Chapitre 5

Classification non supervisée pour la sélection de vues matérialisées

5.1 Introduction

Parmi les techniques adoptées dans l'implémentation des entrepôts de données relationnels afin d'optimiser les performances, la matérialisation des vues et l'indexation sont les plus efficaces [RS03]. Rappelons que les vues matérialisées sont des structures physiques qui améliorent l'accès aux données en précalculant des résultats intermédiaires. Les requêtes des utilisateurs sont alors traitées efficacement en accédant seulement aux données stockées dans les vues matérialisées, sans accéder aux données sources. Cela réduit le coût des jointures, entre la table de faits et les tables dimensions, qui nécessitent le parcours d'un grand volume de données. Cependant, l'utilisation des vues matérialisées nécessite un espace de stockage additionnel et entraîne une surcharge de maintenance lors du rafraîchissement de l'entrepôt de données.

L'une des difficultés les plus importantes dans la conception physique des entrepôts de données est la sélection d'un ensemble de vues matérialisées pertinentes, appelé configuration de vues, qui minimise le coût total d'exécution des requêtes et de maintenance des vues, sous contrainte d'espace de stockage. En effet, si d est le nombre de dimensions dans le schéma d'un entrepôt de données qui ne contient aucune hiérarchie, le nombre de vues candidates à sélectionner est alors égal à $n = 2^d$. La complexité du problème de sélection de vues

matérialisées est de $O(2^n)$, où n est le nombre des vues candidates dans le schéma.

Dans ce chapitre, nous proposons une stratégie de sélection de vues matérialisées exploitant la classification non supervisée de requêtes. L'idée d'utiliser la classification est née du fait que plusieurs requêtes ayant une syntaxe similaire sont susceptibles d'avoir une forte probabilité d'être résolues à partir d'une vue matérialisée dont la syntaxe est également proche de ces requêtes. Le problème qui se pose alors est de regrouper les requêtes afin d'obtenir des classes de requêtes syntaxiquement similaires. C'est pourquoi nous utilisons une méthode de classification. D'autre part, nous ne connaissons pas *a priori* le nombre de classes à obtenir car ce nombre dépend des requêtes de la charge et de leur syntaxe. Cet aspect nous a conduit à utiliser une méthode de classification non supervisée.

Notre stratégie de sélection de vues matérialisées analyse syntaxiquement les requêtes de la charge afin de construire un contexte de classification. Ce contexte nous offre une représentation matricielle des requêtes de la charge en fonction de l'information extraite de ces requêtes. Pour réaliser la classification non supervisée, nous définissons des mesures de similarité et de dissimilarité entre les requêtes du contexte. Ces mesures sont définies sous forme d'une fonction d'homogénéité interne des classes et d'hétérogénéité entre classes. Une classe homogène est alors une classe par laquelle la syntaxe des requêtes est similaire. Les classes de requêtes obtenues par la classification constituent un point de départ pour construire l'ensemble de vues candidates. À ce stade, nous associons à chaque requête différente d'une classe donnée une vue candidate. Le nombre de vues candidates peut être élevé si le nombre de requêtes l'est aussi. Nous développons alors un processus de fusion des requêtes de chaque classe afin de réduire le nombre de vues candidates. Ces vues ont la particularité d'être pertinentes pour plusieurs requêtes. Pour satisfaire la contrainte d'espace de stockage alloué pour stocker les vues sélectionnées, nous proposons un algorithme glouton de sélection de vues qui construit la configuration finale de vues. Cet algorithme exploite un modèle de coût permettant d'estimer le coût d'accès aux données à travers une ou plusieurs vues matérialisées, ainsi que le coût de leur stockage.

Ce chapitre est organisé comme suit. Nous commençons par présenter à la Section 5.2 notre stratégie de sélection de vues matérialisées. Nous montrons ensuite à la Section 5.3 comment est construite la configuration de vues candidates par le processus de fusion. Nous détaillons à la Section 5.4 le modèle de coût et l'algorithme de sélection de vues matérialisées.

Pour valider notre approche, nous présentons des expérimentations à la Section 5.7. Nous terminons enfin par une conclusion et une discussion à la Section 5.8.

5.2 Stratégie de sélection de vues matérialisées

Le principe de notre stratégie de sélection de vues matérialisées est présenté à la Figure 5.1. Nous supposons que nous disposons d'une charge de requêtes représentatives pour laquelle nous souhaitons optimiser le temps d'exécution en matérialisant des vues. La première étape construit à partir de la charge un contexte de classification. Ce contexte est modélisé comme une matrice ayant autant de lignes que de requêtes et autant de colonnes que d'attributs extraits des requêtes de la charge. La classification exploite des mesures de similarité et dissimilarité permettant de regrouper l'ensemble des requêtes qui sont syntaxiquement similaires. Dans chaque groupe de requêtes, nous procédons à un processus de fusion afin de construire une configuration de vues candidates. La configuration finale de vues est ensuite créée à l'aide d'un algorithme glouton. Cette étape exploite des modèles de coût qui évaluent le coût d'accès aux données en utilisant les vues, ainsi que le coût de stockage de ces vues.

Nous procédons comme suit pour proposer une configuration de vues à matérialiser :

- extraction de la charge de requêtes,
- analyse de la charge pour extraire les attributs représentatifs des requêtes,
- construction du contexte de classification non supervisée des requêtes,
- application de l'algorithme de classification Kerouac [JN03a],
- fusion des requêtes de chaque classe obtenue par classification pour construire l'ensemble des vues candidates,
- construction de la configuration finale de vues à matérialiser.

5.2.1 Analyse des requêtes de la charge

La charge que nous considérons est un ensemble de requêtes de projection, sélection et jointure. De telles requêtes sont composées d'opérations de jointures, de prédicats de sélection et d'opérations d'agrégation. Elles peuvent être exprimées en algèbre relationnelle

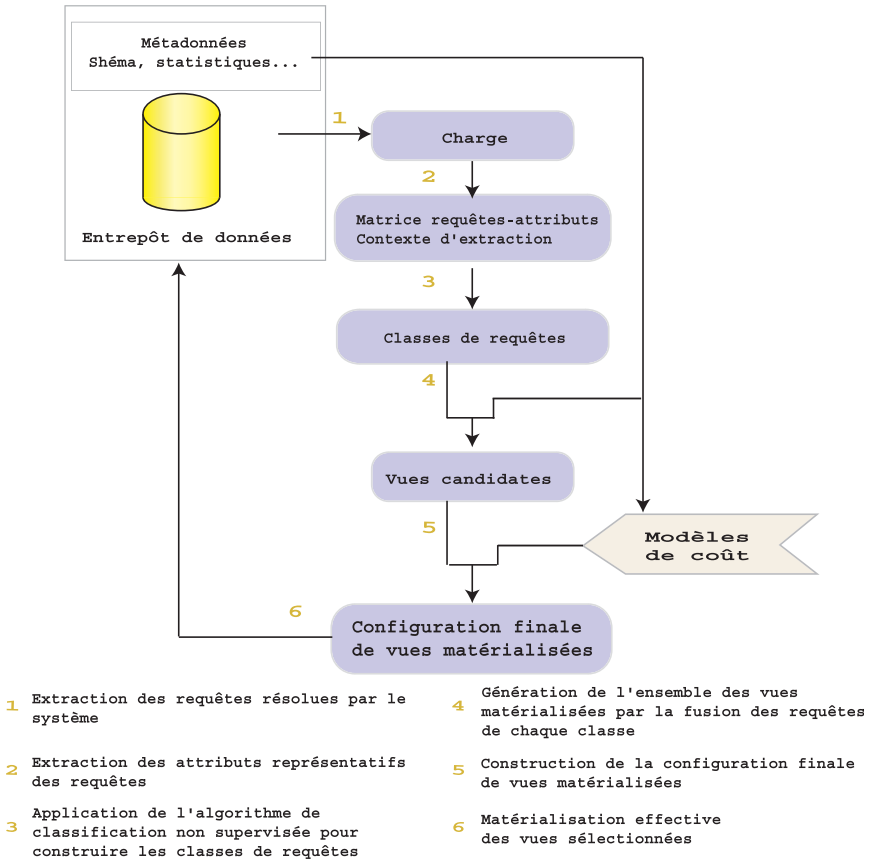

FIG. 5.1 – Principe de notre stratégie de sélection de vues matérialisées

(sur un schéma en étoile) comme suit :

$$\pi_{G,M}\sigma_S(F \bowtie D_1 \bowtie D_2 \bowtie \ldots \bowtie D_d)$$

où S est une conjonction de prédicats simples sur les attributs des tables dimensions, G est un ensemble d'attributs des tables dimensions $D_i, i \in \{1, \ldots, d\}$ (les attributs de la clause `Groupe by`), et M est un ensemble de mesures agrégées, chacune étant définie en appliquant un opérateur d'agrégation distributif sur une mesure de la table de faits F. Par exemple, la requête q_1 de la Figure 5.2 peut être exprimée comme suit :

$$q_1 = \pi_{sales.time\_id,sum(quantity\_sold)}\sigma_{fiscal\_day=2}(sales \bowtie times)$$

.

La première étape de notre stratégie de sélection de vues matérialisées consiste à extraire de la charge des attributs qui sont représentatifs de chaque requête. Nous considérons représentatifs les attributs qui sont présents dans les clauses `Where` (jointures et prédicats de sélection) et `Group by` des requêtes de la charge. Nous conservons aussi pour chaque requête les opérateurs d'agrégation et les tables jointes. Ces informations sont nécessaires lors de la génération des vues matérialisées sélectionnées.

Une requête q_i est vue comme une ligne d'une matrice composée de cellules qui correspondent aux attributs représentatifs. Le terme général qa_{ij} de cette matrice est mis à un si l'attribut extrait est présent dans la requête, et à zéro dans le cas contraire. Nous conservons aussi, dans une matrice annexe, les associations existant entre les attributs de jointure et les requêtes. Cette matrice est utilisée pour fixer les contraintes de la classification. Ces contraintes assurent que chaque classe présente les mêmes prédicats de jointure. C'est une pré-condition du processus de fusion présenté à la Section 5.3. Nous illustrons cette étape d'analyse syntaxique de la charge à travers l'exemple suivant : à partir de la charge présentée à la Figure 5.2, nous construisons le contexte de classification du Tableau 5.1.

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}
q_1	1	1	1	0	0	0	0	0	0	0	0	0
q_2	0	0	0	1	0	1	1	1	1	0	0	0
q_3	1	1	1	1	1	0	0	0	0	1	1	1
q_4	0	0	0	1	0	1	1	1	1	0	0	0
..												

a_1 times.time_id
a_2 times.fiscal_day
a_3 sales.time_id
a_4 products.prod_id
a_5 products.prod_category
a_6 sales.promo_id
a_7 promotions.promo_id
a_8 sales.prod_id
a_9 promotions.promo_category
a_{10} sales.cust_id
a_{11} customers.marital_status
a_{12} customers.cust_id

TAB. 5.1 – Exemple de contexte de classification

```
(q1)  select sales.time_id, sum(quantity_sold)
      from sales, times
      where sales.time_id = times.time_id
      and times.fiscal_day = 2
      group by sales.time_id ;
(q2)  select sales.prod_id, sum(amount_sold)
      from sales, products, promotions
      where sales.prod_id = products.prod_id
      and sales.promo_id = promotions.promo_id
      and promotions.promo_category = 'newspaper'
      group by sales.prod_id ;
(q3)  select sales.cust_id, sum(amount_sold)
      from sales, customers, products, times
      where sales.cust_id = customers.cust_id
      and sales.prod_id = products.prod_id
      and sales.time_id = times.time_id
      and times.fiscal_day = 3
      and customers.cust_marital_status ='single'
      and products.prod_category ='Women'
      group by sales.cust_id ;
(q4)  select sales.prod_id, sum(amount_sold)
      from sales, products, promotions
      where sales.prod_id = products.prod_id
      and sales.promo_id = promotions.promo_id
      and promotions.promo_category = 'TV'
      group by sales.prod_id ;
...
```

FIG. 5.2 – Exemple de charge

5.2.2 Construction de l'ensemble de vues candidates

Dans la pratique, il est difficile de rechercher toutes les vues matérialisées qui sont syntaxiquement pertinentes pour une charge de requêtes, car l'espace de recherche est exponentiel [ACN00]. Pour réduire la taille de cet espace, nous proposons de regrouper les requêtes qui possèdent une certaine similarité au niveau syntaxique. En effet, deux requêtes présentant une forte similarité syntaxique peuvent être résolues en utilisant une seule vue matérialisée dont la syntaxe est proche de celle de ces deux requêtes. C'est pourquoi nous exploitons la classification non supervisée des requêtes comme heuristique pour déterminer l'ensemble des vues candidates.

Dans notre cadre d'étude, les requêtes présentant une forte similarité syntaxique sont

des requêtes dont les représentations binaires sont très proches dans la matrice requêtes-attributs. Ainsi, la matrice requêtes-attributs constitue notre contexte de classification non supervisée.

Pour construire les classes de requêtes, nous utilisons des mesures de similarité et de dissimilarité entre les requêtes. La combinaison de ces deux mesures constitue notre mesure de classification et assure que les classes de requêtes sont homogènes. Nous définissons dans les sections suivantes les mesures de similarité, de dissimilarité, la mesure de classification et l'algorithme de classification non supervisée des requêtes.

5.2.2.1 Mesure de similarité

Soit QA une matrice requêtes-attributs qui a les requêtes $Q = \{q_i, i = 1..n\}$ comme lignes et les attributs $A = \{a_j, j = 1..p\}$ comme colonnes. La valeur q_{ij} est égale à un si l'attribut a_j est extrait de la requête q_i. Sinon, q_{ij} est égale à zéro. Nous décrivons une requête q_i par un vecteur de p valeurs binaires : $q_i = [q_{i1}, ..., q_{ip}]$.

Ce modèle de description permet de comparer deux requêtes. Par exemple, nous pouvons considérer les requêtes q_1 et q_2 comme fortement similaires si leur vecteurs $[q_{11}, ..., q_{1p}]$ et $[q_{21}, ..., q_{2p}]$ présentent une majorité de cellules égales. Par exemple, les requêtes q_2 et q_4 de la Figure 5.2 ont une représentation binaire similaire comme le montre le Tableau 5.1, bien qu'elles soient différentes. Ceci introduit la définition de similarité et de dissimilarité entre les requêtes.

5.2.2.2 Similarité et dissimilarité entre requêtes

Nous définissons la similarité et la dissimilarité entre requêtes par les deux fonctions $\delta_{sim}(q_{k_j}, q_{l_j})$ et $\delta_{dissim}(q_{k_j}, q_{l_j})$, qui mesurent respectivement la similarité et la dissimilarité entre les deux requêtes q_{k_j} et q_{l_j} suivant l'attribut a_j.

$$\delta_{sim}(q_{k_j}, q_{l_j}) = \begin{cases} 1 \text{ si } q_{k_j} = q_{l_j} = 1 \\ 0 \text{ sinon} \end{cases}$$

$$\delta_{dissim}(q_{k_j}, q_{l_j}) = \begin{cases} 0 \text{ si } q_{k_j} = q_{l_j} \\ 1 \text{ si } q_{k_j} \neq q_{l_j} \end{cases}$$

La première fonction définit la similarité entre q_k et q_l suivant l'attribut a_j : deux requêtes q_k et q_l sont considérées similaires suivant l'attribut a_j si et seulement si $q_{k_j} = q_{l_j} = 1$; c'est-à-dire, l'attributs a_j est présent dans ces deux requêtes.

La deuxième fonction définit la dissimilarité entre deux requêtes q_k et q_l suivant l'attribut a_j : les deux requêtes q_k et q_l sont considérées dissimilaires suivant l'attribut a_j si et seulement si $q_{k_j} \neq q_{l_j}$; c'est-à-dire, si une des deux requêtes ne contient pas l'attribut a_j.

Notons qu'il n'existe pas une complète symétrie entre la similarité et la dissimilarité : des requêtes non similaires suivant un attribut ne sont pas nécessairement dissimilaires suivant ce même attribut. Soit q_1 et q_2 deux requêtes telles que $q_{1j} = 0$ et $q_{2j} = 0$, respectivement. Nous avons alors $\delta_{sim}(q_{1_j}, q_{2_j}) = 0$ (q_1 et q_2 ne sont pas considérées comme similaires) et $\delta_{dissim}(q_{1_j}, q_{2_j}) = 0$ (q_1 et q_2 ne sont pas considérées comme dissimilaires). L'absence de symétrie complète souligne un point important de notre définition de la similarité entre requêtes : l'absence d'un attribut dans deux requêtes ne constitue pas un élément de similarité entre ces deux requêtes.

Ces mesures peuvent être étendues sur un ensemble d'attributs A de manière à obtenir le degré de la similarité et dissimilarité globale entre deux requêtes :

$$sim(q_k, q_l) = \sum_{j=1}^{p} \delta_{sim}(q_{k_j}, q_{l_j})$$

$$dissim(q_k, q_l) = \sum_{j=1}^{p} \delta_{dissim}(q_{k_j}, q_{l_j})$$

où $0 \leq sim(q_k, q_l) \leq p$ et $0 \leq dissim(q_k, q_l) \leq p$. Ainsi, plus $sim(q_a, q_b)$ (respectivement, $dissim(q_a, q_b)$) est proche de p, plus q_a et q_b peuvent être considérées comme globalement similaires (respectivement, dissimilaires).

5.2.2.3 Similarité et dissimilarité entre ensembles de requêtes

Comme nous l'avons défini pour deux requêtes, nous introduisons deux fonctions qui prennent en compte le degré de similarité et dissimilarité entre deux ensembles de requêtes.

Soient C_a et C_b deux sous-ensembles de l'ensemble des requêtes Q. Pour rendre compte du niveau de similarité (respectivement, de dissimilarité) entre des ensembles de requêtes, nous utilisons la fonction $Sim(C_a, C_b)$ (respectivement, $Dissim(C_i, C_j)$) qui détermine le nombre de similarités (respectivement, de dissimilarités) entre les deux ensembles de requêtes C_a et C_b ($C_a \neq C_b$) comme suit :

$$Sim(C_a, C_b) = \sum_{q_k \in C_a, q_l \in C_b} sim(q_k, q_l)$$

$$\text{et } Dissim(C_a, C_b) = \sum_{q_k \in C_a, q_l \in C_b} dissim(q_k, q_l)$$

où $0 \leq Sim(C_a, C_b) \leq card(C_a) \times card(C_b) \times p$ et $0 \leq Dissim(C_a, C_b) \leq card(C_a) \times card(C_b) \times p$. Ainsi, plus $Sim(C_a, C_b)$ (respectivement, $Dissim(C_a, C_b)$) est proche de $card(C_a) \times card(C_b) \times p$, plus les ensembles C_a et C_b peuvent être considérés comme similaires (respectivement, dissimilaires).

5.2.2.4 Similarité et dissimilarité dans un ensemble de requêtes

La notion de similarité (respectivement, dissimilarité) dans un ensemble de requêtes correspond au nombre de similarités (respectivement, dissimilarités) entre les requêtes d'un même ensemble C_a. Elle consiste en une extension de la similarité et de la dissimilarité des fonctions définies entre deux ensembles de requêtes :

$$Sim(C_a) = \sum_{q_k \in C_a, q_l \in C_a, k<l} sim(q_k, q_l)$$

$$\text{et } Dissim(C_a) = \sum_{q_k \in C_a, q_l \in C_a, k<l} dissim(q_k, q_l)$$

où $0 \leq Sim(C_a) \leq \frac{card(C_a) \times (card(C_a)-1) \times p}{2}$ et $0 \leq Dissim(C_a) \leq \frac{card(C_a) \times (card(C_a)-1) \times p}{2}$. Ainsi, plus $Sim(C_a)$ (respectivement, $Dissim(C_a)$) est proche de $\frac{card(C_a) \times (card(C_a)-1) \times p}{2}$, plus C_a contient des requêtes qui sont considérées comme globalement similaires (respectivement, dissimilaires).

5.2.2.5 Classification des requêtes

Le problème sous-jacent à la classification non supervisée est : "étant donné un ensemble d'objets O, déterminer une partition P_{nat} de O[1] que l'on dénomme naturelle et qui reflète la structure interne des données" [Jou03]. Cette partition doit être telle que ses classes sont constituées d'objets présentant une forte similarité et que les objets de classes différentes présentent une forte dissimilarité.

En nous basant sur les fonctions définies précédemment, nous construisons la mesure de qualité de la classification comme suit :

$$Q(P_h) \quad = \quad \sum_{\substack{\text{a=1..z,} \\ \text{b=1..z,a < b}}} Sim(C_a, C_b) + \sum_{a=1}^{z} Dissim(C_a)$$

où $0 \leq Q(P_h) \leq \sum_{i=1..z, j=1..z, i<j} card(C_i) \times card(C_j) \times p + \sum_{i=1}^{z} \frac{card(C_i) \times (card(C_i)-1) \times p}{2}$.

Pour atteindre cet objectif, nous utilisons la mesure $Q(P_h)$, qui permet de capturer l'aspect naturel d'une partition. Ainsi, $Q(P_h)$ mesure simultanément les dissimilarités entre objets de même classe de la partition P_h et les similarités entre objets de classes différentes. Nous pouvons donc dire que $Q(P_h)$ est définie comme une fonction d'homogénéité interne des classes et d'hétérogénéité entre classes. Par conséquent, les partitions présentant une forte homogénéité intra-classe et une forte disparité inter-classes posséderont une faible valeur de $Q(P_h)$ et apparaîtront comme les plus naturelles.

L'algorithme de classification que nous avons adopté est l'algorithme Kerouac proposé par Jouve et Nicoloyannis [JN03a]. Nous avons choisi cet algorithme car il permet non seulement de classer les requêtes d'une manière qui nous intéresse en intégrant facilement notre mesure $Q(P_h$, mais aussi d'intégrer des contraintes dans le processus de classification. Ce dernier point est très intéressant car il fournit un moyen de satisfaire une pré-condition du processus de fusion, à savoir que les requêtes d'une classe donnée doivent avoir les mêmes conditions de jointures.

Avant de présenter l'algorithme de classification, nous donnons la définition du voisinage

[1] Dans notre cas d'étude, les objets sont les requêtes de la charge.

utilisé dans cet algorithme [Jou03].

Définition 5.2.1 (Voisinage d'une partition) *Une partition P_g appartient à $Voisinage(P_h)$, le voisinage d'une partition P_h, ou encore l'ensemble des partitions voisines d'une partition P_h, si :*

- *P_g peut être obtenue à partir de P_h par segmentation d'une classe C_j de P_h selon une variable (attribut) a_i,*
- *P_g peut être obtenue à partir de P_h par fusion de deux classes de P_h,*
- *$P_g = P_h$*

L'algorithme 11 de classification non supervisée adopte une heuristique gloutonne de type graphe d'induction. Il procède par segmentations/fusions successives de classes de partitions. À partir d'une partition grossière des requêtes, l'algorithme évolue itérativement de partition en partition en suivant le principe suivant : il passe d'une partition P_i vers une partition P_{i+1} appartenant à son voisinage de telle façon qu'elle réduise le plus la mesure $Q(P_h)$. L'algorithme s'achève lorsque $P_i = Pi + 1$. L'algorithme renvoie la partition $P_{nat} = P_i$ la plus proche de la partition naturelle P_{nat}. Dans l'algorithme de classification, les contraintes sont modélisées par des variables supplémentaires, qui ne peuvent servir pour la segmentation de classes lors du processus de segmentation et dont l'influence dans la valeur de l'aspect naturel des partitions est pondérée.

Algorithme 11 Algorithme de classification non supervisée des requêtes

1: soit P_0 une partition grossière de Q
2: $i \leftarrow 0$
3: Déterminer $voisinage(P_i)$
4: Déterminer la meilleure partition $P_{i+1} \in voisinage(P_i)$ selon la mesure $Q(P_h)$
5: **si** $P_{i+1} = P_i$ **alors**
6: aller en 11
7: **sinon**
8: $i \leftarrow i + 1$
9: aller en 3
10: **fin si**
11: $P_{nat} \leftarrow P_i$

Notons que la classification par l'algorithme Kerouac a d'autres propriétés intéressantes :

– la complexité de calcul est relativement basse car elle est log-linéaire suivant le nombre de requêtes et linéaire suivant le nombre d'attributs ;
– l'algorithme est performant avec un nombre élevé d'objets (requêtes) ;
– l'algorithme peut être appliqué sur des données distribuées [JN03b].

À la Figure 5.3, nous illustrons le résultat de la classification non supervisée sur un contexte composé de vingt requêtes et de quatre attributs extraits de ces requêtes (`cust_gender`, `prod_category`, `chan_category` et `prod_name`). Les cases grisées représentent les classes de requêtes obtenues par cette classification.

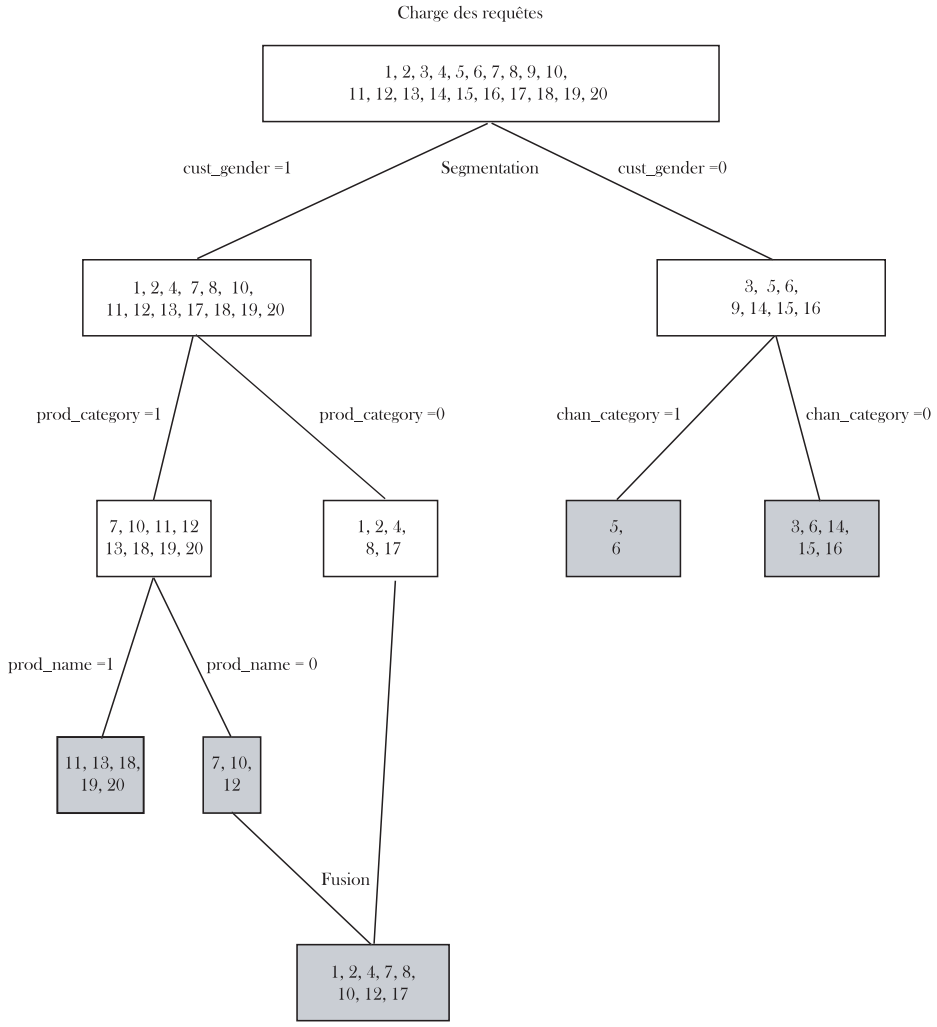

FIG. 5.3 – Exemple de classification des requêtes d'une charge

5.3 Fusion des vues

Si nous matérialisons toutes les vues dérivées à partir des classes de requêtes obtenues dans l'étape précédente, nous pouvons obtenir un nombre élevé de vues matérialisées, surtout si le nombre de requêtes de la charge est élevé. Les vues d'une configuration obtenue de cette manière peuvent ne pas être matérialisables si l'espace de stockage alloué à ces vues par l'administrateur est limité. Au lieu de matérialiser chaque vue, il est plus judicieux de ne matérialiser que les vues qui peuvent être utilisées pour résoudre plusieurs requêtes. Nous proposons donc de fusionner les vues dérivées des requêtes des classes. Pour fusionner les vues, nous devons :

- énumérer l'espace des vues qui peuvent être fusionnées,
- déterminer comment orienter le processus de fusion,
- construire l'ensemble des vues fusionnées.

La fusion de vues est pertinente si les requêtes à partir desquelles sont dérivées ces vues sont fortement similaires. Comme nous regroupons des requêtes fortement similaires, il paraît judicieux d'appliquer le processus de fusion sur l'ensemble des requêtes présentes dans chaque classe. En effet, comme les requêtes d'une classe sont fortement similaires, il est plus aisé d'obtenir des vues fusionnées couvrant toutes les données de ces requêtes. Cela réduit significativement le nombre de combinaisons possibles lors de la fusion des vues.

Notre processus de fusion de vues est similaire à celui proposé dans [ACN01], mais moins complexe car nous réalisons la fusion des vues au sein de chaque classe obtenue par classification au lieu de l'ensemble des vues candidates dérivées directement de la charge.

Proposition 5.3.1 *La fusion de vues réalisée au sein de chaque classe obtenue par classification est moins complexe que la fusion réalisée à partir des vues candidates.*

Preuve 5.3.2 *Soient n le nombre de vues candidates dérivées directement des requêtes de la charge et $k > 1$ le nombre de classes que nous obtenons par la classification non supervisée des requêtes. Soient aussi $n_1, n_2, \ldots, n_k$ le nombre de vues candidates de chaque classe. Le nombre de vues à explorer dans le processus de fusion est égal à 2^n, si la fusion est réalisée à partir de l'ensemble des vues candidates générées à partir de la charge des requêtes. Dans ce cas, la complexité de fusion est de $O(2^n)$.*

D'autre part, le nombre de vues à explorer dans le processus de fusion est égal à $\sum_{i=1}^{k} 2^{n_i}$*, si la fusion est réalisée à partir des vues des k classes, sachant que* $\sum_{i=1}^{k} n_i = n$*. Sans perte de généralité, nous pouvons poser* $n_1 = n_2 = \cdots = n_k = \frac{n}{k}$*. Le nombre de vues à explorer est donc égal à* $\sum_{i=1}^{k} 2^{\frac{n}{k}} = k2^{n/k}$*. Dans ce cas, la complexité de la fusion est de* $O(2^{\frac{n}{k}})$*. La fusion réalisée au sein de chaque classe est donc moins complexe que la fusion réalisée à partir de l'ensemble total des vues candidates.*

Nous détaillons dans la section suivante comment fusionner deux vues et généralisons ce processus sur plusieurs vues.

5.3.1 Fusion d'un couple de vues

La fusion de deux vues candidates doit satisfaire les conditions suivantes :

- toutes les requêtes résolues par chaque vue doivent aussi être résolues par la vue obtenue par fusion,
- le coût d'exploitation du couple de vues ne doit pas être significativement plus grand que le coût d'exploitation de la vue fusionnée.

Soient (v_1, v_2) un couple de vues d'une même classe et $s_{11}, \ldots, s_{1m}$ les prédicats de sélection qui sont présents dans v_1 mais pas dans v_2. De manière duale, soient $s_{21}, \ldots, s_{1n}$ les prédicats de sélection présents dans v_2 mais pas dans v_1. La vue fusionnée v_{12} est obtenue en appliquant l'algorithme 12.

Algorithme 12 Merge_View_Pair

Entrée v_1 et v_2

1: Placer les opérations d'agrégation de v_1 et v_2 dans celles de v_{12}.
2: Placer l'union des attributs des projections et des clauses `Group by` de v_1 et v_2 dans les attributs de projection et la clause `Group by` de v_{12}.
3: Placer tous les attributs $s_{11}, \ldots, s_{1m}$ et $s_{21}, \ldots, s_{1n}$ dans la clause `Group by` de v_{12}.
4: Placer les prédicats de sélection communs à v_1 et v_2 dans les prédicats de sélection de v_{12}.
5: **retourner** v_{12}

La fusion de deux vues v_1 et v_2 est effective si coût$(v_{12}) \geq ((\text{ coût}(v_1) + \text{ coût}(v_2)) * x)$. Le calcul du coût est détaillé à la Section 5.4. La valeur de $x \in [1, 2]$[2] est fixée empiriquement par l'administrateur. Si elle est petite (respectivement, élevée), l'administrateur privilégie (respectivement, désavantage) la fusion des vues.

Proposition 5.3.3 *La vue obtenue par la fusion des vues v_1 et v_2 est la plus petite vue qui réponde aux requêtes résolues par v_1 et v_2.*

Preuve 5.3.4 *Pour démontrer que la vue obtenue par la fusion des vues v_1 et v_2 est la plus petite vue, nous démontrons qu'il n'existe aucune autre vue v'_{12} telle que les données de v'_{12} sont aussi incluses dans v_{12}.*

Dénotons respectivement par $\pi_{G_1,M_1}\sigma_{S_1}(F \bowtie \dots)$, $\pi_{G_2,M_2}\sigma_{S_2}(F \bowtie \dots)$ et $\pi_{G_{12},M_{12}}\sigma_{S_{12}}(F \bowtie \dots)$, les vues v_1, v_2 et v_{12}, où :

- *G_1 et G_2 sont les ensembles d'attributs des clauses* `Group by` *des vues v_1 et v_2 respectivement ;*
- *S_1 et S_2 sont les ensembles d'attributs des prédicats de sélection de v_1 et v_2 respectivement ;*
- *$G_{12} = G_1 \cup G_2 \cup (S_1 \cup S_2 - S_1 \cap S_2)$ est l'ensemble d'attributs de la clause* `Group by` *de la vue fusionnée v_{12} ;*
- *$S_{12} = S_1 \cap S_2$ est l'ensemble d'attributs des prédicats de sélection de la vue fusionnée v_{12}.*

Notons que les ensembles G_{12} et S_{12} sont obtenus en exécutant les lignes 1 et 2 de'Algorithme 12. Supposons maintenant que les données de la vue v'_{12}, dénotée $\pi_{G'_{12},M'_{12}}\sigma_{S'_{12}}(F \bowtie \dots)$ sont dans v_{12}. Cela signifie que les deux conditions suivantes sont satisfaites :

- $G_{12} \subset G'_{12}$
- $S'_{12} \subset S_{12}$

À partir de la première condition, nous pouvons conclure qu'il existe au moins un attribut a tel que $a \in G'_{12}$ et $a \notin G_{12}$. Comme nous avons $a \notin G_{12}$, alors $a \notin G_1$, $a \notin G_2$ et $a \notin S_1 \cup (S_2 - S_1 \cap S_2)$ car $G_{12} = G_1 \cup G_2 \cup (S_1 \cup S_2 - S_1 \cap S_2)$. Comme $a \notin G_1$ et $a \notin G_2$, alors a n'est dans aucune clause de v_2. Cela signifie que $a \notin G'_{12}$; ce qui contredit l'hypothèse

[2] Agrawal *et al.* ont montré que cette plage de valeurs fonctionne bien avec la plupart des bases de données et des charges [ACN00].

$a \in G'_{12}$.

À partir de la deuxième condition, nous pouvons conclure qu'il existe au moins un prédicat p *tel que* $p \in S_{12}$ *et* $p \notin S'_{12}$. *Comme nous avons* $p \in S_{12}$, *alors* $p \in S_1$ *et* $p \in S_2$ *car* $S_{12} = S_1 \cup S_2$. *Comme* $p \in S_1$ *et* $p \in S_2$, *alors* p *doit être dans chaque prédicat de la vue obtenue par la fusion de* v_1 *et* v_2. *Cela signifie que* $p \in S'_{12}$ *; ce qui contredit l'hypothèse* $p \notin S'_{12}$.

5.3.2 Algorithme de génération des vues par fusion

L'algorithme de génération de vues par fusion que nous avons élaboré est similaire à un algorithme de recherche de motifs fréquents. Le treillis des vues d'une classe donnée ressemble à celui des motifs fréquents. La Figure 5.4 montre un exemple de treillis de vues construit à partir de l'ensemble de vues candidates $\{a, b, c, d\}$ d'une classe donnée. Les nœuds du treillis représentent l'espace de recherche des vues que nous pouvons obtenir par fusion.

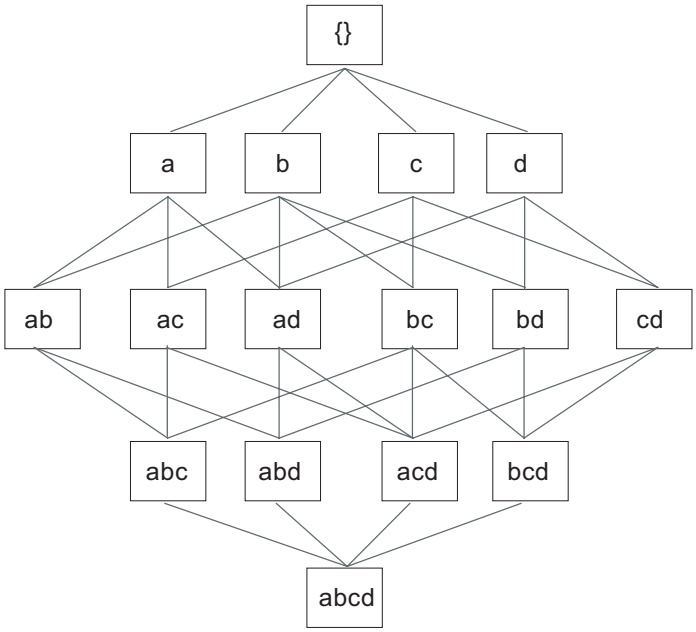

FIG. 5.4 – Treillis de vues

L'algorithme de génération des vues par fusion (Algorithme 13) utilise une approche itérative par niveau pour générer une nouvelle vue. Le treillis de vues est parcouru en largeur.

Algorithme 13 Mergin_View_Generation

1: $V = V_1$
2: **pour** $(k = 2\,;\, V_{k-1} \neq \emptyset\,;\, k++)$ **faire**
3: $C_k = \text{View\_Gen}(V_{k-1})$
4: $V \leftarrow V \cup C_k$
5: **pour tout** vue $v \in V$ **faire**
6: Enlever les parents de v dans V
7: **fin pour**
8: **fin pour**
9: **retourner** V

Algorithme 14 View_Gen

Entrée V_{k-1}
1: $C_k = \emptyset$
2: **pour tout** $v \in V_{k-1}$ **faire**
3: **pour tout** $u \in V_{k-1}$ **faire**
4: **si** $v[1] = u[1] \wedge \cdots \wedge v[k-2] = u[k-2] \wedge v[k-1] < u[k-1]$ **alors**
5: $c = \text{Merge\_View\_Pair}(v,u)$
6: **si** (coût $(c) \geq$ ((coût (v) + coût $(u)) * x)$) **alors**
7: $C_k = C_k \cup \{c\}$
8: **fin si**
9: **fin si**
10: **fin pour**
11: **fin pour**
12: **retourner** C_k

L'entrée de l'algorithme est V_1, l'ensemble de vues candidates extraites d'une classe donnée. Cet algorithme retourne un ensemble de vues candidates obtenues par fusion. Dans la $k^{\text{ème}}$ itération, l'ensemble de vues V_{k-1} obtenu par la fusion des vues du $k-1^{\text{ème}}$ niveau du treillis, calculées dans la dernière itération, est utilisé pour générer l'ensemble C_k de k-vues candidates. Cet ensemble est ajouté à l'ensemble des vues candidates V (ligne 4). Les parents de chaque vue obtenue par fusion sont ensuite supprimés de l'ensemble des vues fusionnées V (lignes 5 à 7).

La fonction de génération de vues par fusion View_Gen (V_{k-1}) (Algorithme 14), appelée à la ligne 3 de l'algorithme 13, prend en argument l'ensemble V_{k-1} et renvoie l'ensemble C_k. Deux vues v et u de V_{k-1} forment une k-vue c si et seulement si elles ont $(k-2)$-vues en commun. Cela est exprimé en utilisant l'ordre lexicographique de la condition de la

ligne 4 de l'algorithme 14. Cet ordre, dénoté $v[1]\dots v[k-2]v[k-1]$ pour la $k-1$-vue v et $u[1]\dots u[k-2]u[k-1]$ pour $k-1$-vue u, permet de déterminer si les deux k-vues peuvent être fusionnées pour obtenir une vue c.

La fonction Merge_View_Pair(v,u) (Algorithme 12), appelée à la ligne 5 de View_Gen génère une nouvelle vue c. La condition de la ligne 6 assure, après génération de la k-vue à partir des $(k-1)$-vues, que la vue candidate n'a pas un coût significativement plus grand que celui de ses parents.

Propriétés de l'algorithme de fusion

Il est possible pour une vue fusionnée générée à la ligne 3 de l'algorithme 13 d'être fusionnée à son tour dans l'itération suivante (lignes 2 à 8). Cela permet de fusionner plusieurs vues.

Le nombre de nouvelles vues à explorer par l'algorithme peut être exponentiel en taille de vues candidates de chaque classe V, dans le pire cas. La condition de la ligne 6 de View_Gen(V_{k-1}) peut réduire cette complexité. En effet, si les vues v et u ne peuvent pas être fusionnées, alors aucune autre vue ne peut être dérivée à partir de ces deux vues dans les itérations suivantes. Cette propriété peut réduire significativement la taille de l'espace de recherche des vues fusionnées.

Notons finalement que les résultats de l'algorithme ne dépendent pas de l'ordre lexicographique utilisé à la condition de la ligne 4 de View_Gen (V_{k-1}). Cet ordre permet d'éviter la redondance lors du calcul des vues fusionnées en parcourant chaque vue du treillis une seule fois.

5.4 Modèles de coût

Le nombre de vues candidates est généralement proportionnel à la taille de la charge en entrée. Il n'est donc pas pratique de matérialiser toutes les vues candidates car l'espace de stockage est souvent limité. Pour pallier ces limitations, nous utilisons des modèles de coût permettant de ne matérialiser que les vues les plus pertinentes. Dans la plupart des modèles de coût proposés dans les entrepôts de données, le coût d'une requête q est supposé proportionnel au nombre de n-uplets de la vue exploitée par q [GR98]. Nous adoptons le même principe dans nos modèles de coût. Dans la section suivante, nous détaillons le modèle

de coût qui estime la taille d'une vue donnée.

Soient $ms(F)$ la taille maximale de la table de faits F, $|F|$ le nombre de n-uplet de F, D_i_ID la clé primaire de la dimension D_i, $|D_i\_ID|$ la cardinalité de l'attribut D_i_ID ou le nombre de n-uplet de la dimension D_i, et d le nombre de dimensions. $ms(F)$ s'exprime comme suit :

$$ms(F) = \prod_{i=1}^{d} |D_i\_ID|.$$

Soient $ms(v)$ la taille maximum d'une vue v ayant les attributs $a_1, a_2, \ldots, a_k$ dans sa clause `Group by`, où k est le nombre d'attributs de la clause `Group by` et $|a_i|$ la cardinalité de l'attribut a_i. $ms(V)$ s'exprime alors :

$$ms(v) = \prod_{i=1}^{k} |a_i|.$$

Golfarelli *et al.* [GR98] ont proposé une estimation du nombre de n-uplets d'une vue v en utilisant la formule de Yao [Yao77] comme suit :

$$|v| = ms(v) \times \left[1 - \prod_{i=1}^{|F|} \frac{ms(F) \times d - i + 1}{ms(F) - i + 1} \right]$$

où $d = 1 - \frac{1}{ms(v)}$.

Lorsque le ratio $\frac{ms(F)}{ms(v)}$ est suffisamment élevé, cette formule est bien approximée par la formule de Cardenas [Car75] :

$$|v| = ms(v) \times \left(1 - \left(1 - \frac{1}{ms(v)} \right)^{|F|} \right).$$

Les formules de Yao et Cardinas se basent sur l'hypothèse que les données sont uniformément distribuées. Tout biais dans les données tend à réduire le nombre de n-uplets de la vue agrégée. Ainsi, l'hypothèse d'uniformité tend à surestimer la taille des vues. Cette estimation ne prend pas en compte le degré du biais et donc donne des estimations grossières. Cependant, ces estimations ont l'avantage d'être simples à implémenter et rapides dans les calculs. D'autres méthodes utilisent l'échantillonnage de données et des lois statistiques pour estimer la taille d'une vue [SDNR96, CM99, NT01]. Ces travaux corrigent la précision dont souffre l'estimation par la formule de Yao et Cardinas, mais sont difficiles à mettre en œuvre.

Notons que de nouvelles méthodes d'estimation peuvent être aisément intégrées dans notre stratégie de sélection de vues car l'architecture de notre système est modulaire. Il suffit alors ce cas de changer le module estimant le coût.

À partir du nombre de n-uplets de v, nous estimons sa taille, en octets, comme suit :

$$taille(v) = |v| \times \sum_{i=1}^{k} taille(d_i)$$

où $taille(d_i)$ dénote la taille, en octets, de la dimension d_i de la vue v, et k est le nombre de ses dimensions. La taille de chaque dimension peut être obtenue directement à partir des métadonnées de l'entrepôt.

5.5 Fonctions objectifs

Dans cette section, nous décrivons trois fonctions objectifs permettant d'évaluer la variation du coût d'exécution des requêtes, en terme de nombre de n-uplets lus, induite par l'ajout d'une nouvelle vue. Le coût d'exécution d'une requête est assimilé au nombre de n-uplets de la table de faits si aucune vue n'est matérialisée ou au nombre de n-uplets des vues exploitées, dans le contraire. Le coût d'exécution de la charge est obtenu en additionnant tous les coûts d'exécution de chaque requête de la charge.

La première fonction objectif privilégie les vues fournissant le plus grand profit lors de l'exécution de requêtes. La deuxième privilégie les vues fournissant le plus grand bénéfice tout en occupant le minimum d'espace de stockage. La troisième fonction combine les deux premières fonctions afin de sélectionner dans un premier temps les vues fournissant le plus grand profit et de conserver ensuite seulement les requêtes occupant le moins d'espace de stockage lorsque celui-ci devient critique. La première fonction est utile lorsque l'espace de stockage n'est pas limité, la deuxième lorsque l'espace de stockage est petit et la troisième lorsque l'espace de stockage est relativement élevé.

5.5.1 Fonction objectif profit

Soient $V = \{v_1, ..., v_m\}$ un ensemble de vues candidates, $Config_V$ un ensemble final de vues à construire et $Q = \{q_1, ..., q_n\}$ une charge de requêtes.

La fonction objectif profit, notée P, est définie comme suit :

$$P_{/Config_V}(v_j) = (C(Q, Config_V) - C(Q, Config_V \cup \{v_j\}) - \beta\, C_{maintenance}(\{v_j\}))$$

où $v_j \notin Config_V$.

- $C(Q, Config_V)$ dénote le coût d'exécution des requêtes lorsque toutes les vues de $Config_V$ sont exploitées. Si cet ensemble est vide, $C(Q, \emptyset) = |Q| \times |F|$ car toutes les requêtes sont résolues en accédant à la table de faits $|F|$. Lorsqu'une vue v_j est ajoutée à $Config_V$, $C(Q, Config_V \cup \{v_j\}) = \sum_{k=0}^{|Q|} C(q_k, \{v_j\})$ dénote le coût d'exécution des requêtes en présence des vues de $Config_V \cup \{v_j\}$. Si la requête q_k exploite v_j, le coût $C(q_k, \{v_j\})$ est alors égal à C_{v_j} (nombre de n-uplets de v_j). Sinon, $C(q_k, \{v_j\})$ est égal à la valeur minimum entre $|F|$ et $C(q_k, \{v\})$ (temps d'exécution de q_k exploitant la vue $v \in Config_V$ avec $v \neq v_j$).
- Le coefficient $\beta = |Q|p(v_j)$ estime le nombre de mises à jour de la vue v_j. La probabilité de mise à jour $p(v_j)$ est égale à $\frac{1}{\text{nombre de vues}} \frac{\%\text{rafraîchissement}}{\%\text{interrogation}}$, où le ratio $\frac{\%\text{rafraîchissement}}{\%\text{interrogation}}$ représente la proportion de rafraîchissement par rapport à la proportion d'interrogation de l'entrepôt de données.
- $C_{maintenance}(\{v_j\})$ représente le coût de maintenance de la vue v_j.

5.5.2 Fonction objectif ratio profit/espace

Si la sélection de vues est réalisée sous la contrainte d'espace de stockage, la fonction objectif ratio profit/espace $R_{/Config_V}(v_j) = \frac{P_{/Config_V}(v_j)}{taille(v_j)}$ est utilisée. Cette fonction calcule le profit fourni par une vue v_j par rapport à l'espace de stockage $taille(v_j)$ qu'elle occupe.

5.5.3 Fonction objectif hybride

La contrainte sur l'espace de stockage peut être relâchée si l'espace de stockage est relativement élevé. La fonction objectif hybride H ne pénalise pas les vues "gourmandes" en espace de stockage si le ratio $\frac{espace\_restant}{espace\_total}$ est plus petit ou égal à un seuil donné α $(0 < \alpha \leq 1)$, où $espace\_restant$ et $espace\_total$ sont respectivement l'espace restant après l'ajout de la vue v_j et l'espace alloué pour stocker toutes les vues sélectionnées. Cette fonction est calculée en combinant les deux fonctions objectifs P et R comme suit :

$$H_{/Config_V}(v_j) = \begin{cases} P_{/Config_V}(v_j) & \text{si } \frac{espace\_restant}{espace\_total} > \alpha, \\ R_{/Config_V}(v_j) & \text{sinon.} \end{cases}$$

5.6 Algorithme de sélection de vues matérialisées

Notre algorithme de sélection de vues (Algorithme 15) est basé sur une recherche gloutonne dans l'ensemble des vues candidates V. La fonction objectif F doit être l'une des fonctions P, R ou H décrite dans la section précédente. Si R est utilisée, nous ajoutons en entrée de l'algorithme l'espace de stockage S alloué aux vues matérialisées. Si la fonction objectif H est utilisée, nous ajoutons en plus le seuil α.

À la première itération de l'algorithme, les valeurs de la fonction objectif sont calculées pour chaque vue de l'ensemble V. Le coût d'exécution de toutes les requêtes de la charge Q (le premier terme de la fonction F) est égal au coût d'exécution des requêtes à partir des tables de base. La vue v_{max} qui maximise F, si elle existe ($F_{/Config_V}(v_{max}) > 0$), est alors ajoutée à l'ensemble $Config$. Si la fonction R ou H est utilisée, l'espace total S est diminué de l'espace de stockage occupé par v_{max}.

Les valeurs de la fonction F sont ensuite recalculées pour chaque vue restante dans $V - Config_V$, car elles dépendent des vues sélectionnées présentes dans $Config_V$. Cela aide à prendre en compte les interactions qui peuvent exister entre les vues.

Algorithme 15 View_Configuration_Construction

1: $Config_V \leftarrow \emptyset$
2: **répéter**
3: $v_{max} \leftarrow \emptyset$
4: $F_{max} \leftarrow 0$
5: **pour tout** $v_j \subseteq V - Config_V$ **faire**
6: **si** $F_{/Config_V}(v_j) > F_{max}$ **alors**
7: $F_{max} \leftarrow F_{/Config_V}(v_j)$
8: $v_{max} \leftarrow v_j$
9: **fin si**
10: **fin pour**
11: **si** $F_{/Config_V}(v_{max}) > 0$ **alors**
12: $Config_V \leftarrow Config_V \cup \{v_{max}\}$
13: **fin si**
14: **jusqu'à** ($F_{/Config_V}(v_{max}) \leq 0$ ou $V - Config_V = \emptyset$)

Nous répétons ces itérations jusqu'à ce qu'il n'y ait plus d'amélioration ($F_{/Config_V}(v_{max}) \leq 0$) ou que toutes les vues aient été sélectionnées ($V - Config_V = \emptyset$). Si la fonction R ou H est utilisée, l'algorithme s'arrête aussi lorsque l'espace de stockage est atteint.

5.7 Expérimentations

Dans le but de valider notre approche de sélection de vues matérialisées, nous avons conduit des expérimentations sur un entrepôt de données test implémenté sous Oracle $9i$, sur un PC Pentium 2.4 GHz doté d'une mémoire de 512 Mo et un disque dur IDE de 120 Go. Cet entrepôt est composé de la table de faits `Sales` et de cinq tables dimensions `Customers`, `Products`, `Times`, `Promotions` et `Channels` (cf. Chapitre 4, Section 4.6).

La charge est composée de soixante et une requêtes décisionnelles (cf. Annexe B). Nous avons mesuré le temps d'exécution total des requêtes de la charge avant et après matérialisation des vues. Nous avons appliqué dans un premier temps notre stratégie de sélection de vues avec la fonction profit. Cela nous a fourni un ensemble de vues matérialisées sans avoir spécifier de contrainte sur l'espace de stockage. Dans ce cas, nous obtenons le nombre maximum de vues matérialisées et par conséquent la borne supérieure de l'espace de stockage que les vues peuvent occuper. Ensuite, nous avons appliqué la fonction objectif ratio profit/espace sous la contrainte d'espace de stockage. Nous avons mesuré le temps d'exécution des requêtes suivant plusieurs pourcentages de l'espace de stockage alloué pour la matérialisation des vues. Ces pourcentages sont calculés à partir de la borne supérieure de l'espace que peuvent occuper les vues (calculée lorsque la fonction profit est appliquée). Cela permet de faire varier l'espace de stockage dans un intervalle plus large.

La Figure 5.5 représente la variation du temps d'exécution des requêtes suivant l'espace de stockage alloué aux vues matérialisées. Le temps d'exécution décroît lorsque l'espace de stockage augmente. Cela est prévisible car nous créons toutes les vues suggérées par notre algorithme, et par conséquent, nous améliorons au maximum le temps d'exécution des requêtes. Nous observons aussi que le gain maximal est égal à $94,86\%$ et qu'il est atteint pour une occupation de l'espace de stockage de 100%. Cet espace correspond à l'espace total occupé par les vues matérialisées sélectionnées en appliquant la fonction objectif profit (aucune contrainte sur l'espace de stockage).

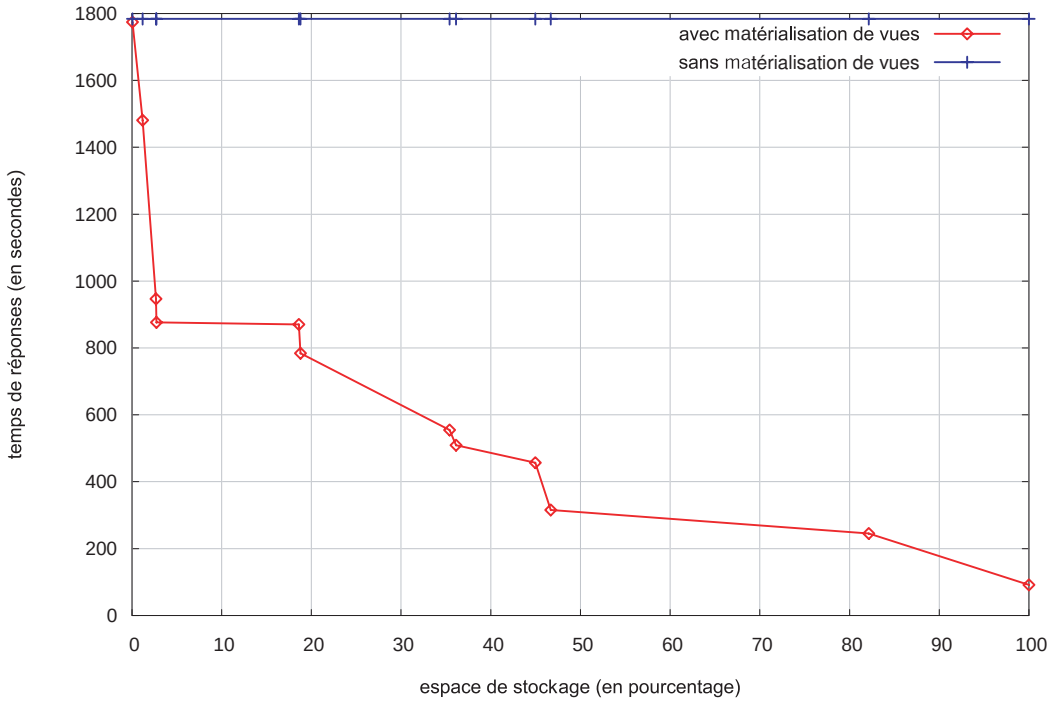

FIG. 5.5 – Résultats expérimentaux obtenus avec la fonction ratio profit/espace

Notre méthode garantit un gain substantiel même si l'espace de stockage est limité. Par exemple, pour une occupation de $35,41\%$ de l'espace de stockage, nous obtenons un gain en performance de $68,93\%$. Ce cas est intéressant lorsque l'administrateur de l'entrepôt de données ne possède pas l'espace suffisant pour matérialiser toutes les vues.

Nous avons reproduit les expérimentations précédentes avec la fonction objectif hybride. Nous avons fait varier les valeurs du paramètre α entre 0,1 et 1 par pas de 0,1. Les résultats obtenus dans ces expérimentations avec α allant de 0,1 à 0,7 et α allant de 0,8 à 1 sont respectivement égaux à ceux obtenus avec $\alpha = 0,1$ et $\alpha = 0,8$. Nous ne représentons donc à la Figure 5.6 que les résultats obtenus avec $\alpha = 0,1$ et $\alpha = 0,8$. Cette figure montre que pour des valeurs du pourcentage de l'espace de stockage inférieures ou égales à 18,6%, les fonction hybrides pour $\alpha = 0,1$ et $\alpha = 0,8$ se comportent comme la fonction objectif ratio espace/profit. Comme l'espace est réduit la fonction hybride converge rapidement vers la fonction ratio profit/espace. En revanche, pour les valeurs du pourcentage de l'espace de stockage supérieures à 18,6%, les résultats obtenus avec $\alpha = 0,8$ sont légèrement meilleurs que ceux obtenus avec $\alpha = 0,1$. Cela s'explique par le fait que pour les valeurs élevées de α,

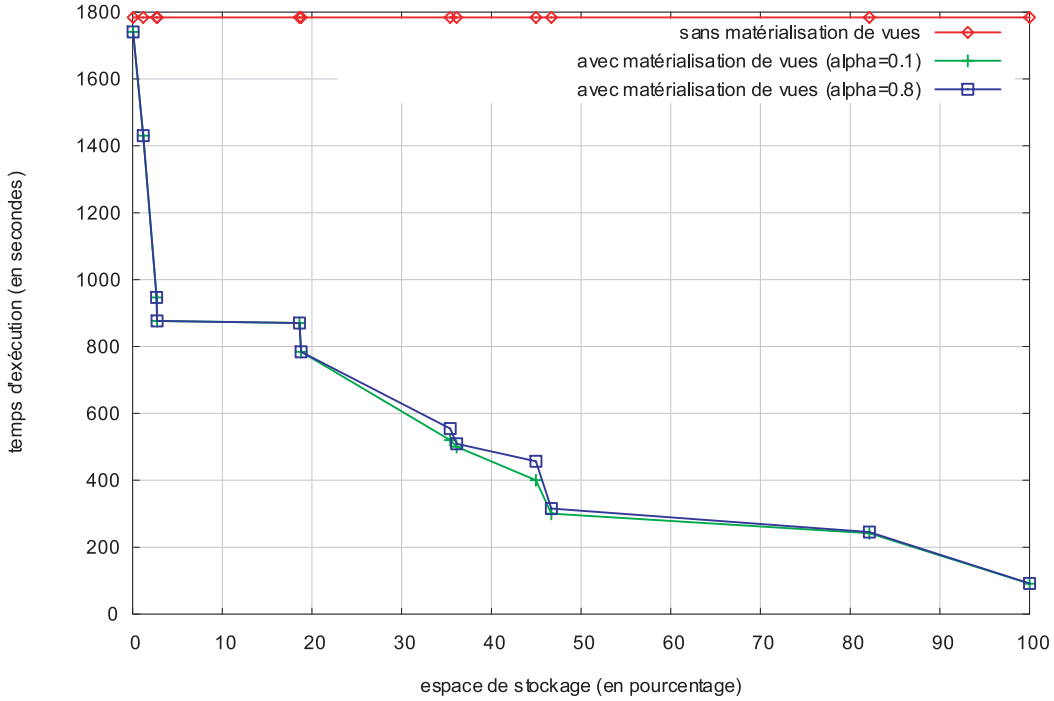

FIG. 5.6 – Résultat avec la fonction hybride

la fonction hybride choisit les vues apportant le plus de profit et donc améliorant le mieux le temps de réponse. Le gain maximal en temps d'exécution observé pour les valeurs 0,1 et 0,8 de α est égal à 94,86%.

Nous notons également que la majorité des données nécessaires pour répondre aux requêtes de la charge sont principalement retrouvées à partir des vues matérialisées au lieu des données sources de l'entrepôt de données. Dans la Figure 5.7, nous représentons le taux de couverture des requêtes par les vues matérialisées sélectionnées en fonction du pourcentage de l'espace de stockage réservé pour ces vues. Nous entendons taux de couverture le rapport entre le nombre de requêtes résolues en utilisant les vues sélectionnées par notre stratégie et le nombre total de requêtes de la charge. Le pourcentage de l'espace de stockage est calculé par rapport à la matérialisation totale des vues obtenues en appliquant la fonction profit. Rappelons que cette fonction donne le plus grand nombre de vues car elle privilégie les vues améliorant le mieux le temps d'exécution sans tenir compte de l'espace de stockage.

Pour une matérialisation maximale[3] (100% d'occupation de l'espace de stockage), les

[3]Il ne s'agit pas de la matérialisation totale de toutes les vues possibles mais de la matérialisation de

douze vues que nous fournit notre stratégie de sélection, couvrent la totalité des réponses des requêtes de la charge. Cela montre la pertinence des vues sélectionnées. Pour une occupation de $0,05\%$ de cet espace, $22,95\%$ des requêtes sont résolues à partir des vues sélectionnées. Ce point montre que même pour un espace de stockage restreint, notre stratégie de sélection permet de construire des vues qui couvrent le maximum de requêtes. Cette couverture augmente en fonction de l'espace de stockage. Plus on dispose d'espace, plus on matérialise et par conséquent plus on couvre de requêtes. Cette expérimentation montre que la sélection de vues matérialisées, orientée analyse syntaxique de la charge, est efficace pour garantir l'exploitation des vues sélectionnées par les requêtes de la charge.

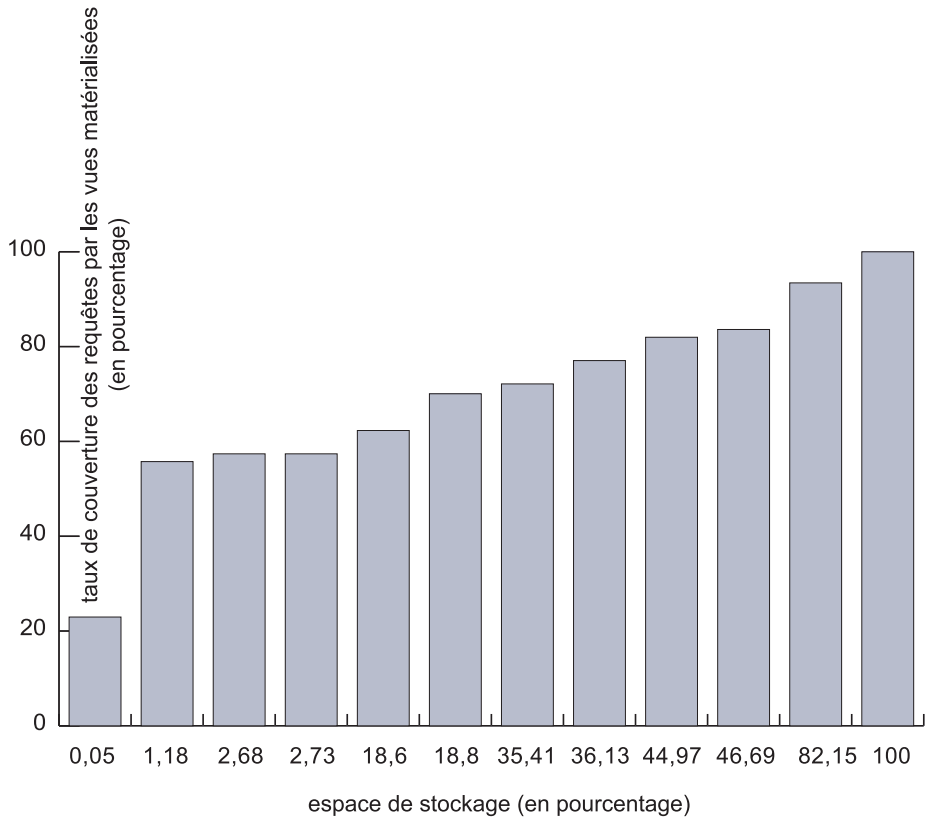

FIG. 5.7 – Couverture des requêtes par les vues matérialisées sélectionnées

toutes les vues suggérées par notre algorithme sans prendre en compte la contrainte d'espace de stockage.

5.8 Conclusion et discussion

Dans ce chapitre, nous avons présenté une stratégie de sélection automatique de vues matérialisées dans un entrepôt de données. Cette stratégie exploite les résultats de la classification non supervisée appliquée sur les requêtes d'une charge donnée. Cette classification permet de construire un ensemble de vues candidates syntaxiquement pertinentes. Pour effectuer la classification, nous avons défini des mesures de similarité et de dissimilarité permettant de construire, à partir de la charge, par une succession de segmentations/fusions, les classes de requêtes. Pour réduire le nombre de vues candidates, nous avons proposé un processus de fusion de requêtes appliqué sur les requêtes d'une même classe. Cela a pour effet de proposer des vues pertinentes pour plusieurs requêtes. À l'aide de modèles de coût, nous ne conservons que les vues candidates les plus avantageuses. Ces modèles estiment le coût d'accès aux données via les vues matérialisées sélectionnées, ainsi que l'espace de stockage de ces vues. Nous avons également proposé trois fonctions objectifs profit, ratio profit/espace et hybride qui exploitent les modèles de coût afin d'évaluer le coût d'exécution des requêtes. Ces fonctions sont à leur tour utilisées par notre algorithme glouton qui recommande la configuration finale de vues à matérialiser. Cela permet à notre stratégie de sélection de respecter la contrainte sur l'espace de stockage alloué aux vues matérialisées. Les résultats expérimentaux montrent que notre stratégie garantit un gain substantiel de performance.

Notre travail montre que l'idée d'utiliser les techniques de fouille de données en général et la classification non supervisée en particulier pour l'auto-administration des entrepôts de données est une approche pertinente. Dans la suite, nous discutons notre approche de sélection de vues matérialisées par rapport à celles présentées au Chapitre 3. Notre discussion s'articule principalement autour des aspects suivants :

- la manière de déterminer l'ensemble des vues candidates ;
- le formalisme utilisé pour mettre en évidence les relations existant entre les vues candidates ;
- l'utilisation des modèles de coût, l'appel à l'optimiseur de requêtes ;
- la sélection de vues dans un environnement relationnel ou multidimensionnel ;
- l'optimisation d'une ou plusieurs requêtes.

Les premiers travaux traitant de la sélection de vues matérialisées introduisent le treillis

des vues pour modéliser et capturer les dépendances (ancêtres et descendants) entre les vues agrégées d'un cube de données [BPT97, KR99, URT99, SDN00]. Par rapport à ces travaux, nous utilisons la mesure de similarité et de dissimilarité entre les requêtes de la charge afin de capturer les relations qui peuvent exister entre les vues candidates. En effet, les vues appartenant à une même classe sont très proches et les parents communs à ces vues sont calculés facilement à l'aide d'un treillis réduit seulement aux vues de cette classe.

Le treillis de vues est parcouru d'une manière gloutonne, en exploitant un modèle de coût, pour sélectionner les meilleures vues. Le parcours du treillis peut être très coûteux en terme de temps de calcul, surtout si le cube de données est à forte dimentionnalité. Le passage à l'échelle est donc délicat. Dans notre cas, le treillis est utilisé dans le processus de fusion des vues candidates d'une même classe. Le parcours de ce treillis est significativement mois important car le nombre de vues par classe est réduit. Nadeau *et al.* ont discuté le problème de la scalabilité [NT02], mais uniquement au niveau d'un seul cube de données. Or, l'optimisation peut être réalisée sur plusieurs cubes dans un environnement multidimensionnel, ou plusieurs tables dans le cas relationnel. D'autres études modélisent les relations entre les vues à l'aide d'un graphe de vues AND-OR [Gup97, GHRU97, CLF99, Gup99, NT02, VVK02]. Le même problème de scalabilité reste posé dans ces travaux, qui, sont théoriques et ne prennent pas en compte la difficulté de construire les graphes de vues AND-OR d'un nombre élevé de requêtes.

Les ondelettes ont été aussi utilisées pour représenter un cube de données multidimensionnel [SLJ04]. Cette approche construit une représentation optimale d'un seul cube de données sous forme d'une base complète d'éléments-vues en ondelettes. Dès lors que le nombre de cubes est élevé, la recherche d'une telle base d'éléments-vues est difficile, voire impossible, car le fondement de cette approche est inhérent à un seul cube de données. Le problème d'optimisation de plusieurs cubes de données peut être réduit en appliquant indépendamment l'optimisation sur chaque cube. Cette séparabilité réduit la complexité du problème d'optimisation, mais ne garantit pas l'optimalité de la solution obtenue car les données des différents cubes ne sont pas indépendantes. Dans notre approche, le problème de séparabilité ne se pose pas car la classification non supervisée est réalisée sur toutes les requêtes qui peuvent être définies sur des tables multiples appartenant à l'entrepôt de données. L'optimisation est donc réalisée sur la globalité de l'entrepôt.

D'autres approches détectent les sous-expressions communes aux requêtes de la charge dans un contexte relationnel [GrL01, BB03b, RS03]. Le problème de sélection de vues consiste à trouver les sous-expressions communes correspondant aux résultats intermédiaires qu'il est souhaitable de matérialiser. Cependant, le parcours des plans d'exécution est coûteux et ces approches peuvent s'avérer difficile lorsque le nombre des requêtes est élevé. De plus, ces travaux construisent un plan d'exécution optimal de requêtes multiples, dit global, à partir des plans optimaux de chaque requête de la charge. Cependant, la fusion des plans optimaux n'assure pas forcement l'optimalité du plan d'exécution global. La solution obtenue par ces approches peut donc être sous-optimale.

Certains travaux assimilent le problème de sélection de vues matérialisées au problème du sac à dos [BB03b] ou à un problème d'optimisation en utilisant les algorithmes génétiques [ZYY01]. Tout comme pour le problème de sélection d'index, le problème de ces travaux est qu'une fois les vues candidates placées dans le sac à dos ou injectées dans l'algorithme génétique, leur coût ne varie plus. Cela empêche de prendre en compte les interactions qui peuvent exister entre les vues matérialisées. Notre algorithme de sélection de vues recalcule à chaque itération le bénéfice apporté par une vue candidate en fonction des vues préalablement sélectionnées. De cette manière, nous prenons en compte les interactions qui peuvent exister entre les vues.

Les approches les plus récentes sont orientées charge, car elles analysent syntaxiquement les requêtes de la charge afin d'énumérer les vues candidates qui semblent pertinentes [ACN00, ACN01]. En faisant appel à l'optimiseur de requêtes, elles construisent d'une manière gloutonne une configuration de vues pertinentes. La charge est en effet un bon point de départ pour prédire les requêtes futures, car ces requêtes sont identiques ou syntaxiquement proches des requêtes de la charge. De plus, l'extraction des vues candidates à partir de la charge peut assurer que les vues sélectionnées ont une forte probabilité d'être exploitées par les requêtes futures adressées à l'entrepôt de données par les utilisateurs. Notre approche est orientée charge. Son originalité consiste à exploiter les connaissances sur la manière dont les vues peuvent être utilisées pour exécuter un ensemble de requêtes. En effet, plusieurs requêtes syntaxiquement proches peuvent être résolues avec une ou plusieurs vues qui sont également proches de ces requêtes. Il s'agit alors de construire des groupes de

requêtes par le moyen d'une classification non supervisée. Par rapport aux travaux de Microsoft [ACN00, ACN01] où les vues candidates sont construites par énumération itérative, notre approche construit l'ensemble des vues candidates à partir des classes de requêtes. Cela permet de construire ces vues candidates à la volée. De plus, nous appliquons la fusion sur les vues candidates présentes dans chaque classe au lieu de l'ensemble des vues candidates obtenu par analyse comme dans [ACN00, ACN01]. Cela réduit considérablement la complexité du processus de fusion car le nombre de vues candidates dans chaque classe est significativement moins élevé.