Algorithmes pour la sélection des vues à matérialiser avec garantie de performance[1]

Nicolas Hanusse*, Sofian Maabout*
Radu Tofan*

*LaBRI. Université Bordeaux 1
nom@labri.fr

Résumé. Nous considérons dans cet article le problème du choix des vues à matérialiser dans le contexte des cubes de données. Contrairement à la plupart des autres travaux, la contrainte que l'on prend en compte ici est relative à la performance des requêtes non pas l'espace mémoire disponible. Nous montrons l'avantage de notre approche avec des arguments théoriques et nous le confirmons par des expériences menées sur des jeux de données différents.

1 Introduction

Dans les applications OLAP, l'optimisation des requêtes analytiques est primordial. Ainsi, on a souvent recours à un pré-calcul ou d'une manière équivalente à la matérialisation des requêtes. Ces dernières peuvent être trop nombreuses pour être toutes pré-calculclées. Ceci pour deux raisons : soit à cause du temps qu'il faut pour toutes les calculer ou bien de l'espace mémoire nécessaire pour toutes les stocker. Ainsi, on a besoin d'algorithmes permettant de choisir les *"meilleures"* vues à stocker. Les performances des algorithmes de sélection sont évalués selon trois critères : (1) l'espace requis pour stocker les vues sélectionnées, (2) le temps d'évaluation des requêtes et (3) le temps nécessaire pour exécuter l'algorithme, i.e sa complexité. La plupart des solutions proposées jusqu'à présent consistent à choisir un sous-ensemble de vues permettant d'optimiser l'évaluation des requêtes sans dépasser une limite d'espace mémoire fixée par l'utilisateur. Certaines variantes de ce problèmes ont été étudiées : (1) la nature des données qu'on peut stocker, e.g seulement des vues ou des index aussi, (2) le modèle de coût e.g minimiser non seulement le temps d'évaluation des requêtes mais aussi la coût de mise à jour ou finalement (3) l'ensemble des requêtes susceptibles d'être posées e.g toutes ou bien un sous-ensemble. Dans ce dernier cas, le problème peut être raffiné en considérant la fréquence ou *charge* des requêtes qu'on veut optimiser. En général, le problème peut être décrit par un problème d'optimisation sous contrainte : la contrainte étant l'espace disponible qu'il ne faut pas dépasser et la fonction à optimiser est le coût moyen des requêtes qu'il faut minimiser. A notre connaissance, aucune des solutions proposées dans la littérature n'offre un bon compromis entre complexité de l'algorithme de sélection, minimalité du coût moyen de la solution et garantie de performance.

L'article est organisé de la manière suivante : nous présentons d'abord quelques notations et défintions nous permettant de formaliser le problème à traiter. Nous décrivons par la suite quelques travaux relatifs. Dans la section 3, nous présentons notre solution pour cas où tous

les cuboïdes d'un cube de données peuvent être interrogés ensuite nous traitons le cas où l'on dispose d'un ensemble de requêtes fixé (Section 4). Dans la section 5, nous présentons quelques unes des expériences que nous avons conduites permettant de valider notre approche et nous terminons par une conclusion récapitulant notre apport en mentionnant quelques perspectives de travaux futurs.

2 Préliminaires

Notation Une table de faits T est une relation où l'ensemble des attributs est divisé en 2 parties : l'ensemble des dimensions $Dim(T)$ et l'ensemble des mesures $Mesures(T)$. En général, $Dim(T)$ forme une clé pour la table T. Le cube de données Jim Gray (1997) construit à partir de T est obtenu en agrégeant T et en regroupant ses tuples dans toutes les manières possibles i.e tous les *Group By c* où c est un sous-ensemble de $Dim(T)$. Chaque c correspond à un *cuboïde*[2]. Le cube de données associé à T est noté $DC(T)$. Par abus de notation $Dim(DC(T))$ désignera $Dim(T)$. Soit $DC(T)$ un cube de données, $Dim(T)$ ses dimensions et $|Dim(T)| = D$. L'ensemble des cuboïdes de $DC(T)$ est noté $\mathcal{C}(T)$. Il est évident que $|\mathcal{C}(T)| = 2^D$. La table T est un cuboïde particulier de $\mathcal{C}(T)$. Il est appelé *cuboïde de base* et est noté c_b. La taille d'un cuboïde c correspond au nombre de ses lignes et est notée $size(c)$. La taille d'un ensemble de cuboïdes $\mathcal{S}$ est notée $size(\mathcal{S})$. Le treillis du cube de données est induit par une relation d'ordre partiel $\preceq$ entre cuboïdes qui est définie comme suit : $v \preceq w$ ssi $Dim(v) \subseteq Dim(w)$. On dit que w est un *ancêtre* de v. Aussi, si $|Dim(w)| = |Dim(v)| + 1$ alors w est un *parent* de v. En fait, si $v \preceq w$ alors v peut être calculé à partir de w. [3]

Modèle de coût Soient $\mathcal{S} \subseteq \mathcal{C}$ l'ensemble des cuboïdes matérialisés et v un cuboïde. Alors, $\mathcal{S}_v = \{w \in \mathcal{S} | v \preceq w\}$ est l'ensemble de cuboïdes matérialisés à partir desquels v peut être calculé. Le coût de v respectivement à $\mathcal{S}$ est défini comme suit : si $\mathcal{S}$ ne contient aucun ancêtre de v alors $cost(v, \mathcal{S}) = \infty$ sinon $cost(v, \mathcal{S}) = arg\min_{w \in \mathcal{S}_v} size(w)$, i.e v est évalué à partir son ancêtre qui a la plus petit nombre de tuples. C'est la mesure généralement utilisée pour évaluer le coût des requêtes (voir par exemple Harinarayan et al. (1996); Shukla et al. (1998); Talebi et al. (2008)). Noter que quand $v \in \mathcal{S}_v$ alors $cost(v, \mathcal{S}) = size(v)$. Ceci est le coût minimal pour v. Nous définissons le coût induit par $\mathcal{S}$ comme étant le coût de toutes les requêtes possibles, i.e $cost(\mathcal{S}) = \sum_{c \in \mathcal{C}} cost(c, \mathcal{S})$. Quand $\mathcal{S} = \mathcal{C}$ i.e tous les cuboïdes sont stockés, $cost(\mathcal{S}) = \sum_{c \in \mathcal{C}} size(c)$. Ceci est le coût minimal et est noté $MinCost$. Si $\mathcal{S} = \{c_b\}$ alors $cost(\mathcal{S}) = |\mathcal{C}| * M$ où $M = size(c_b)$. Ceci représente le coût maximal et est noté $MaxCost$. Ainsi, pour tout $\mathcal{S}$, on a $\sum_{c \in \mathcal{C}} size(c) \leq cost(\mathcal{S}) \leq |\mathcal{C}| * M$[4].

Nous définissons la performance d'un ensemble $\mathcal{S}$ comme suit :

Définition 1 (Facteur de performance)**.** *Soit $\mathcal{S}$ l'ensemble des cuboïdes matérialisés et c un cuboïde de $\mathcal{C}$.* Le facteur de performance *de $\mathcal{S}$ resp. à c est défini par* $f(c, \mathcal{S}) = \frac{cost(c,\mathcal{S})}{size(c)}$. le facteur de performance **moyen** *de $\mathcal{S}$ resp. à $\mathcal{C}' \subseteq \mathcal{C}$ est défini par* $\tilde{f}(\mathcal{C}', S) = \frac{\sum_{c \in \mathcal{C}'} f(c,\mathcal{S})}{|\mathcal{C}'|}$

[2]Dans la suite de l'article, nous utiliserons d'une manière équivalente les termes *cuboïde*, *vue* et requête.

[3]Dans cet article, on ne considère que les mesures algébriques.

[4]On ne considère que les $\mathcal{S}$ qui contiennent c_b car sinon $cost(\mathcal{S}) = \infty$.

Intuitivement, le facteur de performance mesure le ratio entre le temps d'exécution d'une requête à partir de $\mathcal{S}$ par rapport au coût minimal de cette requête. L'exemple ci-dessous permet d'illustrer les notions qu'on a introduites dans cette section.

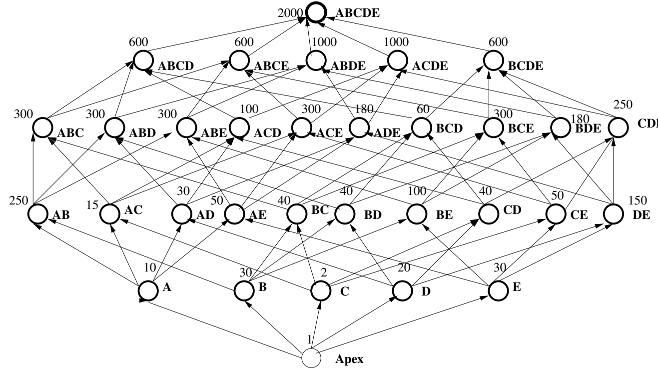


FIG. 1 – *Un exemple de cube de données*

Exemple 1. *Considérons le graphe de la Figure 1. Il représente le treillis du cube de données associé à une table T dont les dimensions sont A, B, C, D et E. Nous allons utiliser cet exemple tout au long de l'article. Chaque nœud du graphe représente un cuboïde décrit par ses dimensions ainsi que sa taille. Le coût minimal pour évaluer toutes les requêtes correspond au cas où tous les cuboïdes sont stockés. Ainsi, $MinCost = \sum_{i=1}^{2^5} size(c_i) = 8928$. Le coût maximal est obtenu en considérant que seul le cuboïde de base est stocké ainsi $MaxCost = 2^5 * size(ABCDE) = 2000 * 32 = 64000$. Noter néanmoins que ceci correspond à l'espace mémoire minimum que l'on doit utiliser pour pouvoir répondre à toutes les requêtes.*

Supposons maintenant que $\mathcal{S} = \{ABCDE, BE\}$. Les mesures de performance de $\mathcal{S}$ sont comme suit : L'espace mémoire utilisé est $Mem(\mathcal{S}) = size(ABCDE) + size(BE) = 2000 + 100 = 2100$. Le coût pour évaluer les 2^5 requêtes possibles est obtenu de la manière suivante. D'abord, prenons le cuboïde stocké BE. Ce dernier sert à calculer BE, B, E et $Apex$[5]. Tous les autres cuboïdes sont calculés à partir de $ABCDE$. Ainsi, $Cost(\mathcal{S}) = 4 * size(BE) + 28 * size(ABCDE) = 56400$. Considérons maintenant les cuboïdes BE et BC. leurs facteurs de performance respectifs sont $Tf(BE, \mathcal{S}) = \frac{cost(BE,\mathcal{S})}{size(BE)} = 100/100 = 1$ et $f(BC, \mathcal{S}) = \frac{cost(BC,\mathcal{S})}{size(BC)} = size(ABCDE)/40 = 2000/40 = 50$. Cela signifie qu'en matérialisant $ABCDE$ et BE, le coût de BE en utilisant $\mathcal{S}$ est égal à son coût minimal, par contre pour BC, le coût est 50 fois son coût minimal.

Formalisation du problème Dans cet article nous traitons le problème suivant :
Etant donnés

- un nombre réel $f \geq 1$ et
- un ensemble de requêtes $\mathcal{Q} \subseteq \mathcal{C}$

trouver un ensemble de cuboïdes $\mathcal{S}$ tel que

[5] $Apex$ est le cuboïde sans dimensions.

– pour tout $c \in \mathcal{Q}$, on a $cost(c, \mathcal{S}) \leq f * size(c)$ et
– $\mathcal{S}$ est de taille minimale

Nous étudions dans cet article deux cas : celui où $\mathcal{Q} = \mathcal{C}$ (appelée plus loin **absence de charge**) et celui où $\mathcal{Q} \subset \mathcal{C}$ (**présence de charge**). Le modèle absence de charge correspond au cas où on ne dispose pas de connaissance préalable sur les requêtes fréquentes.

Travaux relatifs Avant de présenter nos solutions, nous rappelons d'abord quelques travaux relatifs. Aucun des travaux que nous avons trouvés dans la littérature ne pose le problème de sélection des vues dans les termes que nous avons utilisés. La plupart considère une limite $\mathcal{M}$ sur l'espace mémoire disponible et essaye de trouver le sous-ensemble dont la taille ne dépasse pas $\mathcal{M}$ et qui réduit au maximum *la somme* des coûts des requêtes. Dans Harinarayan et al. (1996), les auteurs montrent que ce problème est NP-complet et proposent donc un algorithme glouton pour le résoudre. Ils utilisent pour cela la notion de gain qu'on rappelle ici.

Définition 2 (Gain)**.** *Soit $\mathcal{S} \subset \mathcal{C}$. Le gain de $\mathcal{S}$ est défini par $cost(\{c_b\}) - cost(\mathcal{S})$.*

Noter que $cost(\{c_b\})$ est $MaxCost$. L'algorithme qu'ils proposent tend à trouver un ensemble $\mathcal{S}$ qui maximise le gain. Les auteurs montrent que le gain apporté par leur solution est supérieur à 63% du gain apporté par la solution optimale. Ceci représente leur garantie de performance. Cependant, il faut faire ici quelques remarques : (i) L'algorithme glouton qu'ils proposent a une complexité en $O(n)$ avec n qui est le nombre de cuboïdes au total. On sait bien que ce nombre est 2^D. On est donc en présence d'un algorithme exponentiel en nombre de dimensions. Aussi, (2) l'algorithme suppose la connaissance de la taille de tous les cuboïdes. Ceci est largement non réaliste dans le cas où le nombre de dimensions est grand. Enfin, (3) la garantie du gain qu'ils prouvent pour leur algorithme n'implique pas une garantie de performance. Autrement dit, le rapport entre le coût de leur solution sur le coût de la solution optimale n'est pas borné par une constante. Cette même remarque a été soulevée dans Karloff et Mihail (1999). Ces derniers ont démontré que dans les graphes généraux, l'algorithme glouton ne garantissait pas une performance de requêtes. Ils ont néanmoins laissé ouverte la question quant aux cubes de données. Nous en avons fait la démonstration dans le cadre particulier des cubes de données mais faute d'espace nous ne la présentons pas ici. Pour pallier au problème de complexité de Harinarayan et al. (1996), les auteurs de Shukla et al. (1998) en ont proposé une simplification qui consiste tout simplement à prendre les cuboïdes selon l'ordre croissant de leurs tailles jusqu'à épuiser l'espace mémoire disponible. Ils montrent que souvent les solutions qu'ils trouvent étaient comparables à celle de Harinarayan et al. (1996). Bien sûr, cette méthode souffre des même travers que ceux de Harinarayan et al. (1996) quant aux performances. D'autres auteurs ont eu une vision plus générale du problème. Par exemple, Gupta et al. (1997); Talebi et al. (2008); Maiz et al. (2006) considèrent la possibilité de stocker en plus certains index. Gupta et al. (1997) a utilisé une extension de Harinarayan et al. (1996) alors que Talebi et al. (2008) a montré que le problème pouvait se modéliser par un programme linéaire qu'on pouvait résoudre exactement, i.e trouver la solution optimale, quand le nombre de variables n'était pas trop élevé sinon, ils ont proposé une relaxation pour réduire la taille du problème mais dans ce cas, ils n'ont montré l'efficacité de cette approche que d'une manière expérimentale.

3 Sélection de cuboïdes en l'absence de charge : recherche de bordures

Dans cette section, nous considérons le cas où l'ensemble des requêtes susceptibles d'être posées par l'utilisateur correspond à tous les cuboïdes du cube de données. Cette situation peut avoir lieu lors de la création du cube de données, puisqu'à ce moment on ne sait pas encore quelles sont les requêtes qui seront le plus souvent posées. Nous aborderons par la suite le cas où l'on dispose de cette information. Nous donnons d'abord quelques définitions.

Définition 3 (Petit cuboïde et Bordure)**.** *Soit* $f > 1$ *un nombre réel et* $M = size(c_b)$*.*
- *Un cuboïde* c *est dit* petit *si* $size(c) \leq M/f$*.*
- *Soit* c *un petit cuboïde. Alors* c *est* maximal *ssi aucun de ses parents n'est petit.*
- *L'ensemble des cuboïdes maximaux représente une* **bordure** *du cube de données* $\mathcal{C}$ *relativement à* f*. On la note* $\mathcal{B}_f(\mathcal{C})$ *ou simplement* $\mathcal{B}_f$ *quand* $\mathcal{C}$ *est compris du contexte.*

Exemple 2. *Reprenons le cube de données de la figure 1 et supposons* $f = 10$*. Comme* $size(ABCDE) = 2000$ *alors les petits cuboïdes sont ceux dont la taille ne dépasse pas* $\frac{2000}{10}$*. On peut facilement vérifier que* $\mathcal{B}_f = \{ACD, ADE, BCD, BDE\}$*.*

Tous les cuboïdes qui sont au dessus de la bordure peuvent être calculés à partir du cuboïde de base en ayant une garantie de performance sur leurs coûts :

Lemme 1. *Soient* $f > 1$ *et* $\mathcal{B}_f$*. Pour tout* c *ancêtre d'un élément de* $\mathcal{B}_f$*,* $cost(c, \{c_b\}) \leq f * size(c)$*.*

Comment obtenir la même garantie pour les autres cuboïdes ? Il suffit de calculer les autres bordures $\mathcal{B}_{f^2}, \mathcal{B}_{f^3} \ldots$

Définition 4 (Bordures)**.** *Soient* $f > 1$ *et* $M = size(c_b)$*. Soit* $\mathbb{B}_f = \bigcup_{i=0}^{\lfloor \log_f(M) \rfloor} \mathcal{B}_{f^i}$*. L'ensemble* $\mathbb{B}_f$ *désigne* **les bordures** *de* $\mathcal{C}$*.*

Exemple 3. *Reprenons notre exemple courant. La figure 2 montre l'ensemble des bordures de* $\mathcal{C}$*. Ses éléments sont représentés par des cercles pleins.*

Les bordures d'un cube de données permettent de garantir la performance des requêtes :

Théorème 1 (Hanusse et al. (2009))**.** *Soit* $f > 1$ *et* $\mathbb{B}_f$*. Alors pour tout* $c \in \mathcal{C}$*,* $cost(c, \mathbb{B}_f) \leq size(c) * f$*.*

Ceci signifie que si l'utilisateur fixe une valeur pour f, nous sommes capables de lui retourner un ensemble de cuboïdes qui garantit que le coût de chaque requête ne dépasse pas f fois le coût minimale de cette dernière. Du théorème précédent, on déduit le corolaire suivant :

Corollaire 1. *Soient* $f > 1$ *et* $\mathbb{B}_f$*. Soit* $\mathcal{M} = size(\mathbb{B}_f)$*. Soit* $\mathcal{C}_\mathcal{M} = \{\mathcal{S} \subseteq \mathcal{C} | \, size(\mathcal{S}) \leq \mathcal{M}\}$*. Soit* $\mathcal{S}^* = arg\min_{\mathcal{S} \in \mathcal{C}_\mathcal{M}} cost(\mathcal{S})$*. Alors,* $cost(\mathbb{B}_f) \leq f * cost(\mathcal{S}^*)$*.*

Autrement dit, si avec un espace mémoire disponible égal à $\mathcal{M}$, la meilleure solution, i.e celle qui réduit au maximum le coût des requêtes, est $\mathcal{S}^*$ alors le coût de $\mathbb{B}_f$ est au maximum f fois le coût de $\mathcal{S}^*$.

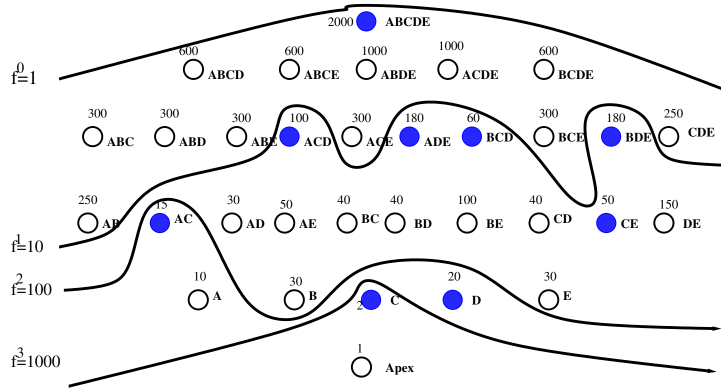


FIG. 2 – *Les bordures du cube de données pour* $f = 10$*.*

Exemple 4. *Pour illustrer notre proposition, reprenons la Figure 2. Les courbes représentent les bordures relatives aux différentes puissances de* f *quand* $f = 10$*. Les nœuds représentés par des cercles remplis sont donc les seuls à être matérialisés. La taille totale du cube de données est de 8928. Ceci représente le coût minimal pour évaluer toutes les requêtes. Le coût maximal est de 32*2000=64000. Par ailleurs,* $size(\mathbb{B}_f) = 2607$ *et* $cost(\mathbb{B}_f) = 27554$*. Noter néanmoins que même si on a fixé le facteur de performance à* $f = 10$*, le ratio entre le coût de* $\mathbb{B}_f$ *et le coût minimal est* $\frac{27554}{8927} = 3.09$*. Ainsi, le coût moyen pour répondre à n'importe quelle requête en utilisant* $\mathbb{B}_f$ *est 3 fois le coût minimal. Avec HRU, l'algorithme de Harinarayan et al. (1996) et avec la contrainte sur l'espace qui ne doit pas dépasser* $size(\mathbb{B}_f)$*, les seuls cuboïdes qui seront retenus sont* $ABCDE$ *et* $BCDE$*. On obtient ainsi un coût de 41600 ce qui représente un ratio, toujours par rapport au coût minimal, de* $\frac{41600}{8927} = 4.67$*. Notons aussi que PBS, l'algorithme de Shukla et al. (1998), retourne 16 cuboïdes (les 16 premiers cuboïdes selon l'ordre croissant de leurs tailles respectives). Le coût de cette solution est de 34518. Ce qui correspond à un ratio de* $\frac{34518}{8927} = 3.87$ *meilleur que HRU.*

Maintenant que nous avons caractérisé les vues que l'on veut matérialiser avec la contrainte de performance, nous allons présenter un algorithme qui permet de les retrouver d'une manière relativement efficace. La notion de bordure que nous avons introduite est analogue à celle qui est utilisée dans le contexte de l'extraction des ensembles fréquents maximaux dans le domaine du datamining Gouda et Zaki (2005); Bayardo (1998); Burdick et al. (2001). En effet, la notion de *"petit cuboïde"* est une propriété antimonotone puisque si c n'est pas un *petit* cuboïde alors aucun de ses ancêtres ne peut l'être et s'il est *petit* alors tous ses descendants le sont. Ainsi, on pourrait adapter les algorithmes de calcul des fréquents maximaux à notre contexte. Cependant, ces algorithmes optimisent le temps de calcul en comptant la fréquence de plusieurs itemsets lors d'une même itération. Ceci n'est pas possible dans le cadre des cubes de données, du moins ce n'est pas efficace de procéder de la sorte, car cela nécessiterait de garder en mémoire plusieurs tables simultanément chacune correspondant à un cuboïde pour lequel on veut calculer la taille. Finalement, nous avons utilisé un algorithme qui calcule la taille d'un seul cuboïde à chaque itération. Pour faciliter la présentation de cet algorithme, nous décrivons d'abord un codage des cuboïdes. On trie les dimensions en considérant par exemple

l'ordre lexicographique. Cela nous permet de définir le *rang* d'une dimension qui correspond à sa position dans l'ordre lexicographique. Par exemple, si $Dim = \{A, B, C, D, E\}$ alors $rang(A) = 1$ et $rang(C) = 3$. Chaque cuboïde est codé par une chaîne de bits de taille $|Dim|$. La position i de la chaîne est égale à 1 ssi la dimension de rang i est présente dans le cuboïde. Par exemple, le cuboïde ADE est codé par 10011. L'algorithme a besoin des paramètres suivants : i représente une position sur la chaîne qui code les cuboïdes, une chaîne s qui code les cuboïdes à traiter, un entier n qui représente le nombre de dimensions du cube de données, un entier $\mathcal{M}$ qui désigne la taille maximale des *petits* cuboïdes et enfin $\mathcal{B}$ qui est la liste des cuboïdes formant la bordure.

Procédure Générer_$\mathcal{B}$(entier i, chaîne s, entier n, entier $\mathcal{M}$, liste $\mathcal{B}$)
    Si $i \leq n$ alors
        $s[i] \leftarrow 1$
        Si s a un ancêtre dans $\mathcal{B}$ Alors
            Générer_$\mathcal{B}(i+1, s, n, \mathcal{M}, \mathcal{B})$
        Sinon
            Si **size**$(s) \leq \mathcal{M}$ Alors
                Ajouter s à $\mathcal{B}$
                Supprimer de $\mathcal{B}$ tous les descendants de s
                Générer_$\mathcal{B}(i+1, s, n, \mathcal{M}, \mathcal{B})$
            Finsi
        FinSi
        $s[i] \leftarrow 0$
        Générer_$\mathcal{B}(i+1, s, n, \mathcal{M}, \mathcal{B})$
    FinSi
Fin Procédure

Générer_$\mathcal{B}(1, 0^n, n, M/f^j, \emptyset)$ est appelée par la procédure **Générer_**$\mathbb{B}(f, M)$ pour j allant de $\lfloor \log_f(M) \rfloor$ à 0 et en faisant l'union des bordures de paramètres M/f^j. f désigne le facteur de performance et M la taille du cuboïde de base.

La partie la plus coûteuse pour le calcul de $\mathbb{B}$ est l'exécution de la fonction **size** qui nécessite le parcours de la table de faits.

4 Sélection de vues en présence de charge

Dans cette section, nous considérons un ensemble $\mathcal{Q}$ qui représente l'ensemble des requêtes que l'on veut optimiser. Rappelons que les requêtes que l'on considère dans cet article sont celle de la forme `select * from c` où `c` est un des cuboïdes de $\mathcal{C}$[6]. Il s'agit donc de trouver un ensemble $\mathcal{S} \subseteq \mathcal{C}$ tel que :

- Pour chaque $q \in \mathcal{Q}$ il existe $c \in \mathcal{S}$ t.q $q \preceq c$ i.e q peut être calculée
- Pour chaque $q \in \mathcal{Q}$ il existe $c \in \mathcal{S}$ t.q $q \preceq c$ et $size(c) \leq f * size(q)$ i.e q peut être calculée efficacement
- $\mathcal{S}$ doit avoir une taille minimale

[6]On pourrait inclure les requêtes du type `select * from c where att=val` sans changer fondamentalement les résultats car on n'a pas considéré la possibilité d'avoir des index.

Notre contribution pour résoudre ce problème est double : (1) nous donnons un algorithme de complexité polynomiale retournant une $\log n$-approximation de l'optimal et (2) un programme linéaire en nombre entiers qui retourne la solution optimale mais en un temps non borné.

Nous introduisons d'abord quelques notations. On note par $\mathcal{A}_f(q)$ l'ensemble des cuboïdes qui sont (1) ancêtres d'une requête $q \in \mathcal{Q}$ et (2) dont la taille est inférieure ou égale à $f * size(q)$. Soit $\mathcal{A}_f(\mathcal{Q}) = \bigcup_{q \in \mathcal{Q}} \mathcal{A}_f(q)$. Noter que $\mathcal{Q} \subseteq \mathcal{A}_f(\mathcal{Q})$. Il est facile de voir que la solution à notre problème est un sous-ensemble de $\mathcal{A}_f(\mathcal{Q})$. En effet,

Lemme 2. *Soit $\mathcal{S}^* \subseteq \mathcal{C}$ le plus petit ensemble de cuboïdes (en terme de taille mémoire) t.q $cost(q, \mathcal{S}^*) \leq f * size(q)$ pour chaque $q \in \mathcal{Q}$. Alors $\mathcal{S}^* \subseteq \mathcal{A}_f(\mathcal{Q})$.*

Considérons maintenant le graphe $G(V, E)$ où $V = \mathcal{A}_f(\mathcal{Q})$ et $(v_1, v_2) \in E$ ssi (i) $v_2 \in \mathcal{Q}$ et (ii) $v_1 \in \mathcal{A}_f(v_2)$. Partant de ce graphe, la sélection du plus petit $\mathcal{S}$ revient à chercher l'ensemble de sommets de V de poids minimum (avec le poids qui correspond à la taille du cuboïde représenté par le sommet) qui couvre tous les éléments de $\mathcal{Q}$. Ce problème est une instance du problème bien connu du *minimal weighted set cover* (MWSC) qui est NP-complet Karp (1972). Cependant, il existe un algorithme glouton de complexité polynomiale qui permet de le résoudre avec une "bonne" approximation de la solution optimale Chvàtal (1979). Par ailleurs, il est bien connu que le problème MWSC peut être modélisé par un programme d'optimisation linéaire qui peut être résolu assez rapidement par les solveurs existants lorsque le nombre de variables n'est pas trop élevé.

Solution approchée Nous présentons d'abord l'algorithme approché et par la suite nous donnons le programme d'optimisation linéaire modélisant la solution exacte.

Procédure PickFrom$f$Ancestors()
$\mathcal{S} = \emptyset$
Tant que $\mathcal{Q} \neq \emptyset$
  $c^* = \arg\min_{c \in \mathcal{A}_f(\mathcal{Q})} \frac{size(c)}{|\Gamma(c)|}$
  `//` $\Gamma(c)$ `sont les voisins de` c
  $\mathcal{S} = \mathcal{S} \cup \{c^*\}$
  $\mathcal{A}_f(\mathcal{Q}) = \mathcal{A}_f(\mathcal{Q}) \backslash (\{c^*\} \cup \Gamma(c^*))$
  $\mathcal{Q} = \mathcal{Q} \backslash \Gamma(c^*)$
Fin Tant Que
Retourner $\mathcal{S}$
Fin Procédure

L'algorithme consiste à choisir lors de chaque itération, l'ancêtre qui a la "charge" minimale. Cette dernière est définie comme étant la taille de l'ancêtre divisée par le nombre de ses voisins (i.e le nombre de requêtes de $\mathcal{Q}$ qu'il couvre). Dès qu'un ancêtre est choisi, on le met dans la solution $\mathcal{S}$ et on le supprime du graphe ainsi que les requêtes qu'il couvre. Ceci a pour incidence de modifier le nombre de voisins des sommets restants. En utilisant le résultat de Chvàtal (1979) nous montrons le facteur d'approximation obtenu par cet algorithme.

Théorème 2. *Soit $f > 1$ et $\mathcal{S}$ la solution retournée par PickFrom$f$Ancestors. Alors*
- *Pour chaque $q \in \mathcal{Q}$, $cost(q, \mathcal{S}) \leq f * size(q)$*

– *Soit $\mathcal{S}^*$ la solution optimale (i.e la plus petite en taille mémoire), alors $size(\mathcal{S}) \leq ln(n) * size(\mathcal{S}^*)$ où $n = arg \max_{c \in \mathcal{A}_f(\mathcal{Q})} |\Gamma(c)|$.*

Solution exacte Nous donnons maintenant le programme d'optimisation linéaire. Avant cela, nous décrivons les variables utilisées :
– s_i désigne la taille du cuboïde $c_i \in \mathcal{A}_f(\mathcal{Q})$,
– $x_i \in \{0, 1\}$ et signifie que c_i est choisi pour être matérialisé (1) ou non (0),
– $y_{ij} \in \{0, 1\}$ telle que $y_{ij} = 1$ ssi $q_i \in \mathcal{Q}$ utilise c_j pour être calculée

$$\min \sum_{j:c_j \in \mathcal{A}_f(\mathcal{Q})} x_j * s_j \tag{1}$$
$$\forall i : q_i \in \mathcal{Q} \sum_{j:c_j \in \mathcal{A}_f(q_i)} y_{ij} = 1 \tag{2}$$
$$\forall i : q_i \in \mathcal{Q}, \forall j : c_j \in \mathcal{A}_f(q_i) \ y_{ij} \leq x_j \tag{3}$$
$$\forall j : c_j \in \mathcal{A}_f(\mathcal{Q}) \ x_j \in \{0, 1\} \tag{4}$$
$$\forall i : c_i \in \mathcal{Q}, \ \forall j : c_j \in \mathcal{A}_f(q_i) \ y_{ij} \in \{0, 1\} \tag{5}$$

La contrainte (2) impose que q_i n'utilise qu'un seul cuboïde et la contrainte (3) implique que q_i ne peut utiliser c_j que si c_j est stocké. Enfin, les contraintes (4) et (5) imposent que les x_i et les y_{ij} sont des variables binaires. Résoudre ce problème revient à affecter des valeurs aux variable x_j et y_{ij} qui permettent de minimiser la valeur de l'expression (1).

La solution "exacte" à notre problème correspond donc à $\mathcal{S} = \{c_j \in \mathcal{A}_f(\mathcal{Q}) : x_j = 1\}$. On peut se demander pourquoi avoir recours à un algorithme approché lorsqu'on peut résoudre exactement un problème. En fait, les solveurs tels que `Cplex`, ou `LP_Solve` [7], sont très puissants jusqu'à un certain nombre de variables. Or, le nombre de x_i et de y_{ij} peut croître assez rapidement avec l'accroissement de $|\mathcal{Q}|$ ou avec l'accroissement de f. Dans les expérimentations que nous avons menées, nous avons utilisé LP_Solve qui est un outil gratuit qui, bien que efficace, ne nous a néanmoins pas permis de traiter des cas où $|\mathcal{Q}|$ était "grand".

5 Expérimentations

Dans cette section nous montrons quelques unes des expérimentations que nous avons entreprises pour valider notre approche. La fonction **size** utilisée calcul la taille *exacte* d'un cuboïde. Nous aurions pu nous contenter d'une estimation de ces tailles en utilisant des techniques présentées par exemple dans Aouiche et al. (2006); Aouiche et Lemire (2007). Nous avons préféré utiliser les tailles exactes car nous désirions surtout montrer expérimentalement la qualité de la solution que nous obtenions. Quand le facteur temps de calcul est primordial, il est bien sûr utile d'avoir recours à ces techniques.

5.1 Le cas sans charge

Nous avons utilisé les données suivantes :

[7]Disponible à l'URL `http ://lpsolve.sourceforge.net/5.5/`

- USData10 : contient 2.5 millions de tuples avec 10 attributs correspondant aux 10 premiers attributs de US Census 1990 data disponible à `http ://kdd.ics.uci.edu/`. Nous avons exclu le premier attribut qui est une clé.
- USData13 : Les mêmes données qu'auparavant mais avec les 13 premières dimensions.
- Objects : contient “seulement" 8000 tuples avec 12 attributs ayant trait à des données issues de fouilles archéologiques[8]. Ce cas est intéressant car le nombre de dimensions est relativement grand par rapport à la taille de la base de faits.
- Zipf10 : est un ensemble de données synthétiques. Il contient 10^6 enregistrements et 10 dimensions. Plusieurs observations ont montré qu'en en général, les valeurs des attributs ne suivaient pas une distribution uniforme mais plutôt une loi en puissance. Intuitivement, si on classe les valeurs prises par un attribut selon un ordre particulier, alors la probabilité que la *i_ème* valeur soit prise est proportionnelle à $\frac{1}{(i)^\alpha}$, $\alpha > 0$. Généralement, α se trouve dans l'intervalle $[2, 3]$. Dans nos expérimentations, nous avons fixé $\alpha = 2$.

La table 1 résume quelques caractéristiques de ces données. Rappelons que $MinCos$ correspond au cas où tout le cube est stocké et $MaxCost$ correspond au cas où seulement le cuboïde de base est stocké.

Dataset	MinCost	MaxCost
USdata10	$4.37 * 10^6$	$5.35 * 10^7$
USdata13	$1.05 * 10^8$	$1.19 * 10^9$
Objects	$1.72 * 10^7$	$3.05 * 10^7$
ZIPF10	$4 * 10^7$	$3.93 * 10^8$

TAB. 1 – *MinCost et MaxCost des Datasets*

Le jeu de données `objects` se distingue par rapport aux 3 autres au niveau de la différence entre les coûts minimaux et maximaux. En effet, pour `objects` nous avons un rapport qui est moins de 2 alors que pour les autres le rapport avoisine 10. Ainsi, `objects` serait qualifié de *non dense*.

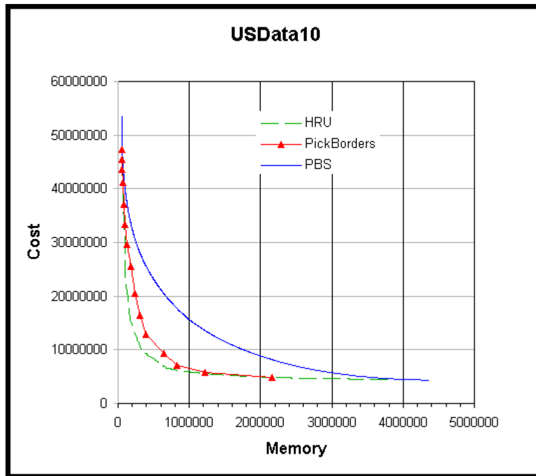
Algorithmes pour la comparaison

A notre connaissance, il n'existe pas d'autres algorithmes permettant de résoudre le problème de sélection de vues tel que nous l'avons formulé. Cependant, pour se comparer avec d'autres travaux, nous avons adapté ces derniers à notre contexte. Nous avons choisi 2 algorithmes bien connus de la littérature : HRU de Harinarayan et al. (1996) et PBS de Shukla et al. (1998). Ces deux algorithmes choisissent le meilleur ensemble de vues à stocker avec comme contrainte, un espace mémoire à ne pas dépasser. Afin d'effectuer une comparaison, nous avons fixé différentes valeurs de f et pour chacune nous avons considéré la taille de notre solution $\mathcal{S}$ comme étant la contrainte pour ces 2 algorithmes.

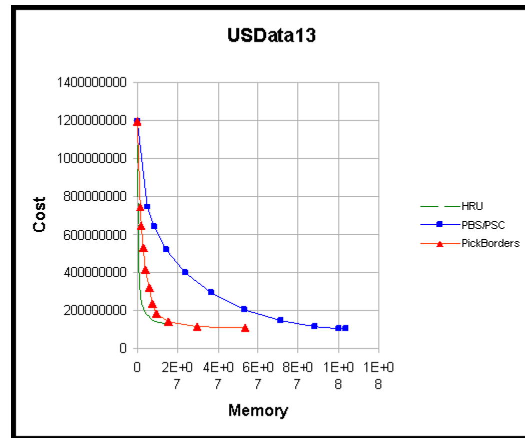
[8]Nous remercions Noël Novelli pour les avoir mises à notre disposition.

Analyse Mémoire/Coût

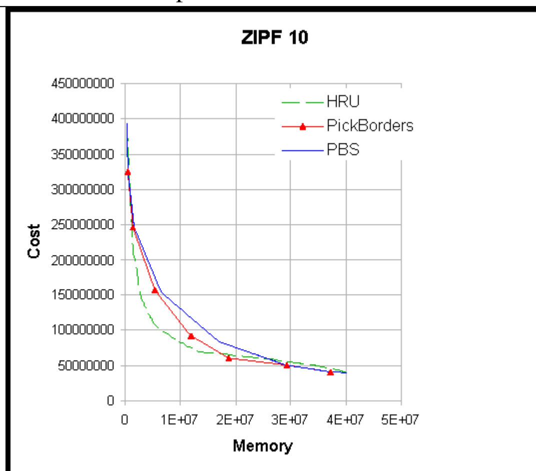
Les figures ci-dessous montrent l'évolution du coût de l'évaluation de toutes les requêtes en fonction de la mémoire utilisée. Dans ces graphiques, la courbe PickBorders représente notre solution.



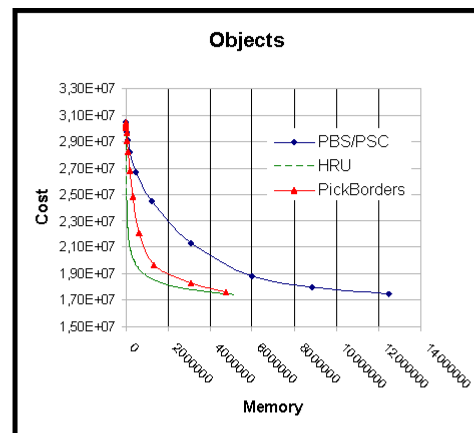
Coûtt/Mémoire pour USData10.



Coût/Mémoire pour USData13.



Coût/Mémoire pour ZIPF10.



Coût/Mémoire pour Objects.

D'après ces graphiques, on voit bien que HRU l'emporte dans tous les cas quand on considère le rapport entre espace utilisé et coût des requêtes. Noter néanmoins que pour exécuter HRU, nous avons besoin de connaître la taille de tous les cuboïdes, au moins une estimation, alors que notre approche n'en a pas besoin. D'ailleurs, nous n'avons pas réalisé des tests sur un nombre élevé de dimensions (USData1990 en compte 68) car ni HRU ni PBS n'aurait pu être exécuté. Aussi, il faut préciser que ce qui est représenté dans ces graphiques c'est la somme des coûts. C'est en quelque sorte une indication sur le coût moyen. Or notre proposition consiste à optimiser "individuellement" chacune des requêtes. Ainsi, le fait d'avoir des sommes de coûts

proches de ceux obtenus par HRU est plutôt un avantage.

Analyse du facteur de performance

Pour analyser plus en détail le comportement de ces algorithmes, nous avons étudié le facteur de performance. Bien que ni PBS ni HRU n'aient été conçus en vue de garantir une performance des requêtes respectivement au coût moyen, nous avons procédé à une comparaison de ces algorithmes avec les résultats qu'on obtient avec notre approche. A cet effet, nous avons procédé de la même manière que précédemment, c'est à dire faire varier la valeur de f et pour chaque valeur prendre la taille de notre solution comme limite pour HRU et PBS. La figure 3 montre que notre solution a un meilleur facteur de performance moyen que ceux des deux autres algorithmes. (On montre juste les résultats pour $f = 3.38$ et $f = 11.39$).

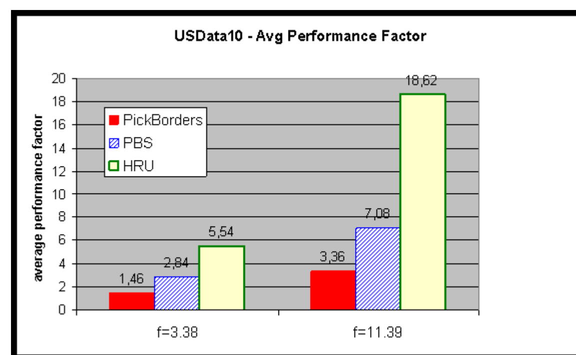
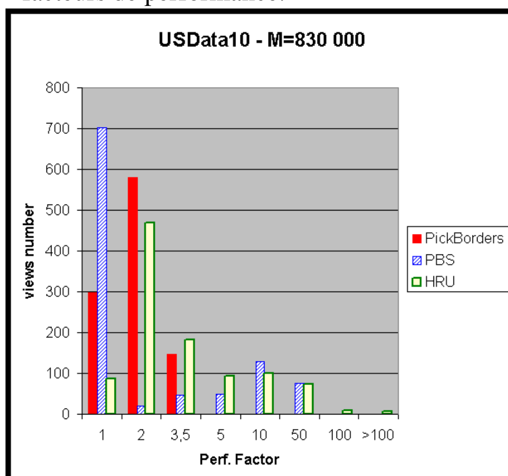
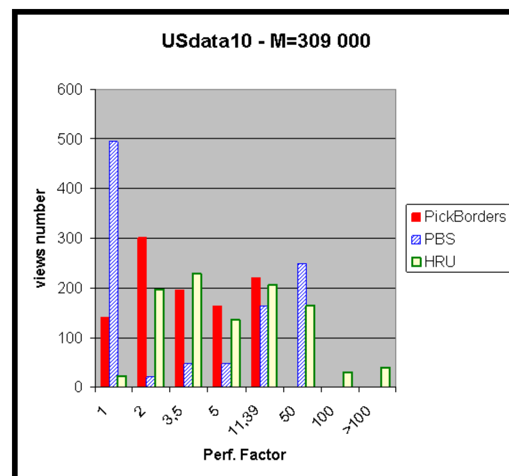


FIG. 3 – *Facteur de performance moyen pour* $f = 3.38$ *et* $f = 11.39$.

Les figures ci-dessous permettent une analyse encore plus détaillée de la distribution des facteurs de performance.



Distribution du facteur de performance quand $f = 3.38$.



Distribution du facteur de performance quand $f = 11.39$.

Par exemple, en exécutant HRU avec une limite d'espace mémoire de 830000, il y a 7 cuboïdes non matérialisés dont le plus petit ancêtre matérialisé a une taille plus que 100 fois plus grande que leurs tailles. Nous avons lancé PBS et HRU avec 309000 et 830000 comme contraintes d'espace. Ce sont les tailles de nos solutions quand f est fixé à 3.38 et 11.39. La première catégorie représente les cuboïdes dont le facteur de performance est égal à 1. Ce sont ceux qui sont matérialisés ou bien leur ancêtre a la même taille qu'eux. La seconde catégorie représente ceux dont le facteur de performance est dans $]1, 2]$. La troisième, ceux dont le facteur est dans $]2, 3.5]$ et ainsi de suite. On peut noter que la première catégorie reflète l'esprit des différents algorithmes : PBS a tendance à stocker beaucoup de cuboïdes de petites taille (puisqu'il commence par eux jusqu'à épuiser l'espace imparti), HRU tend plutôt à privilégier les cuboïdes de grande taille, quant à notre approche, elle a tendance à choisir une combinaison de cuboïdes de différentes tailles.

5.2 Le cas avec charge

Nous avons considéré USdata10. La taille du cuboïde de base est de 52278 (une fois les doublons de la table de faits enlevés). Nous avons généné $\mathcal{Q}$ de manière aléatoire uniforme en faisant varier la cardinalité de la charge entre 4 et 1024. Nous avons comparé la mémoire de $\mathcal{Q}$ avec la mémoire de la solution optimale S_{opt} ainsi que la solution retournée par PickFrom$f$Ancestors pour la valeur $f = 10$. PickFrom$f$Ancestors se comporte trés bien par

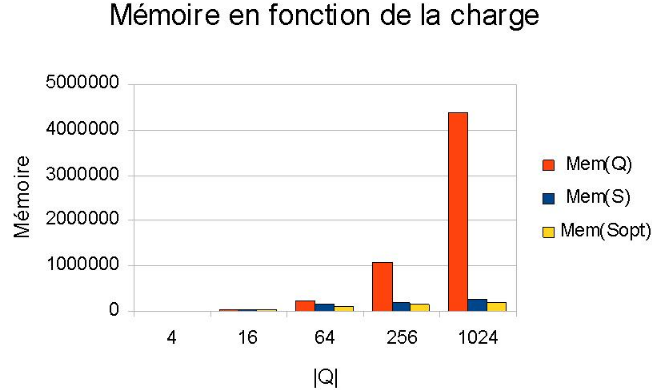


FIG. 4 – *Comparaison entre PickFromfAncestors et la solution optimale.*

rapport à la solution optimale et le gain de mémoire par rapport à la matérialisation complète de $\mathcal{Q}$ est extrêmement important. Le véritable apport de PickFrom$f$Ancestors par rapport à la programmation linéaire est le temps d'exécution : pour PickFrom$f$Ancestors, cela va de $0.01s$ ($|\mathcal{Q}|$ =4) à $0.29s$ ($|\mathcal{Q}|$ =1024). Or le programme linéaire exécuté sur CPLEX met de $0.1s$ ($|\mathcal{Q}|$ =4) à $3366s$ ($|\mathcal{Q}|$ =1024)[9]. En dimension supérieure, la programmation linéaire prend

[9] LP_Solve était incapable de résoudre des programme d'une taille aussi grande.

beaucoup trop de temps. En dimension $\geq$ 20, PickFrom$f$Ancestors donne une solution en temps inférieur à la minute. Bien entendu, cela dépend de la valeur de f et de $|\mathcal{Q}|$.

6 Conclusion

Nous avons présenté dans cet article des techniques permettant de sélectionner les vues à matérialiser dans le cadre des cubes de données. L'intérêt de notre approche est que le critère de sélection est basé sur le facteur de performance de l'évaluation des requêtes non pas sur l'espace mémoire requis. Nous pensons que ce critère est peut être plus fondé car d'un coté une solution qui occupe un espace mémoire M à un instant t_i occupera un espace $M + m$ à t_{i+1}. Faudra-t-il alors recalculer les vues à matérialiser ? Aussi, nous pensons que les solutions basées sur le facteur de performance seront plus stables dans le temps car il y a bon espoir que les rapports entre les tailles soient préservés lorsque les données évoluent avec le temps. Ceci représente une des voies que l'on compte explorer : analyser l'évolution des bordures sous certaines hypothèses de distribution des données. Enfin, nous avons étudié deux cas de figures : celui où toutes les requêtes sont possibles et celui où on restreint l'ensemble des requêtes. On peut se dire que le premier cas n'est en fait qu'un cas particulier du second ($\mathcal{Q} = \mathcal{C}$). En effet, lorsque le nombre de dimensions est réduit, on peut utiliser les techniques de programmation linéaire pour résoudre le problème. Nous comptons dans nos prochains travaux enrichir notre modèle pour pouvoir tenir compte des coûts de mise à jour dans le même esprit que Baralis et al. (1997); Baril et Bellahsene (2003) et pour intégrer la possibilité d'avoir des requêtes qui ont besoin de plusieurs vues (cf. Gupta et Mumick (2005); Palpanas et al. (2008)). Enfin, nous n'avons pas considéré des dimensions structurées sous forme de hiérarchies. Or dans l'analyse en ligne des cubes de données, les opérations de *RollUp* et *DrillDown* sont communes. La prise en compte de ces hiérarchies ne pose pas de problèmes majeurs. En effet, comme cela a été montré dans Harinarayan et al. (1996), les hiérarchies elles mêmes forment des treillis. Comme le produit de deux treillis est lui même un treillis, on se retrouve donc finalement avec la même structure. Il faudra néanmoins modifier quelque peu le programme (ou le codage des cuboïdes) de recherche des bordure car e.g c_1 peut être un parent de c_2 alors que les deux cuboïdes ont exactement le même nombre de dimensions. Pour terminer, nous comptons étudier la possibilité d'intégrer notre approche avec celle adoptée par Bellatreche et al. (2008) en proposant de matérialiser des parties (horizontales) des tables d'un schéma en étoile.

Références

Aouiche, K., P.-E. Jouve, et J. Darmont (2006). Clustering-based materialized view selection in data warehouses. In *Proceedings of ADBIS'06 conference*.

Aouiche, K. et D. Lemire (2007). A comparison of five probabilistic view-size estimation techniques in olap. In *Proceedings of DOLAP'07 conference*.

Baralis, E., S. Paraboschi, et E. Teniente (1997). Materialized views selection in a multidimensional database. In *Proceedings VLDB'97 conference*.

Baril, X. et Z. Bellahsene (2003). Selection of materialized views : A cost-based approach. In *Proceedings of CAiSE'03 conference*.

Bayardo, Jr., R. J. (1998). Efficiently mining long patterns from databases. In *Proceedings of SIGMOD'98 conference*.

Bellatreche, L., K. Boukhalfa, et P. Richard (2008). Data partitioning in data warehouses : Hardness study, heuristics and ORACLE validation. In *Proceedings of DaWaK'08 conference*.

Burdick, D., M. Calimlim, et J. Gehrke (2001). Mafia : A maximal frequent itemset algorithm for transactional databases. In *Proceedings of ICDE'01 conference*.

Chvàtal, V. (1979). A greedy heuristic for the set covering problem. *Mathematics of operation research 4*(3), 233–235.

Gouda, K. et M. J. Zaki (2005). GenMax : An efficient algorithm for mining maximal frequent itemsets. *Data Mining and Knowledge Discovery 11*(3), 223–242.

Gupta, H., V. Harinarayan, A. Rajaraman, et J. D. Ullman (1997). Index selection for olap. In *Proceedings of ICDE'97 conference*.

Gupta, H. et I. S. Mumick (2005). Selection of views to materialize in a data warehouse. *IEEE Trans. on Knowl. and Data Eng. 17*(1), 24–43.

Harinarayan, V., A. Rajaraman, et J. D. Ullman (1996). Implementing data cubes efficiently. In *Proceedings of SIGMOD'96 conference*.

Jim Gray, *et al.* (1997). Data cube : A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Mining and Knowledge Discovery 1*(1), 29–53.

Karloff, H. et M. Mihail (1999). On the complexity of the view-selection problem. In *Proceedings of PODS'99 conference*.

Karp, R. (1972). Reducibility among combinatorial problems. In R. E. Miller et J. W. Thatcher (Eds.), *Complexity of Computer Computations*, pp. 85–103. Plenum Press.

Maiz, N., K. Aouiche, et J. Darmont (2006). Sélection automatique d'index et de vues matérialisées dans les entrepôts de données. In *Proceedings de (EDA 06)*.

Hanusse, N., S. Maabout, et R. Tofan (2009). A view selection algorithm with performance guarantee. In *Proceedings of EDBT'09 Conference*.

Palpanas, T., N. Koudas, et A. Mendelzon (2008). On space constrained set selection problems. *Data Knowl. Eng. 67*(1), 200–218.

Shukla, A., P. Deshpande, et J. F. Naughton (1998). Materialized view selection for multidimensional datasets. In *Proceedings of VLDB'98 conference*.

Talebi, Z. A., R. Chirkova, Y. Fathi, et M. Stallmann (2008). Exact and inexact methods for selecting views and indexes for olap performance improvement. In *Proceedings of EDBT'08*.

Summary

In this paper we consider the problem of selecting materialized views in the context of datacubes. In contrast to most other approaches, the constraint we aim to satisfy is related to query performance instead of memory space limit. We show the advantages of our approach both theoretically and experimentally.