Third International Conference on Data Engineering(ICDE'87)*, pages 30–35, 1987.

[CY99a] C. Chee-Yong. Indexing techniques in decision support systems. Phd. thesis, University of Wisconsin - Madison, 1999.

[CY99b] C. Chee-Yong. Indexing techniques in decision support systems. Ph.d. thesis, University of Wisconsin - Madison, 1999.

[D.96] Simpson D. Build your warehouse on mpp. *Available at http://www.datamation.com/servr/12mpp.html*, December 1996.

[Dar99] J. Darmont. Étude des performances de méthodes de regroupement dynamique dans les bases de données orientées-objet. Thèse de doctorat, Université Clermont-Ferrand II, Janvier 1999.

[DRSN98] P. Deshpande, K. Ramasamy, A. Shukla, and J. F. Naughton. Caching multidimensional queries using chunks. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 259–270, June 1998.

[DRT99] A. Datta, K. Ramamritham, and H. Thomas. Curio: A novel solution for efficient storage and indexing in data warehouses. *Proceedings of the International Conference on Very Large Databases*, pages 730–733, September 1999.

[EB95] C. I. Ezeife and K. Barker. A comprehensive approach to horizontal class fragmentation in distributed object based system. *International Journal of Distributed and Parallel Databases*, 3(3):247–272, 1995.

[EN94] R. ElMasri and S. B. Navathe. *Fundamentals of Database Systems.* Benjamin Cummings, Redwood City, CA, 1994.

[ES76] M. J. Eisner and D. G. Severance. Mathematical techniques for efficient record segmentation in large shared databases. *Journal of the ACM*, 23(4):619–635, October 1976.

[Fir97] J. M. Firestone. Data warehouses and data marts: A dynamic view. White Paper 3, Executive Information Systems, Inc., March 1997.

[FKL97] C. W. Fung, K. Karlapalem, and Q. Li. Cost-driven evaluation of vertical class partitioning in object oriented databases. *in Fifth International Conference On Database Systems For Advanced Applications (DASFAA'97), Melbourne, Australia*, pages 11–20, April 1997.

[FKL98] C. W. Fung, K. Karlapalem, and Q. Li. Structural join index hierarchy: A mechanism for efficient complex object retrieval. *in the 5th International Conference on Foundations of Data Organization and Algorithms (FODO'98)*, November 1998.

[Fra97] J-M. Franco. *Le Data Warehouse, Le Data Mining.* Eyrolles, 1997.

[FST88] S. Finkelstein, M. Schkolnick, and P. Tiberio. Physical database design for relational databases. *ACM Transactions on Database Systems*, 13(1):91–128, March 1988.

[Gar83] G. Gardarin. *Bases de Données : Les Systèmes et Leurs Langages.* Editions Eyrolles, 1983.

[GCA01] E. B. Guerrero, C. Collet, and M. Adiba. Entrepôts de données: caractéristiques et problématique. *Technique et Science Informatiques*, 20(2-2001):145 – 178, 2001.

[GGT96] G. Gardarin, J.-R. Gruser, and Z.-H. Tang. A cost-based selection of path expression processing algorithms in object-oriented databases. *22th International Conference on Very Large Data Bases, VLDB'96*, pages 390–401, 1996.

[GHRU97] H. Gupta, V. Harinarayan, A. Rajaraman, and J. Ullman. Index selection for olap. *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 208–219, April 1997.

[GM95] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *Data Engineering Bulletin*, 18(2):3–18, June 1995.

[GMR99] M. Golfarelli, D. Maio, and S. Rizzi. Vertical fragmentation of views in relational data warehouses. *Proceedings of Sistemi Evoluti per Basi di Dati*, pages 19–33, 1999.

[GMS93] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 157–166, June 1993.

[GO96] G. Gardarin and Gardarin O. *Le Client-Serveur.* Edition Eyrolles, 1996.

[Gra93] G. Graefe. Query evaluation technique for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[Gru96] J. R. Gruser. Modèle de coût pour l'optimisation de requête objet. Thèse de doctorat, Université de Paris VI, Décembre 1996.

[Gup97] H. Gupta. Selection of views to materialize in a data warehouse. *Proceedings of the 6th International Conference on Database Theory (ICDT '97)*, pages 98–112, 1997.

[Gup99] H. Gupta. Selection and maintenance of views in a data warehouse. Ph.d. thesis, Stanford University, September 1999.

[GV90] G. Gardarin and P. Valduriez. *SGBD AVANCEES : Bases de données objets, déductives, réparties.* Edition Eyrolles, 1990.

[HMA99] C. A. Hurtado, O. A. Mendelzon, and Vaisman A. A. Maintaining data cubes under dimension updates. *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 346–355, March 1999.

[HN79] M. Hammer and B. Niamir. A heuristic approach to attribute partitioning. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 93–101, 1979.

[HRU96] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes effeciently. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 205–216, June 1996.

[HS75] J. A. Hoffer and D. G. Severance. The 1st international conference on very large databases (vldb'75). In *The Use of Cluster Analysis in Physical Database Design*, pages 69–86. Morgan Kaufman, 1975.

[HS95] P. J. Haas and A. N. Swami. Sampling-based selectivity estimation for joins using augmented frequent value statistics. *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 522–531, 1995.

[Huy97] N. Huyn. Multiple-view self-maintenance in data warehousing environments. *Proceedings of the International Conference on Very Large Databases*, pages 26–35, August 1997.

[Hyp] Hyperion. *Hyperion Essbase OLAP Server.* http://www.hyperion.com/.

[IC91] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 268–277, June 1991.

[Inm92] W. H. Inmon. *Building the Data Warehouse.* John Wiley, 1992.

[Ioa93] Y. E. Ioannidis. Universality of serial histograms. *Proceedings of the International Conference on Very Large Databases*, pages 256–267, August 1993.

[Jai91] R. Jain. *The Art of Computer Systems Performance Analysis.* Willy Professional Computing, 1991.

[JLS99a] H. Jagadish, L. V. S. Lakshmanan, and D. Srivastava. Snakes and sandwiches: Optimal clustering strategies for a data warehouse. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 37–48, June 1999.

[JLS99b] H. Jagadish, L. V. S. Lakshmanan, and D. Srivastava. What can hierarchies do for your data warehouses. *Proceedings of the International Conference on Very Large Databases*, pages 530–541, September 1999.

[JMS95] H. Jagadish, I. S. Mumick, and A. Silberschatz. Maintenance issues for the chronicle data model. *roceedings of the Symposium on Principles of Databases Systems (PODS'95)*, pages 113–124, May 1995.

[Kar96] K. Karlapalem. Redesign of distributed relational databases. Ph.d. thesis, Georgia Institute of Technology, November 1996.

[Ked99] Z. Kedad. Techniques d'intégration dans les systèmes d'information multi-source. Thèse de doctorat, Université de Versailles Saint-Quentin-en-Yvelines, Janvier 1999.

[Kim96] R. Kimball. *The Data Warehouse Toolkit.* John Wiley & Sons, 1996.

[Kim97] R. Kimball. A dimensional modeling manifesto. *DBMS Magazine*, August 1997.

[KL70] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, pages 291–307, February 1970.

[KL90] O. Kiyoshi and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. *Proceedings of the International Conference on Very Large Databases*, pages 314–325, August 1990.

[KL95] K. Karlapalem and Q. Li. Partitioning schemes for object oriented databases. *in Proceeding of the Fifth International Workshop on Research Issues in Data Engineering- Distributed Object Management, RIDE-DOM'95*, pages 42–49, March 1995.

[KLV96] K. Karlapalem, Q. Li, and S. Vieweg. Method induced partitioning schemes in object-oriented databases. *in 16th International Conference on Distributed Computing System (ICDCS'96), Hong Kong*, pages 377–384, May 1996.

[KNM94] K. Karlapalem, S.B. Navathe, and M. M. A. Morsi. Issues in distributed design of object-oriented databases. In *Distributed Object Management*, pages 148–165. Morgan Kaufman Publishers Inc., 1994.

[KP97] K. Karlapalem and N. M Pun. Query driven data allocation algorithms for distributed database systems. *in 8th International Conference on Database and Expert Systems Applications (DEXA'97), Toulouse, Lecture Notes in Computer Science 1308*, pages 347–356, September 1997.

[KR98] Y. Kotidis and N. Roussopoulos. An alternative storage organization for rolap aggregate views based on cubetrees. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 249–258, June 1998.

[KR99] Y. Kotidis and N. Roussopoulos. Dynamat: A dynamic view management system for data warehouses. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 371–382, June 1999.

[KS95] R. Kimball and K. Strehlo. Why decision support fails and how to fix it. *SIGMOD Record*, 24(3):92–97, September 1995.

[LDH+84] M. G. Lohman, D. Daniels, L. M. Haas, R. Kistler, and P. G. Selinger. Optimization of nested queries in a distributed relational database. *Proceedings of the International Conference on Very Large Databases*, pages 403–415, August 1984.

[Lev] A. Levy. Answering queries using views: a survey. *To appear in the VLDB Journal.*

[LHM+86] B. G. Lindsay, L. M. Haas, C. Mohan, H. Pirahesh, and P. F. Wilms. A snapshot differential refresh algorithm. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 53–60, June 1986.

[LQA97] W. Labio, D. Quass, and B. Adelberg. Physical database design for data warehouses. *Proceedings of the International Conference on Data Engineering (ICDE)*, 1997.

[LR98] H. Lei and K. A. Ross. Faster joins, self-joins and multi-way joins using join indices. *Data and Knowledge Engineering*, 28(3):277–298, November 1998.

[Mar98] P. Marcel. Manipulation de données multidimensionnelles et langages de règles. Thèse de doctorat, Institut National des Sciences Appliquées de Lyon, Décembre 1998.

[Men97] A. Mendelzon. Olap: Concepts and products. *Talk at University of Toronto*, 1997.

[Mic98] MicroStrategy. The case for relational olap. Technical report, White paper: http://www.microstrategy.com, 1998.

[MK00] M. Mohania and Y. Kambayashi. Making aggregate views self-maintainable. *Data and Knowledge Engineering*, 32(1):87–109, January 2000.

[MLJ98] H. G. Molina, W. J. Labio, and Yang J. Expiring data in a warehouse. *Proceedings of the International Conference on Very Large Databases*, pages 500–511, August 1998.

[MLWZ98] H. Molina, W. J. Labio, J. L. Wiener, and Y. Zhuge. Distributed and parallel computing issues in data warehousing. *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, page 7, June 1998.

[MSRK00] M. Mohania, S. Samtani, J. F. Roddick, and Y. Kambayashi. Advances and research directions in data warehousing technology. *To appear in the Australian Journal of Information Systems*, 2000.

[MSW72] W. T. McCormick, P. J. Schweitzer, and T. W. White. Problem decomposition and data reorganization by a clustering technique. *Operation Research*, 20(5):993–1009, September 1972.

[MW89] A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph database. *Proceedings of the International Conference on Very Large Databases*, pages 185–193, August 1989.

[Naa99] H. Naacke. Modèles de coût pour médiateurs de bases de données hétérogènes. Thèse de doctorat, Université de Versailles Saint-Quentin-en-Yvelines, Séptembre 1999.

[NB99] A. Y. Noaman and K. Barker. A horizontal fragmentation algorithm for the fact relation in a distributed data warehouse. *in the 8th International Conference on Information and Knowledge Management (CIKM'99)*, pages 154–161, November 1999.

[NCWJ84] S.B. Navathe, S. Ceri, G. Wiederhold, and Dou J. Vertical partitioning algorithms for database design. *ACM Transaction on Database Systems*, 9(4):681–710, December 1984.

[Nic91] J-C. Nicolas. Machines bases de données parallèles: Contribution aux problèmes de la fragmentation et de la distribution. Thèse de doctorat, Université des Sciences et Techniques de Lille, Flandres Artois, Janvier 1991.

[Nil80] N. J. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufman Publishers Inc., 1980.

[NKR95] S.B. Navathe, K. Karlapalem, and M. Ra. A mixed partitioning methodology for distributed database design. *Journal of Computer and Software Engineering*, 3(4):395–426, 1995.

[NR89] S.B. Navathe and M. Ra. Vertical partitioning for database design : a graphical algorithm. *ACM SIGMOD*, pages 440–450, 1989.

[ODV92] M. T. Özsu, Dayal, and P. Valduriez. An introduction to distributed object management. *International Workshop on Distributed Object Management*, pages 1–24, August 1992.

[OQ97] P. O'Neil and D. Quass. Improved query performance with variant indexes. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 38–49, May 1997.

[Ora99] *Oracle8i Concepts*. Oracle Corporation, Release 8.1.5, 1999.

[OV91] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1991.

[OV99] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems: Second Edition*. Prentice Hall International, Inc., 1999.

[Pag] TPC Home Page. Tpc benchmark^{TM}d (decision support). *http://www.tpc.org*.

[Pas00] N. Pasquier. Data mining : Algorithmes d'extraction et de réduction des rgles d'association dans les bases de données. Thèse de doctorat, Université Clermont-Ferrand II, Janvier 2000.

[PC98] N. Pendse and R. Creeth. The olap report, business intelligence inc. *http://www.olapreport.com*, 1998.

[PH89] A. Poli and L. Huguet. *Codes Correcteurs: Théorie et Applications.* Masson, 1989.

[PKN91] G Pernul, K. Karlapalem, and S. B. Navathe. Relational database organization based on views and fragments. *in International Workshop on Database and Expert Systems Applications (DEXA'91)*, pages 380–386, 1991.

[PS91] G. Piatetsky-Shapiro. Discovery, analysis, and presentation of strong rules. *in Knowledge Discovery in Databases, AAAI/MIT Press*, pages 229–248, 1991.

[QGMW96] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. *in Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems*, pages 158–169, December 1996.

[QW97] D. Quass and J. Widom. On-line warehouse view maintenance for batch updates. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 393–404, May 1997.

[Ram98] R. Ramakrishnan. *Database Management Systems.* WCB/McGraw Hill, 1998.

[Rav96] R. Ravat. od^3: contribution méthodologique à la conception de bases de données orientées objet réparties. Thèse de doctorat, Université Paul Sabatier, September 1996.

[RSS96] K. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 447–458, June 1996.

[SAD+79] P. G. Selinger, M. M. Astrahan, Chamberlin D. D., R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 23–34, 1979.

[SDJL96] D. Srivastava, S. Dar, H. Jagadish, and A. Y. Levy. Answering queries with aggregation using views. *Proceedings of the International Conference on Very Large Databases*, pages 318–329, September 1996.

[SDN98] A. Shukla, P. Deshpande, and J. F. Naughton. Materialized view selection for multidimensional datasets. *Proceedings of the International Conference on Very Large Databases*, pages 488–499, August 1998.

[Sel88] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, March 1988.

[SM77] D. Stanat and D. McAllister. *Discrete Mathematics in Computer Science*. Printice Hall, 1977.

[SP89] A. Segev and J. Park. Maintaining materialized views in distributed databases. *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 262–270, February 1989.

[SS89] A. Simonet and M. Simonet. Les classes d'équivalence induites par des dépendances inter-attributs: fondements pour une représentation des objets partagés dans les bases de connaissances. Technical Report R.R. 769-I, Laboratoire Artemis - Grenoble, 1989.

[SS94] A. Simonet and M. Simonet. Objects with views and constraints: From databases to knowledge bases. *in the Proceeding of the International Conference on Object Oriented Information Systems, OOIS'94*, pages 182–195, December 1994.

[SSN00] A. Sanjay, C. Surajit, and V. R. Narasayya. Automated selection of materialized views and indexes in microsoft sql server. *To appear in VLDB'2000*, September 2000.

[Sys97] Red Brick Systems. Star schema processing for complex queries. *White Paper*, July 1997.

[TK78] D. Tsichritzis and A. Klug. The ansi/x3/sparc framework. *AFIPS Press, Montval, N.J.*, 1978.

[TS97] D. Theodoratos and T. K. Sellis. Data warehouse configuration. *Proceedings of the International Conference on Very Large Databases*, pages 126–135, August 1997.

[Ull89] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*. vol. II. Computer Science Press, 1989.

[Ull96] J. Ullman. Efficient implementation of data cubes via materialized views. *in the Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining (KDD'96)*, pages 386–388, 1996.

[Val85] P. Valduriez. Optimisation des opérateurs relationnels dans les machines bases de données. Docteur ès sciences, Université Pierre et Marie Curie, September 1985.

[Val87] P. Valduriez. Join indices. *ACM Transactions on Database Systems*, 12(2):218–246, June 1987.

[VS99] P. Vassiliadis and T Sellis. A survey of logical models for olap databases. *SIGMOD Record*, 28(4):64–69, December 1999.

[WB97] M-C. Wu and A. Buchmann. Research issues in data warehousing. *in Datenbanksysteme in Büro, Technik und Wissenschaft(BTW'97)*, pages 61–82, March 1997.

[WLDH96] Y. Wang, J. C.L. Liu, Du, and J. Hsieh. Video file allocation over disk arrays for video-on-demand. *in Proceedings of the International Conference on Multimedia Computing and Systems(ICMCS'96)*, pages 160–173, June 1996.

[YC84] C. T. Yu and Chang C. C. Distributed query processing. *ACM Computing Surveys*, 16(4):399–433, December 1984.

[YKL97] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. *Proceedings of the International Conference on Very Large Databases*, pages 136–145, August 1997.

[ZO94] Y. Zhang and M-E. Orlowska. On fragmentation for distributed database design. *Information Sciences*, 1(3):117–132, 1994.