Chapitre 3

Problème de sélection d'index et de vues matérialisées

Dans la première partie de ce chapitre, nous étudions le problème de sélection d'index et rapportons les travaux traitant ce problème. De la même manière, dans la deuxième partie, nous présentons le problème de sélection de vues matérialisées et détaillons les travaux qui s'y consacrent. Nous proposons à la fin de chaque partie une classification des travaux de sélection d'index et de vues matérialisées selon des critères que nous jugeons pertinents. Dans la troisième partie, nous exposons quelques travaux qui s'intéressent au problème de la sélection simultanée des index et des vues matérialisées.

3.1 Problème de sélection d'index

Le problème de sélection d'index consiste à construire une configuration d'index optimisant le coût d'exécution d'une charge donnée. Cette optimisation peut être réalisée sous certaines contraintes, comme l'espace de stockage alloué aux index à sélectionner.

Plus formellement, si $I = \{i_1, ..., i_n\}$ est un ensemble d'index candidats, $Q = \{q_1, ..., q_m\}$ un ensemble de requêtes de la charge et S la taille de l'espace disque alloué par l'administrateur pour stocker les index à sélectionner, alors il faut trouver une configuration d'index $Config_I$ tel que :

– le coût d'exécution des requêtes de la charge soit minimal, c'est-à-dire

$$C_(Q, Config_I) = Min\left(C_{/I}(Q)\right);$$

– l'espace de stockage des index de $Config_I$ ne dépasse pas S, c'est-à-dire

$$\sum_{i \in Config_I} taille(i) \leq S.$$

Le problème de sélection d'index est NP–Complet [Com78]. De ce fait, il n'existe pas d'algorithme qui propose une solution optimale en un temps fini. Plusieurs travaux de recherche proposent des solutions proches de la solution l'optimale en utilisant des heuristiques réduisant la complexité du problème.

3.1.1 Construction d'un ensemble d'index candidats

L'administrateur, de par son expertise, peut manuellement fournir, à partir d'une charge donnée, un ensemble d'index candidats. Dans ce cas, la sélection d'index est effectuée à partir de ces candidats [FON92, CBC93a, CBC93b]. Le choix des index candidats est alors subjectif car il dépend du degré d'expertise de l'administrateur.

Par opposition, les index candidats peuvent être extraits de manière automatique en réalisant une analyse syntaxique des requêtes de la charge [CN97, VZZ+00, GRS02]. L'analyse syntaxique dépend du SGBD utilisé pour l'optimisation des performances car chaque SGBD peut utiliser une syntaxe spécifique dérivée du standard SQL.

3.1.2 Construction de la configuration finale d'index

3.1.2.1 Construction ascendante ou descendante

Les méthodes ascendantes partent d'une configuration d'index candidats vide [KY87, FON92, CBC93b, CN97]. Elles ajoutent continuellement des index en minimisant le coût de la charge. Ce processus d'ajout s'arrête lorsque le coût ne diminue plus suite à un ajout. En revanche, les méthodes ascendantes ont comme point de départ l'ensemble de tous les index candidats. À chaque itération, des index sont élagués [KY87, CBC93a]. Si le coût de la charge avant élagage est inférieur (respectivement, supérieur) au coût de cette même charge

après élagage, les index élagués sont inutiles (respectivement, utiles) pour réduire le coût de la charge. Le processus d'élagage s'arrête lorsque le coût augmente suite à un élagage.

3.1.2.2 Algorithmes génétiques

Les algorithmes génétiques sont couramment utilisés pour résoudre des problèmes d'optimisation. Ils ont été adaptés au problème de sélection d'index [KLT03]. La population de départ est l'ensemble d'index candidats fourni en entrée. Un index est donc assimilé à un individu. La fonction objectif à optimiser est le coût de la charge en présence d'une configuration d'index. La construction combinatoire des configurations d'index est réalisée à l'aide des opérateurs génétiques de croisement, de mutation et de sélection.

3.1.2.3 Problème du sac à dos

Le problème de sélection d'index a été assimilé dans certains travaux au problème du sac à dos [ISR83, Gun99, VZZ+00, FR03]. Ce dernier est posé comme suit. Soit un ensemble d'objets $O = \{o_1, o_2, \ldots, o_n\}$. Chaque objet o_i a un poids $poids(o_i)$ et un bénéfice $benefice(o_i)$. Le sac à dos est de taille S. Le but est de trouver un sous-ensemble d'objets $N \subseteq O$ tels que sa taille ait comme borne supérieure S et que son bénéfice soit maximum, ou plus formellement : $\sum_{o_i \in N} poids(o_i) \leq S$ et $\forall N' \subseteq O, \sum_{o_i \in N} benefice(o_i) < \sum_{o_i \in S} benefice(o_i)$.

Par analogie avec le problème de sélection d'index, un objet est un index, le poids et le bénéfice d'un objet représentent le coût de stockage d'un index et le coût de la charge en présence de cet index, la taille du sac à dos correspond à l'espace disque alloué par l'administrateur pour stocker les index sélectionnés et enfin l'ensemble N correspond à la configuration finale d'index.

3.1.3 Travaux de Frank *et al.*

Les travaux de Frank *et al.* , menés à l'Université de Georgia Tech, ont pour but de proposer un outil d'aide à la décision pour le choix d'index dans une base de données [FON92]. Cet outil utilise une configuration initiale d'index et une charge de requêtes. Un dialogue est établi entre l'outil de sélection d'index et l'optimiseur de requêtes du SGBD afin de calculer le gain en performance qu'apporte l'utilisation d'un index pour la charge. Le gain

est défini comme la différence entre le coût d'exécution des requêtes sans index et celui de leur exécution avec index.

L'estimation du coût d'un index est réalisée en faisant appel à l'optimiseur de requêtes. Le gain est représenté sous forme d'un graphe d'index. Le graphe d'index représenté à la Figure 3.1 correspond au dialogue suivant entre l'outil et l'optimiseur, donné en langage naturel pour des raisons de clarté et de compréhensibilité.

- Outil : Pour une requête q, quel est l'index à choisir parmi un ensemble d'index candidats $p_1 = \{a, b, c, d\}$ et quel est son coût d'utilisation ?
- Optimiseur : Je propose l'index $i_1 = \{b\}$. Le coût estimé pour la requête q utilisant cet index est $c_1 = 23$.
- Outil : Parmi les index qui restent, c'est-à-dire $p_2 = p_1 - i_1 = \{a, c, d\}$, quel est le meilleur index et son coût pour la requête q ?
- Optimiseur : L'index $i_2 = \{d\}$ avec un coût $c_2 = 27$.

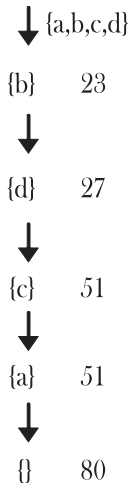


FIG. 3.1 – Graphe d'index simple

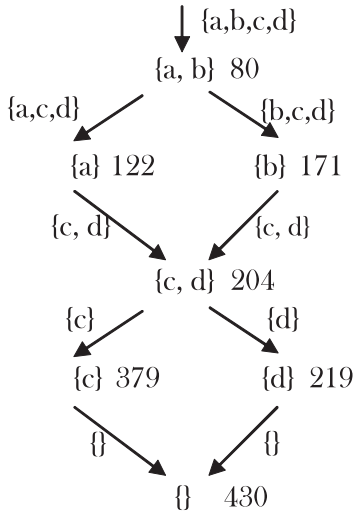


FIG. 3.2 – Graphe d'index simplifié

Ce jeu de questions-réponses continue jusqu'à ce qu'il n'y ait plus d'index à proposer ($i_k = \emptyset$). Dans le cas général, un sous-ensemble d'index peut être une configuration d'index pour une requête q. Par exemple, si $\{a, b, c, d\}$ est utilisé, le sous-ensemble $\{a, b, c\}$ peut être un bon index pour la requête q. Dans ce cas, un graphe d'index peut être construit en prenant en compte toutes les combinaisons possibles. Cependant, la construction de ce graphe n'est

pas faisable vu le nombre exponentiel des sous-ensembles d'index à générer dans chaque nœud. Une solution a été proposée pour remédier à ce problème en utilisant la propriété d'inclusion des ensembles. Si l'optimiseur de requêtes propose une configuration $\{a, b\}$ pour un ensemble d'index $\{a, b, c, d\}$, il n'est pas judicieux d'interroger l'optimiseur de requêtes pour chercher une configuration parmi l'ensemble d'index a, b, c inclus dans $\{a, b, c, d\}$, car elle sera forcément $\{a, b\}$. La Figure 3.2 montre un exemple de graphe d'index simplifié.

Nous résumons cette méthode par les cinq points suivants.

1. Une requête de la charge est soumise à l'optimiseur de requêtes avec un ensemble d'index initial.
2. L'ensemble des index utilisés pour la requête courante est stocké avec le gain de performance pour la requête.
3. De nouveaux ensembles d'index sont générés et l'étape 2 est réitérée jusqu'à effectuer un parcours séquentiel.
4. Les gains en performance de chaque index sont additionnés.
5. Les index présentant un gain total positif sont enfin proposés à l'utilisateur.

3.1.4 Travaux de Choenni *et al.*

Le problème de sélection d'index, tel qu'il a été posé dans [CBC93b], est basé sur un modèle mathématique. En effet, l'ajout d'un index dans une configuration peut augmenter son coût, comme il peut aussi le réduire. Ce comportement est similaire à celui des fonctions mathématiques dites super–modulaire et sous–modulaire. Nous donnerons ultérieurement la définition de ces fonctions.

À travers l'exemple suivant, nous montrons le comportement de ces fonctions. Considérons la table `Persons` de 400000 n-uplets composée des attributs `Pers_ID, Pers_name, Pers_salary, Pers_age, Pers_sexe, Pers_education` et la requête suivante :

```
select  pers_name
from    persons
where   oers_education = 'second'
        and pers_salary = 2000 and pers_age = 21
```

Supposons que les facteurs de sélectivité SF des attributs `Pers_education`, `Pers_salary` et `Pers_age` sont respectivement $SF_{Pers\_education} = \frac{1}{10}$, $SF_{Pers\_salary} = \frac{1}{40}$ et $SF_{Pers\_age} = \frac{1}{50}$, et qu'une page disque contient vingt n-uplets. Étudions le comportement de l'ajout d'un index construit sur l'attribut `Pers_education` dans les trois ensembles d'index suivants : $A = \emptyset$, $A' = \{Pers\_salary\}$ et $A'' = \{Pers\_salary, Pers\_age\}$. Les coûts, qui représentent le nombre de pages lues, dans chaque configuration, avec ajout et sans ajout d'un index construit sur l'attribut `Pers_education`, sont résumés au Tableau 3.1.

Configuration	sans ajout	avec ajout
A	$\frac{400\,000}{20} = 20\,000$ pages	$\frac{400\,000}{20} = 20\,000$ pages
A'	$\frac{400\,000}{40} = 10\,000$ pages	$\frac{400\,000}{40*10} = 1\,000$ pages
A''	$\frac{400\,000}{40*50} = 200$ pages	$\frac{400\,000}{40*50*10} = 20$ pages

TAB. 3.1 – Résultat de calcul des coûts

L'ajout d'un index construit sur l'attribut `Pers_education` dans la configuration vide n'apporte aucun gain. Le coût avant et après l'ajout de cet index reste inchangé. En revanche, l'ajout du même index dans la configuration A' apporte un gain important. En effet, le coût passe de 10000 pages à 1000 pages. Le même ajout dans A'' apporte un gain, quoique moindre comparé au précédent. Ces observations montrent les caractéristiques de base de la fonction de coût similaires aux caractéristiques des fonctions super et sous–modulaires. Nous détaillons dans la suite le formalisme mathématique de ces fonctions et son adaptation au problème de sélection d'index.

Définition 3.1.1 (Fonction super et sous-modulaire) *Soient N un ensemble fini et C une fonction définie comme suit $C : \mathbb{P}(N) \rightarrow \mathbb{R}$, où $\mathbb{P}(N)$ désigne l'ensemble des parties de N. La fonction C est dite super–modulaire si :*

$$C(X) + C(Y) \leq C(X \cup Y) + C(X \cap Y), \forall X, Y \subseteq N$$

et sous–modulaire si :

$$C(X) + C(Y) \geq C(X \cup Y) + C(X \cap Y), \forall X, Y \subseteq N.$$

En posant $X = I \cup \{i'\}$ et $Y = I'$, et pour tout $I \subseteq I' \subseteq N$ avec $i' \notin I'$, où I, I'

sont des sous-ensembles d'index de l'ensemble de tous les index candidats N et i' un index n'appartenant pas à I', les deux formules suivantes sont obtenue.

$$C(I' \cup \{i'\}) - C(I') \geq C(I \cup \{i'\}) - C(I), \forall I \subseteq I' \subseteq N \wedge i' \notin I'$$

$$C(I' \cup \{i'\}) - C(I') \leq C(I \cup \{i'\}) - C(I), \forall I \subseteq I' \subseteq N \wedge i' \notin I'$$

Les deux équations, ci-dessus, montrent la relation existante entre le problème de sélection d'index et la définition des fonctions super– et sous–modulaires. Elles peuvent être exploitées pour faire la distinction entre les index candidats qui font partie de la configuration optimale et ceux qui n'en font pas partie.

Les propriétés suivantes fournissent des techniques évitant d'effectuer une recherche exhaustive afin de trouver l'ensemble d'index $Config_I \subseteq N$ qui minimise la fonction C, c'est-à-dire, de trouver un ensemble $Config_I \subseteq N$ tel que $C(Config_I) \leq C(I)$ pour tout $I \subseteq N$.

Les deux premières propriétés sont spécifiques à une fonction super–modulaire et les deux dernières à une fonction sous–modulaire.

Proposition 3.1.1 *Supposons un index i' à ajouter à l'ensemble d'index I. Si la valeur de C ne décroît pas suite à cet ajout, elle ne décroît alors pas non plus en ajoutant un index i' à tout ensemble I' contenant I. Cela signifie que l'index i' n'appartient pas à la configuration d'index optimale $Config_I$.*

Proposition 3.1.2 *Supposons que l'index i' soit à éliminer de I. Si la valeur de C n'est pas réduite suite à cette élimination, la valeur de C ne peut pas l'être pour tout index i' éliminé de l'ensemble I' contenu dans I. Cela signifie que i' appartient à la configuration optimale.*

Proposition 3.1.3 *Si l'ajout d'un index i' est avantageux pour un ensemble d'index I, il l'est aussi pour tout ensemble d'index I' contenant I. Cela signifie que i' fait partie de la configuration optimale.*

Proposition 3.1.4 *Si un index i' réduit le coût d'un ensemble d'index I suite à son élimination de cet ensemble, ce coût est aussi réduit pour tout ensemble I' contenu dans I. Cela signifie que i' ne fait pas partie de la configuration optimale.*

Notons que les propositions 3.1.1 et 3.1.4 sont équivalentes. Il est de même pour les propositions 3.1.2 et 3.1.3.

L'exemple des Figures 3.3 et 3.4 illustre l'emploi de la fonction super–modulaire. Chaque graphe (treillis) représente l'espace de recherche à explorer pour construire une configuration d'index optimale à partir des attributs de la table $R(a, b, c, d)$ sur laquelle est définie une charge. Les nombres se trouvant sur les nœuds du graphe de la Figure 3.4 symbolisent le coût d'exécution de la charge estimé par la fonction de coût C.

La fonction de coût qui estime le temps d'exécution de la charge se comporte comme une fonction super-modulaire. Sans appliquer les propositions super-modulaires d'optimisation, chaque nœud du graphe représenté à la Figure 3.3 est évalué (2^4 évaluations). En appliquant les proposions 3.1.1 et 3.1.2, le graphe de la Figure 3.3 est réduit au graphe présenté à la Figure 3.4.

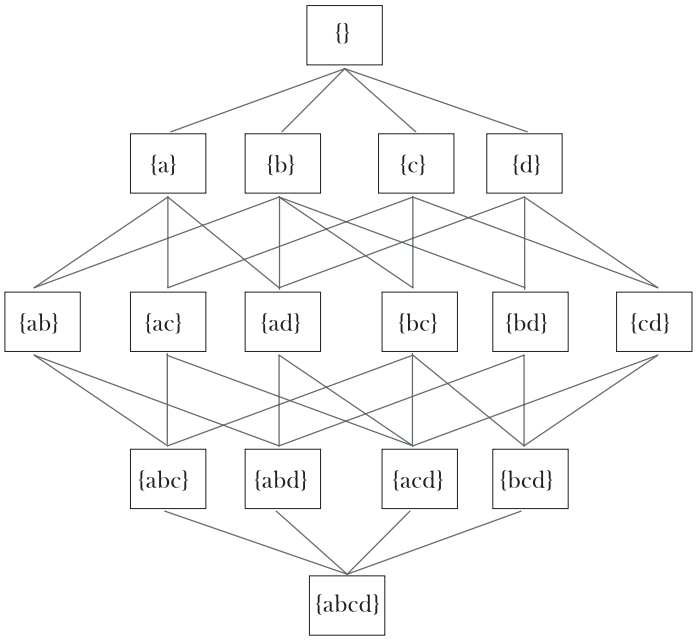


FIG. 3.3 – Exemple de graphe représentant un espace de recherche exhaustif

Le processus d'optimisation d'une fonction super–modulaire commence à partir de l'ensemble vide $A = \emptyset$ correspondant à la borne inférieure du graphe et de l'ensemble total $\{a, b, c, d\}$ correspondant à la borne supérieure du graphe. L'ajout de l'index $i' = b$ à la

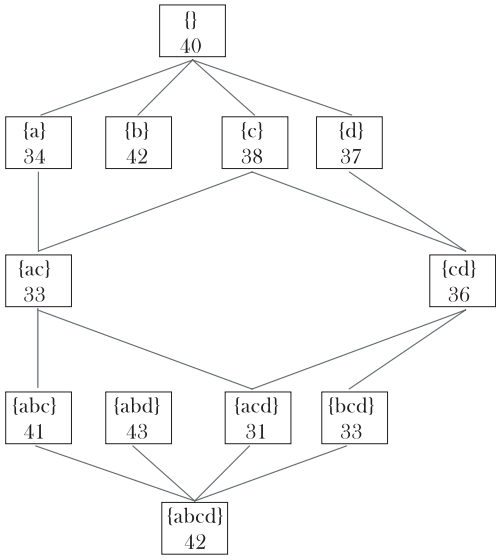


FIG. 3.4 – Exemple de graphe représentant un espace de recherche réduit

borne inférieure augmente le coût (le coût passe de 40 unités à 42 unités), alors, et en appliquant la proposition 3.1.1, b est éliminé de la configuration optimale. En revanche, partant de l'ensemble $\{a, b, c, d\}$ et en éliminant $i' = c$ de cet ensemble, le coût augmente. Cela signifie, d'après la proposition 3.1.2, que l'index c appartient à la configuration optimale. Cela a pour conséquence de considérer que les sous-ensembles qui ne contiennent pas b ne doivent pas contenir c. Partant de l'ensemble des sous-ensembles restants, ce processus est réitéré pour un niveau plus bas jusqu'à ce qu'il n'y ait plus d'index à ajouter ou à enlever.

3.1.4.1 Algorithme de sélection d'index

L'algorithme de sélection d'index prend en entrée une fonction de coût C[1], une charge Q_{red}[2] extraite de la charge Q, un ensemble d'attributs à indexer selon l'administrateur du système et une configuration initiale d'index pertinente pour chaque requête de la charge.

L'algorithme renvoie un ensemble d'index avantageux et un autre ensemble d'index désavantageux, pour des groupes de requêtes constituant des sous–ensembles de Q_{red}. Un index

[1] Trois fonctions de coût ont été proposées.

[2] Q_{red} est l'ensemble de requêtes de la charge Q pour lesquelles aucun index parmi la configuration initiale n'est avantageux.

avantageux appartient à la configuration d'index optimale et, par opposition, un index désavantageux n'appartient pas à la configuration optimale.

Pour que l'algorithme fonctionne, il est nécessaire de déterminer si un index exploité par une requête de la charge appartient à la partie super–modulaire ou sous–modulaire de la fonction de coût. Ensuite, les propositions des fonctions super-modulaires (respectivement, sous–modulaires) sont utilisées pour réduire l'espace de recherche. L'évaluation des coûts des requêtes de Q_{red} mène généralement à une décomposition des requêtes en deux sous–groupes. Un sous-groupe G_1 contient les requêtes rendant la fonction de coût C super–modulaire pour l'ensemble d'index exploité par ces requêtes, et un deuxième sous-groupe G_2 contenant d'autres requêtes rendant C sous–modulaire pour l'ensemble d'index exploité par ces requêtes.

Pour le groupe G_1, l'optimisation est réalisée en utilisant les propriétés d'une fonction super–modulaire. La borne inférieure du graphe de l'espace de recherche à explorer est appliquée.

Le processus d'optimisation de G_2 peut se poursuivre en appliquant les propriétés d'une fonction super–modulaire. En revanche, le groupe G_2 peut être traité de plusieurs manières. La méthode proposée est de réitérer le même processus que celui appliqué pour l'ensemble des requêtes Q_{red} en ne prenant que les opérations de G_2.

Notons que l'algorithme ne propose pas une configuration d'index pour la charge Q, mais il propose un sous-ensemble d'index avantageux et désavantageux d'un sous-groupe d'opérations. Cependant, il est indispensable de combiner tous les ensembles d'index avantageux et désavantageux pour avoir un ensemble d'index avantageux et un autre d'index désavantageux, et d'ensuite résoudre le problème de sélection d'index avec une recherche exhaustive. Le temps et la faisabilité de cette recherche dépend du nombre d'index engendré.

3.1.4.2 Travaux de Whang – Algorithmes ADD and DROP

L'algorithme DROP de K. Whang commence la sélection d'index en considérant l'ensemble des index possibles [Kin74]. À chaque étape, il élimine l'index qui engendre la plus grande décroissance du coût. Quand la valeur de la fonction ne peut plus être réduite en éliminant un seul index, l'algorithme tente d'éliminer deux index à la fois, trois index à la

fois, et ainsi de suite, jusqu'à ce que ça ne soit plus possible. La technique d'élimination d'un seul index à la fois est l'application de la propriété 3.1.4 d'une fonction sous–modulaire.

L'algorithme ADD initialise le processus d'optimisation par l'ensemble vide. À chaque étape, il ajoute un index susceptible de réduire la fonction de coût et s'arrête quand il n'y a plus de réduction de coût. Cette règle est similaire à la propriété 3.1.3 d'une fonction sous–modulaire.

3.1.4.3 Sélection des index primaires et des index secondaires

D'autres travaux de Choenni *et al.* [CBC93a] traitent le problème de sélection d'index en distinguant les index primaires et les index secondaires, le but est de trouver un ensemble d'index optimal ou proche de l'optimal en considérant au plus un index primaire et zéro ou plusieurs index secondaires. Ces travaux supposent que la fréquence d'insertion et de suppression des n-uplets est telle que le nombre total des n-uplets de chaque table reste constant dans deux choix consécutifs d'un ensemble d'index. De plus, les attributs sont mutuellement indépendants et leurs valeurs suivent une distribution uniforme.

Un modèle de coût général $C(\{A\}, \alpha_p, q)$ est utilisé pour estimer le coût d'utilisation d'un index donné, où $\{A\}$, α_p et q représentent respectivement l'ensemble des index secondaires, un index primaire et une opération de la charge. La fonction générale de coût proposée se décompose en deux fonctions : une fonction de coût pour la sélection des index primaires et une autre fonction similaire à celle proposée dans [CBC93b] pour la sélection des index secondaires. Deux méthodes heuristiques, dans le but d'optimiser la fonction de coût, ont été proposées.

Heuristique 1

La première méthode choisit d'abord un index primaire et ensuite les index secondaires. Pour chaque attribut α_p la fonction de coût $C(\emptyset, \alpha_p, q)$ est évaluée (sans index primaire), et l'attribut α_p ayant un coût minimal est choisi comme index primaire. Cet index est injecté dans la fonction de coût générale et elle est optimisée pour la sélection d'index secondaires. La méthode heuristique 1 divise la charge en deux parties. Le choix d'un index primaire signifie qu'une partie de la charge n'est traitée efficacement avec cet index. La sélection

d'index secondaires est basée sur la réduction du coût du reste de la charge en prenant en compte la partie précédente.

Heuristique 2

Dans cette méthode, un ensemble optimal d'index secondaires A_{opt} est déterminé sans aucun index primaire en utilisant les algorithmes proposés dans [BP90, CBC93b]. La fonction générale de coût est optimisée en choisissant l'attribut qui apporte la plus importante réduction du coût comme index primaire. Le succès de la méthode heuristique 1 dépend du choix de l'index primaire. Si le choix est "bon", la configuration éventuelle d'index l'est aussi. L'heuristique de la méthode 1 mène à une solution proche de l'optimal car le gain apporté par un index primaire est "fortement" dépendant de l'ensemble des index secondaires. L'heuristique de la méthode 2 peut mener également à une solution plus proche de l'optimal car le gain apporté par un index secondaire dépend fortement du choix de l'index primaire.

3.1.5 Travaux de Gündem

Gündem pose le problème des choix multiples pour la sélection d'index [Gun99]. Il propose une méthode qui prend en compte le fait qu'un attribut donné peut être indexé suivant plusieurs techniques et que, par conséquent, les index construits sur cet attribut suivant différentes techniques apportent des gains et occupent des espaces de stockage différents. En effet, un index candidat peut être un B-arbre, un index d'ordre, un ensemble d'index partiels, etc. L'ensemble des index associés à un attribut donné est appelé une classe d'équivalence. L'ensemble de ces classes constitue une partition de l'ensemble des index correspondants aux attributs des tables de la base considérée.

La méthode présentée utilise un ensemble d'index candidats fourni par l'administrateur pour y sélectionner un sous-ensemble I qui minimise, suivant une erreur tolérée, le coût de traitement des opérations de mise à jour et de sélection sans violer la contrainte d'espace de stockage. Pour utiliser cette méthode, les choix multiples d'index pour chaque attribut et les fréquences respectives des sélections et des mises à jour sont supposés fournis par l'administrateur du système.

Une première étape consiste à effectuer une optimisation locale pour calculer un ensemble

disjoint d'index I. Ce dernier est défini telle que pour chaque paire d'index différents de cet ensemble, les attributs qui ont servi à leur construction sont également différents. De ce fait, il ne peut y avoir au plus q'un seul index par attribut. Pour chaque index candidat, le gain total est calculé en utilisant une fonction de coût. Ce gain est la différence entre les gains apportés par l'utilisation d'un index pour les opérations de sélection et de mise à jour et le coût de construction de cet index. Les index ayant un gain négatif sont éliminés de l'ensemble d'index candidats. Le résultat de cette étape est un ensemble disjoint d'index I (un index par classe d'équivalence) contenant l'index candidat qui maximise le gain.

La deuxième étape de cette approche consiste à réaliser une optimisation globale. Le coût total d'utilisation d'un ensemble d'index I est la différence entre le coût de traitement des opérations de la charge sans index et le gain total apporté par tous les index dans I. Partant de l'ensemble des classes d'équivalence d'index données (au plus une classe d'équivalence pour chaque attribut) et de la taille de l'espace maximum de stockage S, il s'agit de trouver l'ensemble disjoint d'index I (parmi les ensembles disjoints d'index possibles) de l'étape précédente tels que :

- la fonction de gain total ait sa valeur minimum ;
- l'espace de stockage requis pour les index dans I soit inférieur à S.

Gündem assimile le problème de sélection d'index ainsi posé au problème du sac à dos binaire à choix multiple (*multiple choice 0-1 knapsack optimization*). Comme le problème posé est NP-complet, une solution approximative avec un taux d'erreur fixé par l'utilisateur a été proposée. En résumé, la recherche d'une configuration d'index I proche de la configuration optimale revient à résoudre le problème du sac à dos binaire.

3.1.6 Travaux de Chaudhuri *et al.*

Le projet de recherche AutoAdmin a été lancé par Microsoft dans le but de trouver de nouvelles techniques pour auto-administrer une base de données, tout en assurant des performances comparables à celles d'une base de données gérée uniquement par un administrateur humain [Mic01].

L'outil de sélection d'index IST (*Index Selection Tool*) [CN97] extrait un ensemble d'index, dit configuration, souhaitable pour une base et une charge données. Un index peut être

mono ou multi-attributs. Pour extraire une configuration, il faut être en mesure de comparer l'utilité de deux configurations. Étant données une configuration et une charge, la somme des coûts estimés pour toutes les requêtes de la charge est utilisée comme une mesure d'utilité.

IST, dont l'architecture générale est représentée à la Figure 3.5, ne considère que les index mono-attribut dans la première itération. Dans la seconde itération, il ne considère que les index à un seul attribut (trouvés dans la première étape) et à deux attributs ; et ainsi de suite dans les itérations suivantes.

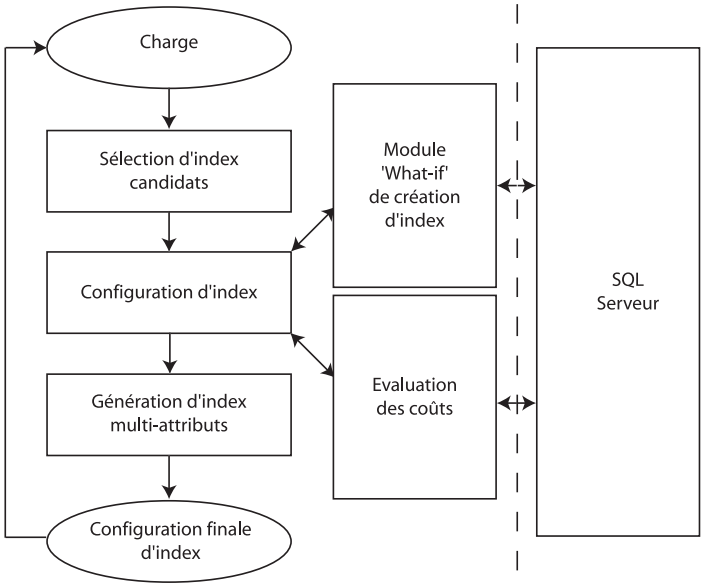


FIG. 3.5 – Architecture générale d'IST

L'algorithme de sélection d'index passe par trois phases. Le module de sélection des index candidats élimine un nombre important d'index qui n'apportent aucun bénéfice aux requêtes de la charge. Le module de sélection de configurations recherche les différents sous-ensembles d'index et extrait la configuration optimale. Le module générateur d'index multi-attributs génère les attributs multi-attributs à considérer à chaque nouvelle itération. IST se base dans son choix sur le coût estimé par l'optimiseur de requêtes du SGBD [CN97]. Cette tâche est assurée par le module d'évaluation du coût qui maintient une table des coûts représentée à la Figure 3.6. Dans le cas où un index à évaluer ne se trouverait pas dans la base de données, IST a besoin de simuler sa présence pour l'optimiseur. IST fait donc appel au module *What–if creation index* [CN98].

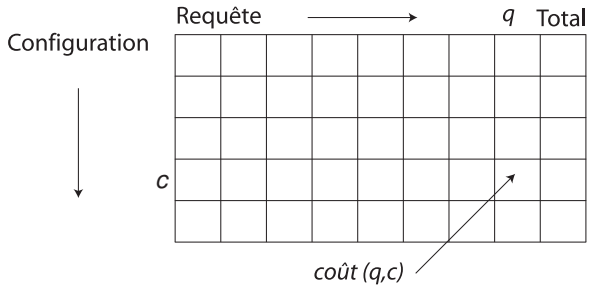


FIG. 3.6 – Table d'évaluation des coûts

3.1.6.1 Sélection des index candidats

Pour chaque requête de la charge, IST considère l'ensemble des attributs indexables comme les index candidats de départ. Les attributs indexables sont ceux présents dans une des clauses : `Where`, `Group by`, `Order By` et les attributs mis à jour dans une requête `Update`. Ensuite, IST construit une configuration d'index à partir de ces index candidats. La construction se fait grâce aux deux modules : *cost–evaluation* et *What–if index creation* [CN98] qui calculent le coût des différentes configurations d'index de l'ensemble de départ. La meilleure configuration est retenue. L'ensemble d'index résultat est l'union des configurations obtenues pour chaque requête.

3.1.6.2 Élagage des configurations d'index

L'élagage permet de supprimer un certain nombre d'index qui ne sont d'aucune utilité ou qui amènent un gain de performance peu significatif. À partir de n index candidats construits dans l'étape précédente, les k meilleurs sont sélectionnés à l'aide d'un algorithme glouton. Ce module est en dialogue permanent avec l'optimiseur de requêtes et suit les étapes suivantes :

1. initialisation avec la meilleure configuration de taille m petite ($m << k$) ;
2. augmentation de la configuration courante avec l'index minimisant le coût ;
3. tant que le coût diminue et que le nombre des k index n'est pas atteint, revenir à l'étape 2.

Le résultat est un ensemble d'index constitué de l'union des configurations obtenues après élagage.

3.1.6.3 Génération des configurations d'index multi-attributs

Les index obtenus après élagage sont mono-attribut. Pour construire les index multi-attributs, IST utilise ces index mono-attribut. Deux démarches : MC-LEAD et MC-ALL ont été proposées. Nous expliquons la méthode de construction d'un index multi-attributs de taille 2 à partir des deux attributs a et b.

- **MC-LEAD** : Avec MC-LEAD, il doit exister un index mono-attribut sur a, mais pas forcément sur b (qui doit tout de même être un attribut indexable). Donc un index candidat est un index mono-attribut combiné avec un attribut indexable.
- **MC-ALL** : Avec MC-ALL, il doit exister un index mono-attribut sur a et sur b. Un index candidat est donc une combinaison d'index mono-attributs.

L'ensemble d'index construit par MC-ALL est donc inclus dans celui construit par MC-LEAD. Les index de taille supérieure à deux sont construits de la même façon.

3.1.7 Travaux de Kratica *et al.*

Kratica *et al.* proposent un algorithme génétique pour résoudre le problème de sélection d'index [KLT03]. Dans la suite, nous présentons le modèle mathématique du problème de sélection d'index et les aspects importants de l'algorithme génétique proposé : encodage, fonction objectif d'évaluation et opérateurs génétiques de croisement, de mutation et de sélection.

3.1.7.1 Modèle mathématique

Soient $I = \{1, 2, ..., n\}$ un ensemble d'index et $Q = \{1, 2, ..., m\}$ un ensemble de requêtes. Chaque index peut être construit ou non ; la construction d'un index i requiert un coût $f_i > 0$. Une configuration d'index $P = \{1, 2, ..., p\}$ de taille $k \in P$ (k est le nombre total d'index de cette configuration) est associée à un sous-ensemble d'index $I_k \subseteq I$. Une configuration est dite active si tous ses index sont construits. Si une configuration $k \in P$ est active, le gain apporté par celle-ci lors de l'exécution d'une requête $q \in Q$ est $g_{qk} \geq 0$. Le but est de construire des index tel que le temps total nécessaire pour l'exécution de toutes les requêtes soit minimal, c'est-à-dire, le total des gains est maximal. Formellement, le problème de sélection d'index, formulé dans les notations utilisées pour les algorithmes génétiques, est

posé comme suit :

$$\max\left(\sum_{q\in Q}\sum_{k\in P} g_{qk}.x_{qk} - \sum_{i\in I} f_i.y_i\right)$$

où :

$$\sum_{k\in P} x_{qk} \leq 1, q \in Q$$

$$x_{qk} \leq y_i, x_{qk}, y_q \in \{0,1\}, q \in Q, i \in I, k \in P.$$

Dans cette formulation, chaque solution est représentée par un ensemble d'index construits $Config_I \subseteq I$, où y représente un vecteur de caractéristiques de la solution $Config_I$; c'est-à-dire :

$$y_i = \begin{cases} 1 & \text{si } i \in Config_I \\ 0 & \text{sinon.} \end{cases}$$

Pour chaque paire requête–configuration $(q,k) \in Q \times P$:

$$x_{qk} = \begin{cases} 1 & \text{si la requête } q \text{ utilise la configuration k} \\ 0 & \text{sinon.} \end{cases}$$

3.1.7.2 Algorithme génétique pour la sélection d'index

Dans [KLT03], l'algorithme génétique proposé s'applique sur une population d'individus N_{pop}. N_{elit} est le nombre d'individus élus pour survivre à la prochaine génération. Les valeurs de la fonction objectif à évaluer sont le temps de traitement des requêtes. Le temps calculé est stocké dans une table cache de taille N_{cache} pour accélérer le temps des calculs. Avant de calculer une valeur objectif d'un individu, la table cache est consultée en premier.

3.1.7.3 Encodage et fonction objectif

Pour chaque requête q, trois variables sont stockées : le nombre des valeurs de gains strictitement positives, les valeurs de ces gains et les indices des configurations d'index

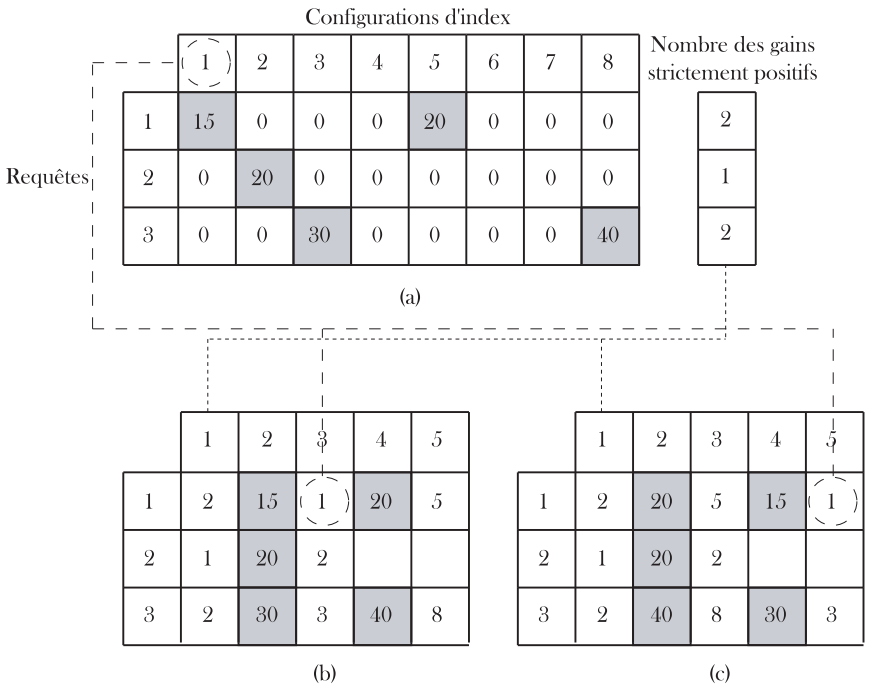


FIG. 3.7 – Encodage et valeurs de la fonction objectif

apportant ces gains. La Figure 3.7 montre un exemple d'encodage et d'évaluation des valeurs de la fonction objectif.

Dans la Figure 3.7 (a), les lignes représentent les requêtes et les colonnes les indices des configurations d'index possibles. Les valeurs des cellules sont les gains g_{ik}. Un gain nul signifie que la configuration d'index n'est pas bénéfique à la requête correspondante. La matrice de la Figure 3.7 (a) est réduite de telle façon que la première colonne donne le nombre des valeurs strictement positives du gain, suivi des valeurs de gain et de l'indice de la configuration apportant ce gain comme le montre la Figure 3.7 (b). La Figure 3.7 (c) est le résultat du tri suivant les valeurs décroissantes du gain de la matrice précédente.

Pour chaque requête q, la configuration k^* apportant un gain maximal est recherchée. Enfin, la différence entre la somme des gains apportés par une configuration pour l'ensemble de toutes les requêtes et le coût de maintenance des index de la configuration est calculée. Si la différence est négative, la construction des index n'apporte aucun bénéfice. Dans ce cas, $y_i = 0$ pour tout $i \in I$ et il n'existe aucune configuration active ; c'est-à-dire, la valeur de la fonction objectif est à zéro.

3.1.7.4 Opérateurs génétiques

Pour éviter une convergence prématurée, les individus multiples sont éliminés de la population. Pour recombiner deux individus, l'opérateur de croisement uniforme, décrit dans [Sys89], est utilisé. La recombinaison d'une paire d'individus sélectionnés a une probabilité p_{cro}.

Pour un taux de mutation donné, le nombre de gènes en mutation dans une génération est prédit en utilisant le théorème central limite pour une distribution gaussienne. La procédure de mutation est réalisée uniquement sur le nombre précédent de gènes choisis aléatoirement. Le taux de mutation change d'une génération à une autre suivant une loi donnée dans [KLT03]. La méthode de sélection utilisée est la *fine grained tournament selection* proposée dans [Fil00].

3.1.8 Travaux de Feldman *et al.*

DINNER est un outil à base de connaissances qui assiste l'administrateur dans la sélection d'index [FR03]. Étant donnés un ensemble de tables, leur statistiques et un ensemble de requêtes sur ces tables, DINNER recommande une configuration d'index qui contient un index primaire et un ensemble d'index secondaires.

Un SGBD peut résoudre une requête suivant plusieurs chemins d'accès [3] possibles, correspondant aux index disponibles ou npn. Pour trouver les index possibles, les chemins d'accès qui utilisent ces index, et le coût des chemins d'accès, DINNER utilise une base de connaissances extraite de différentes sources : l'expertise de l'administrateur de la base de données, le manuel du SGBD utilisé, des cours sur l'administration des bases de données, etc. Ces connaissances sont formalisées en utilisant un schéma de représentation de connaissances.

DINNER construit pour chaque requête un graphe représentant l'ensemble des solutions possibles que peut utiliser cette requête. Les requêtes de jointures sont décomposées en plusieurs requêtes définies sur une seule table. Les feuilles du graphe d'une jointure contiennent les requêtes définies sur une seule table, représentée par un autre graphe qui lui est propre. La racine du graphe représente l'ensemble des solutions possibles. En revanche, un nœud du

[3] Il existe généralement plusieurs méthodes pour calculer la réponse d'une requête. Ces méthodes sont appelées chemins d'accès. Le choix d'un chemin d'accès pour le traitement d'une requête est du ressort de l'optimiseur de requêtes.

graphe représente les solutions intermédiaires. Il existe deux sortes de nœuds : des nœuds OR et des nœuds AND. Un nœud parent est dit OR si ses fils immédiats offrent plusieurs solutions alternatives à ce nœud parent. Par contre, il peut être un nœud AND si les solutions qu'il représente sont composées des solutions représentées par ses fils immédiats.

La Figure 3.8 illustre une représentation sous forme d'un graphe de la requête de jointure donnée par la requête (q1) ci-dessous. Le nœud A de la Figure 3.8 correspond à une solution réalisant une jointure où l'attribut `item` de `T1` est dans une table OUTER et l'attribut `item-warehouse` de `T3` est dans une table INNER. Le nœud B correspond à la solution réalisant la jointure dans le sens inverse. Chacun de ces nœuds a deux fils : boucle imbriquée (*nested loop*) et tri–fusion (*merged–sort*), qui sont des nœuds AND. Chaque nœud possède deux fils représentant une requête définie sur une seule table.

```
(q1)    select  T1.name
        from    Item T1, Item-Wharehouse T3
        Where   T3.catalog-number = T1.catalog-number
                and T3.quantity > 10000
                and T1.supplier-code = 4
                and T1.price > 1000
```

Après la génération des graphes de chaque solution, DINNER trouve le chemin d'accès possédant une forte probabilité d'être choisi par l'optimiseur lors de l'exécution d'une requête et estime le temps de parcours et l'espace de stockage de ce chemin d'accès. L'estimation du coût se propage des feuilles vers la racine. Le résultat de cette étape est un graphe pour chaque requête avec les coûts et les espaces de stockage estimés.

L'étape suivante, consiste à élaguer le graphe en éliminant les mauvaises solutions suivant quatre critères. L'élimination ne se fait pas sur les nœuds descendant d'un nœud AND car ces nœuds représentent une solution partielle. Les quatre critères, sauf le premier, sont applicables sur le graphe de requêtes de jointure et le graphe de requêtes défini sur une seule table.

1. Le premier critère est appliqué uniquement sur les graphes des requêtes définies sur une seule table. Les nœuds ayant un coût estimé supérieur à celui d'un nœud utilisant

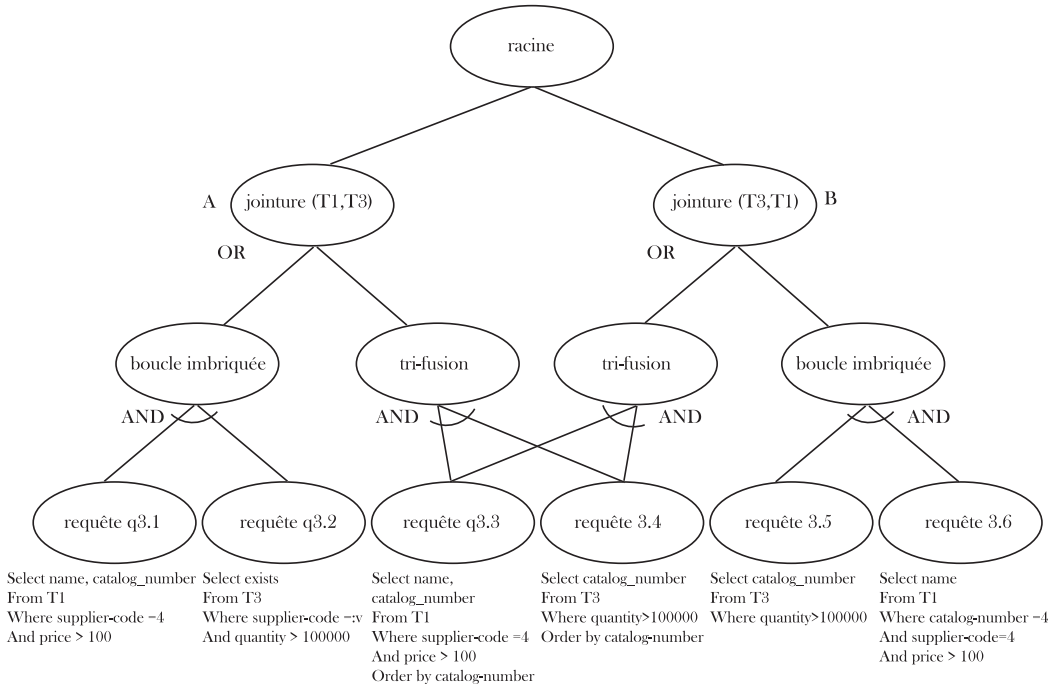


FIG. 3.8 – Graphe de solutions de la requête q1

une méthode de parcours séquentiel sont élagués, car la solution associée n'est pas utilisée par l'optimiseur de requêtes.

2. Le deuxième critère élimine les nœuds dont le coût est supérieur au coût des contraintes définies sur la requête.
3. Le troisième critère élimine les nœuds dont l'espace de stockage dépasse l'espace requis par l'administrateur du système.
4. Le dernier et quatrième critère concerne la fonction de compromis (*trade–off*) "coût–espace de stockage". Cette fonction est une combinaison linéaire (à coefficients positifs) du temps (coût) et de l'espace de stockage.

Après l'étape d'élagage, DINNER procède à l'unification, sans perte de solutions, des graphes des requêtes définies sur une seule table. Pour cela, les graphes sont transformés en une structure simple composée d'un seul nœud, d'un arbre à deux niveaux et d'un arbre à trois niveaux où la racine est un nœud OR et où les fils peuvent être des feuilles ou des nœuds AND ayant des feuilles pour fils. Le but de cette unification est de regrouper, sous un même arbre, les arbres résultant de la transformation et référençant la même table. Un

système de caches est préconisé pour accélérer l'unification.

Le coût des jointures est ensuite calculé sur la base des coûts des graphes des requêtes définies sur une seule table. Les meilleures solutions de la jointure sont ainsi trouvées. À la fin de cette étape, une configuration d'index, avec leur coût et leur espace de stockage, appartenant aux meilleurs graphes de jointure est obtenue. Le problème consiste à trouver dans cette configuration les meilleurs index. Ce problème est assimilé au problème du sac à dos.

3.1.9 Travaux de Valentin *et al.*

Le système DB2 Advisor, présenté à la Figure 3.9, est une boite noire pour recommander un ensemble d'index [VZZ+00]. Ce système admet en entrée une charge de requêtes SQL et renvoie en sortie la configuration d'index qui optimise les requêtes de la charge.

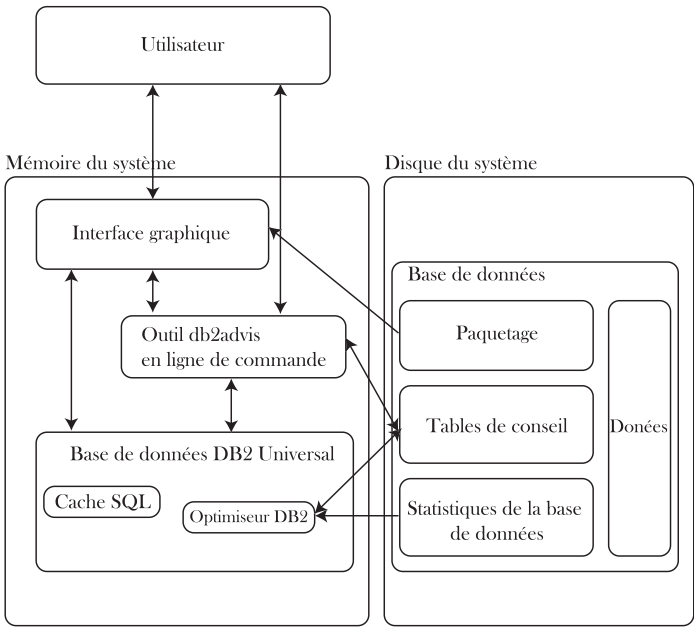


FIG. 3.9 – Architecture du système DB2advisor de Valentin *et al.*

Le système DB2advisor est composé :

- d'une interface graphique pour la sélection d'index,
- d'un outil en ligne de commande pour recommander des index,
- de l'optimiseur de requêtes de DB2 qui évalue et recommande des index,

– de tables de conseil, créées temporairement dans un but de conseil utilisées comme un moyen de communication entre l'optimiseur et l'outil db2advis.

L'invocation de DB2 Advisor peut se faire par deux modes : interface graphique ou ligne de commande. En mode graphique, l'utilisateur peut optimiser des requêtes se trouvant dans le cache, compilées dans des paquetages ou saisies. Ces requêtes sont stockées dans une table de conseil créée à cet effet. L'utilisateur peut aussi spécifier des contraintes sur l'espace disque alloué pour stocker les index à sélectionner ou le temps maximum nécessaire pour recommander des index par DB2 Advisor.

L'optimisation en mode graphique appelle l'outil db2advis. Pour chaque requête, cet outil invoque l'optimiseur de requêtes de DB2 afin de recommander ou d'évaluer des index. L'optimiseur stocke les index dans une autre table de conseil. L'outil db2advis peut être invoqué directement en ligne de commande.

L'algorithme de sélection d'index est vu comme une extension de l'optimiseur de requêtes de DB2. En effet, l'optimiseur de DB2 est amélioré par l'injection de plusieurs index, dits index virtuels, et de leurs métadonnées stockées temporairement le temps du processus d'optimisation dans une table de conseil. Les index virtuels sont extraits des requêtes de la charge en effectuant une analyse syntaxique. En présence de ces index et des statistiques de la base de données, l'optimiseur génère le plan d'exécution optimal d'une requête donnée. Si ce plan contient un ou plusieurs index, ces derniers sont recommandés.

La génération du plan optimal et la recommandation d'index sont réitérés pour chaque requête de la charge. Après la recommandation des index, l'outil db2advis modélise le problème de sélection d'index comme le problème du sac à dos.

3.1.10 Travaux de Golfarelli *et al.*

Golfarelli *et al.* proposent un système de sélection d'index dans les entrepôts de données [GRS02]. L'architecture du système est présentée à la Figure 3.10. La description des composants du système est la suivante.

– Le schéma logique décrit le schéma relationnel des tables de faits, des tables dimensions et éventuellement des vues matérialisées pré-sélectionnées par l'administrateur.
– La charge est un ensemble de requêtes à exécuter au niveau de l'entrepôt de données.

Les requêtes comportent des projections, des sélections, des jointures et des agrégations.

- Le volume des données contient des informations quantitatives sur les données, telles que la taille des tables de base et la cardinalité du domaine de leur attributs.
- Les contraintes du système incluent l'espace de stockage alloué pour stocker les index et la taille du cache utilisé lors de l'exécution des jointures.
- La borne de la charge met en correspondance chaque requête de la charge avec une référence de la table de faits utilisée pour résoudre cette requête.
- Les attributs qui sont susceptibles d'être de bons index sont appelés attributs indexables. L'ensemble des index candidats associe chaque attribut indexable à une technique d'indexation qui lui convient le mieux. Dans cette approche, les index créés sont tous construits sur un seul attribut, sauf les index construits sur une clé primaire d'une table de faits. L'ensemble des index optimaux à créer est un sous-ensemble des index candidats qui s'ajoutent aux index construits sur les clés primaires des tables de faits et des dimensions.

Les éléments liés au traitement sont représentés sous forme d'ellipse sur la Figure 3.10. Leur description est la suivante.

- Le navigateur d'agrégats exploite la charge et le schéma logique de l'entrepôt de données pour sélectionner la meilleure vue avec laquelle chaque requête doit être résolue, sans prendre en compte les index.
- Le sélecteur d'attributs indexables se base sur la structure des requêtes pour déterminer quels sont les attributs des tables dimensions qui sont pertinents à indexer.
- Le sélecteur d'index candidats évalue quelle est la technique d'indexation qui convient le mieux à un attribut indexable.
- Le sélecteur d'index optimaux détermine les index à créer parmi les index candidats.
- L'évaluateur de coût estime aussi bien le coût des index que celui d'un plan d'exécution.
- Le générateur de plans partant du schéma physique de l'entrepôt de données, d'une requête et d'éventuelles vues, génère le meilleur plan d'exécution de cette requête.

La génération des plans d'exécution est basée sur un modèle de règles. Étant données une requête, la tables de base et les vues, l'optimiseur de requêtes construit le plan d'exécution estimé comme le meilleur. Ce modèle est basé sur l'optimiseur de requêtes d'Informix Red

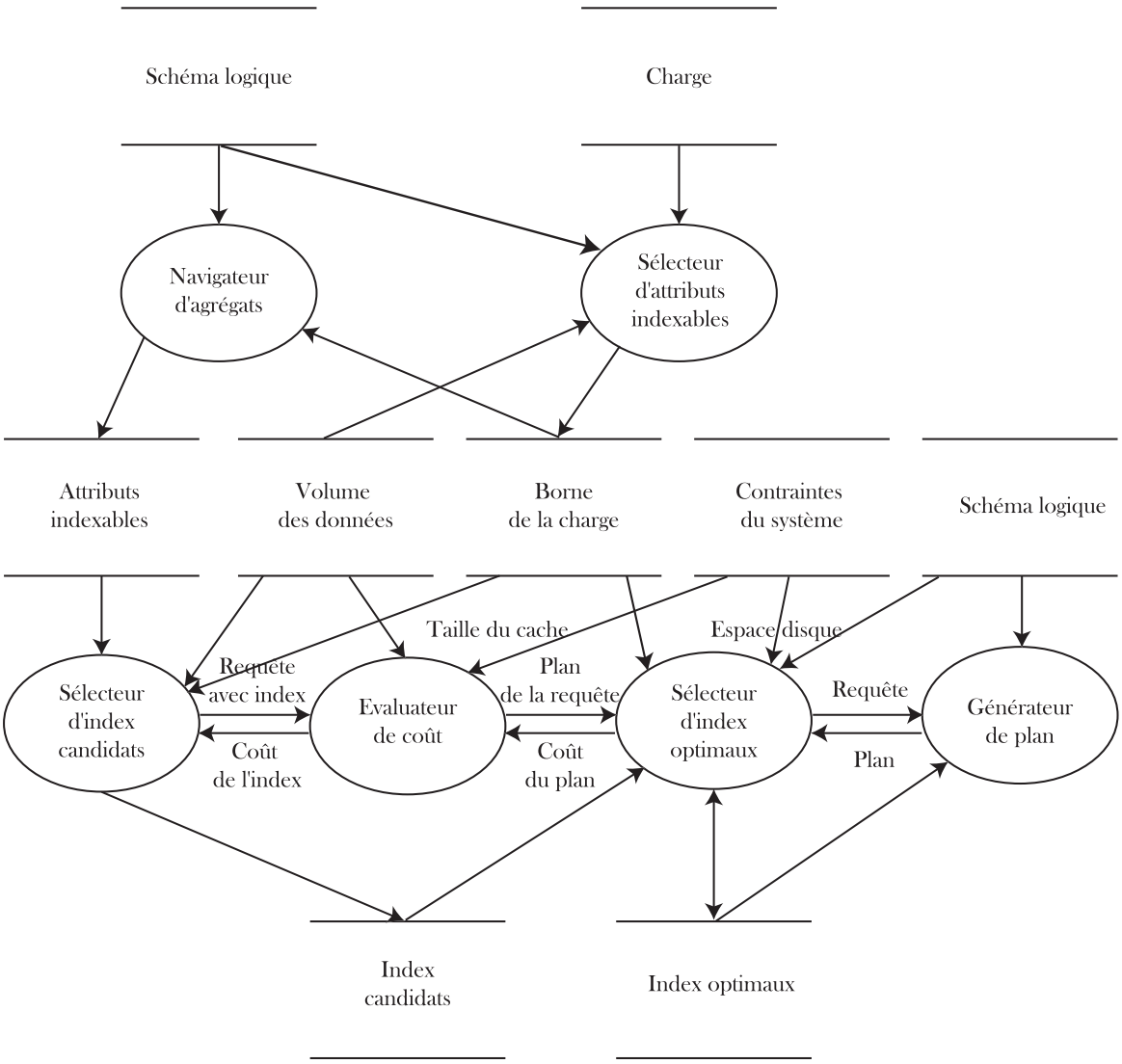


FIG. 3.10 – Architecture du système de sélection d'index proposé par Golfarelli *et al.*

Brick 6.0.

Un plan d'exécution est vu comme une séquence d'opérateurs élémentaires appliqués sur le schéma physique de l'entrepôt de données. Chaque opérateur est caractérisé par une entrée composée des tables de bases, d'index ou des résultats d'un autre opérateur, et d'une sortie. Certains opérateurs admettent des prédicats locaux afin de filtrer la sortie.

Ces opérateurs sont décrit brièvement ci-dessous.

- Le parcours séquentiel d'une table retourne un ensemble de n-uplets qui satisfont un prédicat de sélection.
- Le parcours d'un index accède à un index et recherche la liste des identifiants des n-uplets qui satisfont un prédicat de sélection.
- L'accès à une table retourne les n-uplets correspondant à une liste d'identifiants.
- L'accès à un index retourne une liste d'identifiants correspondant à une clé de l'index.
- La jointure est réalisée par hachage hybride pour les n-uplets de différentes tables.
- L'intersection des identifiants de deux listes retourne les n-uplets qui satisfont plusieurs prédicats de sélection.

Après la génération des plans d'exécution d'une requête, il est nécessaire de choisir le meilleur. Ce choix est orienté par un modèle de coût. Le coût d'un plan est exprimé en nombre de pages disques à lire pour répondre à une requête donnée.

L'évaluation d'un plan d'exécution requiert une fonction de coût pour chaque opérateur. Le coût d'un opérateur dépend de la cardinalité des résultats trouvés par l'opérateur qui le précède. Le coût total d'un plan est la somme des coûts de tous ses opérateurs. Le détail du modèle de coût est présenté dans [GRS02]. Nous présentons dans la suite l'algorithme de sélection d'index.

L'algorithme proposé par Golfarelli *et al.* procède en trois étapes distinctes. La première consiste à initialiser l'ensemble des index candidats I et optimaux O, ainsi que l'espace de stockage S nécessaire pour stocker les index à sélectionner. La deuxième étape sélectionne d'une manière gloutonne, à partir de l'ensemble des index candidats, ceux apportant un meilleur bénéfice par unité de stockage. Si, un tel index existe, il est ajouté à l'ensemble O. Si après un ajout, il arrive que tous les attributs composant la clé primaire d'une table de faits soient indexés, alors ces index sont transformés en un seul index multi-attributs

construit sur la clé primaire de la table de faits. Après avoir sélectionné tous les index, la troisième étape choisit les index primaires pour les tables de faits restantes.

3.1.11 Conclusion

Le Tableau 3.2 résume la classification des travaux traitant du problème de sélection d'index en tenant compte de ces critères suivant :

- la méthode de construction des index candidats : manuelle ou automatique ;
- la méthode de construction de la configuration finale d'index : algorithme glouton (de manière ascendante ou descendante), algorithme génétique ou résolution de problème du sac à dos ;
- le modèle de coût : fonction mathématique ou appel à l'optimiseur de requêtes.

	Sélection des index candidats		**Construction de la configuration finale d'index**			**Estimation du coût**	
TRAVAUX	*Manuelle*	*Automatique*	*Algorithme glouton*	*Algorithme génétique*	*Sac à dos*	*Fonction mathématique*	*Optimiseur de requêtes*
Frank *et al.* [FON92]	X		Descendant				X
Choenni *et al.* [CBC93a, CBC93b]	X		Descendant et ascendant			X	
Whang *et al.* [KY87]	X		Descendant et ascendant			X	
Gündem *et al.* [Gun99]	X		Descendant et ascendant		X	X	
Chaudhuri *et al.* [CN97]		X	Ascendant				X
Kratica *et al.* [KLT03]	X			X		X	
Feldman *et al.* [FR03]		X			X	X	
Valentin *et al.* [VZZ+00]		X	Ascendant				X
Golfarelli *et al.* [GRS02]		X	Ascendant				X

TAB. 3.2 – Classification des travaux sur la sélection d'index

3.2 Problème de sélection de vues matérialisées

Le problème de sélection de vues matérialisées consiste à construire une configuration de vues optimisant le coût d'exécution d'une charge donnée. Cette optimisation peut être réalisée sous certaines contraintes telles que l'espace de stockage alloué aux vues sélectionnées ou une borne supérieure du coût de maintenance des vues sélectionnées.

Le problème de sélection de vues matérialisées dans les entrepôts de données a été largement étudié tant pour l'approche MOLAP (*Multidimentional On-Line Analytical Processing*) que pour l'approche ROLAP (*Relational On-Line Analytical Processing*). Le problème de sélection de vues matérialisées est NP-complet [Gup99]. En effet, si d est le nombre de dimensions dans le schéma d'un entrepôt de données, et qui ne contient aucune hiérarchie, alors le nombre de vues candidates à sélectionner est égal à $n = 2^d$. La complexité du problème est de $O(2^n)$, où n est le nombre de vues candidates dans le schéma.

De nombreux algorithmes ont été développés pour élaborer une solution optimale ou quasi-optimale pour le problème de sélection de vues en réduisant la complexité de sélection. Ils peuvent être divisés en deux classes différentes selon l'objectif de l'optimisation :

- algorithmes dirigés par la contrainte d'espace,
- algorithmes dirigés par le temps de maintenance.

Pour les algorithmes dirigés par le temps de maintenance, nous citons les travaux de Gupta *et al.* [GM99]. Le problème de sélection de vues consiste à sélectionner un ensemble de vues afin de minimiser le temps de réponse des requêtes de manière à ce que le temps de maintenance des vues sélectionnées ne dépasse pas un seuil fourni par l'utilisateur.

Ces algorithmes supposent que les requêtes qui sont l'objet de l'optimisation sont connues *a priori*. Dans ce cas, le problème de sélection de vues est qualifié de statique. Si une évolution des requêtes est enregistrée, alors il est nécessaire de reconsidérer le problème de sélection de vues en reconstruisant les vues à matérialiser. La sélection devient alors dynamique. Dans ce contexte, nous pouvons citer les travaux de Kotidis. Le système DynaMat de Kotidis *et al.* [KR99] rafraîchit les vues sélectionnées si la taille de ces dernières dépasse la capacité de l'espace de stockage allouée par l'administrateur suite aux opérations de mise à jour. Il procède alors à certaines éliminations selon des critères de remplacement. Par exemple, les vues les moins utilisées sont éliminées.

L'objet de notre étude porte sur la sélection des vues matérialisées dans le cas statique. Le problème de sélection de vues admet en entrée le schéma d'un entrepôt de données, une charge de m requêtes fréquemment utilisées (les requêtes sont donc connues *a priori*) et une ressource d'espace de stockage. Le problème posé suppose donc que l'ensemble des requêtes n'évolue pas dans le temps.

Plus formellement, si $V = \{v_1, ..., v_n\}$ est un ensemble de vues candidates, $Q = \{q_1, ..., q_m\}$ un ensemble de requêtes et S la taille de l'espace disque réservé par l'administrateur pour stocker les vues à sélectionner, alors il faut trouver une configuration de vues $Config_V$ telle que :

– le coût d'exécution des requêtes de la charge soit minimal, c'est-à-dire

$$C_{/Config_V}(Q) = Min\left(C_{/V}(Q)\right);$$

– l'espace de stockage des index de $Config_V$ ne dépasse pas S, c'est-à-dire

$$\sum_{v \in Config_V} taille(v) \leq S.$$

Les algorithmes de sélection de vues matérialisées proposés dans la littérature opèrent en deux étapes. La première étape consiste à construire un ensemble de vues candidates à partir des requêtes de la charge. Cette construction matérialise les relations qui peuvent exister entre les vues candidates dans un treillis, un graphe de vues AND-OR, un plan d'exécution des requêtes, dans un graphe d'éléments-vues en ondelettes ou en analysant syntaxiquement les requêtes de la charge. La deuxième étape consiste à construire, à partir de l'ensemble de vues candidates, une configuration de vues optimisant le temps d'exécution des requêtes sous la contrainte d'espace de stockage. Cette sélection est alors guidée par un modèle de coût ou en faisant appel à l'optimiseur de requêtes du SGBD. Elle est réalisée à l'aide d'un algorithme glouton qui sélectionne progressivement des vues à matérialiser, un algorithme génétique ou en assimilant le problème de sélection au problème du sac à dos.

Dans la suite, nous nous intéressons plus particulièrement aux algorithmes de sélection de vues matérialisées dirigés par la contrainte d'espace de stockage, en vue d'optimiser le temps d'exécution des requêtes d'une charge donnée.

3.2.1 Sélection de vues matérialisées à base de treillis

Un cube de données multidimensionnel est composé de cellules présentant des mesures observées aux croisements des dimensions. La Figure 3.11 montre un exemple de cube composé de la mesure `sales` observée suivant les trois dimensions `customers`, `times` et `products`. Dans chaque cellule (c, t, p) de ce cube est stocké le montant des ventes du produit p acheté au mois t par le client c. Dans ce cube, on peut être amené à observer les ventes d'un produit p pour un client c donné. La valeur `ALL`, ajoutée au domaine de chaque dimension, a été proposé pour répondre à ce type de requête [GCB+97]. La requête est résolue à partir des cellules (p, ALL, c).

Cet exemple montre que les valeurs (mesures) de plusieurs cellules sont calculables à partir d'autres cellules du même cube. De telles cellules sont appelées dépendantes. Par exemple, les valeurs des cellules (p, ALL, c) peuvent être calculées à partir des cellules $(p, t_1, c), \ldots, (p, t_{|t|}, c)$, où $|t|$ désigne la cardinalité de la dimension t.

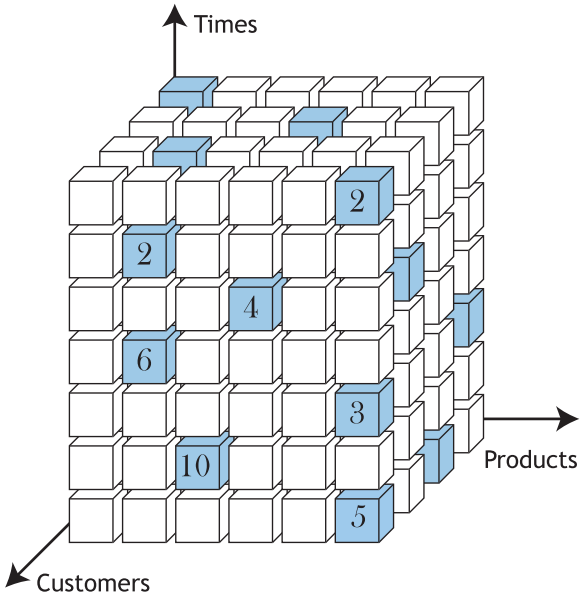


FIG. 3.11 – Cube de données multidimensionnel

Toute cellule ayant la valeur `ALL` est dite dépendante. Sa valeur peut donc être calculée à partir d'autres cellules. En revanche, si une cellule ne comporte pas la valeur `ALL`, elle ne peut pas être calculée à partir d'autres cellules. Le problème posé est de choisir quelles sont

```
select products, customers, sum(sales)
from R
group by products, customers
```

FIG. 3.12 – Exemple de requête SQL avec `Group by`

les cellules du cube à matérialiser [HRU96].

Les cellules sont organisées dans différents ensembles suivant la position de la valeur `ALL` dans leur adresse. Par exemple, toutes les cellules $(-, ALL, -)$ (le caractère - joue le rôle d'un jocker) sont placées dans le même ensemble. Chacun de ces ensembles correspond à une requête SQL. Par exemple, la requête SQL de la Figure 3.12 prend ses résultats dans l'ensemble des cellules (p, ALL, c). La table R de la clause `From` contient l'ensemble de toutes les cellules. Chaque requête est ainsi caractérisée par les attributs de la clause `Group by`. Par exemple, la requête de la Figure 3.12 est caractérisée par $(products, customers)$.

Sélectionner les ensembles de cellules à matérialiser revient à décider quelles requêtes SQL ou vues matérialiser. Ce problème a été formalisé sous forme de treillis capturant les relations existante entre les vues, requêtes ou cellules [HRU96, BPT97, KR99, URT99, SDN00, KMP02]. Nous donnons dans la suite quelques définitions utilisées pour formaliser le problème de sélection de vues en utilisant un treillis, les modèles de coût proposés et les algorithmes de sélection de vues exploitant le treillis de vues.

3.2.1.1 Formalisme

Définition 3.2.1 (Dépendance entre vues) *Soient deux vues v_1 et v_2. La vue v_1 est dépendante de la vue v_2 et dénotée $v_1 \preceq v_2$, si v_1 peut être calculée à partir de v_2.*

Par exemple, la vue $(products)$ peut être résolue en utilisant la vue $(products, customers)$ car $(products) \preceq (products, customers)$. D'autres vues ne sont pas comparables en utilisant l'opérateur $\preceq$. Par exemple, $(products) \not\preceq (customers)$ et $(customers) \not\preceq (products)$. L'opérateur $\preceq$ forme alors un ordre partiel sur l'ensemble des vues. L'ensemble des vues muni de l'opérateur $\preceq$ a été modélisé sous forme d'un treillis.

Définition 3.2.2 (Treillis de vues) *Un treillis de vues, noté $L(V, \preceq)$, est formé par un ensemble de vues V et l'opérateur $\preceq$ défini précédemment. La borne supérieure du treillis correspond à la vue dont laquelle toutes les vues sont dépendantes (la totalité du cube) et sa borne inférieure correspond à la vue générée en agrégeant complètement les données de n'importe quelle vue du treillis.*

La Figure 3.13 illustre un exemple de treillis de vues construit à partir des vues `products` (p), `time` (t) et `customers` (c). À chaque vue est associée le nombre de cellules qu'elle contient (la lettre M symbolise l'unité million). La vue *none* correspond à la vue résultant d'une requête ne contenant pas de clause `Group By`.

Les liens existant entre les vues de treillis et les requêtes de charge peuvent être conservés [BPT97]. Dans ce cas, chaque requête pointe vers un nœud du treillis. Les vues du treillis, quant à elles, sont liées par des relations d'ancêtre et de descendant. Les ancêtres et les descendants d'un élément (vue) du treillis $(V, \preceq)$ sont définis comme suit [HRU96].

$$ancestor(a) = \{b, a \preceq b\}$$

$$descendant(a) = \{b, b \preceq a\}$$

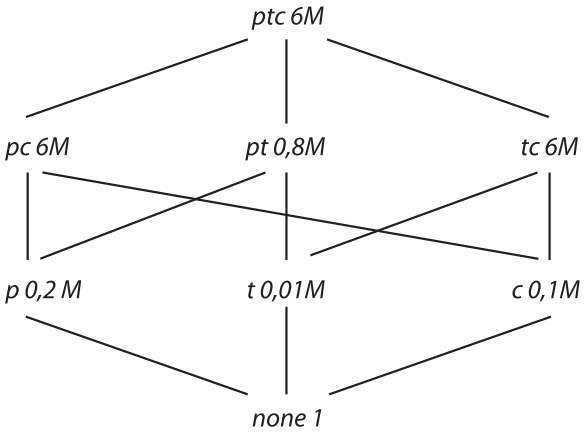


FIG. 3.13 – Exemple de treillis de vues

Lorsque les dimensions présentent des hiérarchies, la dépendance entre les vues est formalisée à l'aide d'un treillis appelé le produit direct d'un treillis dimensionnel (*direct product*

of the dimensional lattice) [TM75]. À la Figure 3.14 (a), la dimension `cutomers` c est organisée en deux niveaux : les individus clients dénotés c qui sont groupés suivant leur pays de résidence dénotés n. La dimensions `products`, quant à elle, est organisée suivant les produits dénotés p regroupés par leur tailles dénotées s. Les produits sont aussi regroupés par leur types dénotés t. À partir des deux treillis représentant les hiérarchies des dimensions c et p est construit le treillis de ces deux dimensions. Cela est réalisé à l'aide du produit direct de ces treillis. La Figure 3.14 (b) illustre le treillis produit.

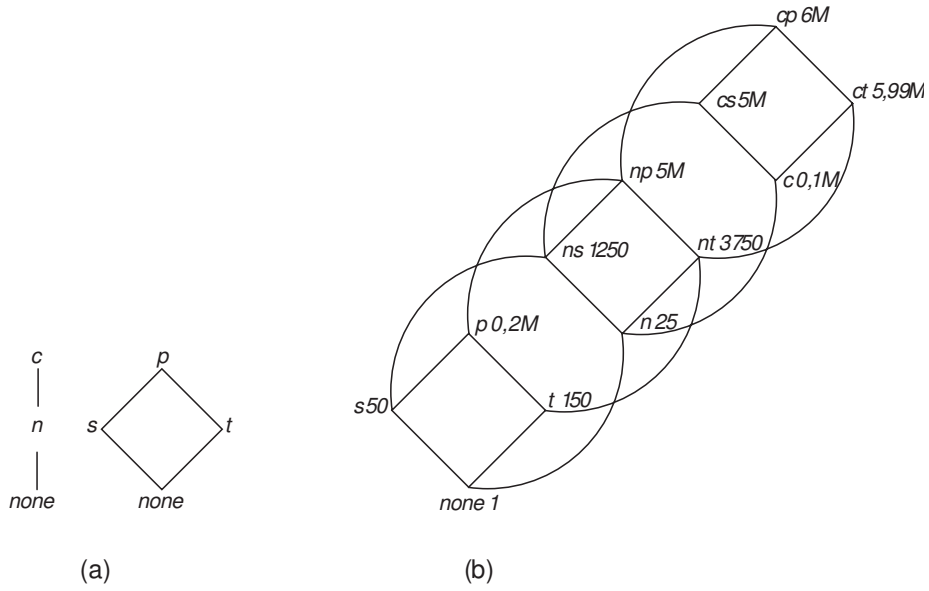


FIG. 3.14 – Treillis combinant des dimensions comportant des hiérarchies

Les hiérarchies au niveau des dimensions peuvent être prises en compte en utilisant les dépendances fonctionnelles [BPT97]. La présence de hiérarchies permet de supprimer certains éléments du treillis. En effet, si on considère une requête opérant des opération de regroupement sur une dimension clé d et aussi sur l'attribut a de la même dimension, alors la vue (d, a) produit le même résultat que (d) car l'attribut a dépend fonctionnellement de d. Cette observation a été introduite dans la définition des ancêtres et des descendants d'une vue. Le treillis est donc doté des deux opérateurs *meet* et *join* pour calculer les ancêtres ou les descendants des vues, respectivement [BPT97]. L'opérateur *meet*, dénoté $\oplus$, s'applique sur deux vues et construit l'union des attributs de la clause `Group by` de ces vues en éliminant de cette union les attributs pour lesquels il existe une dépendance fonctionnelle.

Le résultat de l'opérateur $\oplus$ est la plus petite vue contenant toute l'information nécessaire pour répondre aux vues mères. L'opérateur *join*, dénoté $\ominus$, s'applique aussi sur deux vues. Il étend les attributs de la clause `Group By` des vues données en argument par les attributs qui peuvent être dérivés. Il considère ensuite leur intersection et élimine finalement les attributs présents dans la partie droite des dépendances fonctionnelles dont les attributs de la partie gauche sont présents dans la clause de la vue donnée comme premier argument. L'opérateur $\ominus$ calcule le plus grand ensemble d'attributs caractérisant une vue qui peut être calculée à partir des deux vues données en argument.

3.2.1.2 Modèles de coût

Plusieurs modèles de coût ont été développés pour optimiser le compromis espace–temps lorsque le treillis de vues est implémenté. Le problème est approché selon trois points de vues :

- optimiser les temps ;
- optimiser l'espace de stockage ;
- optimiser le temps tant que la contrainte d'espace de stockage n'est pas atteinte.

Le coût d'exécution d'une requête est assimilé au nombre de n-uplets lus pour répondre à cette requête. En absence d'index construits sur la vue exploitée par une requête, le coût de cette dernière est égale au nombre de n-uplet que contient cette vue.

Plusieurs modèles de coût ont été proposés pour évaluer l'exploitation d'une vue et par conséquent le bénéfice qu'elle apporte. Le coût de stockage des vues a aussi été estimé pour satisfaire la contrainte d'espace.

Soit $C(v)$ le coût d'une vue v. Le bénéfice $B(v, Config_V)$ apporté par la vue v, sachant qu'un ensemble $Config_V$ est préalablement sélectionné, est défini comme suit [HRU96].

1. Pour chaque vue $w \preceq v$, définir la quantité B_w :
   (a) soit u la vue de plus petit coût telle que $w \preceq u$,
   (b) si $C(v) < C(u)$, alors $B(w) = C(v) - C(u)$. Sinon, $B(w) = 0$.
2. $B(v, Config_V) = \sum_{w \preceq u} B(w)$.

Ce modèle a été étendu en considérant le bénéfice de chaque vue par unité d'espace de stockage de cette vue. Dans ce modèle, certaines hypothèses sont prises en compte :

- les attributs de regroupement sont supposés avoir la même cardinalité r,
- les données du cube sont distribuées aléatoirement.

Sous ces conditions, la taille d'une vue v est donnée par la formule suivante :

$$|v| = \begin{cases} r^i & \text{si } r^i < m, \\ m & \text{sinon} \end{cases}$$

où m désigne le nombre de cellules dans la borne supérieure du treillis et i le nombre d'attributs de regroupement de la vue v. Une adaptation de ce modèle a été proposée dans le cas où les attributs de regroupement ont des cardinalités différentes.

Dans ce modèle, la fréquence des requêtes et le coût de maintenance des vues ne sont pas pris en compte. Les travaux de Baralis *et al.* [BPT97] et Uchiyama *et al.* [URT99] intègrent ces paramètres dans leur modèle de coût.

Dans [BPT97], la fonction de coût s'exprime alors

$$C(Q,V) = \sum_{q_i \in Q} f_{q_i} C(q_i, V) + \underbrace{\sum_{v_i \in V} f_u C_u(v_i)}_{M(V)}$$

où f_{q_i}, f_u, $C(q_i,V)$ et $C_u(v_i)$ sont respectivement la fréquence d'une requête d'interrogation q_i, la fréquence des mises à jour des vues, le coût de la requête q_i en présence des vues de l'ensemble V et le coût de maintenance de la vue v_i.

La notion de la plus proche vue matérialisée parente NMPV (*Nearest Materialized Parent View*) est introduite pour calculer le bénéfice apporté par une vue dans le processus d'optimisation [URT99]. Une vue u est dite NMPV de v si u est un ancêtre de v et que la différence entre la taille de v et la taille de u est minimale, ou plus formellement, $NMVP(v) = min(|u|, |v|) \forall u \in$ treillis et $v \preceq u$. Le bénéfice d'une vue v est calculé par

$$B(v, Config_V) = \left(\frac{|NMPV(v)| - |v|}{BF} \sum_{p \in child(v) \cup v} f_p \right) T_{rba}$$

où T_{rba} désigne le temps d'un accès aléatoire à un bloc disque, BF le facteur de bloc des vues et f_p la fréquence d'utilisation de la vue p. La fonction $child(v)$ donne les fils d'une vue

v de NMPV. Le coût de maintenance des vues est aussi pris en compte pour calculer le profit apporté par une vue. Le profit s'exprime par $P(v, Config_V) = B(v, Config_V) - M(v)$.

3.2.1.3 Algorithmes de sélection de vues

Dans cette section, nous présentons quelques algorithmes d'optimisation du treillis de vues. Ces algorithmes admettent principalement en entrée un treillis de vues et exploitent un modèle de coût pour construire une configuration de vues à matérialiser.

L'algorithme 1, proposé par Harinarayan *et al.* [HRU96], initialise l'ensemble des vues à matérialiser à la vue représentant la racine du treillis de vues. À chaque itération, dont le nombre est fixé à k, l'algorithme parcourt le treillis pour chercher la vue apportant le meilleur bénéfice. Cette vue est ajoutée à l'ensemble $Config_V$.

Algorithme 1 Algorithme de sélection de vues de Harinarayan *et al.*

$Config_V \leftarrow$ {vue de la borne supérieure du treillis}
pour $i = 1$ à k **faire**
  Sélectionner une vue $v \notin Config_V$ tel que $B(v, Config_V)$ soit maximal
  $Config_V \leftarrow Config_V \cup \{v\}$
fin pour
retourner $(Config_V)$

Dans le cas où le modèle de coût prend en compte la taille des vues, au lieu de fixer le nombre k, l'espace de stockage alloué aux vues est prédéfini.

Baralis *et al.* proposent deux heuristiques pour la construction de la configuration de vues à matérialiser. La première consiste à réduire le treillis des vues candidates en ne considérant que les vues associées aux requêtes de la charge. La deuxième heuristique élague les vues qui ne contribuent pas à l'amélioration des performances. Cette heuristique est basée sur l'estimation de la taille des vues en utilisant la technique d'estimation de Shukla *et al.* [SDNR96]. Selon la taille estimée des vues, il est possible de déterminer à partir de quel niveau d'agrégation il n'est plus pertinent de matérialiser.

Soient FD et A des de dépendances fonctionnelles et d'attributs, respectivement. L'algorithme 2 considère les dépendances fonctionnelles $fd_i \in FD$ pour identifier quand des attributs dans A, qui apparaissent dans la partie droite A^r des dépendances fonctionnelles, produisent une vue dont la taille estimée est non significativement (suivant la valeur d'un

seuil $\overline{p}$ fournie par l'utilisateur) différente de la taille de la vue obtenue en prenant les attributs de la partie gauche A_l. Lorsque ce cas se produit, les attributs de la partie droite sont remplacés par ceux de la partie gauche. En effet, la vue v_1 obtenue en utilisant les attributs de la partie gauche des dépendances fonctionnelles est remplacée par la vue v_2 obtenue en utilisant la partie droite car $v_1 \preceq v_2$.

Algorithme 2 Algorithme de l'heuristique de réduction de Baralis *et al.*

Entrée A un ensemble d'attribut et $\overline{p}$ un seuil
  répéter
    $stop \leftarrow$ true
    pour tout $fd_i \in FD$ **faire**
      si $\Big((A \cap A_i^r \neq \emptyset)$ et $\overline{p}$ x $(|v^{A_i^l}| < |v^{A \cap A_i^r}|)\Big)$ **alors**
        $A \leftarrow A - A_i^r$
        $A \leftarrow A \cup A_i^l$
        $stop \leftarrow$ false
      fin si
    fin pour
  jusqu'à *stop*
  retourner A

Nadeau *et al.* discutent le problème de scalabilité des algorithmes de sélection de vues matérialisées dans un treillis de vues [NT02]. En effet, les auteurs montrent que l'algorithme de Harinarayan *et al.* n'est pas optimal car, à chaque itération, tous les nœuds non sélectionnés du treillis de vues sont évalués. L'algorithme de Nadeau *et al.* , appelé PGA (*Polynomial Greedy Algorithm*), procède en plusieurs phases : nomination et sélection, comme illustré à la Figure 3.15.

La phase de nomination commence à partir de la borne supérieure du treillis. Le nœud fils, noté v_1, qui a la taille minimale est nommé. L'ensemble de vues candidates contient alors v_1. Les fils de v_1 sont examinés pour nommer le meilleur fils. Le fils trouvé est ajouté à l'ensemble des vues candidates. Cette phase de nomination se poursuit jusqu'à atteindre la borne inférieure du treillis. Un chemin de vues candidates est ainsi construit.

La phase de sélection évalue chaque vue appartenant à un ensemble de vues candidates et sélectionne d'une manière gloutonne la vue apportant le bénéfice maximal. La seconde itération commence par une autre phase de nomination, partant de l'un de fils non encore nommés de la borne supérieure du treillis. Si plusieurs choix de nomination se présentent, un

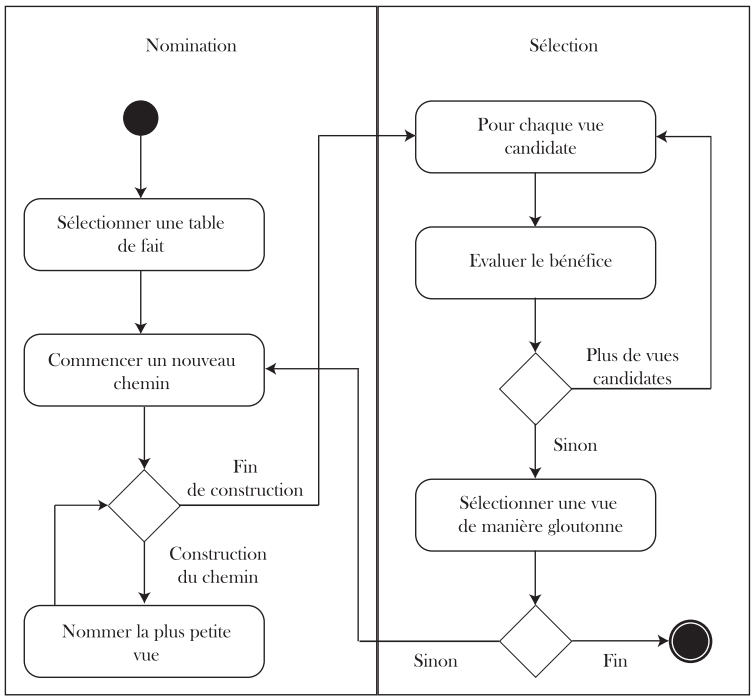


FIG. 3.15 – Diagramme d'activité de l'algorithme PGA

nœud est pris au hasard. Après la construction du chemin de vues candidates, la phase de sélection entre en jeu. L'alternance entre la phase de nomination et de sélection se poursuit jusqu'à atteindre les contraintes de sélection comme l'espace disque alloué au stockage des vues matérialisées.

3.2.2 Sélection de vues matérialisées à partir des graphes de vues AND-OR

Nous présentons dans cette section les travaux portant sur la sélection de vues matérialisées à l'aide des graphes de vues AND-OR, exploités pour modéliser les relations existant entre les vues candidates. Nous présentons dans un premier temps le formalisme des graphes de vues AND-OR. Nous introduisons ensuite les modèles de coût proposés pour estimer le coût des requêtes de la charge et détaillons les algorithmes de sélection de vues matérialisées dans un graphe de vues AND-OR.

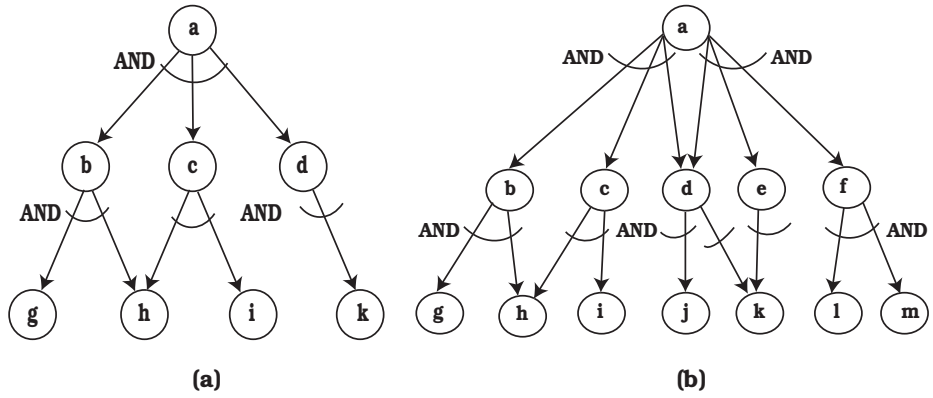


FIG. 3.16 – Exemple d'expression AND-DAG (a) et d'expression AND-OR-DAG (b)

3.2.2.1 Formalisme

Définition 3.2.3 (Expression AND-DAG) *Une expression AND-DAG pour une requête ou une vue v est un graphe orienté acyclique (Directed Acyclic Graph) ayant pour feuilles (aucune arrête sortante) les tables de la base de données source et v pour racine (aucune arrête rentrante). Si un nœud u correspondant à une vue a une arrête sortante vers d'autres nœuds $v_1, v_2, \ldots, v_k$, alors toutes les vues $v_1, v_2, \ldots, v_k$ sont requises pour calculer u. Cette dépendance est indiquée en traçant un demi-cercle, appelé un arc AND, à travers les arrêtes $(u, v_1), (u, v_2), \ldots, (u, v_k)$. À un tel arc est associé un opérateur (jointure, union, agrégation, etc.) et un coût exprimant le coût de calcul de u à partir de $v_1, v_2, \ldots, v_k$.*

Un exemple d'expression AND-DAG pour une vue a est présentée à la Figure 3.16 (a). Le nœud intermédiaire b (représentant la vue b) peut être calculé à partir des vues g et h. Les expressions AND-DAG sont structurées comme un arbre. De ce fait, elles ne sont pas capables d'exprimer plusieurs alternatives pour évaluer une vue. Les expressions AND-OR-DAG sont définies pour pallier ce problème. Une expression AND-OR-DAG doit avoir plus d'un arc AND à chaque nœud, le transformant en une expression AND/OR.

Définition 3.2.4 (Expression AND-OR-DAG) *Une expression AND-OR pour une requête ou une vue v est un graphe orienté acyclique avec v comme racine et les tables de la base de données comme nœuds feuilles. Chaque nœud intermédiaire est associé avec un ou*

plusieurs arcs AND, chacun imbriquant un sous-ensemble des ses arrêtes sortantes. Comme dans la définition précédente, à chaque arc AND est associé un opérateur et un coût. Plusieurs arcs AND au niveau d'un nœud désignent les différentes manières de calculer ce nœud.

La Figure 3.16 (b) montre un exemple d'expression AND-OR-DAG. Le nœud a peut être calculé à partir de l'ensemble des vues $\{b, c, d\}$ ou $\{d, e, f\}$. La vue a peut être calculée à partir de l'ensemble $\{j, k, f\}$ car d peut être calculée à partir de j ou de k et e peut être calculée à partir de k.

Définition 3.2.5 (Graphe de vues AND-OR) *Un graphe orienté acyclique G ayant les tables de base comme nœuds feuilles est appelé un graphe de vues AND-OR pour les requêtes ou vues $v_1, v_2, \ldots, v_k$ si, pour chaque v_i, il existe un sous-graphe G_i de G qui est une expression AND-OR pour v_i. Chaque nœud n dans le graphe de vues AND-OR a les paramètres suivants : son espace de stockage S_n, la fréquence f_n (fréquence des requêtes d'interrogation définies sur n), fréquence de mise à jour g_n (fréquence des requêtes de mise à jour définies sur n) et le coût de lecture R_n de la vue matérialisée n correspondant à ce nœud.*

3.2.2.2 Construction du graphe de vues AND-OR

Étant donné un ensemble de requêtes, la construction d'un graphe AND-OR pour ces requêtes est réalisée en trois étapes :

1. construction des expressions D_i AND-OR-DAG de chaque requête,
2. fusion de ces expressions pour construire le graphe G de vues AND-OR,
3. obtention des paramètres des vues.

Le processus de construction d'une expression D_i AND-OR-DAG correspondant à une requête q_i consiste à identifier plusieurs plans d'évaluation de cette requête à partir des tables de la base de données. Une requête peut être évaluée en utilisant des expressions multiples d'arbres dépendant de séquence et de l'ordre d'application des opérateurs de cette requête.

Le graphe de vues AND-OR pour l'ensemble de requêtes est construit en intégrant et fusionnant les expressions AND-OR-DAG $D_1, D_2, \ldots, D_k$. Soit G_{i-1} un graphe de vues AND-OR formé en intégrant les expressions $D_1, D_2, \ldots, D_{i-1}$. Le processus d'intégration de l'expression D_i dans le graphe G_{i-1} consiste à :

1. mettre en correspondance les nœuds de D_i avec ceux de G_{i-1} présentant les mêmes expressions relationnelles,
2. identifier si les nœuds de D_i n'ayant pas de correspondants peuvent être dérivés à partir de l'ensemble de nœuds de G_{i-1} et *vice-versa*,
3. imbriquer les relations de dérivation entre les nœuds trouvés à l'étape 2.

Chaque nœud final du graphe de vues AND-OR représente une vue candidate. Pour chaque vue candidate, un certain nombre de paramètres sont calculés :

- sa fréquence calculée à partir des fréquences des requêtes de la charge susceptibles d'exploiter cette vue en interrogation,
- sa fréquence de mise à jour calculée à partir des fréquences de mise à jour des tables utilisées pour dériver cette vue,
- l'espace de stockage de cette vue déterminé en estimant le nombre de n-uplets qu'elle peut contenir.

3.2.2.3 Modèles de coût

Étant donnés un graphe de vues AND-OR G et un espace de stockage S, le problème de sélection de vues consiste à construire une configuration de vues $Config_V$ correspondant à un sous-ensemble de nœuds de G, qui minimise le temps de réponse des requêtes de la charge sous la contrainte que l'espace de stockage occupé par $Config_V$ ne dépasse pas S. Le temps s'exprime via la fonction suivante :

$$C(G,V) = \sum_{q_i \in Q} f_{q_i} C(q_i, V) + \sum_{v \in V} f_u C_u(v_i)$$

sous la contrainte $\sum_{v \in V} taille(v) \leq S$, où S_v désigne l'espace de stockage de la vue v.

Le coût de réponse $C(q,V)$ à une requête q_i en présence d'un ensemble de vues V d'un graphe de vues AND-OR est évalué par le coût le moins élevé d'un sous-graphe de vues AND H_v de la vue associée à q. H_v est un sous-graphe de G dont les nœuds sont dans V. $C(q,\emptyset)$ est le coût de réponse à la requête q à partir des données sources.

Soit V un ensemble de vues préalablement sélectionnées. Le bénéfice apporté par un

ensemble de vues V' sachant V est égal à $B(V', V) = C(G, V) - C(G, V \cup V')$. Le bénéfice apporté par l'ensemble de V' par unité de stockage est égal à $B(V', V)/taille(V')$, où $taille(V')$ désigne la taille de l'ensemble de vues V'. Le bénéfice $B(V', \emptyset)$ est appelé bénéfice absolu des vues V'.

3.2.2.4 Algorithmes de sélection des vues matérialisées

Les algorithmes de sélection de vues matérialisées dépendent de la nature du graphe de vues : graphe de vues OR, AND ou AND-OR. Nous présentons dans la suite les algorithmes adoptés dans chaque cas.

Graphe de vues OR

Un graphe de vues OR est un graphe ne contenant pas d'expression AND-DAG. Dans ce cas, un nœud est calculé à partir de l'un de ses fils. Ce type de graphe est observé dans un cube de données car chaque vue est calculée à partir de zéro ou plusieurs autres vues et chaque vue représente une seule manière d'effectuer ce calcul.

L'algorithme 3 réalise une sélection gloutonne d'un ensemble de vues d'un graphe de vues OR pour minimiser le coût des requêtes sous la contrainte de l'espace de stockage S [Gup97, GM05]. À chaque itération, la vue dont le bénéfice est maximum par unité de stockage est sélectionnée.

Algorithme 3 Algorithme de sélection de vues dans un graphe de vues OR

Entrée Un graphe de vue OR G et un espace de stockage S
  $Config_V \leftarrow \emptyset$
  tant que $(taille(Config_V) < S)$ **faire**
    Soit v la vue dont le bénéfice par unité de stockage $B(\{v\}, Config_V)$ est maximal.
    $Config_V \leftarrow config_V \cup \{v\}$
  fin tant que
  retourner $(Config_V)$

En présence de plusieurs index construits sur les vues, le graphe de vues OR est étendu. Dans ce cas, il peut exister plusieurs arcs partant d'un nœud u vers un nœud v, à raison d'un arc pour chaque index construit sur v. Une étiquette (i, t_i) est associée à chaque arc partant de u vers v, où t_i est le coût de calcul de u à partir de v en utilisant le $i^{\text{ème}}$ index

de v. Si $i = 0$, aucun index n'est associé à la vue v.

L'algorithme 4, appelé *Inner-Level Greedy Algorithm*, est proposé pour la sélection des vues dans un graphe OR avec index [Gup97, GM05]. À chaque itération, l'algorithme choisi une vue et certains de ses index sélectionnés à leur tour d'une manière gloutonne ou un index dont la vue a préalablement été sélectionnée dans une itération antérieure.

Chaque itération peut être subdivisée en deux phases. Dans la première, pour chaque vue v_i est construit un ensemble IG_i contenant initialement v_i. Ensuite, ses index sont ajoutés un à un jusqu'à ce le bénéfice par unité de stockage ait atteint son maximum sous la contrainte de l'espace de stockage S. Si ce bénéfice est supérieur à celui apporté par l'ensemble courant des vues sélectionnées V, alors le contenu IG_i est ajouté à V. Dans la deuxième phase, l'index dont le bénéfice par unité de stockage est maximal sous la contrainte S est sélectionné.

Algorithme 4 Algorithme de sélection de vues dans un graphe de vues OR avec index

Entrée Un graphe de vues OR G avec index et un espace de stockage S
  $Config_V \leftarrow \emptyset$
  tant que $(taille(Config_V) < S)$ **faire**
    $TMP \leftarrow \emptyset$
    pour tout $v_i \in Config_V$ **faire**
      $IG \leftarrow v_i$
      tant que $taille(IG) < S$ **faire**
        Soit I_{ic} l'index de la vue v_i dont le bénéfice par unité de stockage $B(IG, V \cup IG)$ est maximal.
        $IG \leftarrow IG \cup I_{ic}$
      fin tant que
      si $B(IG, Config_V)/taille(IG) > B(TMP, Config_V)/taille(Config_V)$ **alors**
        $TMP \leftarrow IG$
      fin si
    fin pour
    pour tout index I_{ij} tel que ses vues $v_i \in Config_V$ **faire**
      si $B(\{I_{ij}, V)\}/taille(IG) > B(TMP, Config_V)/taille(Config_V)$ **alors**
        $TMP \leftarrow \{I_{ij}\}$
      fin si
    fin pour
    $Config_V \leftarrow Config_V \cup TMP$
  fin tant que
  retourner $(Config_V)$

Graphe de vues AND

Un graphe de vues AND est un graphe AND-OR ne comportant que des expressions AND-DAG. Dans un tel graphe, un nœud est calculé d'une seule manière : à partir de ses nœuds fils. Le graphe AND peut être vu comme un plan global pour des requêtes multiples. Ce plan prend en compte les sous-expressions communes de ces requêtes.

L'algorithme 3 opère aussi sur un graphe de vues AND. L'algorithme 5, appelé *Greedy-Interchange algorithm*, est proposé pour l'améliorer [Gup97]. En effet, l'algorithme 5 commence avec la solution fournie par l'algorithme 3 et améliore par la suite cette solution en réalisant des échanges entre les vues sélectionnées et celles qui ne sont pas encore sélectionnées. Ces échanges sont poursuivis jusqu'à ce que la solution ne puisse plus être améliorée.

Algorithme 5 Algorithme de sélection de vues dans un graphe de vues AND

Entrée Un graphe de vues AND G et un espace de stockage S
  Exécuter l'algorithme 3 pour obtenir $Config_V$
  répéter
    Soit (v_1, v_2) un couple de vues tel que $v_1 \in Config_V$ et le bénéfice apporté par $(Config_V - \{v_1\}) \cup \{v_2\}$ est supérieur à celui apporté par $Config_V$
    $Config_V = (Config_V - \{v_1\}) \cup \{v_2\}$
  jusqu'à aucune couple $(v_1, v_{2)}$ n'existe
  retourner $(Config_V)$

Comme dans le cas du graphe de vues OR, le problème de sélection de vues a été généralisé dans les graphe de vues AND en introduisant des index pour chaque vue. Chaque vue ou nœud du graphe de vue AND peut être calculé à partir de ses fils, mais en présence d'index, le coût de calcul dépend aussi de ces index.

Graphe de vues AND-OR

Les résultats développés pour la sélection de vues matérialisées dans un graphe de vues OR et AND sont étendus pour un graphe de vues AND-OR. Ce graphe est converti en un autre graphe appelé graphe requête–vue [GHRU97, GM99, GM05].

Définition 3.2.6 (Graphe requête–vue) *Un graphe requête–vue G est un graphe bipartite $(Q \cup \zeta, E)$, où Q est l'ensemble de requêtes de la charge, ζ l'ensemble des parties de l'ensemble des vues V et E l'ensemble des arrêtes de G. Une arrête (q, σ) appartient à E si*

et seulement si la requête q peut être résolue à partir de l'ensemble de vues σ et le coût associé à cette arrête est le coût requis pour répondre à q. Une fréquence f_q est également associée à chaque requête q. L'ensemble des tables sources $\rho \in \zeta$ est tel que $(q, \rho) \in E, \forall q \in Q$.

Le problème de sélection de vues dans un graphe de vues AND-OR est posé comme suit. Étant donnés un graphe requête–vue $G = (Q \cup \zeta, E)$ et un espace de stockage S, il s'agit de sélectionner une configuration de vues $Config_V \subseteq V$ qui minimise le temps de réponse total aux requêtes de Q, sous la contrainte que l'espace de stockage des vues de $Config_V$ ne dépasse pas S.

L'algorithme proposé, appelé *AO-Greedy Algorithm*, manipule des sous-graphes appelés graphes d'intersection [Gup97, GM05]. Un graphe d'intersection $F\zeta$ de ζ est un graphe ayant respectivement ζ et D comme son ensemble de nœuds et d'arrêtes telle qu'une arrête (α, β) appartient à D si et seulement si l'intersection entre l'ensemble des vues α et β n'est pas vide.

L'algorithme AO-Greedy opère d'une manière itérative comme suit. À chaque itération, l'algorithme prend un sous graphe connexe H de F_ζ dont l'ensemble des vues V_H offre un bénéfice maximal par unité de stockage. L'ensemble des vues V_H est ainsi ajouté à $Config_V$. L'algorithme s'arrête lorsque les vues $Config_V$ remplissent l'espace de stockage S.

Dans un graphe requête-vue d'un graphe OR, un élément $\sigma \in \zeta$ consiste en une seule vue, et ainsi F_ζ ne contient pas d'arrête. L'algorithme AO se comporte exactement comme l'algorithme 3.

Un autre algorithme, appelé *Multilevel Greedy Algorithm*, est proposé pour améliorer les performances de l'algorithme AO. Cet algorithme décompose le graphe d'intersection F_ζ d'un graphe requête-vue G en plusieurs composantes connexes $G_1, G_2, \ldots, G_l, l > 1$, où $G_i = (Q \cup \zeta, E_i)$ est un sous graphe de G et l le nombre de composantes connexes. L'algorithme procède de manière itérative. À chaque itération, l'ensemble des vues V_i de chaque graphe G_i ayant un bénéfice maximal par unité de stockage est recherché en utilisant la fonction récursive `Inner-Greedy` [Gup97, GM05]. Parmi les ensembles V_i trouvés, l'ensemble maximisant le bénéfice est à ajouter à la configuration de vues $Config_V$. La configuration $Config_V$ est retournée lorsque l'espace de stockage S est plein.

3.2.3 Sélection de vues matérialisées exploitant le plan d'exécution des requêtes

Nous présentons dans cette section les travaux portant sur la sélection de vues matérialisées à l'aide des plans d'exécution de requêtes, exploités pour déterminer l'ensemble des vues candidates. Nous présentons ensuite trois algorithmes de sélection d'index. Le premier assimile le problème de sélection de vues au problème du sac à dos, le deuxième introduit la notion de pertinence de vues pour estimer le bénéfice de matérialisation d'une vue et le troisième s'appuie sur le principe des algorithmes évolutifs pour construire une configuration de vues minimisant le coût d'exécution des requêtes d'une charge.

3.2.3.1 Formalisme

Un plan d'exécution d'une requête traduit les opérations réalisées à partir des tables de base pour fournir les résultats de cette requête. Chaque plan d'exécution d'une requête[4] peut être vu comme une expression AND-DAG. Tous les plans d'exécution d'une requête, considérés ensemble, forment une expression AND-OR-DAG. L'union des expressions AND-OR-DAG de chaque requête d'une charge donnée forme un graphe de vues AND-OR [VVK02].

Un graphe de vues AND-OR est un graphe bipartite, acyclique et orienté composé de deux types de nœuds : des nœuds d'opération et des nœuds d'équivalence [BB03b].

Définition 3.2.7 *Un nœud d'opération d'un graphe de vues AND-OR est un nœud AND correspondant à une opération algébrique relationnelle comme une jointure, sélection, projection, agrégation, etc.*

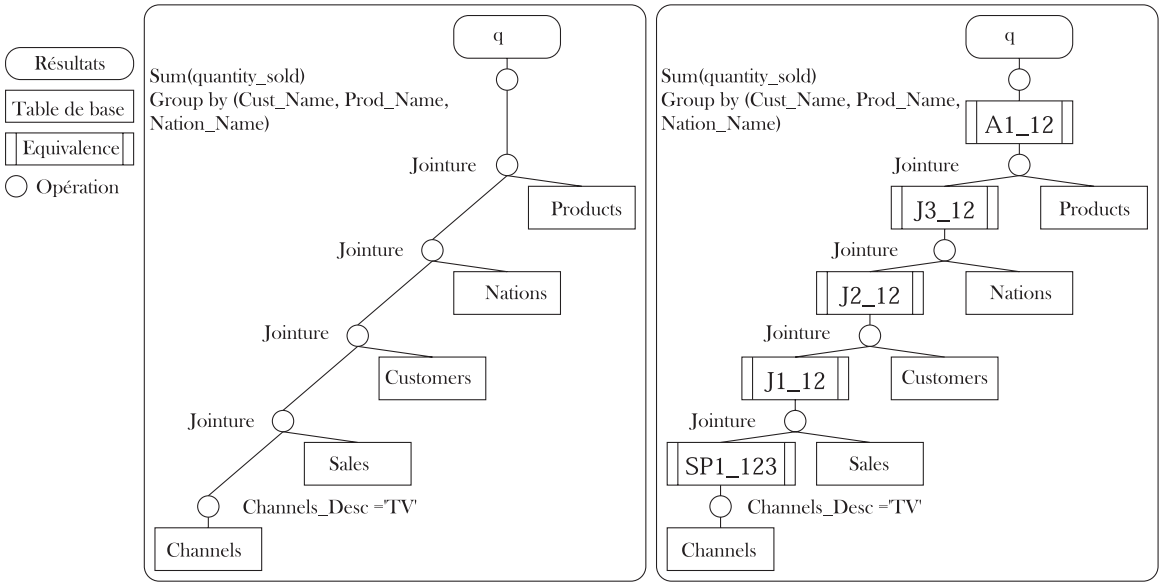
Définition 3.2.8 *Un nœud d'équivalence d'un graphe de vues AND-OR est un nœud OR représentant un ensemble d'expressions algébriques équivalentes qui génèrent les mêmes résultats.*

Les fils d'un nœud d'équivalence sont un ou plusieurs nœuds d'opération et les fils d'un nœud d'opération sont un ou plusieurs nœuds d'équivalence. Les nœuds feuilles d'un graphe AND-OR sont des nœuds d'équivalence représentant les tables de base. En général, à chaque

[4]Une requête donnée peut avoir plusieurs plans d'exécution.

table de base correspond un nœud feuille sauf dans les cas d'une jointure réflexive. Les nœuds d'équivalence correspondent aux vues candidates. La Figure 3.17 illustre un plan d'exécution de la requête suivante et son graphe de vues AND-OR.

```
(q)     select      cust_name, prod_name, nation_name, sum(quantity_sold)
        from        sales, customers, products, nations, channels
        where       sales.cust_id = customers.cust_id
                    and sales.prod_id = products.prod_id
                    and customers.prod_id = nations.prod_id
                    and sales.chan_ID= channels.chan_id
                    and channels.chan_desc = 'TV'
        group by    cust_name, prod_name, nation_name
```



(a) Plan d'exécution possible pour la requête q

(b) Expression AND-OR DAG de la requête q

FIG. 3.17 – Plan d'exécution d'une requête et son graphe de vues AND-OR

La construction d'un graphe AND-OR est réalisée comme suit. Pour chaque requête les expressions AND-DAG équivalentes à un plan d'exécution sont construites. Ces expressions sont regroupées pour construire les expressions AND-OR-DAG de chaque requête. La fusion des expressions AND-OR-DAG forment un graphe de vues AND-OR, appelé aussi un graphe

de matérialisation multi-vues (*Multi View Materialization Graph*), où les sous-expressions communes sont représentées une seule fois [BB03b].

3.2.3.2 Algorithmes de sélection de vues matérialisées

Nous présentons dans cette section trois approches exploitant le plan d'exécution de requêtes pour la sélection des vues matérialisées.

L'algorithme 6 de sélection de vues matérialisées proposé par Baril *et al.* se base sur le calcul de niveaux de matérialisation pour chaque requête du graphe AND-OR, réalisé par l'algorithme 7 [BB03b]. Un niveau de matérialisation est associé à chaque vue représentée par un nœud d'équivalence. Un niveau défini comme suit :

- $level(root) = 1$, la racine du graphe *root* représente le résultat de la requête ;
- $level(view) = level(parent(view)) + 1$, où la fonction *parent* donne le parent d'une vue du graphe.

La première étape de l'algorithme 6 consiste à sélectionner un ensemble de vues candidates. L'idée est de chercher, pour chaque requête, un niveau de matérialisation intermédiaire entre le premier niveau correspondant à la racine et le dernier niveau correspondant aux tables de base. En effet, la matérialisation du premier niveau donne un coût de traitement peu élevé et un coût de maintenance très élevé. En revanche, la matérialisation du dernier niveau donne un coût de traitement élevé et un coût de maintenance nul[5].

Pour chaque requête, l'algorithme 7 calcule le niveau de matérialisation intermédiaire suivant la fréquence d'interrogation et de mise à jour des tables impliquées dans cette requête. Ce calcul est réalisé en deux temps. Dans un premier temps, les vues apportant un bénéfice local positif sont pré-sélectionnées. Dans un deuxième temps, le coût total de chaque niveau de matérialisation, puis le niveau dont le coût d'exécution et de maintenance est minimal, sont calculés.

Après la recherche des vues candidates de chaque requête, la deuxième partie du processus d'optimisation consiste à chercher parmi ces vues celles qui optimisent le bénéfice global sous la contrainte d'espace de stockage. Dans la première étape de cette partie, les vues ayant un bénéfice global négatif sont élaguées. La configuration finale de vues $Config_V$

[5]Seules les tables de bases sont maintenues.

Algorithme 6 Algorithme général de sélection de vues matérialisées

Entrée Un graphe MVMG G et une contrainte d'espace de stockage S
  $V \leftarrow \emptyset$
  pour tout $q \in MVMG$ **faire**
    $V \leftarrow V \cup LevelSelection(MVMG, q)$
  fin pour
  pour tout $\{v\} \in V$ **faire**
    si $GlobalBenefit(v) < 0$ **alors**
      $C \leftarrow C - \{v\}$
    fin si
  fin pour
  $Config_V \leftarrow Knapsack(V, S, S(), GlobalBenefit())$
  retourner $(Config_V)$

est ensuite sélectionnée en assimilant le problème de sélection de vues au problème du sac à dos, dont les objets sont les vues candidates, la contrainte est S, le poids des objets est défini par la fonction $S()$ et la fonction à optimiser est $GlobalBenefit()$. Le problème du sac à dos est présenté à la Section 3.1.2.3.

Valluri *et al.* introduisent la notion de pertinence d'une vue matérialisée dans la sélection de vues basée sur le plan d'exécution des requêtes [VVK02]. La pertinence d'une vue indique comment la présence d'une vue dans une configuration peut affecter le calcul du bénéfice des autres vues de cette configuration et, par conséquent, le coût de traitement des requêtes et de maintenance des vues. La pertinence entre deux vues dépend de la relation mère-fille qui peut exister entre ces vues, de l'appartenance ou de la non-appartenance de ces vues à la configuration et du bénéfice qu'apportent ces vues séparément. Les auteurs ont défini douze cas possibles de calcul de la pertinence entre deux vues matérialisées, qui sont résumés au Tableau 3.3. La fonction *Pertinence* se base sur le calcul du bénéfice et la fonction *Materialization* indique dans quel cas il est pertinent de matérialiser ou pas et quelles sont les vues à matérialiser. Le symbole ∞ signifie que le bénéfice est maximal. Par exemple, dans le cas 3, la pertinence des vues v_1 et v_2 sachant que seulement v_1 est matérialisée et que ces deux vues ne sont pas connectées est égale au bénéfice de matérialisation apporté par v_2.

L'algorithme 8 initialise $Config_V$ à l'ensemble vide et parcourt le graphe de vues AND G tant que ses nœuds ne sont pas étiquetés comme "non visités". À chaque itération, une matrice

Cas	**Condition**	$Pertinence(v_1, v_2)$	$Materialization(v_1, v_2)$
1	$v_1 = v_2$	$B(Config_V \cup \{v_1\})$	v_1
2	Si $v_1 \notin Config_V$, $v_2 \notin Config_V$, et v_1, v_2 ne sont pas connectées	$B(Config_V \cup \{v_1, v_2\})$	v_1 et v_2
3	Si $v_1 \in Config_V$, $v_2 \notin Config_V$, et v_1, v_2 ne sont pas connectées	$B(Config_V \cup \{v_2\})$	v_2
4	Si $v_1 \notin Config_V$, $v_2 \in Config_V$, et v_1, v_2 ne sont pas connectées	$B(Config_V \cup \{v_1\})$	v_1
5	Si $v_1 \in Config_V$, $v_2 \in Config_V$, et v_1, v_2 ne sont pas connectées	∞	$\emptyset$
6	Si $v_1 \notin Config_V$, $v_2 \notin Config_V$, et v_1, v_2 sont connectées	$Max(B(Config_V \cup \{v_1\}, B(Config_V \cup \{v_2\})$	v_1 ou v_2
7	Si $v_1 \in Config_V$, $v_2 \notin Config_V$, et v_2 est le fils de v_1	$Max(B((Config_V \cup \{v_1) - \{v_2\}\}, B(Config_V \cup \{v_2\})$ ou ∞	v_2 ou $\emptyset$
8	Si $v_1 \notin Config_V$, $v_2 \in Config_V$, et v_2 est le fils de v_1	$B((Config_V \cup \{v_2\}) - \{\{v_1\}\})$	v_1
9	Si $v_1 \in Config_V$, $v_2 \in Config_V$, et v_2 est le fils de v_1	∞	$\emptyset$
10	Si $v_1 \in Config_V$, $v_2 \notin Config_V$, et v_1 est le fils de v_2	$B((Config_V \cup \{v_1\}) - \{\{v_2\}\})$	v_2
11	Si $v_1 \notin Config_V$, $v_2 \in Config_V$, et v_1 est le fils de v_1	$Max(B((Config_V \cup \{v_{2)} - \{v_1\}\}, B(Config_V \cup \{v_1\})$ ou ∞	v_1 ou $\emptyset$
12	Si $v_1 \in Config_V$, $v_2 \in Config_V$, et v_1 est le fils de v_2	∞	$\emptyset$

TAB. 3.3 – Calcul de la pertinence entre deux vues matérialisées

Algorithme 7 Sélection de niveau de matérialisation d'une requête $LevelSelection(G, q)$

Entrée Un graphe MVMG G et une requête q
  $P \leftarrow \emptyset$
  pour tout $v \in AllChildren(q)$ **faire**
    $b \leftarrow LocalBenefit(q, v)$
    si b>0 **alors**
      $P \leftarrow P \cup \{v\}$
    fin si
  fin pour
  $mlc \leftarrow \infty$
  pour tout $L \cup AllViewsOfLevel(q)$ **faire**
    $PL \leftarrow L \cap P$
    $lc \leftarrow TotalCost(q, PL)$
    si $mc < mlc$ **alors**
      $Config_V \leftarrow PL$
      $mlc \leftarrow lc$
    fin si
  fin pour
  retourner $(Config_V)$

de pertinence des vues est calculée selon les cas présentés au Tableau 3.3. Le couple de vues maximisant la pertinence est recherché. Sa valeur de pertinence est ensuite mise à ∞ car ce couple de vues n'est pas considéré dans les itérations futures. La fonction $Materialization$ indique si, pour ce couple de vue, la matérialisation est pertinente ou pas et quelles sont les vues à matérialiser. La matérialisation est effective si l'espace de stockage S n'est pas atteint. Le nœud du graphe correspondant aux vues du couple est finalement étiqueté comme visité.

Zhang *et al.* proposent les algorithmes évolutifs (*evolutionary algorithms*) pour la sélection de vues matérialisées en se basant sur le plan d'exécution des requêtes [ZYY01]. Les algorithmes évolutifs sont utilisés pour résoudre des problèmes d'optimisation en se basant sur des stratégies de recherche stochastique dans une population d'individus. Le problème de sélection de vues est spécifié comme suit. Étant donnés n plans d'exécution locaux $P_i, P_2, \ldots, P_n$, où $P_i = \{p_{i1}, \ldots, p_{ik}\}$ est l'ensemble des plans d'exécution possibles de chaque requête $q_i \in Q$, le problème de sélection de vues matérialisées consiste à matérialiser les vues du plan d'exécution global obtenu en fusionnant les n plans locaux (un plan pour chaque ensemble de plan P_i) tels que la somme du coût des requêtes et de leur maintenance soit minimale.

Algorithme 8 Sélection de vues basée sur la pertinence

Entrée Un graphe AND G et un espace de stockage S
  $Config_V \leftarrow \emptyset$
  Étiqueter chaque nœud de G comme "non visité"
  tant que tous les nœuds de G ne sont pas visités **faire**
    Calculer la matrice de pertinence des vues
    Soit (v_1, v_2) un couple de vues tel que leur pertinence est maximale
    si $Pertinence(v_1, v_2) < 0$ **alors**
      Quitter
    fin si
    $Pertinence(v_1, v_2) = \infty$
    si $Materialization(v_1, v_2) \neq \emptyset$ **alors**
      $L \leftarrow Materialization(v_1, v_2)$
      si $taille(Config_V \cup L) < S$ **alors**
        $Config_V \leftarrow Config_V \cup L$
      fin si
      $Materialization(v_1, v_2) \leftarrow \emptyset$
    fin si
    Étiqueter comme "visité" le nœud correspondant aux vues matérialisées
  fin tant que
  retourner $Config_V$

L'algorithme proposé procède en deux niveaux. Le niveau le plus haut recherche, à partir des plans d'exécution locaux de chaque requête, le meilleur plan d'exécution global de ces requêtes. Le niveau le plus bas choisi le meilleur ensemble de vues minimisant le coût de plan d'exécution global. Dans chaque niveau, le problème d'optimisation est résolu à l'aide d'un algorithme évolutionniste. Nous décrivons dans la suite les principes de base utilisés dans les algorithmes évolutifs.

Représentation des solutions

Deux représentations des solutions correspondant à chaque niveau sont à distinguer : représentations du plan d'exécution global et des vues matérialisées.

1. **Plan d'exécution global** : Pour n requêtes $q_i, i..n$, un plan d'exécution global peut être représenté sous forme d'un vecteur de n entiers $P_{1i}, \ldots, P_{kn}$, où P_{ki} indique le $k^{\text{ème}}$ plan d'exécution local de la requête q_i. Par exemple, si les trois requêtes q_1, q_2 et q_3 ont respectivement 12, 120 et 120 plans d'exécutions locaux, le vecteur $\{[4], [89], [70]\}$

représente le plan d'exécution global composé du quatrième plan d'exécution local de q_1, du quatre-vingt neuvième plan de q_2 et le soixante dixième plan de q_3. Chaque élément du vecteur est un gêne.

2. **Vues matérialisées** : La représentation des vues matérialisées dans le niveau bas est basée sur les graphes de vues AND. Chaque graphe est encodé sous forme d'une chaîne de caractères binaires. Chaque caractère correspond à un nœud du graphe. Si un caractère est à un, alors la vue dérivée de ce nœud correspondant à ce caractère est matérialisée. Dans le cas contraire, c'est-à-dire un caractère mis à zéro, la vue n'est pas matérialisée. La représentation binaire des graphes de vues facilite l'implémentation des opérations de croisement, de mutation et de sélection des algorithmes évolutionnistes.

Fonction de fitness

La fonction de fitness est la fonction à maximiser dans un algorithme évolutionniste. La fonction de fitness peut être dérivée à partir de la fonction de coût comme suit :

$$f(x) = \begin{cases} C_{max} - C(x) & \text{si } C(x) < C_{max}, \\ 0 & \text{sinon} \end{cases}$$

où C_{max}, $C(x)$ et $f(x)$ dénotent respectivement le coût maximum dans chaque itération, le coût et la fitness d'un individu x. Dans le niveau bas de l'algorithme, un individu est une vue matérialisée dont la fitness dépend du coût d'exécution des requêtes exploitant cette vue et de son coût de maintenance. Dans le niveau haut de l'algorithme, un individu est un plan d'exécution global dont la fitness dépend des vues sélectionnées.

Croisement

Le croisement encourage l'échange d'information entre les différents individus. Il aide à propager les meilleurs gènes. Dans le niveau bas de l'algorithme, l'opérateur de croisement appelé *cut and swap* est implémenté [Gol89]. Dans le niveau haut, le croisement entre deux gènes est réalisé à partir d'un point de croisement prédéfini.

Mutation

La mutation permet à l'algorithme de créer de nouveaux gènes qui ne font pas partie de la population initiale d'individus. Cela permet d'atteindre, en théorie, toutes les solutions possibles de l'espace de recherche. La mutation dans le niveau bas de l'algorithme, appelée *bit-flipping*, est réalisée en changeant l'état d'un bit choisi aléatoirement. Dans le niveau haut, la mutation est réalisée en changeant aléatoirement un élément du vecteur représentant le plan d'exécution global.

Traitement des solutions invalides

Il se peut, lors d'une mutation ou d'un croisement dans le niveau bas de l'algorithme, que le gène généré soit invalide. Par exemple, si une vue candidate v_2 a les mêmes ancêtres qu'une autre vue matérialisée v_1, alors il n'est pas nécessaire de matérialiser v_2. Les solutions invalides peuvent être prévenues ou corrigées après leur génération.

Sélection

La sélection dans un algorithme évolutionniste détermine la probabilité qu'un individu soit reproduit dans une prochaine génération. Un nombre prédéfini d'individus sont tirés aléatoirement de la population d'individus. Un "tournoi" est conduit parmi ces individus. Le meilleur individu est sélectionné pour survivre et se reproduire ainsi dans l'itération suivante.

3.2.4 Sélection de vues matérialisées à base d'ondelettes

Smith *et al.* proposent une méthode pour représenter un cube de données en utilisant le principe des ondelettes [SLJ04]. Cette méthode décompose un cube de données en une hiérarchie d'éléments-vues structurés sous forme d'un graphe. Ces éléments fournissent une structure granulaire pour synthétiser les vues agrégées. Un algorithme de sélection d'un ensemble d'éléments-vues est proposé pour matérialiser un cube de données. Dans la suite, nous donnons les définitions d'un élément-vue, du graphe d'éléments-vues et détaillons l'algorithme de sélection des éléments-vues proposé.

Soit A un cube de données comportant d dimensions et plusieurs mesures. Le volume de

ce cube est $vol(A) = \pi_{n=0}^{d-1} n_m$, où n_m désigne la taille du domaine de la dimension i_m de A. Les auteurs supposent que $\forall m, n_m = 2^{k_m}, k_m \in N$.

Définition 3.2.9 (Opérateur somme partielle) *La première somme partielle est un couple* (P_1^m, R_1^m) *pour une dimension* i_m *de* A *défini comme suit :*

$$P_1^m(A) = \sum_{l=0}^{1} A[i_0, \ldots, 2i_m + l, \ldots, i_{d-1}]$$

$$R_1^m(A) = \sum_{l=0}^{1} (-1)^l A[i_0, \ldots, 2i_m + l, \ldots, i_{d-1}]$$

La $i^{ème}$ *somme partielle est obtenue en appliquant le couple* (P_1^m, R_1^m) *i fois sur la même dimension* i_m*.*

P_1^m désigne la première somme partielle et R_1^m la première somme résiduelle. P_1^m prend la somme des agrégats des couples de cellules voisines au long de chaque dimension i_m de A. En revanche, R_1^m prend la différence. La Figure 3.18 montre un exemple de cube de données A et le résultat de l'application de l'opérateur de première somme partielle (P_1^m,P_1^m) sur la dimension i_1 de ce cube. La partie grisée du résultat représente la somme et celle non grisée représente la différence. Le cube initial A peut être reconstruit sans perte à partir de la somme et de la différence comme le montre la Figure 3.18.

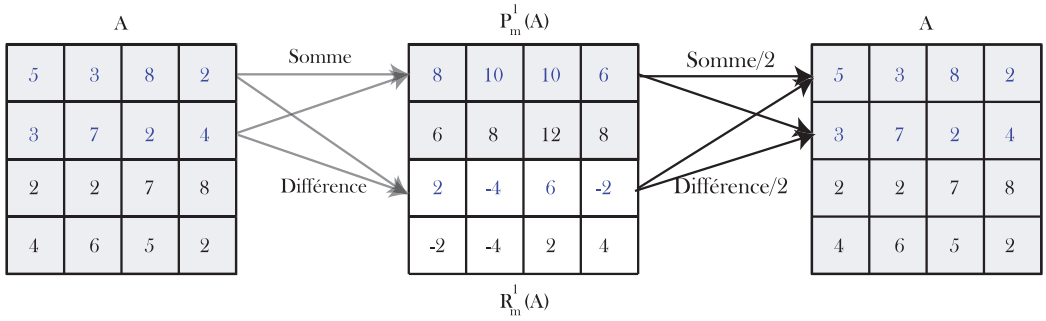


FIG. 3.18 – Décomposition d'un cube en ondelettes

Définition 3.2.10 (Agrégation totale) *Une agrégation totale* S^m *le long d'une dimension* i_m *est calculée en appliquant* P_1^m *dans une succession de cascade de longueur* $log_2 n_m$*, où* n_m *désigne la taille du domaine de* i_m*.*

$$S^m = P^m_{log_2 n_m}(A) = \underbrace{P^m_1(P^m_1(\ldots(P^m_1}_{log_2 n_m}(A))))$$

Définition 3.2.11 (Vue agrégée) *Une vue agrégée d'un cube de données A est générée en agrégeant totalement A le long d'une ou plusieurs dimensions.*

Définition 3.2.12 (Elément-vue) *Un élément-vue d'un cube de données A est générée par une agrégation totale S^m, une agrégation partielle P^m ou une agrégation résiduelle R^m sur un nombre quelconque de dimensions i_m.*

Deux types d'éléments-vues sont à distinguer : les éléments-vues résiduels et les éléments-vues intermédiaires. Les premiers sont générés par une agrégation résiduelle R^m. Tout élément-vue non résiduel est dit intermédiaire. La Figure 3.18 illustre un cas d'éléments-vues résiduels et partiels. Le cube A est agrégé en deux éléments-vues en appliquant l'opérateur P^1_m (partie grisée) et P^1_m (partie non grisée). Ces éléments-vues peuvent être utilisés pour synthétiser ce cube. Un ensemble d'éléments-vues est un ensemble d'agrégation partielle ou résiduelle d'éléments-vues. Cet ensemble est dit complet si le cube initial peut être reconstruit à partir des éléments-vues de cet ensemble. Ce dernier est dit redondant s'il n'existe pas d'éléments-vues pouvant être générés à partir d'autres éléments du même ensemble.

Définition 3.2.13 (Base d'éléments-vues) *Un ensemble d'éléments-vues forme une base pour un cube de données A si cet ensemble est complet. De plus, si cet ensemble n'est pas redondant, l'ensemble forme une base non redondante.*

La notion de bases d'éléments est importante pour la sélection des éléments-vues à matérialiser. Une base d'éléments-vues non redondante peut représenter le cube de données sans surcharge au niveau du volume des données.

Définition 3.2.14 (Graphe d'éléments-vues) *Un graphe d'éléments-vues organise ces éléments et fournit une structure de données pour évaluer la complétude, la non redondance et le bénéfice d'un ensemble d'éléments-vues. Il organise l'ensemble des éléments-vues suivant leurs dépendances directes et indirectes.*

La Figure 3.19 montre un exemple de graphe d'éléments-vues d'un cube de profondeur trois [6]. Le graphe décompose le cube de données A en ondelettes d'éléments-vues organisées suivant deux dépendances représentées par des flèches en pointillés. Les vues résiduelles sont représentées en blanc, les vues intermédiaires en gris et les vues agrégées en noir.

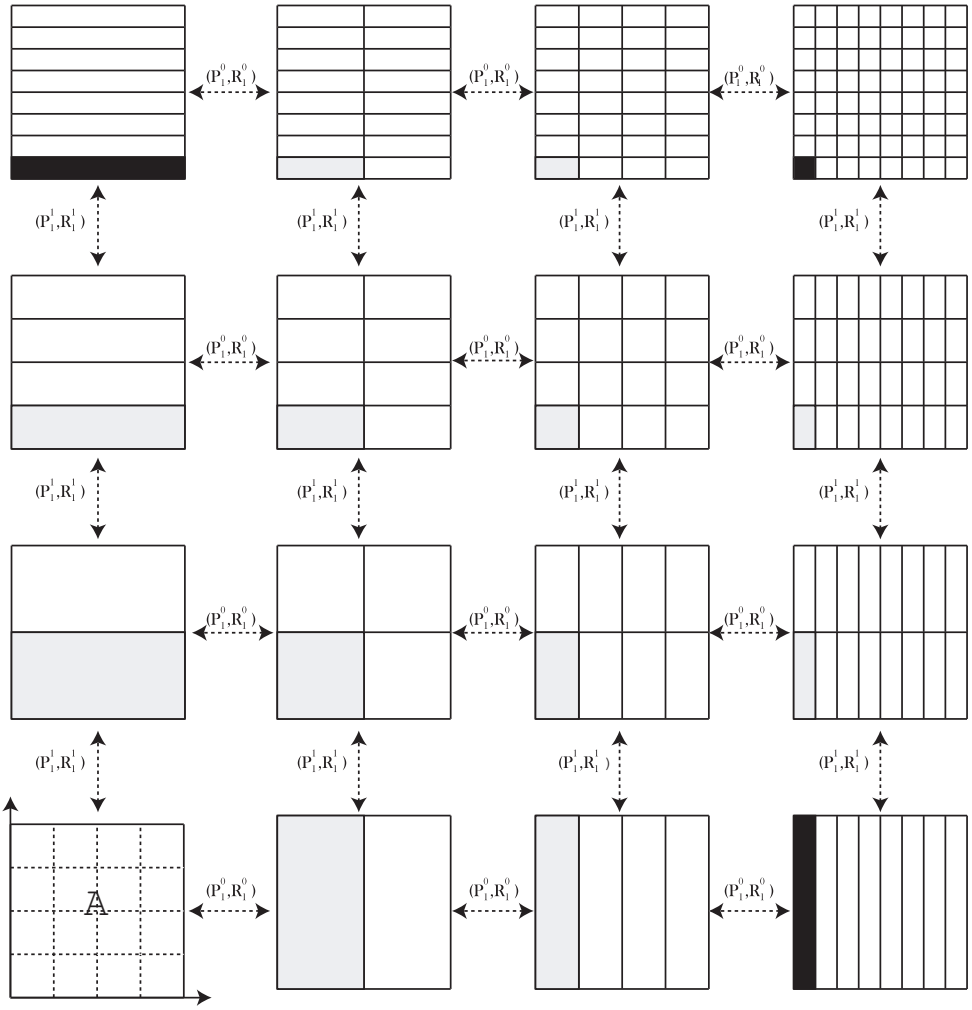


FIG. 3.19 – Graphe d'éléments-vues

La complétude et la redondance d'un ensemble d'éléments-vues peuvent être déterminées en évaluant sa couverture dans un plan de fréquence à d dimensions (*d-dim frequency plane*) correspondant à chaque bloc du graphe d'éléments-vues. À chaque élément-vue est associé une position X_m et une taille W_m dans le plan à d dimensions. La position et la taille

[6] La profondeur est obtenue en appliquant un log_2 sur la taille du domaine de chaque dimension.

sont déterminées par la cascade des opérateurs d'agrégation partielle et résiduelle appliquée pour obtenir cet élément-vue. Par exemple, la position de la racine représentant le cube de données A est $X(A) = [0, 0, \ldots, 0]$ et sa taille est $W(A) = [1, 1, \ldots, 1]$ car A couvre tout le plan.

L'application de l'agrégation partielle au long de la dimension i_m sur le cube A génère un élément-vue dont la position et la taille sont respectivement $X(P_1^m(A)) = [0, 0, \ldots, 0, \ldots, 0]$ et $W(P_1^m(A)) = [0, 0, \ldots, \frac{1}{2}, \ldots, 0]$. D'autre part, l'application de l'agrégation résiduelle R_m^1 au long de la dimension i_m sur le cube A génère un élément-vue dont la position et la taille sont respectivement $X(R_1^m(A)) = [0, 0, \ldots, \frac{1}{2}, \ldots, 0]$ et $W(P_1^m(A)) = [0, 0, \ldots, 0, \ldots, 0]$.

De manière plus générale, si un élément-vue ev dont la position et la taille sont respectivement $X(ev) = [x_0, x_1, \ldots, x_m, \ldots, x_{d-1}]$ et $W(ev) = [w_0, w_1, \ldots, w_m, \ldots, w_d]$, alors l'application des opérateurs P_m^1 et R_m^1 donne les postions et les tailles suivantes.

$$X(P_1^m(ev)) = [x_0, x_1, \ldots, x_m, \ldots, x_{d-1}]$$
$$W(P_1^m(ev)) = [w_0, w_1, \ldots, \frac{w_m}{2}, \ldots, w_d]$$
$$X(R_1^m(ev)) = [x_0, x_1, \ldots, x_m + \frac{w_m}{2}, \ldots, x_{d-1}]$$
$$W(R_1^m(ev)) = [w_0, w_1, \ldots, \frac{w_m}{2}, \ldots, w_d]$$

Le chevauchement d'éléments-vues dans le plan à d-dimension est déterminé en évaluant l'intersection des rectangles multidimensionnels, représentant les élément-vues, définis par leurs tailles et positions. L'intersection $ev_1 \cap ev_2$ des éléments-vues ev_1 et ev_2 est donnée par la formule suivante :

$$ev_1 \cap ev_2 = \begin{cases} 0 & \text{si } \exists m \; x_m^{ev_1} > x_m^{ev_2} + w_m^{ev_2} \\ & \text{ou } x_m^{ev_2} > x_m^{ev_1} + w_m^{ev_1} \\ I(ev_1, ev_2) & \text{sinon} \end{cases}$$

où $I(ev_1, ev_2) = \prod_{m=0}^{d-1} (min(x_m^{ev_1} + w_m^{ev_1}, x_m^{ev_2} + w_m^{ev_2})) - max(x_m^{ev_1}, x_m^{ev_2})$.

Un ensemble d'éléments-vues est non-redondant si et seulement si les éléments-vues de cet ensemble ne se chevauchent pas ; c'est-à-dire : $\forall ev_1 \; ev_2, ev_1 \cap ev_2 = 0$. Cet ensemble est complet si et seulement s'il couvre tout le plan multidimensionnel.

Ensemble des éléments-vues	**Base**	**Redondance**	**Coût de traitement**	**Coût de stockage**
$\{ev_3, ev_6, ev_7\}$	oui	non	3	4
$\{ev_1, ev_5, ev_6\}$	oui	non	3	4
$\{A\}$	oui	non	4	4
$\{ev_1, ev_4\}$	oui	non	4	4
$\{ev_7, ev_8\}$	oui	non	4	4
$\{ev_2, ev_3, ev_5, ev_6\}$	oui	non	4	4
$\{ev_0, ev_1, ev_7\}$	oui	oui	0	8
$\{ev_1, ev_7\}$	non	oui	0	8
$\{ev_3, ev_7\}$	non	non	3	4
$\{ev_2, ev_3, ev_5\}$	non	non	4	3

TAB. 3.4 – Résumé du coût de traitement et de stockage de certains ensembles d'éléments-vues du graphe

3.2.4.1 Algorithmes de sélection d'éléments-vues

Étant donnée la fréquence d'accès aux vues, deux algorithmes de sélection d'éléments-vues représentant un cube de données sont proposés. Dans le premier algorithme, les éléments-vues sélectionnés constituent une base non redondante et complète minimisant le coût des requêtes définies sur ce cube et le coût de stockage des éléments-vues de la base. Le deuxième algorithme relaxe la contrainte de non redondance pour sélectionner un ensemble d'éléments-vues minimisant le temps d'exécution des requêtes sous la contrainte d'espace de stockage. Pour des raisons de simplicité, nous présentons dans les sections suivantes les deux algorithmes à travers un exemple.

Considérons deux éléments-vues ev_1 et ev_7 qui sont à rechercher avec une fréquence de $f_{ev_1} = f_{ev_2} = 0,5$. Les autres éléments-vues ont une fréquence nulle. Le coût de traitement pour chaque transition dans le graphe d'éléments-vues est indiqué à la Figure 3.20 par (i, j), où i est le coût d'agrégation et j est le coût de synthèse. Le Tableau 3.4 donne le coût de stockage et de traitement de différents ensembles d'éléments-vues.

Les six premiers ensembles listés au Tableau 3.4 sont des bases redondantes du cube de données A. Deux de ces ensembles $\{ev_3, ev_6, ev_7\}$ et $\{ev_1, ev_5, ev_6\}$ ont le coût de traitement minimal. Par exemple, le coût de traitement de $\{ev_1, ev_5, ev_6\}$ est calculé à partir de $ev_1 \rightarrow^0 ev_1 + ev_5 \rightarrow^1 ev_7, ev_1 \rightarrow^1 ev_2, ev_2 \rightarrow^1 ev_7$. Le reste des ensembles d'éléments ne sont pas optimaux.

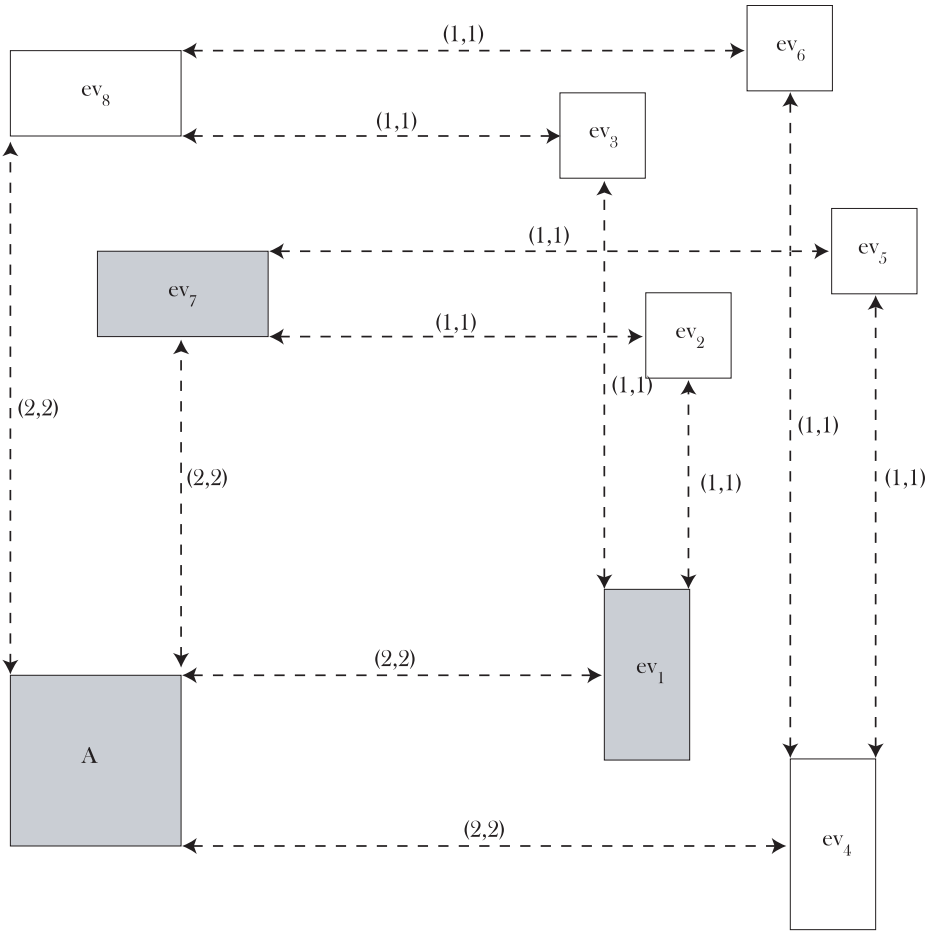


FIG. 3.20 – Exemple d'un graphe d'éléments-vues avec le coût de synthèse et d'agrégation des éléments-vues

D'autre part, la matérialisation du cube A donne un temps de traitement de 4 ($A \rightarrow^2 ev_1 + A \rightarrow^2 ev_7$). De plus, l'ensemble redondant d'éléments-vues $\{ev_0, ev_1, ev_7\}$ a un coût de traitement nul mais augmente l'espace de stockage de huit unités. L'ensemble de vues incomplet $\{ev_1, ev_7\}$ a un coût de traitement nul mais il est impossible de reconstruire toutes les vues du cube A à partir de cet ensemble d'éléments-vues.

3.2.5 Sélection de vues matérialisées en analysant la charge de requêtes

Agrawal *et al.* proposent une approche automatique de sélection d'index et de vues matérialisées basée sur l'analyse syntaxique de la charge des requêtes [ACN00, ACN01].

L'architecture de leur système est présentée à la Figure 3.21. Le système prend en entrée une charge de requêtes estimée représentative et procède en plusieurs étapes pour proposer une configuration de vues à matérialiser.

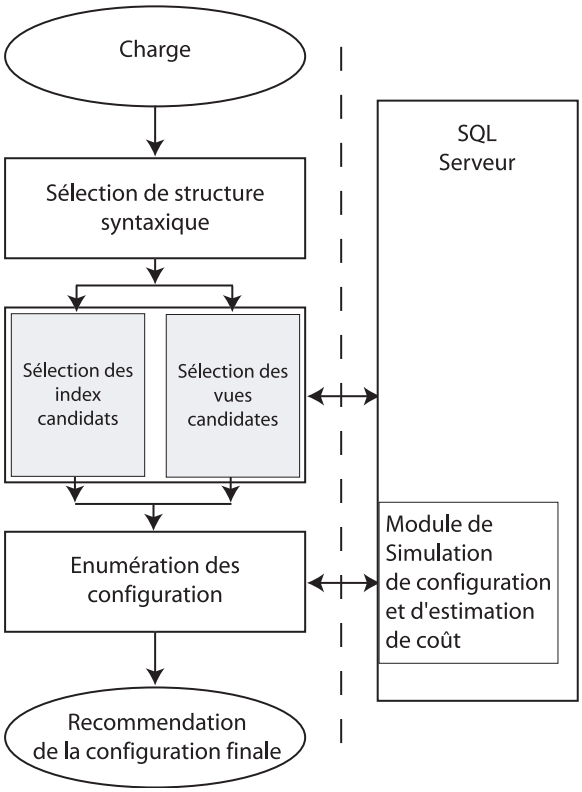


FIG. 3.21 – Architecture du système de Agrawal *et al.*

La première étape consiste à identifier des index, des vues matérialisées et des index sur ces vues qui soient syntaxiquement pertinents et potentiellement exploitables par les requêtes de la charge. Le stratégie adoptée pour la sélection d'index est présentée à la Section 3.1.6. Nous ne nous intéressons dans cette section qu'à l'étude de la sélection des vues matérialisées.

Les vues matérialisées syntaxiquement pertinentes sont déterminées à partir de sous-ensembles de tables (*table-subsets*). Un sous-ensemble de tables est composé d'une ou plusieurs tables présentes dans la base de données. Une vue est dite pertinente si l'ensemble des sous-tables dont cette vue est dérivée le sont aussi. Un sous-ensemble de tables T est dit syntaxiquement pertinent si la matérialisation d'une vue sur T est bénéfique pour au moins une requête de la charge et que cette vue réduit le coût de cette charge. La pertinence du

sous-ensemble de tables T est mesurée par les deux métriques suivantes :

- $TS - Cost(T)$ le coût total des requêtes exploitant les tables de T
- $TS - Weight(T) = \sum_i cost(q_i) \frac{\text{somme des tailles des tables dans } T}{\text{somme des tailles des tables exploitées par } q_i}$

Dans un premier temps, les sous-ensembles de tables dont le coût, calculé par $TS-Cost$, ne dépasse pas un seuil donné par l'utilisateur, sont élagués. La métrique $TS - Weight$ est appliquée sur les sous-ensembles de tables restantes.

La deuxième étape consiste à solliciter l'optimiseur de requêtes afin d'élaguer certaines vues matérialisées dérivées des sous-ensembles de tables générés dans l'étape précédente. Une vue est élaguée si elle n'est bénéfique à aucune requête de la charge. L'idée est que si une vue n'est bénéfique à aucune requête de la charge, elle ne fait pas, par conséquent, partie de la configuration finale de vues. Étant donnés une requête q_i de la charge, un ensemble de vues V_i, et éventuellement un ensemble d'index sur ces vues, la meilleure configuration de vues et d'index est choisie pour q_i. Ce choix est réalisé à l'aide d'un algorithme glouton.

À la fin de la deuxième étape, la meilleure configuration de vues et d'index est sélectionnée pour chaque requête de la charge. Dans la troisième étape, il s'agit de fusionner les vues proposées afin qu'elles soient bénéfiques à plusieurs requêtes. La fusion est réalisée en gardant la structure en commun des vues parentes et en généralisant les différences de ces vues parentes. Cette fusion est guidée par un modèle de coût. En effet, si le coût d'exploitation de la vue v_{12}, obtenue par la fusion des vues v_1 et v_2, est seulement légèrement supérieure (en fonction d'un seuil fixé par l'utilisateur) au coût d'exploitation des vues parentes, alors la fusion n'est pas pertinente.

3.2.6 Conclusion

Nous résumons au Tableau 3.5 les travaux traitant le problème de sélection de vues matérialisées selon :

- la méthode de construction des vues candidates : treillis, graphe de vues AND-OR, plan d'exécution des requêtes, ondelettes d'éléments-vues ou analyse syntaxique de la charge ;
- la méthode de construction de la configuration finale de vues : algorithme glouton, algorithme génétique ou résolution du problème du sac à dos ;

– le modèle de coût utilisé : fonction mathématique ou appel à l'optimiseur de requêtes.

	Construction de l'ensemble de vues candidates					**Construction de la configuration finale de vues**			**Estimation du coût**	
TRAVAUX	*Treillis*	*Graphe de vues AND-OR*	*Plan d'exécution*	*Ondelettes*	*Analyse de la charge*	*Algorithme glouton*	*Algorithme génétique*	*Sac à dos*	*Fonction mathématique*	*Optimiseur de requêtes*
Harinarayan *et al.* [HRU96]	X					X			X	
Baralis *et al.* [BPT97]	X					X			X	
Uchiyama *et al.* [URT99]	X					X			X	
Shukla *et al.* [SDN00]	X					X			X	
Nadeau *et al.* [NT02]	X					X			X	
Gupta *et al.* [Gup97, Gup99, GM05]		X				X			X	
Baril *et al.* [BB03b]			X					X	X	
Valluri *et al.* [VVK02]			X			X			X	
Zhang *et al.* [ZYY01]			X				X		X	
Smith *et al.* [SLJ04]				X					X	
Agrawal *et al.* [ACN00]					X	X				X

TAB. 3.5 – Classification des travaux sur la sélection de vues matérialisées

3.3 Problème de sélection simultanée d'index et de vues matérialisées

Les travaux de recherche traitant la sélection de vues matérialisées et d'index s'orientent dans leur majorité vers une sélection séquentielle des vues et des index sur les vues, ou vers une sélection isolée des index ou des vues matérialisées. Cependant, les index et les vues matérialisées sont fondamentalement des structures physiques similaires [ACN00]. En effet, les deux structures sont redondantes, accélèrent le temps d'exécution des requêtes, partagent la même ressource de stockage et impliquent une surcharge de maintenance pour le système suite aux mises à jour des données. Les vues et les index peuvent alors être en interaction. La présence d'un index sur une vue matérialisée peut en effet rendre celle-ci plus "attractive" et *vice versa*.

Peu de travaux se sont portés sur la sélection conjointe des vues matérialisées et des index. Dans [ACN00], trois alternatives pour l'énumération conjointe de l'espace des index et des vues matérialisées sont présentées. La première alternative, dénotée MVFIRST, tend à sélectionner des vues matérialisées en premier, puis les index pour une charge donnée en présence des vues préalablement sélectionnées. La deuxième alternative, dénotée INDFIRST, sélectionne en premier les index, puis les vues. La troisième alternative, dénotée *joint enumeration*, traite la sélection des index, des vues matérialisées et des index sur ces vues en une seule itération. Les auteurs ne donnent cependant aucun détail sur le fonctionnement de cette alternative, mais ils affirment qu'elle est meilleure que les deux premières.

Dans la suite, nous présentons en détail les travaux de Bellatreche *et al.* et de Rizzi et Saltarelli, qui traitent la sélection simultanée d'index et de vues matérialisées. Bellatreche *et al.* proposent une sorte de compétition entre deux agents qui se disputent l'espace de stockage tantôt pour les index et tantôt pour les vues matérialisées. Rizzi et Saltarelli proposent une approche orientant l'indexation ou la matérialisation en fonction de la sélectivité des attributs de la clause `Where` des requêtes et de la granularité de leur clause `Group by`.

3.3.1 Travaux de Bellatreche *et al.*

Bellatreche *et al.* ont traité le problème de distribution de l'espace de stockage entre les vues matérialisées et les index de manière itérative afin de minimiser le coût total d'exécution des requêtes d'une charge donnée [BKS00]. L'intuition sous-jacente de l'approche proposée est la suivante. Initialement, l'administrateur estime les espaces disques S_V et S_I alloués pour stocker les vues matérialisées et les index respectivement. L'estimation peut être établie en utilisant une distribution uniforme ou aléatoire de l'espace de stockage $S = S_V + S_I$ alloué pour stocker les vues et les index. Les vues et les index sont sélectionnés séquentiellement à l'aide d'algorithmes de sélection sous contrainte d'espace de stockage. L'ensemble de vues et d'index est désigné comme une solution initiale au problème de sélection d'index et de vues.

L'approche reconsidère itérativement la solution initiale dans le but de réduire davantage le coût d'exécution des requêtes en redistribuant efficacement l'espace de stockage entre les vues et les index. Elle s'appuie sur une compétition perpétuelle entre deux agents, l'espion des index et l'espion des vues, dont les rôles sont décrits comme suit.

- L'espion des index vole de l'espace réservé pour stocker les vues. L'espace ainsi récupéré est utilisé pour créer d'autres index à la place des vues élaguées. L'opération est validée si le coût d'exécution des requêtes est réduit.
- De manière analogue, l'espion des vues dérobe de l'espace réservé pour stocker les index afin de créer d'autres vues dans le but de minimiser le coût total d'exécution des requêtes.

La sélection d'index et de vues commence par appliquer l'espion qui réduit le plus le coût des requêtes. Le processus de sélection s'arrête lorsqu'il n'y a plus de réduction du coût des requêtes. L'architecture générale du système de sélection est présentée à la Figure 3.22.

Le contrôle d'admission de vol vérifie que chaque tentative de vol par un espion contribue à la réduction du coût des requêtes. La contribution est évaluée à l'aide d'un modèle de coût. Le contrôleur d'espion, quant à lui, s'assure qu'à tout moment, un seul espion est actif. Les algorithmes de sélection d'index et de vues sont présentés et discutés dans [YKL97, BSMB02].

Lors du remplacement d'une vue (respectivement, d'un index) par un ou plusieurs index (respectivement, vues), plusieurs politiques de remplacement sont considérées. Les politiques

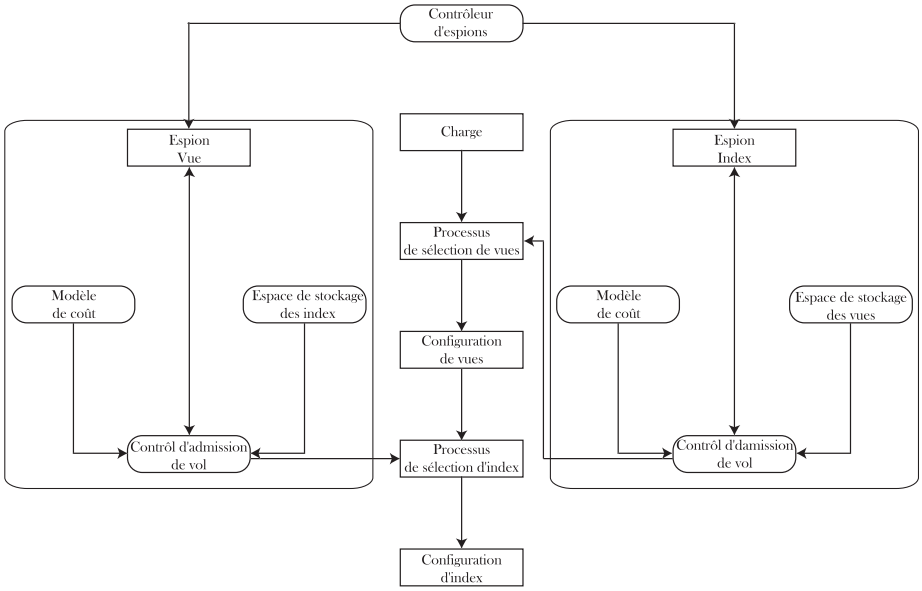


FIG. 3.22 – Architecture du système de Bellatreche *et al.*

de remplacement lors de l'application de l'espion des index sont la vue la moins fréquemment utilisée VMFU et la vue de plus petite taille VPPT. Dans le cas VMFU, l'espion des index élimine la vue la moins utilisée car elle contribue le moins à la réduction du coût. Dans le cas VPPT, la vue de plus petite taille est remplacée par un ou plusieurs index car la taille d'une vue est en moyenne plus grande que celle d'un index. De la même manière, les politiques de remplacement lors de l'application de l'espion des vues sont l'index le moins fréquemment utilisé IMFU et le plus grand index PGI. En politique IMFU, l'espion des vues élimine l'index le moins utilisé car il contribue le moins à la réduction du coût. En mode PGI, le plus grand index est éliminé car la taille d'un index est généralement plus petite que celle d'une vue.

3.3.2 Travaux de Rizzi et Saltarelli

Rizzi et Saltarelli proposent une approche qui détermine *a priori* un compromis entre l'espace de stockage alloué aux index et aux vues matérialisées en se basant sur les requêtes de la charge [RS03]. L'idée de Rizzi et Saltarelli est que le facteur clé dans l'optimisation des performances des requêtes est leur niveau d'agrégation, défini par la liste des attributs de la clause `Group by`, et la sélectivité des attributs présents dans les clauses `Having` et

`Where`. En effet, la matérialisation offre un grand bénéfice aux requêtes comportant des agrégations de granularité grossière (nombre faible d'attributs dans la clause `Group by`) car elles génèrent peu de groupes dans un grand nombre de n-uplets et, par conséquent, l'accès à une petite vue est moins coûteux que l'accès aux tables de base. D'autre part, les index donnent leur meilleur bénéfice avec des requêtes contenant des attributs dont la sélectivité est élevée car elles sélectionnent peu de n-uplets et, par conséquent, l'accès à un nombre élevé de n-uplets inutiles est évité. Les requêtes avec des agrégations fines et de fortes sélectivités encouragent l'indexation. En contrepartie, les requêtes avec des agrégations grossières et de faibles sélectivités encouragent la matérialisation. La Figure 3.23 résume ces cas. Cependant, aucune prédiction ne peut être envisagée pour le cas des requêtes qui ne relèvent pas des deux cas précédents.

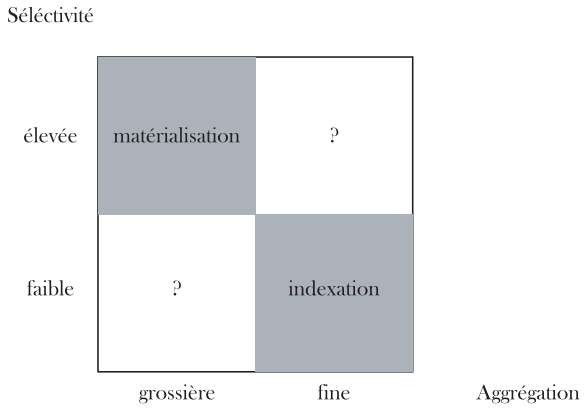


FIG. 3.23 – Recommandation de la technique d'optimisation en fonction de la sélectivité et du niveau d'agrégation

L'approche de Rizzi et Saltarelli estime les fractions de l'espace de stockage S destinées à stocker l'ensemble des vues matérialisées V et celui des index I, notées S_V et S_I, respectivement. Étant donnée une charge Q, la matérialisation totale se produit lorsque toute requête $q \in Q$ peut être résolue par une ou plusieurs vues de V sans aucun recours aux tables de base. Dans ce cas, $S_V^{total} = \sum_{v \in V_Q} taille(v \in V_Q)$, où V_Q est l'ensemble de toutes les vues utilisées par les requêtes de Q. Une vue matérialisée est complètement indexée si pour toute requête $q \in Q$ exploitant cette vue, tous les index utiles pour cette requête sont créés. Étant donné un ensemble de vues V occupant un espace S_V, l'espace $(S_V)_I^{total}$ dénote l'espace requis pour stocker tous les index construits sur les vues de V et les tables de base.

$(0)_I^{total}$ est l'espace requis pour totalement indexer seulement les tables de base. $(S_V^{total})_I^{total}$ est l'espace requis pour stocker tous les index dans le cas d'une matérialisation totale.

Rizzi et Saltarelli ont pu déterminer le compromis optimal de l'espace alloué pour la matérialisation et celui alloué pour l'indexation comme suit :

$$S_I = \begin{cases} S_I^{inf} & \text{si } S_I^{benefice} < S_I^{inf}, \\ S_I^{benefice} & \text{si } S_I^{inf} \leq S_I^{benefice} \leq S_I^{sup}, \\ S_I^{supp} & \text{si } S_I^{benefice} > S_I^{sup}. \end{cases}$$

$$\text{et } S_V = S - S_I$$

où $S_I^{inf} = max(0, S - S_V^{total})$ dénote la borne inférieure de l'espace de stockage des index, $S_I^{sup} = min((S_V)_I^{total}, S)$ dénote la borne supérieure de l'espace de stockage des index et $S_I^{benefice} = \frac{S}{1+\frac{\sum_{q \in Q} benefice_V(q)}{\sum_{q \in Q} benefice_I(q)}}$ mesure le ratio entre le bénéfice de matérialisation $benefice_V$ et d'indexation $benefice_I$ calculé sur la totalité de la charge.