Dépendances fonctionnelles et matérialisation partielle des cubes de données

Eve Garnaud, Sofian Maabout, Mohamed Mosbah

LaBRI. Université de Bordeaux
351, cours de la Libération, 33405 Talence
{garnaud, maabout, mosbah}@labri.fr

Résumé. La sélection de vues à matérialiser dans des entrepôts de données de plus en plus volumineux est une nécessité. Dans cet article, nous montrons qu'il existe un lien très étroit entre recherche des cuboïdes à matérialiser dans un cube de données afin d'optimiser les traitements et les dépendances fonctionnelles sur celui-ci. La contrainte que nous imposons sur les vues que l'on matérialise ne porte pas sur une borne d'espace de stockage à ne pas dépasser comme c'est le cas dans la plupart des travaux relatifs, mais elle porte sur le facteur de performance f que celles-ci vérifient. Nous tentons cependant d'utiliser le moins d'espace mémoire pour atteindre cet objectif. Nous caractérisons formellement toute solution optimale (en terme d'espace mémoire) répondant à ce critère. On prouve que ce problème est NP-difficile et on démontre l'efficacité de nos algorithmes gloutons pour répondre à ce problème en respectant la contrainte de performance fixée par l'utilisateur.

1 Introduction

De nos jours, la quantité d'information stockée par les entreprises est de plus en plus importante ce qui rend le travail d'analyse et d'exploitation de ces informations souvent long et complexe. Une interface sous forme de cube de données permet d'optimiser ces traitements puisqu'il est possible de choisir de ne matérialiser qu'une partie des cuboïdes qui le composent. Le problème est de savoir quelle est la meilleure partie à stocker pour répondre, dans un temps minimum, à toutes les requêtes.

Nous développons donc une technique basée sur les dépendances fonctionnelles pour trouver la meilleure solution à ce problème. Cette solution est optimale dans le sens où, étant donné un facteur de performance $f \geq 1$, on peut répondre à toutes les requêtes posées sans que le coût de calcul de toutes ces requêtes ne dépasse de plus de f fois leur coût de calcul si tous les cuboïdes étaient matérialisés. Il serait trop couteux en terme d'espace mémoire et de coût de maintenance de stocker tous les cuboïdes, c'est pourquoi une sélection s'impose.

Nous prouvons que ce problème de sélection est NP-difficile, cependant, nous sommes en mesure de caractériser précisément, grâce aux dépendances fonctionnelles, les sous-ensembles de cuboïdes solutions et cela, quel que soit le f fixé par l'utilisateur. Cette caractérisation nous assure que la solution apportée est bien de taille minimale par rapport à une autre sélection qui respecterait elle aussi la contrainte f.

Afin de mieux formaliser le problème, nous présentons dans la seconde section quelques notations et définitions originales nous permettant de mieux comprendre les enjeux du problème. Ensuite, nous montrons, dans la section 3, que cette question de sélection des cuboïdes peut être résolue à l'aide des dépendances fonctionnelles qui existent entre les différentes dimensions du cube. Nous prouvons que ce problème est NP-difficile. Nous proposons donc des algorithmes gloutons permettant de répondre efficacement à la demande de sélection en respectant la contrainte $f \geq 1$. Puis, un rapide de l'art nous permet de constater que, à notre connaissance, le problème a rarement était abordé dans une optique de garantie de performance, cela explique pourquoi les résultats proposés ne sont qu'une préparation permettant de poursuivre nos travaux dans le futur. Les différents axes de recherche que nous souhaitons traiter sont développés dans la section 5.

2 Définitions et notations

Nous commençons par présenter quelques notations et définitions qui seront utilisées tout au long du papier. Afin de mieux visualiser l'entrepôt de données, on utilise la modélisation multidimentionnelle qui permet de représenter les données sous forme d'un cube Gray et al. (1997). Pour simplifier la présentation, nous allons considérer un schéma constitué uniquement d'une table de faits. Chaque attribut est une dimension du cube. Le cube C obtenu à partir d'une table de faits T est défini par une requête de la forme

```
SELECT    dimensions, mesures
FROM      T
GROUP BY  CUBE(dimensions)
```

`dimensions` représentent les attributs de la table T et `mesures` représente un ensemble de fonction d'agrégation distributives telles que `COUNT`, ou `SUM`. Cette requête est équivalente à l'union de toutes les requêtes où `GROUP BY` est appliqué à un sous ensemble des dimensions. Ainsi, si n est le nombre d'attributs, le nombre de requêtes correspondant au cube est 2^n. Chaque résultat de ces requêtes est appelé *cuboïde* ou *vue*.

Le cube C est donc un ensemble des cuboïdes (ou vues). On pose $n = |Dim(C)|$ où $Dim(C)$ est l'ensemble des n dimensions du cube. Dans cet article, nous considérons la situation où l'on ne dispose pas de charge (workload). On suppose donc que toutes les vues du cube sont équi-probablement utilisées par les requêtes. Les requêtes sont de la forme :

```
SELECT    attributs, mesure
FROM      T
GROUP BY  attributs
```

Soit $\preceq$ la relation d'inclusion des dimensions entre les cuboïdes, $< C, \preceq >$ forme le treillis que nous allons considérer.

Définition 1 (Ordre partiel du treillis)**.** *Soit $\preceq$ un ordre partiel entre les vues du treillis $< C, \preceq >$. Pour deux vues u et v, $v \preceq u$ si et seulement si on peut répondre à toute requête posée sur v en utilisant uniquement la vue u. On peut définir ainsi, pour un élément a du treillis :*

- *l'ensemble de ses ancêtres : $A_a = \{b \mid a \preceq b\}$, on a $Dim(a) \subseteq Dim(b)$*
- *l'ensemble de ses descendants : $D_a = \{b \mid b \preceq a\}$*

– *l'ensemble des parents :* $P_a = \{b \mid a \preceq b, \nexists c \mid a \preceq c \wedge c \preceq b\}$*, on peut en déduire* $|Dim(b)| = |Dim(a)| + 1$

Pour optimiser les requêtes sur le cube C, l'idéal serait de calculer et stocker tous ses cuboïdes. Mais cette solution n'est pas viable du fait de l'espace mémoire trop important qu'elle occuperait et la nécessité de mettre à jour tous ses cuboïdes à la moindre modification de la table de faits ce qui prendrait trop de temps. En pratique, seul un sous ensemble S des cuboïdes sera matérialisé. Nous cherchons à savoir quel est le meilleur sous ensemble qui nous permettra de répondre à toutes les requêtes en respectant le facteur de performance f tout en utilisant un minimum d'espace mémoire.

Comme dit précédemment, les requêtes que l'on considère consistent simplement à retourner le contenu d'un cuboïde. Lorsque celui-ci est déjà pré-calculé, alors le temps de la réponse sera proportionnel à sa taille puisqu'il suffit juste de le parcourir. Quand par contre il n'est pas stocké, il faut chercher le plus petit (en termes de taille) de ses ancêtres stockés et utiliser ce dernier pour évaluer la requête. Il existe essentiellement deux méthodes pour évaluer les requêtes avec `GROUP BY` Graefe (1993); Chaudhuri (1998), (1) trier la table pour constituer les groupes puis effectuer les opérations d'agrégation pour chaque groupe. On se retrouve donc avec un coût de l'ordre de $m \log m$ avec m qui est la taille de la table ou bien (2) on utilise le hachage ce qui permet d'atteindre un coût linéaire de l'ordre de m. Nous allons supposer l'utilisation du hachage. Nous donnons maintenant le modèle de coût d'une requête en fonction de l'ensemble S des cuboïdes matérialisés.

Définition 2 (Modèle de coût)**.** *Soit* $S \subseteq C$ *l'ensemble des cuboïdes qui sont matérialisés.* $\mathcal{C}(q, S)$*, le coût d'une requête* q *respectivement à* S*, est défini par* $min_{w \in A_q}|w|$ *avec* $w \in S$ *et* $|w|$ *est le nombre de tuples de la vue* w *ancêtre de* q*.*

Le lecteur aura noté que nous confondons taille mémoire d'un cuboïde avec le nombre de ses tuples. En effet, pour être plus précis, la taille d'un cuboïde correspond à la place mémoire occupée par un tuple multipliée par le nombre de tuples. Mais, afin de ne pas alourdir davantage les calculs, nous ne considérons que le nombre de tuples pour définir la taille d'un cuboïde.

Exemple 1. *Soit la table de fait* T *ci-dessous (nous allons l'étudier tout au long de l'article afin d'illustrer nos propos). Nous avons quatre attributs* A*,* B*,* C *et* D*, donc le cube que nous considérons a quatre dimensions.*

A	B	C	D
1	1	1	1
2	1	1	1
3	2	1	1
4	2	1	2

FIG. 1 – *Exemple de table de faits.*

Considérons le cube C *défini par*

```
SELECT    A, B, C, D, COUNT(*) AS NOMBRE
FROM      T
GROUP BY  CUBE (A, B, C, D)
```

C est composé de $2^4 = 16$ cuboïdes. Le cuboïde de base c_b est la table T elle même, le cuboïde A contient 4 tuples tandis que le cuboïde BC n'est composé que des 2 tuples $(B = 1, C = 1, NOMBRE = 2)$ et $(B = 2, C = 1, NOMBRE = 2)$. Supposons que les cuboïdes matérialisés sont $S = \{ABCD, BCD, BC, CD\}$. Soit la requête q

```
SELECT    D, COUNT(*)
FROM      T
GROUP BY  D
```

Pour évaluer cette requête, nous pouvons utiliser soit le cuboïde $ABCD$ soit BCD ou encore CD (ce sont les seules ancêtres de D qui soient stockés). $|CD| = 2$, $|BCD| = 3$ et $|ABCD| = 4$. Nous avons donc intérêt à utiliser CD comme suit

```
SELECT    D, SUM(NOMBRE)
FROM      CD
GROUP BY  D
```

Nous avons donc $\mathcal{C}(q, S) = 2$.

Nous donnons à présent les définitions relatives aux dépendances fonctionnelles que nous utilisons dans cet article. Bien entendu, les dépendances fonctionnelles, exactes comme approximatives, on déjà été définies mais notre approche est originale et c'est pourquoi nous souhaitons attirer l'attention des lecteurs sur ce qui suit.

Définition 3 (Dépendance fonctionnelle)**.** *Soient R un schéma relationnel, X et Y deux sous ensembles de R. On dit que l'instance r de R satisfait la dépendance fonctionnelle $X \rightarrow Y$ avec un degré de correction de $c \leq 1 \in \mathbb{Q}$ si et seulement si $\frac{|X|}{|XY|} = c$ où $|X|$ est le nombre de tuples (ou la taille de X et $|XY|$ la taille de $|XY|$. On note dans ce cas, $r \models X \rightarrow Y(c)$.*
On dit que la dépendance fonctionnelle $X \rightarrow Y$ est exacte si nous avons $c = 1$, sinon, elle est approximative.

Intuitivement, le degrés de correction d'une dépendance approximative $X \rightarrow Y$ reflète le nombre *moyen* de valeurs de Y qui sont associées à chaque valeur de X. Plusieurs définitions de dépendances approximatives ont été proposées dans la littérature Giannella et Robertson (2004). Malgré nos recherches bibliographiques, nous n'avons pas trouvé de travaux ayant étudié le type de dépendances définies ci-dessus. Ceci nous a poussé à vérifier les axiomes d'Armstrong pour ce "nouveau" type de dépendances. Ceci sera développé dans la Section 3.

Exemple 2. *Reprenons la table illustrée dans Figure 1. Parmi les dépendances que cette table satisfait, on trouve :$A \rightarrow B(1)$, $B \rightarrow C(1)$, $C \rightarrow D(0,5)$ et $C \rightarrow B(0,5)$. Le lecteur peut remarquer que les deux dernières dépendances ont le même degrés de correction alors que pour les autres approches qu'on rencontre dans la littérature, $C \rightarrow D$ est "plus correcte" que $C \rightarrow B$ car, pour qu'elle soit exacte, il suffit de supprimer un seul tuple (le dernier). Alors que pour $C \rightarrow B$, il faut supprimer les deux derniers tuples.*

Le lecteur peut vérifier que pour tout X et tout Y ensembles d'attributs, $r \models X \rightarrow Y(c)$ avec $\frac{1}{n} \leq c \leq 1$ où n est le nombre de tuples de r.
Précisions maintenant, grâce aux dépendances fonctionnelles, les relations d'ordre que nous avons définies précedemment dans le treillis.

Définition 4 (f_Ancêtre)**.** *Soient X un cuboïde et XY un ancêtre de X. XY est un f_ancêtre de X si et seulement si $\frac{|XY|}{|X|} \leq f$.*

Nous définissons maintenant la solution au problème que nous voulons résoudre. A savoir, l'ensemble des cuboïdes qu'il faut matérialiser étant donné un facteur de performance f ($f \geq 1$) fixé par l'utilisateur.

Définition 5 (Solution f_optimale)**.** *Une solution S est f_optimale si et seulement si :*

1. *$\forall q \in C, fp(q,S) \leq f$ avec $fp(q,S) = \mathcal{C}(q,S)/|q|$. Ce qui s'écrit également, $\exists\, q' \in S$ telle que q' est un f_ancêtre de q.*
2. *S est de taille minimale parmi pour les ensembles de solutions S' qui satisfont la contrainte 1.*

Exemple 3. *Revenons à l'exemple précédent. On voit que le coût d'exécution de q sur D est le même que le coût d'exécution de cette requête sur le plus petit ancêtre de D appartenant à S. Nous respectons ici un facteur de performance $f = \frac{\mathcal{C}(q,S)}{\mathcal{C}(q,D)} = \frac{2}{2} = 1$.*

Intuitivement, la solution que nous cherchons permet d'évaluer chaque requête à partir d'un ancêtre matérialisé dont la taille ne dépasse pas f fois la taille du résultat de la requête. D'une manière équivalente, sachant que si la requête elle même est déjà matérialisée alors on aurait le coût minimal, notre solution permettra d'avoir un coût au plus f fois supérieur au coût minimal.

3 Contributions

Commençons par caractériser la solution 1_optimale. C'est-à -dire que l'on cherche une solution dont le coût total serait le même que si on matérialisait tous les cuboïdes. On souhaite cependant trouver une solution de taille minimale.

3.1 Dépendances fonctionnelles exactes ($c = 1$) et solution 1_optimale

Etudions le lien entre dépendances fonctionnelles "exactes" et solution 1_optimale par le biais de quelques lemmes et propositions.

Définition 6 (Cuboïde clos)**.** *X est clos si et seulement si $\forall Y \mid X \subset Y \Rightarrow |X| < |Y|$. Autrement dit, tous les ancêtres de X ont une taille strictement supérieure à celle de X.*

Définition 7 (Fermeture d'un ensemble d'attributs)**.** *Soient $\mathcal{F}$ un ensemble de dépendances fonctionnelles et X un ensemble d'attributs. La fermeture de X par rapport à $\mathcal{F}$, notée X^+ est l'ensemble des attributs A tels que $\mathcal{F} \models X \rightarrow A(1)$.*

Lemme 1. *X est clos si et seulement si $X^+ = X$.*

Preuve. $\Rightarrow$: si X est clos alors, par définition, $\forall Y$ tel que $X \subset Y$ on a $|X| < |Y|$. Ainsi, $\frac{|X|}{|Y|} < 1$. D'où, $X \not\rightarrow Y(1)$ et donc $X^+ = X$.

$\Leftarrow$: Si $X^+ = X$ alors $\nexists A \mid X \subset A$ et $\mathcal{F} \models X \rightarrow A(1)$. Ainsi, $\frac{|X|}{|A|} < 1$ et donc, $|X| < |A|$. D'où X qui est clos par définition. □

Exemple 4. *Dans notre exemple, C est de taille 1 et tous ses ancêtres sont de taille 2, on dit donc que C est clos et nous avons $C^+ = C$.*

Revenons plus concrètement au problème de sélection des vues à matérialiser pour répondre à toutes les requêtes $\mathcal{Q}$ de C en optimisant le temps de maintenance sur le cube de données sans pour autant perdre en performance. On cherche donc une solution 1_optimale.
Puisqu'un cuboïde clos a une taille strictement inférieure à celle de tous ses ancêtres, il semble pertinent de le matérialiser pour trouver la solution 1_optimale.

Proposition 1. *Soient F_1 l'ensemble des cuboïdes clos et S la solution 1_optimale au problème de sélection de vues. On a alors $S = F_1$.*

Preuve. Soit S la solution optimale au problème de sélection de vues.
Supposons qu'il existe un cuboïde $c \in S$ non clos. c est non clos donc il existe un ancêtre c' de c tel que $|c| = |c'|$. Le coût d'évaluation d'une requête sur c est donc le même que le coût d'évaluation de la même requête sur c'. Il est donc préférable de stocker uniquement c' et non c puisque c' permet de répondre un plus grand nombre de requêtes. On procède de la même façon sur c' s'il n'est pas clos non plus et ainsi de suite.
c n'appartient donc pas à S et tous les cuboïdes de S sont des cuboïdes clos.
Supposons maintenant qu'il existe un cuboïde c clos qui n'appartient pas à S. Pour évaluer une requête q posée sur c, il va donc falloir utiliser un ancêtre c' de c appartenant à S. Or, étant donné que c est clos, on sait que $|c| < |c'|$ donc $\mathcal{C}(q, c') > \mathcal{C}(q, c)$. On en déduit naturellement que S n'est pas la solution 1_optimale si elle ne contient pas c.
On voit donc que S contient tous les cuboïdes clos du tréillis et uniquement ceux là. □

Les algorithmes suivants nous permette de trouver la solution 1_optimale S au problème de sélection des vues dans le cube de données :

L'algorithme 1 peut être utilisé si l'on suppose que l'on connait la taille de tous les cuboïdes du cube. Si ce n'est pas le cas mais que nous disposons de l'ensemble $\mathcal{F}$ des dépendances fonctionnelles, nous pouvons appliquer l'algorithme 2.

Théorème 1. *Le problème de sélection des cuboïdes clos est NP-difficile.*

Preuve. On prouve la NP-difficulté du problème de recherche des cuboïdes clos par réduction au problème de sélection des vues à matérialiser pour obtenir une solution 1_optimale puisqu'on sait que ce dernier est NP-difficile.
Pour qu'une solution S soit 1_optimale, il faut que $\mathcal{C}(S) = \mathcal{C}(C)$. Le problème de sélection des vues à matérialiser consiste donc à choisir les vues c telles que $\forall d \in D_c$ non matérialisée on a $\mathcal{C}(\mathcal{Q}_d, d) = \mathcal{C}(\mathcal{Q}_c, c)$. Or on sait que $\mathcal{C}(\mathcal{Q}_d, d) = |d|$ si toutes les vues sont matérialisée et $\mathcal{C}(\mathcal{Q}_d, d) = |c|$ si c est le premier ancêtre de d matérialisé. On cherche donc à matérialiser les vues qui ont la même taille que leurs descendants et une taille inférieure à celle de leurs

Algorithm 1 Algorithme de sélection des cuboïdes clos

$S \leftarrow c_b$
for all $c \in \mathcal{Q}$ **do**
  clos $\leftarrow$ true
  for all $c' \in \mathcal{Q}$ parent de c et clos = true **do**
    if $|c| = |c'|$ **then**
      clos $\leftarrow$ false
    end if
  end for
  if clos = true **then**
    $S \leftarrow S \cup c$
  end if
end for
retourner S

Algorithm 2 Algorithme de sélection des cuboïdes clos

Require: $\mathcal{F}$
for all $c \in \mathcal{Q}$ **do**
  clos $\leftarrow$ true
  for all $X \rightarrow Y \in \mathcal{F}$ **do**
    if $X \subset c$ et $Y \not\subset c$ **then**
      clos $\leftarrow$ false
    end if
  end for
  if clos = true **then**
    $S \leftarrow S \cup c$
  end if
end for
retourner S

ancêtres pour assurer la 1_optimalité. Ce qui se traduit donc par chercher les cuboïdes c tels que $c^+ = c$. □

Exemple 5. *Avec la table de faits T de la figure 1, nous pouvons dessiner le cube de données à quatre dimensions A, B, C et D suivant :*

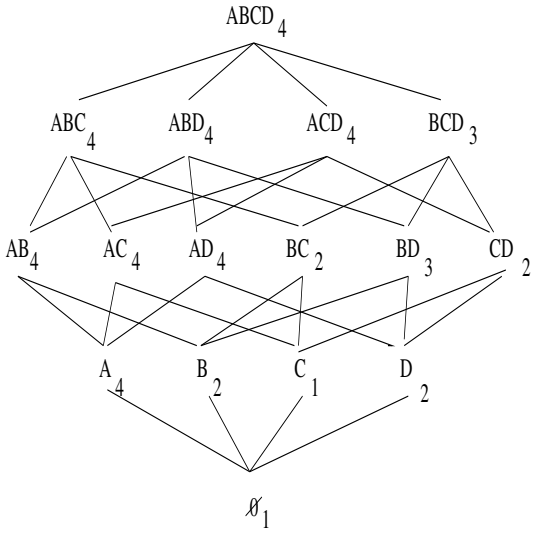


FIG. 2 – *Cube de données construit à partir de T*

On voit ici que la solution 1_optimale consiste à matérialiser les cuboïdes C, BC, CD, BCD et $ABCD$ puisqu'aucun de leurs ancêtres n'a une taille égale à la leur et de ce fait, il serait plus couteux d'interroger un de ces ancêtres plutôt que les cuboïdes eux-mêmes directement. On a ici $\mathcal{F}_{exacte} = \{A \rightarrow B(1), A \rightarrow C(1), A \rightarrow D(1), B \rightarrow C(1), D \rightarrow C(1)\}$. On note $size(S)$ la taille de la solution S, et on a, pour S la solution 1_optimale, $size(S) = 4 + 3 + 2 + 2 + 1 = 12 << size(C) = 50$ ce qui représente donc un gain considérable en terme d'espace mémoire d'autant que $\forall q, \mathcal{C}(q, S) = \mathcal{C}(q, C)$.

Nous avons vu que l'on peut élargir le sens de dépendances fonctionnelles "exactes" en associant à une valeur x_i du cuboïde X plusieurs valeurs y de la table Y. Cela nous permet de caractériser tous les ensembles d'attributs entre eux et, de ce fait, nous pouvons exploiter un plus grand nombre de propriétés sur le cube.

3.2 Dépendances fonctionnelles approximatives et solution f_optimale

Etudions maintenant comment, par ce biais, nous pouvons trouver une solution f_optimale.

Lemme 2. *X est un f_ancêtre de Y si et seulement si*

1. *X est un ancêtre de Y et*
2. *$X \setminus Y \rightarrow Y (c \geq \frac{1}{f})$.*

Preuve. X est un ancêtre de $Y \Leftrightarrow X = X'Y$ où X' est un ensemble d'attributs et $Y \cap X' = \emptyset$.
Donc X est un f-ancêtre de $Y \Leftrightarrow X' \rightarrow Y(c \geq \frac{1}{f})$ par définition.
$X' \rightarrow Y(c \geq \frac{1}{f}) \Leftrightarrow \frac{X'Y}{Y} \leq f$. Or $X = X'Y$ donc, par définition, on a bien que X est un f_ancêtre de Y. □

Exemple 6. *Dans l'exemple précédent, $ABCD$ est un 1_ancêtre de A ($|ABCD| = |A|$) mais aussi un 2_ancêtre de B et D ($|ABCD| = 2 \times |B| = 2 \times |D|$) et un 4_ancêtre de C ($|ABCD| = 4 \times |C|$). On peut ainsi calculer avec quel degré chaque cuboïde est ancêtre de ses descendants.*

Avec cette nouvelle définition de dépendance fonctionnnelle approximative, vérifions les axiomes d'Armstrong.

Lemme 3 (Réduction). *$X \rightarrow Y(c_1)$ et $X \rightarrow Z(c_2)$ alors $X \rightarrow YZ(c_3)$ où $c_3 \leq min(c_1, c_2)$. Nous pouvons transposer ce lemme en termes de tailles de cuboïdes comme suit : $\frac{|XY|}{|X|} = f_1$ et $\frac{|XZ|}{|X|} = f_2$ alors $\frac{|XYZ|}{|X|} \geq max(f_1, f_2)$.*

Preuve. Soient $\frac{|XY|}{|X|} = \frac{b}{a}$ et $\frac{|XZ|}{|X|} = \frac{c}{a}$.
On sait que $\frac{|XYZ|}{|X|} = \frac{d}{a}$ avec $d \geq max(b, c, |YZ|)$, donc on a $\frac{|XYZ|}{|X|} \geq max(\frac{b}{a}, \frac{c}{a})$. □

Lemme 4 (Transitivité). *$X \rightarrow Y(c_1)$ et $Y \rightarrow Z(c_2)$ alors $X \rightarrow Z(c_3)$ où $c_3 \geq c_1 \times c_2$. Nous pouvons l'écrire si $\frac{|XY|}{|X|} = f_1$ et $\frac{|YZ|}{|Y|} = f_2$ alors $\frac{|XZ|}{|X|} \leq f_1 \times f_2$.*

Preuve. (par l'absurde) On suppose $\frac{|XZ|}{|X|} > \frac{|XY|}{|X|} \times \frac{|YZ|}{|Y|} \Leftrightarrow |XZ| > \frac{|XY| \times |YZ|}{|Y|} \geq \frac{|XYZ| \times |Y|}{|Y|} = |XYZ|$ or, on sait que $|XZ| \leq |XYZ|$ donc il y a une incohérence et on peut en déduire qu'on a bien $\frac{|XZ|}{|X|} \leq \frac{|XY|}{|X|} \times \frac{|YZ|}{|Y|}$. □

Lemme 5 (Augmentation). *$X \rightarrow Y(c_1)$ alors $\forall Z : XZ \rightarrow Y(c_2)$ avec $c_2 \geq c_1$. Ce qui est équivalent à dire que $\frac{|XY|}{|X|} \geq \frac{|XYZ|}{|XZ|}$.*

Preuve. Posons DF1 : $XZ \rightarrow X(c_3 = 1)$ car $\frac{|XZ|}{|XZ|} = 1$ et DF2 : $X \rightarrow Y(c_1)$.
Par transitivité, on déduit de DF1 et DF2 : $XZ \rightarrow Y(c_2)$ avec $c_2 \geq c_3 \times c_1 = 1 \times c_1 = c_1$. □

Grace aux dépendances fonctionnelles approximatives, nous pouvons étendre les définitions de cuboïdes clos et de fermeture comme suit :

Définition 8 (Cuboïde f_clos). *X est f_clos si $\forall Y \in A_X$, $|X| \leq f \times |Y|$.*

Définition 9 (f_fermeture). *Soit X un ensemble d'attributs. La f_fermeture de X notée X_f^+ est l'ensemble des attributs A tels que $X \rightarrow A(c)$ et $c \geq \frac{1}{f}$. X est f_fermé si et seulement si $X_f^+ = X$.*

Exemple 7. *En reprenant toujours l'exemple utilisé jusque là, nous avons par exemple $C_2^+ = \{BC, CD\}$ et C est 1_clos.*

Voyons maintenant les liens qui existent entre ces ensembles d'attributs f_fermés.

Proposition 2. *$\forall f, f'$ tels que $f' \leq f$, on a $F_f \subseteq F_{f'}$ où F_f (resp. $F_{f'}$) est l'ensemble des cuboïdes f_fermés (resp. f'_fermés).*

Preuve. Si un ensemble X d'attributs est f_fermé alors aucun de ses ancêtres n'a une taille inférieure à f fois sa taille. Puisque $f' \leq f$, alors X est également f'_fermé par définition. □

Proposition 3. *Soit S^* la solution 1_optimale, alors, pour toute solution S f_optimale, on a $S \subseteq S^*$*

Preuve. Commençons par montrer qu'il existe une solution S f_optimale telle que $S \subseteq S^*$.
Soit $c \notin S^*$, alors $\exists c' \in A_c$ tel que $c' \in S^*$. Stocker l'un ou l'autre des cuboïdes nous est indifférent en terme d'occupation mémoire mais nous savons que $\mathcal{Q}_c \subset \mathcal{Q}_{c'}$. La contrainte de performance étant élargie, nous ne trouverons aucun intérêt supplémentaire à matérialiser c à la place de son ancêtre.
Si $c \in S^*$, alors $c_1^+ = c$ et son importance au sein du cube de données dans la recherche d'une solution n'est pas à démontrer. La question est de savoir si, étant donné que $f > 1$, sa matérialisation est toujours indispensable.
Voyons maintenant si une solution S' f_optimale peut contenir des cuboïdes non clos. Soient $c \notin S^*$ et $c^+ \in S^*$. Soient $||\mathcal{Q}_c||$ et $||\mathcal{Q}_{c'}||$ la cardinalité des ensembles. On a $||\mathcal{Q}_{c'}|| \geq ||\mathcal{Q}_c|| + 1$ donc

- si nous supposons c^+ f_clos, alors matérialiser c à la place de c^+ ne permettrait pas de garantir la f_optimalité de la solution.
- si nous supposons c^+ non f_clos, alors matérialiser c à la place de c^+ ne serait pas légitime puisqu'on ne gagne pas en espace mémoire et il est possible qu'il existe un cuboïde v tel que $v^+ = c^+$ donc il faudrait matérialiser v également et dans ce cas, S' ne serait pas f_optimale. Afin d'éviter des tests supplémentaires et d'optimiser le calcul de la solution, il est donc plus judicieux de considérer la matérialisation de c^+.

Toute solution f_optimale est donc composée uniquement de cuboïdes clos, nous avons bien $S \subseteq S^*$. □

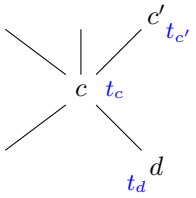
Pour trouver une solution 1_optimale, il suffit de matérialiser tous les cuboïdes 1_clos. Il est donc naturel de penser que pour calculer une solution f_optimale, nous devons matérialiser l'ensemble de tous les cuboïdes f_clos.

Proposition 4. *Soit S une solution f_optimale, alors $F_f \subseteq S$.*

Preuve. D'après la proposition 3, on sait qu'il est suffisant de ne s'intéresser qu'aux cuboïdes de la solution 1_optimale pour savoir s'ils appartiendront ou non à la solution f_optimale.
On cherche une solution S dont le coût d'exécution est au plus f fois supérieur à celui de la solution 1_optimale S^*.
Si c n'est pas f_clos et $c \in S^*$, alors, au moins un de ses ancêtres c' a une taille moins de f fois supérieur à la sienne et on respecte bien la contrainte f en utilisant c' à la place de c. Si $c' \notin S^*$ alors cela signifie qu'une requête posée sur c' peut être calculée sur un de ses ancêtres dans S^* sans perte de performance. c ne sera donc pas dans S si, pour tous ses descendants, on garde, par transitivité, un facteur de performance inférieur ou égal à f. Si ce n'était pas le cas, on conserve c dans la solution. Si $c' \in S^*$ alors on réitère le même raisonnement.
Si c est f_clos et $c \in S^*$ alors on ne peut pas calculer une requête posée sur c à partir d'un de ses ancêtres sans dépasser le facteur de performance f fixé donc $c \in S$. □

On voit donc bien qu'une solution f_optimale contient tous les cuboïdes f_fermés, les autres cuboïdes de la solution étant des cuboïdes 1_fermés.

La figure suivante illustre la preuve ci-dessus : On a d, c et c' des cuboïdes clos (i.e. $d, c, c' \in S^*$) avec $t_c \leq f \times t_d$ et $t_{c'} \leq f \times t_c$.



Supposons que tous les descendants de d ont la même taille que celui-ci. Puisque $c \in d_f^+$ alors il n'est plus nécessaire de conserver d dans S dans la mesure où c est également dans la f_fermeture de tous les descendants de d. Appliquons maintenant le même raisonnement sur c : on sait que $c' \in c_f^+$ donc il semble naturel, de la même façon que précédemment, de ne pas matérialiser c mais plutot c'. En revanche, nous devons cette fois prendre garde à ne pas dépasser la contrainte de performance fixée si l'on pose une requête sur d. Elle serait donc évaluée sur c' (premier ancêtre de d appartenant à S) et, par transitivité, $\frac{t_{c'}}{t_d}$ pourrait être supérieur à f. Nous ne pourrons enlever c de la solution f_optimale, uniquement si $c' \in d_f^+$. Sinon, pour la même raison que plus haut, il est plus judicieux de matérialiser c et ainsi la contrainte de performance fixée est bien respectée.

Algorithm 3 Algorithme glouton de solution f_optimale

```
1:  S ← S*
2:  for all c ∈ S* do
3:    for all c' ∈ Q ancêtre de c do
4:      if |c'| ≤ |c| * f then
5:        for all d ∈ S* et d ∉ S descendant de c do
6:          if |c'|/|d| ≤ f then
7:            S ← S \ c
8:          end if
9:        end for
10:     end if
11:   end for
12: end for
13: retourner S
```

Exemple 8. *Fixons le facteur de performance $f = 2$ et appliquons l'algorithme 3 sur le cube de données défini dans les exemples précédents. Pour une solution 1_optimale, on matérialise les cuboïdes $ABCD$, BCD, BC, CD et C. Voyons alors s'il est nécessaire de tous les conserver pour la solution 2_optimale.*

- *$|BC| = |CD| \leq 2 \times |C|$ donc il n'est plus nécessaire de conserver le cuboïde C.*

- $|BCD| \leq 2 \times |BC| = 2 \times |CD|$, *les cuboïdes* BC *et* CD *ne sont donc pas f_fermés et il ne semble plus indispensable de les matérialiser. Or* $BCD \notin C_2^+$ *donc nous devons stocker l'un ou l'autre des ancêtre de* C*. Etant donné qu'ils ont la même taille, nous ne pouvons choisir le plus petit mais prenons par exemple* BC*.*
- $|ABCD| \leq 2 \times |BCD|$ *et* $ABCD \in CD_2^+$ *donc il n'est pas optimal de matérialiser* BCD *ni* CD*.*

Nous avons donc sélectionné pour la solution S_2 *2_optimale les cuboïdes* BC *et* $ABCD$*, ce qui nous donne* $size(S_2) = 4 + 2 = 6$*.*

En suivant le même raisonnement pour une solution S_4 *4_optimale, on trouve* $S_4 = ABCD$ *et* $size(S_4) = 4$*. On voit donc que, plus le facteur de performance augmente, plus la taille de la solution diminue.*

4 Travaux relatifs

Le problème que l'on cherche à résoudre dans la plupart des articles est le suivant : étant donnés un cube de données C, un ensemble de requêtes $\mathcal{Q}$ et un espace mémoire de taille b, trouver l'ensemble des vues S à matérialiser en respectant la contrainte b pour répondre à toutes les requêtes de $\mathcal{Q}$ dans un temps minimum. On parle donc ici de solution optimale lorsque celle-ci a une taille inférieure à b et qu'aucune autre sélection de cuboïdes occupant le même espace mémoire n'a un coût total inférieur à celui de cette solution.

Li et al. (2005) ont utilisé des techniques de programmation linéaire pour résoudre exactement ce problème à l'aide de solveurs tels que CPLEX. Ce procédé permet donc, dans un temps non borné, de trouver la solution optimale ce qui permet alors de comparer entre elles les autres méthodes.

Un des premiers algorithmes proposant une solution approchée est celui de Harinarayan et al. (1996). Leur solution garantit un gain d'au moins 63% par rapport la solution optimale. Le gain d'une solution est la différence entre le coût des requêtes lorsque seul le cuboïde de base est stocké moins le coût des requêtes lorsqu'un sous ensemble S est stocké. Cependant, Karloff et Mihail (1999) ont montré que la garantie sur le bénéfice ne donne aucune certitude sur le coût total de la solution.

Une autre approche développée par Kotidis et Roussopoulos (1999) consiste à matérialiser les vues les plus pertinentes à un instant donné. On répond ainsi mieux au besoin d'analyse pour l'aide à la décision puisque le système s'adapte, en temps réel aux requêtes mais, de ce fait, si ce sont toujours les mêmes n requêtes qui sont posées en suivant un cycle régulier et que $\sum_{i=1}^{n} q_i > b$ alors le système ne se trouvera jamais dans un état "stable". Or nous devons considérer le temps de mise à jour de la sélection et c'est donc pour cette méthode un inconvénient.

L'article de Hanusse et al. (2009a) s'intéresse au même problème que le notre. Leur solution cependant ne permet d'obtenir qu'une approximation de la solution f_optimale dans le sens où elle peut être f fois plus volumineuse.

5 Conclusion et travaux futurs

Nous poursuivons les travaux de Hanusse et al. (2009b) sur la matérialisation partielle des cubes de données en considérant comme contrainte, non pas l'espace mémoire disponible pour la matérialisation comme c'est le cas dans la plupart des autres travaux, mais une contrainte sur la performance avec laquelle l'utilisateur veut que ses requêtes soient évaluées.

Pour résoudre, en partie ce problème, nous avons établi quelques liens avec les dépendances fonctionnelles qu'elles soient exactes ou approximatives. Pour ces dernières, bien que des travaux assez anciens les aient déjà traitées (voir par exemple Mannila et Räihä (1994); Kivinen et Mannila (1995)), nos recherches bibliographiques ne nous ont pas permis de trouver des travaux qui utilisent la définition de dépendance approximative que nous avons proposée. En effet, la plupart des références utilisent la mesure g_3 introduite dans Kivinen et Mannila (1995) qui désigne le rapport entre le nombre minimum de tuples à supprimer d'une table et le nombre de tuples de celles-ci afin de rendre une dépendance exacte. Nous n'avons pas utilisé cette définition mais nous comptons analyser dans nos futurs travaux les liens entre notre facteur d'approximation et ceux qu'on rencontre dans la littérature.

Nous avons démontré que le problème de sélection des cuboïdes à matérialiser était équivalent au problème de recherche des dépendances fonctionnelles. Ces dernières pouvant être en nombre exponentiel en fonction du nombre d'attributs, ceci montre que notre problème est aussi exponentiel en nombre d'attributs. C'est la complexité des algorithmes que nous avons proposés. Ceci ne nous interdit pas de chercher dans le futur des algorithmes polynomiaux en fonction, non pas du nombre d'attributs, mais du nombre de dépendances. Ceci est un autre axe que nous comptons explorer. En effet, dans les cubes de données, notamment dans les schémas en étoile, le concepteur connaît les dépendances existant entre les niveaux de chaque hiérarchie. Ainsi, les 1_fermés peuvent être calculés d'une manière statique.

Les requêtes que nous avons considérées sont trop restreintes pour que nos solutions soient envisageables en pratique. En effet, dans le but d'intégrer des requêtes avec conditions de restriction (clause `WHERE`), nous envisageons d'étudier les dépendances conditionnelles Diallo et Novelli (2010); Fan et al. (2011) qui ont été utilisées dans Bra et Paredaens (1983) pour définir des décompositions horizontales de tables relationnelles. Ceci s'apparente aux travaux sur le partitionnement proposées par exemple dans Bellatreche et al. (2004). Nous notons néanmoins le travail de Ciaccia et al. (2003) qui ont utilisé les dépendances de cardinalité, un cas particulier des dépendances que nous avons considérées, pour estimer la taille des cuboïdes.

Un autre axe de recherche pour nos travaux futurs serait d'adapter notre méthode dans le cas où nous avons plusieurs bases de données à interroger. Ce sujet a déjà été étudié dans l'article de Bauer et Lehner (2003). Mais les auteurs ne traitent pas des questions relatives à la complétude des réponses à apporter. En effet, deux tables T et T' sur deux entrepôts différents et ayant les mêmes attributs doivent être considérées toutes les deux pour répondre à une seule requête et assurer que la réponse contient bien toutes les données présentes sur l'ensemble des entrepôts à notre disposition. Pour palier ce problème, Shukla et al. (2000) proposent de faire la jointure entre les tables de faits (c_b) de tous les cubes et d'appliquer un algorithme de sélection des vues sur le cube résultat. Il reste maintenant à savoir où stocker les cuboïdes résultant de jointures depuis plusieurs entrepôts. On souhaite donc avoir la possibilité d'interroger plusieurs cuboïdes distants pour répondre à une seule requête.

Références

Bauer, A. et W. Lehner (2003). On solving the view selection problem in distributed data warehouse architectures. In *Proceedings of SSDBM conference*.

Bellatreche, L., M. Schneider, H. Lorinquer, et M. K. Mohania (2004). Bringing together partitioning, materialized views and indexes to optimize performance of relational data warehouses. In *Proceedings of DaWaK conference*.

Bra, P. D. et J. Paredaens (1983). Conditional dependencies for horizontal decompositions. In *Proceedings of ICALP conference*, pp. 67–82.

Chaudhuri, S. (1998). An overview of query optimization in relational systems. In *Proceedings of PODS conference*, pp. 34–43.

Ciaccia, P., M. Golfarelli, et S. Rizzi (2003). Bounding the cardinality of aggregate views through domain-derived constraints. *Data and Knowledge Engineering 45*(2), 131–153.

Diallo, T. et N. Novelli (2010). Découverte des dépendances fonctionnelles conditionnelles fréquentes. In *Actes de la conférence EGC*, pp. 315–326.

Fan, W., F. Geerts, J. Li, et M. Xiong (2011). Discovering conditional functional dependencies. *To appear in TKDE*.

Giannella, C. et E. L. Robertson (2004). On approximation measures for functional dependencies. *Inf. Syst. 29*(6), 483–507.

Graefe, G. (1993). Query evaluation techniques for large databases. *ACM Computing Surveys 25*(2), 73–170.

Gray, J., S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, et H. Pirahesh (1997). Data cube : A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Min. Knowl. Discov. 1*(1), 29–53.

Hanusse, N., S. Maabout, et R. Tofan (2009a). Algorithmes pour la sélection des vues à matérialiser avec garantie de performance. In *Entrepôts de Données et l'Analyse en ligne*, pp. 107–122.

Hanusse, N., S. Maabout, et R. Tofan (2009b). A view selection algorithm with performance guarantee. In *Proceedings of EDBT conference*, pp. 946–957.

Harinarayan, V., A. Rajaraman, et J. Ullman (1996). Implementing data cubes efficiently. In *Proceedings of ACM SIGMOD*, pp. 205–216.

Karloff, H. et M. Mihail (1999). On the complexity of the view-selection problem. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 167–173. ACM.

Kivinen, J. et H. Mannila (1995). Approximate inference of functional dependencies from relations. *Theoretical Computer Science 149*(1), 129–149.

Kotidis, Y. et N. Roussopoulos (1999). DynaMat : a dynamic view management system for data warehouses. *ACM SIGMOD Record 28*(2), 371–382.

Li, J., Z. Talebi, R. Chirkova, et Y. Fathi (2005). A formal model for the problem of view selection for aggregate queries. In *Advances in Databases and Information Systems*, pp. 125–138. Springer.

Mannila, H. et K.-J. Räihä (1994). Algorithms for inferring functional dependencies from relations. *Data and Knowledge Engineering 12*(1), 83–99.

Shukla, A., P. Deshpande, et J. Naughton (2000). Materialized view selection for multi-cube data models. In *Proceedings of EDBT conference*, pp. 269–284. Springer.

Summary

Selecting the views to materialize is often required for very large data warehouses. In this work, we show that there exists a very tight relationship between searching the views to materialize in a data cube in order to optimize query processing and the functional dependencies present between its dimensions. In contrast to most related works, our constraint is not a storage space budget fixed by the user but a performance factor f by which the queries are evaluated. Hence, our solutions respect this constraint while they minimize the storage space required. We formally characterize all optimal solutions (in terms of memory space) satisfying this criterion by resorting to properties of functional dependencies. We prove that the problem we investigate is NP-hard and we present greedy algorithms which are efficient with respect to the performance constraint set by the user.