Techniques d'optimisation des requêtes dans les data warehouses

Ladjel Bellatreche

Techniques d'optimisation des requêtes dans les data warehouses

Ladjel Bellatreche
LISI/ENSMA
Téléport2 - 1, Avenue Clément Ader
86960 Futuroscope - FRANCE
bellatreche@ensma.fr

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 papier, nous allons présenter les techniques d'optimisation des requêtes les plus utilisées dans le contexte des entrepôts de données. Ces techniques interpellent deux niveaux de la conception des entrepôts : le niveau logique (fragmentation de l'entrepôt et la sélection des vues) et le niveau physique (la sélection des index). Nous allons également dégager quelques ouvertures de recherches.

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, et de nombreux projets ont été développés au sein des universités (le projet *WHIPS* de l'université de Stanford, le projet *H2O* de l'université du Colorado en collaboration avec la Southern California University, etc.).

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, 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, etc.) comme nous le verrons tout au long de ce papier. Ce dernier est divisé en six 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 [1]. En fonction de la manière dont le cube est stocké, deux approches existent pour construire des systèmes multidimensionnels : Relational OLAP (ROLAP), et Multidimensional OLAP (MOLAP).

La section 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 [8].

La section 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 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 5 nous présentons ce concept et les algorithmes associés pour les modèles relationnel qui peuvent être adaptés aux entrepôts de données.

La section 6 présente les incidences pratiques des techniques présentées et la section 7 conclut le papier.

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 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.

2.1 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 [18].

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.

2.1.1 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".*

Plus formellement, la structure d'un cube de données est la suivante [1] :

- 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 $< v'_1, v'_2, ..., v'_k >$.

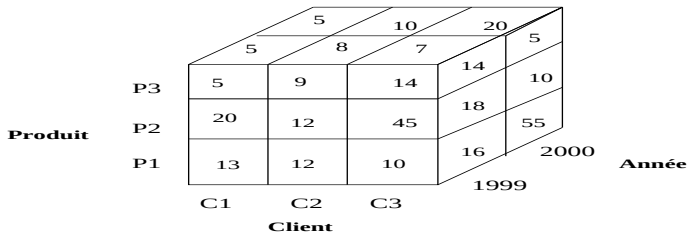

FIG. 1 – Le cube VENTES

Exemple 2 *La Figure 1 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* $dom_{quantite} \subseteq N$.

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

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. 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.
- **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.

2.1.3 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 : $RollUp^{niveau\_sup}_{niveau\_inf}(CUBE)$.

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 : $DrillDown^{niveau\_sup}_{niveau\_inf}(CUBE)$.

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 [16]) 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 [12]) utilise un SGBD relationnel pour gérer et stocker le cube de données.

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). 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.

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. Mais, lorsque les données sont éparses, ces systèmes sont consommateurs d'espace [9] et des techniques de compression doivent être utilisées. Les produits de Hyperion Essbase OLAP Server [16] ont adopté cette technique de stockage.

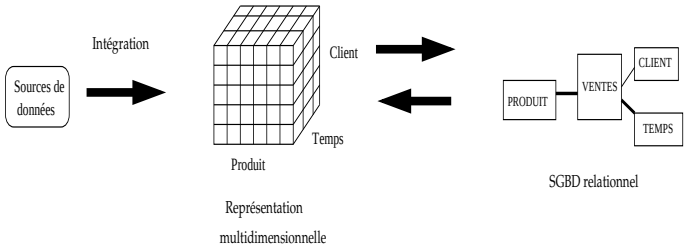

FIG. 2 – Systèmes MOLAP et ROLAP

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).

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* [22] et *MetaCube d'Informix* [12].

Mendelzon [21] fait une comparaison entre l'approche ROLAP et l'approche MOLAP (voir la Table 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. 1 – ROLAP vs. MOLAP

Deux schémas principaux sont utilisés pour modéliser les systèmes ROLAP : (1) *le schéma en étoile*, (2) le *schéma en flocon de neige*.

Le schéma en étoile Dans ce type de schéma, 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* [29], ou encore *Informix* [12]).

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>

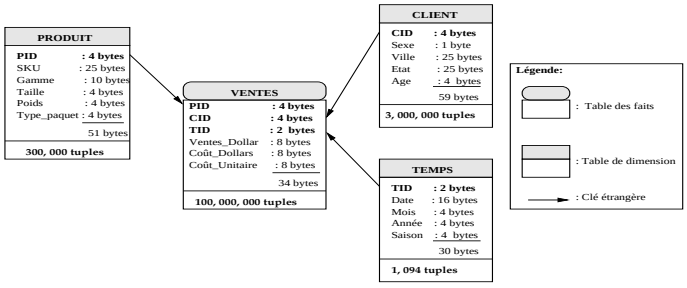

FIG. 3 – Un exemple d'un schéma en étoile

La Figure 3 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 3, 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).

Le schéma en flocon de neige Le schéma en étoile ne reflète pas les hiérarchies associées à une dimension [17]. 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 [17]. 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. Ce schéma a été fortement déconseillé par Kimball [18] 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. [17]) et des industriels de AT&T Labs-Research [17].

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.
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.

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.

- 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. 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 [31] :

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 [14, 31] :
Étant donné une contrainte de ressource S (capacité de stockage, par exemple), 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 4).

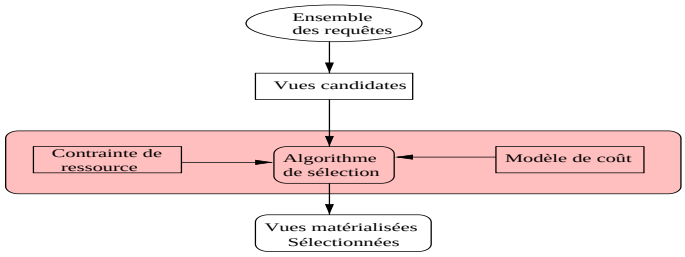

FIG. 4 – Le processus de sélection des vues matérialisées

Ce problème a été largement étudié tant pour l'approche MOLAP [19, 31] que pour l'approche ROLAP [14, 30, 34] 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 [14].

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. [19] 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.

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* [26, 34, 2, 30], (2) *algorithmes dirigés par la contrainte d'espace* [14] et (3) *algorithmes dirigés par la contrainte du temps total de maintenance des vues* [14].

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. [34] proposée dans le contexte ROLAP, et l'approche de Baralis [2] proposée dans le contexte MOLAP.

Les travaux de Yang et al. [34] 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* [2]. 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. 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 (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 5 montre que les requêtes Q_1 et Q_2 ont une expression commune (Exp_1). Ces noeuds sont de bons candidats pour la matérialisation.*

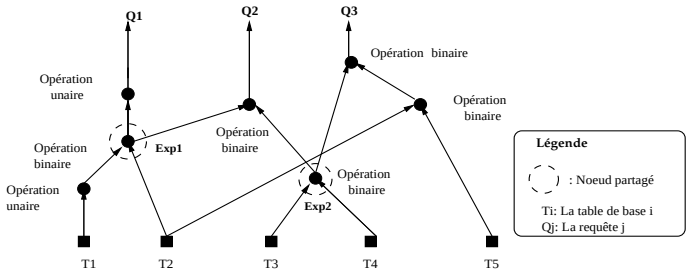

FIG. 5 – Le principe de base de la sélection de Yang et al. [34]

Les travaux de Baralis et al. [2] Dans le contexte MOLAP, Baralis et al. [2] 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 et V_j un descendant. Les ancêtres et les descendants d'un noeud V_i du treillis sont définis comme suit :*

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

.

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 [15]. 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.
- 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 [14].

3.2.2 Algorithmes dirigés par la contrainte d'espace

Dans le travail initial réalisé pour la sélection des vues, Harinarayan et al. [15] 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.

Les auteurs ont modélisé le problème sous la forme d'un treillis de vues. 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 :
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 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 [27].

3.2.3 Algorithmes dirigés par le temps de maintenance

Ce type d'algorithmes a été étudié par Gupta [14]. 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^* 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.

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 [20]. Dans ce cas, ces vues peuvent être considérées comme des photographies (snapshots). Dans la deuxième stratégie [8], 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. 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.

3.4 La réécriture des requêtes

Les vues matérialisées sont stockées sous forme de tables relationnelles [7]. 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* [28]. 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 [7]. La plupart des solutions proposées sont basées sur des modèles de coût.

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.

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 [24, 12].

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 [32] 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.

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 [24] 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 [24].

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

4.1.2 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. Plusieurs méthodes ont été proposées pour accélérer ces opérations. Ces méthodes incluent les boucles imbriquées, le hachage, la fusion, etc.

Valduriez [32] 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. 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. 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 [29].

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*).

Pour résoudre ce problème, Red Brick [29] 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* :

1. Comme son nom l'indique, ce type d'index est exclusivement adapté aux schémas en étoile [17]. Par contre, pour d'autres schémas comme le flocon de 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.
2. 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.

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 [10]. 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 6.

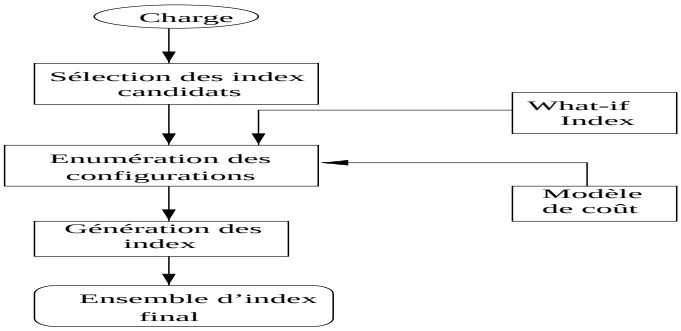

FIG. 6 – L'architecture de l'outil de sélection d'index

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 [1] 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. [11] 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 :

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

- 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.

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 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. 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 ?*

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 [25]. Dans ce papier, nous ne ferons pas de différence entre les deux concepts. Rappelons qu'un schéma de fragmentation est le résultat du processus de la fragmentation [3].

5.1 Les types de fragmentation

Deux sortes de fragmentation sont possibles : la fragmentation horizontale et 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. 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.

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 [2]. 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 [4] : les fragmentations *primaire* et *dérivée*. La fragmentation primaire d'une relation est effectuée grâce à des prédicats de sélection définis sur la relation [25]. La fragmentation horizontale dérivée s'effectue avec des prédicats de sélection définis sur une autre relation.

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

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

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 semi-jointure suivante : $S_i = S \ltimes R_i$.
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 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.

5.2 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 [6].

Wu et al. [33] 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 [33] 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. [24]. Les index de projection ressemblent à un fragment vertical d'une relation.
Dernièrement, Datta et al. [13] 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.

Pour ce qui est de la fragmentation horizontale, peu de travaux ont été réalisés, excepté celui de Noaman et al. [23]. Ces auteurs ont proposé une technique de construction d'un entrepôt réparti en utilisant la stratégie descendante [25]. 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. Dans [5, 6], nous avons proposé un algorithme de fragmentation horizontale d'un schéma en étoile en se basant sur un ensemble de requêtes de départ. Nous avons également évalué la performance de cet algorithme en utilisant des benchmarks.

6 Incidences pratiques des techniques présentées

Les problèmes et les approches considérés dans ce papier 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. Nous indiquons en particulier comment certaines de ces approches ont été incorporées dans ORACLE 9i et Microsoft SQL server.

6.1 Partitionnement des données

ORACLE 9i incorpore le fragmentation horizontale pour gérer les grands objets (tables, vues matérialisées, index) et les décomposer en parties plus petites appelées partitions. *ORACLE 9i* propose plusieurs méthodes

de fragmentation : 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. *Par contre, la fragmentation dérivée n'est pas supportée par les SGBDs existents.*

6.2 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.

6.3 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 papier 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.

7 Conclusion et perspectives

Dans cet article, nous avons présenté les techniques principales utilisées pour optimiser le temps d'exécution des requêtes dans les entrepôts de données. Il s'agit des vues matérialisées, de la fragmentation, et de la création d'index. Ces techniques ont été présentées en se basant sur les différentes modélisations des entrepôts de données : le modèle ROLAP et le modèle MOLAP. Nous avons mis en évidence les problèmes liés à chaque technique. Certains problèmes restent toujours ouverts : (1) le problème de distribution de l'espace entre les vues et les index, (2) le problème de la sélection des vues matérialisées ou des index en prenant en considération *simultanément* les trois types de contraintes classiques : le coût de stockage, le coût de maintenance et le temps de génération, (3) la *fragmentation sous contraintes* (par exemple : le nombre de fragments voulus prédéfinis) : la plupart des algorithmes de fragmentation horizontale existants donne un nombre de fragments que l'on ne peut pas contrôler. Alors que fréquemment l'administrateur de l'entrepôt de données souhaiterait décomposer son entrepôt en un ensemble de partitions choisi à l'avance.

Remerciements Je tiens à remercier Monsieur Guy Pierra directeur du laboratoire LISI et Monsieur Yamine AIT AMEUR pour leurs commentaires et ainsi que la relecture de cet article.

Références

[1] R. Agrawal, A. Gupta, and S. Sarawagi. Modeling multidimensional databases. *Research Report : IBM Almaden Research Center, San Jose, CA*, 1996.

[2] 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.

[3] L. Bellatreche. Partitioning strategies in distributed object-oriented database systems. *in Proceeding of the 3rd International Symposium Programming and Systems (ISPS'97), Algiers*, pages 360–373, April 1997.

[4] 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.

[5] 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.

[6] L. Bellatreche, M. Schneider, M. Mohania, and B. Bhargava. Partjoin : An efficient storage and query execution for data warehouses. *Proceedings of the 4th International Conference on Data Warehousing and Knowledge Discovery (DAWAK'2002)*, LNCS(2454) :296–306, September 2002.

[7] 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.

[8] 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.

[9] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *Sigmod Record*, 26(1) :65–74, March 1997.

[10] 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.

[11] S. Chaudhuri and V. Narasayya. Index merging. *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 296–303, March 1999.

[12] 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.

[13] 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.

[14] H. Gupta. Selection and maintenance of views in a data warehouse. Ph.d. thesis, Stanford University, September 1999.

[15] 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.

[16] Hyperion. *Hyperion Essbase OLAP Server*. http ://www.hyperion.com/.

[17] 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.

[18] R. Kimball. A dimensional modeling manifesto. *DBMS Magazine*, August 1997.

[19] 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.

[20] 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.

[21] A. Mendelzon. Olap : Concepts and products. *Talk at University of Toronto*, 1997.

[22] MicroStrategy. The case for relational olap. Technical report, White paper : http ://www.microstrategy.com, 1998.

[23] 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.

[24] 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.

[25] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems : Second Edition*. Prentice Hall, 1999.

[26] 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.

[27] 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.

[28] 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.

[29] Red Brick Systems. Star schema processing for complex queries. *White Paper*, July 1997.

[30] D. Theodoratos and T. K. Sellis. Data warehouse configuration. *Proceedings of the International Conference on Very Large Databases*, pages 126–135, August 1997.

[31] 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.

[32] P. Valduriez. Join indices. *ACM Transactions on Database Systems*, 12(2) :218–246, June 1987.

[33] M-C. Wu and A. Buchmann. Research issues in data warehousing. *in Datenbanksysteme in Bro, Technik und Wissenschaft(BTW'97)*, pages 61–82, March 1997.

[34] 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.