

**Rapport de recherche
N° 01 -2008**

**Fragmentation Primaire et Dérivée: Étude de
Complexité, Algorithmes de Sélection et Validation
sous ORACLE10g**

Kamel BOUKHALFA, Ladjel BELLATRECHE, Pascal RICHARD

Fragmentation Primaire et Dérivée: Étude de Complexité, Algorithmes de Sélection et Validation sous ORACLE10g

Kamel Boukhalfa *, Ladjel Bellatreche* Pascal Richard *

* LISI/ENSMA - Université de Poitiers
Futuroscope 86960 France
(boukhalk, bellatreche, richardp)@ensma.fr

Résumé. La fragmentation horizontale a été largement adoptée par la communauté des bases de données. Elle a une place à part entière dans la conception physique. Plusieurs systèmes de gestion de bases de données (SGBD) commerciaux ont proposé un langage de définition de données pour partitionner des tables relationnelles en utilisant différents modes. Dans ce papier, nous présentons d'abord l'évolution de la fragmentation ces dernières années au sein des SGBDs. Deuxièmement, nous étudions le problème de sélection de schéma de fragmentation d'un entrepôt de données relationnel, et nous montrons qu'il est NP-complet. Vu sa complexité, nous développons un algorithme de hill climbing (méthode de voisinage) pour sélectionner un schéma de fragmentation quasi optimale. Nous effectuons des expérimentations afin de comparer cet algorithme avec deux autres algorithmes: un génétique et un recuit simulé en utilisant un modèle de coût mathématique. Finalement, nous effectuons une validation réelle de nos algorithmes sous ORACLE10g en utilisant les données issues du banc d'essai APB1.

1 Introduction

Les entrepôts de données sont connus par leur volumétrie et requêtes complexes caractérisées par des jointures, sélections et agrégations. Pour optimiser ces opérations et faciliter la gestion des ces données, la fragmentation horizontale est devenue un candidat sérieux (Papadomanolakis et Ailamaki, 2004; Sanjay et al., 2004). Elle permet de partitionner les tables, les vues matérialisées ou les index en plusieurs ensembles disjoints de tuples stockés physiquement et généralement accédés séparément. Une caractéristique intéressante liée à la fragmentation horizontale est le fait qu'elle ne duplique pas les données, et par conséquent, elle réduit les coûts de stockage et de maintenance.

Deux versions de la fragmentation horizontale ont été définies (Özsu et Valduriez, 1999) : primaire et dérivée. La fragmentation horizontale primaire d'une relation est effectuée en utilisant les attributs définis sur cette table. La fragmentation horizontale dérivée consiste à partitionner une table en utilisant les attributs définis sur une ou plusieurs autres(s) table(s). En d'autres termes, la fragmentation horizontale dérivée d'une table est basée sur le schéma de

fragmentation des autres tables ¹. La fragmentation horizontale dérivée d'une table R en fonction du schéma de fragmentation de S est possible si, et seulement si, il existe un lien de jointure entre R et S (R contient une clé étrangère de S). A partir de ces deux définitions, nous constatons que la fragmentation primaire pourrait accélérer les opérations de sélection tandis que la fragmentation dérivée les opérations de jointure.

Cette optimisation est assurée par le mécanisme d'élimination de partitions non pertinentes : si une requête contient un attribut de partition dans la clause WHERE, l'optimiseur dirige automatiquement la requête vers les partitions valides : si nous fragmentons une table CLIENT en utilisant l'attribut SEXE et si une requête possède une condition sur cet attribut, l'optimiseur ne charge que la partition pertinente.

Le partitionnement de données a été largement étudié dans les bases de données traditionnelles (Bellatreche et al., 2000) et les bases de données distribuées et parallèles (Navathe et al., 1995; Özsu et Valduriez, 1999). Récemment, plusieurs travaux commerciaux et académiques ont montré l'intérêt de la fragmentation horizontale (Sanjay et al., 2004; Papadomanolakis et Ailamaki, 2004; Corp., 2007). Plusieurs SGBDs commerciaux ont proposé des langages de définitions de données pour supporter la fragmentation. Pour étudier cette évolution, nous nous concentrons sur le SGBD ORACLE qui offre plusieurs modes de partitionnement.

Le premier mode de partitionnement supporté par Oracle a été le partitionnement par intervalle (Range) dans Oracle8i. Il est défini par un tuple (c, V) , où c est un type de colonne et V une séquence ordonnée de valeurs dans le domaine de c . Oracle9 et 9i ont ajouté d'autres modes de fragmentation qui sont : le partitionnement par hachage (Hash) le partitionnement par liste (List) et le partitionnement composé (Range-Hash et Range-List). Le partitionnement par hachage décompose la table selon une fonction de hachage (fournie par le système) appliquée sur les valeurs des attributs de fragmentation. Le partitionnement par liste, décompose une table selon les listes de valeurs d'une colonne. Le partitionnement composé est utilisé à l'aide des instructions PARTITION-SUBPARTITION ². Les partitionnements Range, List et Hash sont des modes de base supportés par la plupart des SGBDs commerciaux.

Récemment, Oracle 11g fait évoluer la fragmentation horizontale en proposant plusieurs modes : (1) Partitionnement par une colonne virtuelle (virtual column partitioning) dans lequel, une table est fragmentée en utilisant un attribut virtuel, qui est défini par une expression utilisant un ou plusieurs attributs. Cette colonne est stockée seulement dans les méta-données. (2) Le partitionnement par référence (referential partitioning) : qui permet de fragmenter une table en utilisant une autre table (à condition il y a une relation de type père-fils entre les deux tables (Corp., 2007)). Ce partitionnement est similaire à la fragmentation dérivée, mais son inconvénient majeur est qu'une table est fragmentée en fonction d'une seule autre table. Dans les entrepôts de données, une table des faits doit être fragmentée en utilisant les schémas de fragmentation de plusieurs tables de dimension pour garantir une optimisation des requêtes complexes. (3) Toutes les combinaisons de modes de base c'est-à-dire, Range, List et Hash sont possibles : Range-Range, List-List, Hash-Hash, List-Hash, etc. Malheureusement, une table donnée ne pourra pas être fragmentée selon la combinaison de trois modes de base. Pour une réelle utilisation de la fragmentation dans les bases de données, ces modes doivent être supportés ou implémentés par un administrateur.

¹Un schéma de fragmentation est le résultat du processus de fragmentation d'une table donnée

²Ces modes de partitionnement sont aussi supportés par les autres SGBDs commerciaux comme SQL Server, Sybase, DB2, etc.

Cette évolution rapide de la fragmentation horizontale nous a motivée pour l'étudier en détail. En explorant les travaux académiques et industriels les plus importants sur la manière de sélectionner des schéma de fragmentation horizontale, nous constatons que cette sélection suppose une décomposition du domaine des valeurs d'attributs participant au processus de fragmentation³. Cette décomposition peut être réalisée de deux manières : (1) une décomposition orientée utilisateur et (2) une décomposition orientée requêtes. Dans la première catégorie, l'administrateur décompose le domaine de valeurs de chaque attribut de fragmentation en se basant sur ses connaissances des applications (requêtes) et *impose à priori* le nombre de fragments horizontaux générés. Les principaux inconvénients de ce mode de partitionnement sont (i) l'absence d'une métrique garantissant l'efficacité du schéma de fragmentation obtenu et (ii) la manière de décomposer chaque domaine en plusieurs sous domaines.

Dans le partitionnement orienté requêtes, les domaines des valeurs des attributs de fragmentation sont décomposés en se basant sur les requêtes définies sur le schéma de la base de données. Dans ce mode, plusieurs algorithmes ont été proposés dans les bases de données traditionnelles, que nous pouvons classer en trois principales approches : approches basées sur les prédicats (Özsu et Valduriez, 1999; Ceri et al., 1982), approches basées sur l'affinité (Navathe et al., 1995) et approches basées sur un modèle de coût (Bellatreche et al., 2000). Le principal inconvénient de cette catégorie, est que l'administrateur n'a aucun contrôle sur le nombre de fragments générés.

Dans ce papier, nous montrons la NP-complétude du problème de sélection de schéma de fragmentation avec une contrainte représentant le nombre de fragments que l'administrateur souhaite avoir. Puis, nous proposons un algorithme de type hill climbing que nous comparons ensuite avec deux algorithmes de sélection de schéma de fragmentation. Les schémas générés sont ensuite validés sous Oracle10g avec les données du banc d'essai APB-1 Council (1998). Pour réaliser cette expérimentation, nous avons implémenté la fragmentation horizontale dérivée puisqu'elle n'est pas bien nativement supportée par Oracle10g, ainsi que la fragmentation combinant trois ou plusieurs modes.

Ce papier est divisé en 6 sections. Dans la section 2, nous formalisons le problème de sélection d'un schéma de fragmentation horizontale et proposons une preuve de sa NP-Complétude. La section 3 présente l'algorithme de Hill Climbing basé sur un mécanisme de codage permettant d'éviter les solutions redondantes ainsi que le modèle de coût mathématique que nous avons utilisé pour estimer le coût d'exécution des requêtes. La section 4 présente une étude expérimentale afin de comparer trois algorithmes, à savoir le hill climbing, le recuit simulé et le génétique, tous basés sur le même codage. La Section 5 présente une validation de notre travail sur ORACLE10G. La section 6 conclut le papier en résumant le travail effectué et en citant quelques perspectives.

2 Étude de complexité

Dans le contexte des entrepôts de données relationnels, nous avons proposé dans (Bellatreche et Boukhalfa, 2005), une méthodologie pour la fragmentation des différentes tables composant un schéma en étoile (tables de dimension et table de faits) : (1) fragmenter quelques/toutes les tables de dimension en utilisant la fragmentation horizontale primaire, et (2) fragmenter la

³La décomposition des domaines de valeurs des attributs d'une table implique une fragmentation horizontale de cette table.

table des faits en utilisant les schémas de fragmentation des tables de dimension, fragmentées à première étape. Cette méthodologie peut générer un nombre important de fragments de la table des faits (noté par N) $N = \prod_{i=1}^g m_i$ où m_i et g sont le nombre de fragments de la table de dimension D_i et le nombre de tables de dimension qui ont participées dans le processus de fragmentation, respectivement.

Pour éviter l'explosion de ce nombre, nous formalisons le problème de sélection d'un schéma de fragmentation horizontale comme un problème d'optimisation sous contrainte : étant donné un ensemble de requêtes représentative Q définies sur un entrepôt de données composé d'une table de faits F et n tables de dimension $\{D_1, D_2, \dots, D_n\}$, et une contrainte de maintenance W représentant le nombre de fragments de faits que l'administrateur souhaite avoir, fragmenter la table des faits F en N fragments tel que $(\sum_{q_j \in Q} freq_{q_j} \times Cost(q_j))$ ($freq_{q_j}$ représente la fréquence d'accès de la requête q_j) soit minimisé et que la contrainte ($N \leq W$) soit respectée.

2.1 Complexité du problème de sélection d'un schéma de fragmentation horizontale

Pour étudier la complexité du problème de sélection d'un schéma de fragmentation horizontale nous considérons un problème de décision simplifié qui prend en considération un seul domaine d'un attribut d'une table de dimension pour partitionner la table des faits. Nous appelons ce problème, *Problème de Fragmentation Horizontale à un Seul Domaine (PFHSD)*. Le problème d'optimisation correspondant consiste à partitionner la table des faits telle que le nombre de partitions de cette table soit borné par une constante et le nombre d'opérations d'entrées/sorties (E/S) nécessaires soit minimisé. Nous présentons ce problème d'optimisation comme suit :

Problème : Fragmentation Horizontale à un Seul Domaine

- **Instance :**
 - un ensemble D de sous domaines disjoints $\{d_1, \dots, d_n\}$ d'un attribut d'une table de dimension et le nombre d'E/S nécessaires pour lire les données correspondant au sous domaine d_i dans la table des faits, noté $l(d_i)$, $1 \leq i \leq n$.
 - un ensemble de requêtes $\{q_1, \dots, q_m\}$ et pour chaque requête q_j la liste $f(q_j) \subseteq D$ des sous domaines utilisés pour l'exécution de la requête : $\{d_{j1}, \dots, d_{jn_j}\}$, où n_j est le nombre de sous domaines utilisés pour exécuter q_j .
 - deux entiers positifs K et L , où K est le nombre maximum de partitions qui peuvent être créés et L est le nombre maximum d'E/S nécessaires pour chaque requête, $L \geq \sum_{d \in f(q)} l(d)$.
- **Question :** Est-ce que D peut être partitionné au maximum en K sous domaines, D_1, \dots, D_K tels que chaque requête nécessite au plus L opérations d'entrées sorties.

Le nombre optimal d'opération d'E/S nécessaires pour une requête q_j est : $l(q_j) = \sum_{d \in f(q_j)} l(d)$. Nous partons du principe que seules les données nécessaires sont chargées en mémoire pour exécuter q_j . Selon une partition donnée, le nombre des opérations d'E/S augmente puisque toutes les données d'une partition sont chargées lorsqu'elles sont utilisées par une requête donnée, même si cette requête ne nécessite pas toutes les données de la partition (c'est-à-dire un sous-ensemble des domaines de la partition).

Ainsi, le nombre des opérations d'entrées/sorties requises par une requête après fragmentation ne dépend pas des sous domaines utilisés, mais uniquement des partitions utilisées. Le nombre d'E/S pour le chargement d'une partition D_i est défini par : $l(D_i) = \sum_{d \in D_i} l(d)$. En conséquence, le nombre d'E/S nécessaires à l'exécution d'une requête peut être défini comme : $l(q) = \sum_{D \in F(q)} l(D)$, où $F(q)$ est la liste des partitions utilisées par une requête q .

L'objectif est de réaliser un partitionnement horizontal de la table de faits tel que le nombre de partitions est limité à K et le nombre d'E/S est limité par L pour chaque requête. Il est évident que si $K \geq n$, le partitionnement horizontal optimal est obtenu en définissant exactement une partition pour chaque $d_i \in D$. De cette manière, toutes les requêtes ne chargent que les données requises pendant leur exécution. Nous verrons que notre problème de décision simplifié devient difficile lorsque $K < n$. Nous supposons aussi que $L \geq \sum_{d \in f(q)} l(d)$ car sinon il n'existe pas de partitionnement d'un domaine respectant cette contrainte.

2.2 NP-Complétude du problème de fragmentation horizontale dérivée

Le problème de partitionnement horizontal a été montré NP-Complet dans Sacca et Wiederhold (1985). Toutefois, l'architecture considérée est un cluster de processeurs et la transformation utilisée repose sur des relations avec exactement une ligne et une colonne, donc pour lesquelles aucune fragmentation supplémentaire n'est possible (voir preuve dans Sacca et Wiederhold, 1985, p.37). Dans notre cas, l'architecture est centralisée et nous allons montrer que dans ce cas la fragmentation horizontale est un problème NP-Complet *au sens fort*.

Théorème 1 *Le problème de fragmentation horizontale à un seul domaine est NP-Complet au sens fort.*

Preuve 1 *Le PFHSD appartient clairement à NP car si on effectue un partitionnement de D , alors un algorithme polynomial peut vérifier qu'au plus K partitions sont utilisées et que chaque requête nécessite au plus L opérations d'E/S. Nous prouvons maintenant que le problème PFHSD est NP-Complet au sens fort. Nous utilisons le problème 3-Partition qui est NP-Complet au sens fort (Garey et Johnson, 1990) et se définit comme suit :*

Problème : 3-Partition

- **Instance :** Un ensemble A de $3m$ éléments, une borne $B \in \mathbb{Z}^+$, et une taille $s(a) \in \mathbb{Z}^+$ pour chaque $a \in A$ tel que $B/4 < s(a) < B/2$ et que $\sum_{a \in A} s(a) = mB$.
- **Question :** Est ce que A peut être partitionné en m ensembles disjoints A_1, \dots, A_m tels que, pour $1 \leq i \leq m$, $\sum_{a \in A_i} s(a) = B$? (notons que chaque A_i doit obligatoirement contenir trois éléments de A)

Pour prouver la NP-Complétude du problème PFHSD, nous réduisons à partir du problème 3-Partition. Pour chaque instance du problème 3-Partition, une instance du problème PFHSD est définie comme suit :

- pour chaque $a_i \in A$, un sous domaine d_i est créé de sorte que $l(d_i) = s(a_i)$, $1 \leq i \leq 3m$.
- $3m$ requêtes sont créées telles que chaque requête utilise exactement un sous domaine : $f(q_i) = \{d_i\}$, $1 \leq i \leq 3m$.
- $K = L = B$

Il est clair que la transformation est effectuée en un temps polynomial car elle consiste à une correspondance un à un des éléments de 3-partition, les sous domaines et les requêtes. Nous

prouvons maintenant que nous trouvons une solution à une instance du problème 3-partition, si et seulement si, nous trouvons une solution à une instance du problème PFHSD.

(*Condition nécessaire*) Supposons que nous avons une solution au problème PFHSD, alors elle satisfait les conditions suivantes :

- puisque $B/4 < l(d) < B/2$, chaque sous ensemble de D doit être défini avec exactement 3 sous domaines (comme dans chaque instance 3-partition).
- Puisque nous avons une solution faisable du problème PFHSD, alors aucune requête ne nécessite plus de B opérations d'E/S. Par construction nous vérifions que : $\sum_{d \in D} l(d) = mB$. Par conséquent, chaque requête nécessite exactement B E/S dans la table des faits (sinon, ce n'est pas une solution). En utilisant une correspondance un à un des sous domaines en éléments de 3-Partition, une solution faisable à l'instance 3-partition est obtenue.

(*Condition suffisante*) Supposons que nous avons une solution à une instance 3-Partition. Alors, chaque sous ensemble A_i a une taille totale de B et il est composé de exactement 3 éléments de A . Commenant par A_1 , nous définissons une partition de sous-domaines en utilisant les mêmes indices des éléments appartenant à A_1 . Puisque chaque requête est associée à exactement un sous-domaine et que trois sous domaines sont groupés dans chaque partition, alors trois requêtes utilisent une partition donnée. Par conséquent, le nombre d'E/S associées à ces trois requêtes est exactement B . En répétant ce processus pour chaque sous ensemble restant A_i , alors une solution faisable du problème de fragmentation à un seul domaine est obtenue.

Notre problème de fragmentation horizontale est au moins aussi complexe à résoudre que notre problème simplifié car plusieurs domaines sont considérés et cela pour plusieurs tables de dimension. Une instance de notre problème de fragmentation est une combinaison de plusieurs instances du problème PFHSD, par conséquent il est NP-Complet au sens fort.

3 Algorithmes de Sélection de Schéma de Fragmentation

En raison de la complexité du problème de fragmentation horizontale, le développement des heuristiques pour la sélection d'une solution satisfaisante est recommandé. Donc, dans cette section nous présentons un algorithme de Hill Climbing. Avant de détailler cet algorithme, nous présentons un codage pour représenter un schéma de fragmentation sur lequel des opérations seront définies. Nous présentons par la suite le modèle de coût mathématique que nous avons utilisé pour évaluer le coût d'exécution des requêtes sur un schéma fragmenté ou non.

3.1 Codage d'un schéma de fragmentation

Rappelons que la fragmentation horizontale est effectuée sur les attributs de prédicats de sélection définis généralement sur les tables de dimension. Chaque attribut a un domaine de valeur. Chaque prédicat de sélection est défini par : $A \theta Valeur$ tel que A est un attribut d'une table de dimension, $\theta \in \{=, <, >, \leq, \geq\}$, et $Valeur \in Domaine(A)$ (Ceri et al., 1982). Tout attribut participant dans le processus de partitionnement est appelé un *attribut de fragmentation*.

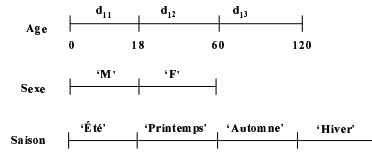


FIG. 1 – Exemple de sous domaines

La fragmentation horizontale permet d’abord de partitionner les domaines de valeurs des attributs de fragmentation. Pour illustrer cela, supposons que les domaines des attributs Age et Sexe de la table de dimension CLIENT et Saison de la table TEMPS sont :

$Dom(Age) =]0, 120]$, $Dom(Sexe) = \{ 'M', 'F' \}$ et $Dom(Saison) = \{ \ll Printemps \gg, \ll Été \gg, \ll Automne \gg, \ll Hiver \gg \}$. Nous supposons que l’administrateur décompose les domaines de ces attributs en sous domaines comme suit : $Dom(Age) = d_{11} \cup d_{12} \cup d_{13}$, avec $d_{11} =]0, 18]$, $d_{12} =]18, 60[$, $d_{13} = [60, 120]$, $Dom(Sexe) = d_{21} \cup d_{22}$, avec $d_{21} = \{ 'M' \}$, $d_{22} = \{ 'F' \}$ et $Dom(Saison) = d_{31} \cup d_{32} \cup d_{33} \cup d_{34}$, avec $d_{31} = \{ \ll Printemps \gg \}$ et $d_{32} = \{ \ll Été \gg \}$, $d_{33} = \{ \ll Automne \gg \}$ et $d_{34} = \{ \ll Hiver \gg \}$. Les différents sous domaines des trois attributs de fragmentation sont représentés sur la figure 1.

Le partitionnement du domaine de chaque attribut peut être représenté par un tableau multidimensionnel, où chaque ligne représente le partitionnement du domaine de l’attribut de fragmentation. La valeur de chaque cellule d’un tableau donné représentant un attribut A_i appartient à l’intervalle $[1..n_i]$, où n_i représente le nombre de sous domaines de l’attribut A_i . En se basant sur cette représentation, le schéma de fragmentation de chaque table est généré comme suit : (1) si toutes les cellules d’un attribut donné ont des valeurs différentes alors tous les sous domaines sont considérés pour fragmenter la table de dimension correspondante, (2) Si toutes les cellules d’un attribut donné ont la même valeur cela signifie que cet attribut ne participe pas au processus de fragmentation, (3) Si certaines cellules d’un attribut ont la même valeur alors leurs sous domaines correspondants sont fusionnés en un seul.

La figure 2 montre un exemple de codage d’un schéma de fragmentation basé sur trois attributs de fragmentation, Sexe, Saison et Age. Donc la table CLIENT sera fragmentée en utilisant les attributs Age et Sexe et la table TEMPS en utilisant l’attribut Saison. Pour matérialiser cette fragmentation, l’administrateur utilise l’instruction suivante pour créer les tables CLIENTS et TEMPS fragmentées.

```
CREATE TABLE CLIENT
(CID NUMBER, Nom Varchar2(20), Sexe CHAR, Age Number)
PARTITION BY RANGE (Age)
SUBPARTITION BY LIST (SEXE)
SUBPARTITION TEMPLATE (SUBPARTITION Female VALUES ('F'),
SUBPARTITION Male VALUES ('M'))
(PARTITION Cust_0_60 VALUES LESS THAN (61),
PARTITION Cust_60_120 VALUES LESS THAN (MAXVALUE));

CREATE TABLE TEMPS
(TID NUMBER, Saison VARCHAR2(10), Année Number)
PARTITION BY LIST(Saison)
(PARTITION Time_Summer VALUES('Summer'),
PARTITION Time_Spring VALUES ('Spring'),
PARTITION Time_Autumn_Winter VALUES('Automn', 'Winter'));
```


Sexe	1	2		
Saison	1	2	3	3
Age	1	1	2	

FIG. 2 – Exemple de codage d'un schéma de fragmentation

Puisque les tables CLIENT et TEMPS ont été fragmentées en 3 et 4 fragments respectivement, la table des faits sera fragmentée en 12 partitions.

3.2 Partitionner un ensemble et les fonctions à croissance restreintes

Notre codage souffre d'un problème de multi-instanciation. Pour illustrer ce problème, nous considérons un ensemble $D = \{d_1, d_2, d_3\}$ alors chaque ensemble de partitions de D, par exemple $\{\{d_1, d_3\}, \{d_2\}\}$ peut être représenté par un tableau d'entiers. Néanmoins, un même partitionnement peut être représenté par différents tableaux d'entiers comme dans l'exemple présenté table 1.

Sous Domaines	d_1	d_2	d_3
Tableau 1	0	1	0
Tableau 2	1	0	1

TAB. 1 – Valeurs de paramètres de l'AG

Il est clair que tableau 1 et 2 diffèrent seulement par les numéros utilisés pour représenter les sous-ensembles. Dans les deux représentations d_1 et d_3 appartiennent au même sous-ensemble et d_2 appartient au deuxième sous-ensemble. Pour résoudre ce problème, nous utilisons les fonctions à croissance restreinte (*Restricted Growth Functions* (Er, 1988; Tucker et al., 2005)).

Soit $[n]$ un ensemble $\{1, \dots, n\}$, une fonction à croissance restreinte est une fonction $f : [n] \rightarrow [n]$ tel que :

$$\begin{aligned}
 f(1) &= 0 \\
 f(i+1) &\leq \max\{f(1), \dots, f(i)\} + 1
 \end{aligned}$$

$f(i)$ définit l'indice du sous ensemble où l'élément i appartient. Par exemple, la partition $\{\{1, 3, 5\}, \{2, 6\}, \{4\}\}$ est représentée en utilisant les fonctions à croissance restreintes par le codage suivant : $[0, 1, 0, 2, 0, 1]$ où 0 est l'indice du premier sous-ensemble. Il y a une correspondance d'équivalence entre les partitions d'un ensemble et les fonctions à croissance restreinte. Dans l'exemple précédant, seul tableau 1 respecte l'ordre lexicographique introduit par le codage, tandis que tableau 2 ne sera jamais considéré durant le partitionnement de l'ensemble D .

Théorème 2 *Il y a une correspondance un à un entre l'ensemble des partitionnements et l'ensemble des fonctions à croissance restreinte.*

Plusieurs algorithmes sont connus pour générer toutes les partitions d'un ensemble D dans l'ordre lexicographique (voir Er (1988) par exemple). Les propriétés des fonctions à croissance restreinte ont été étudiées pour définir efficacement les opérations de mutation et de croisement d'un algorithme génétique dans Tucker et al. (2005).

Pour évaluer la qualité d'un schéma de fragmentation horizontale, nous avons défini un modèle de coût mathématique qui estime le nombre de pages nécessaires à charger pour exécuter un ensemble de requêtes que nous présentons dans la section suivante.

3.3 Modèle de coût

Le modèle de coût que nous avons défini permet d'estimer le nombre d'E/S nécessaires (exprimé en nombre de pages chargées) pour exécuter une requête définie sur un schéma en étoile. Nous ne prenons pas en compte le temps CPU que nous supposons négligeable par rapport au temps nécessaire pour effectuer les E/S. Étant un schéma en étoile défini par une table des faits F et d tables de dimension. Chaque requête Q défini sur cet entrepôt comportant plusieurs prédicats de sélection (définis sur des attributs de dimension) et des jointures entre la table des faits et les tables de dimensions. Le modèle de coût que nous proposons utilise des données collectées sur : (1) l'entrepôt de données (taille des tables, taille d'un tuple, nombre de pages stockant une table, etc.), (2) le système physique (taille du buffer, taille de la page système, etc.) et (3) la charge de requêtes (fréquence d'accès, sélectivité des prédicats, etc.). Le tableau 2 montre les principaux paramètres utilisés dans nos formules de coût.

Paramètre	Signification
F	la table des faits
D_i	i^{eme} table de dimension
$ R $	nombre de pages nécessaires pour stocker la table R
$\ R\ $	nombre de tuples de la table R
B	taille du buffer en nombre de pages
$\ R \bowtie S\ $	nombre de tuples du résultat de la jointure entre R et S
$Sel(R \bowtie S)$	sélectivité de la jointure entre R et S
PS	taille de la page système
TS_R	taille d'un tuple de la table R

TAB. 2 – Paramètres utilisés dans les formules de coût

Les requêtes prise en compte dans l'élaboration de notre modèle de coût sont des requêtes de jointure en étoile ayant la structure suivante :

```
SELECT [SGA], FC(AA)
FROM F, D1, D2, ... Dk
WHERE PJOIN AND PSEL
[GROUP BY GA]
```

où

- *SGA* : l'ensemble des attributs retournés dans la réponse à la requête (il peut être vide pour les requêtes retournant un calcul). Ces attributs peuvent être des attributs de dimension et/ou de faits.
- *FC* : les fonctions de calcul (MIN, MAX, COUNT, SUM, AVG).
- *AA* : l'ensemble des attributs d'agrégation, ces attributs sont généralement des mesures définis dans la table des faits.
- *k* : le nombre de tables de dimension utilisées par la requête
- *PJOIN* : un ensemble de prédicats de jointure entre la table de faits et les tables de dimension. Un prédicat de jointure (équi-jointure) a la forme suivante : $F.f_{k_i} = D_i.k_i$. où f_{k_i} , k_i représentent respectivement la clé étrangère et la clé primaire de la table de dimension D_i .
- *PSEL* : un ensemble de prédicats de sélection sous forme de conjonction, chaque conjonction est constituée de plusieurs prédicats définis sur la même table.
- *GA* : l'ensemble des attributs de groupement. Ces attributs peuvent être des attributs de faits et/ou de dimension.

Dans notre modèle de coût nous avons considéré que la jointure est implémentée par un algorithme de hachage. Cet algorithme est simple et donne de meilleures performances pour des tables volumineuses. Pour déterminer l'ordre des jointures, nous avons utilisé la méthode de sélectivité minimum. Cette méthode propose de choisir la jointure qui génère moins de résultats intermédiaires. Cela nous permettra de mieux exploiter le buffer pour garder ces résultats en mémoire.

3.3.1 Coût d'une requête sur un schéma non fragmenté

Pour exécuter une requête de jointure en étoile, nous avons retenu le plan d'exécution suivant :

Soit une requête de jointure en étoile Q respectant la syntaxe que nous avons définie précédemment et comporte k jointures. Nous supposons que l'ordre de jointure a été établi (en utilisant la méthode de sélectivité minimum) et les tables de dimension sont jointes avec l'ordre suivant : D_1, D_2, \dots, D_k . Le plan d'exécution comporte les étapes suivantes :

1. effectuer la première jointure entre F et D_1 .
2. si le résultat de la jointure tient en mémoire, le garder et passer à la jointure suivante, sinon, mettre le surplus de ce résultat sur le disque, et le recharger pour la prochaine jointure.
3. répéter 1 et 2 pour toutes les jointures présentes dans Q .
4. soit R_k le résultat issu de toutes les jointures, si R_k tient en mémoire alors nous supposons que le groupement et le calcul se font en mémoire (pas de coût supplémentaire). Si R_k ne tient pas en mémoire, mettre le surplus sur le disque et le recharger pour effectuer les groupements et/ou les calculs.

Nous voyons bien que la taille des résultats intermédiaires a un effet considérable sur le coût d'exécution, nous considérons trois cas, (1) tous les résultats intermédiaires tiennent en mémoire, (2) une partie de ces résultats tient en mémoire et (3) aucun résultat intermédiaire ne tient en mémoire.

1. *Tous les résultats intermédiaires tiennent en mémoire* : dans ce cas l'optimiseur effectue la première jointure ensuite charge les tables de dimension une à une pour effectuer une

jointure par hachage de ces dernières avec les résultats intermédiaires. Le coût peut être calculé par la formule suivante :

$$Cost = 3 \times (|F| + |D_1|) + \sum_{i=2}^k 3 \times |D_i|$$

2. *Une partie des résultats intermédiaires tiennent en mémoire* : si un résultat intermédiaire ne tient pas en mémoire, alors le surplus sera écrit sur le disque et rechargé ensuite pour effectuer la prochaine jointure. Nous supposons que les résultats intermédiaires R_1, \dots, R_j ($j < k$) ne tiennent pas en mémoire et R_{j+1}, \dots, R_k tiennent en mémoire. La formule de coût dans ce cas est la suivante :

$$Cost = 3 \times (|F| + |D_1|) + \sum_{i=2}^j [2 \times (|R_{i-1}| - B + 1) + 3 \times |D_i|] + \sum_{i=j+1}^k 3 \times |D_i|$$

3. *Tous les résultats intermédiaires ne tiennent pas en mémoire* : dans ce cas le surplus de chaque résultat est écrit sur le disque et rechargé ensuite pour effectuer la prochaine jointure. Nous considérons deux scénarii selon que la requête contient ou non des groupements. Pour les requêtes contenant que des agrégations sans groupement, le surplus du dernier résultat R_k est chargé pour effectuer les calculs. La formule de coût est la suivante :

$$Cost = 3 \times (|F| + |D_1|) + \sum_{i=2}^k [2 \times (|R_{i-1}| - B + 1) + 3 \times |D_i|] + 2 \times [|R_k| - B + 1]$$

Pour les requêtes contenant les groupements, nous ajoutons le coût d'écriture et de chargement pour effectuer ces groupements. La formule de coût est donnée par :

$$Cost = 3 \times (|F| + |D_1|) + \sum_{i=2}^k [2 \times (|R_{i-1}| - B + 1) + 3 \times |D_i|] + 4 \times [|R_k| - B + 1]$$

Calcul de la taille des résultats intermédiaires : Les formules de coût que nous utilisons reposent sur la taille des résultats intermédiaires. Le nombre de tuples d'un résultat intermédiaire ($R_i = R_{i-1} \bowtie D_i$) dépend de la sélectivité de cette jointure. La cardinalité de R_i est donnée par : $\|R_i\| = Sel(R_{i-1} \bowtie D_i) \times \|R_{i-1}\| \times \|D_i\|$. Notons que le nombre de pages nécessaires pour stocker un résultat intermédiaire R_i est donné par : $|R_i| = \frac{\|R_i\| \times TS_{R_i}}{PS}$.

3.3.2 Coût d'une requête sur un schéma fragmenté :

Le modèle de coût que nous avons présenté est utilisé pour estimer le coût d'une requête défini sur un schéma en étoile. Comme la fragmentation horizontale préserve le schéma de l'entrepôt, ce modèle peut être utilisé pour estimer le coût d'une requête sur un sous schéma en étoile généré par le processus de fragmentation. Chaque sous schéma en étoile sc_i est composé d'un fragment de faits F_i lié à un ensemble de fragments de dimension qui ont permis de le construire.

Pour expliquer comment nous avons appliqué le modèle de coût sur un schéma fragmenté, nous présentons un exemple de fragmentation d'un entrepôt de données.

Exemple 1 Soit un schéma en étoile constitué de trois tables de dimension Client, Produit et Temps et une table des faits Ventes dont la population est donnée dans la figure 3.

Supposons trois attributs de sélection Client.Ville, Produit.Catégorie et Temps.Mois.

Le domaine de l'attribut Ville est subdivisé en trois sous domaines comme suit :

$$Dom(Ville) = \{Poitiers, Paris, Nantes\}$$

Le domaine de l'attribut Catégorie est subdivisé en cinq sous domaines comme suit :

$$Dom(Catégorie) = \{Beauté, Jouets, Jardinage, Fitness\}$$

Le domaine de l'attribut Mois est subdivisé en six sous domaines comme suit :

Fragmentation: Complexité, Algorithmes de Sélection et Validation sous ORACLE10g

Client			
RID ^c	CID	Nom	Ville
6	616	Gilles	Poitiers
5	515	Yves	Paris
4	414	Patrick	Nantes
3	313	Didier	Nantes
2	212	Eric	Poitiers
1	111	Pascal	Poitiers

Produit			
RID ^p	PID	Nom	Catégorie
6	106	Sonoflore	Beauté
5	105	Clarins	Beauté
4	104	WebCam	Multimédia
3	103	Barbie	Jouet
2	102	Manure	Jardinage
1	101	SlimForm	Fitness

Temps			
RID ^t	TID	Mois	Année
6	11	Janvier	2003
5	22	Février	2003
4	33	Mars	2003
3	44	Avril	2003
2	55	Mai	2003
1	66	Juin	2003

Ventes				
RID ^v	CID	PID	TID	montant
1	616	106	11	25
2	616	106	66	28
3	616	104	33	50
4	545	104	11	10
5	414	105	66	14
6	212	108	55	14
7	111	101	44	20
8	111	101	33	27
9	212	101	11	100
10	313	102	11	200
11	414	102	11	102
12	414	102	55	103
13	515	102	66	100
14	515	103	55	17
15	212	103	44	45
16	111	105	66	44
17	212	104	66	40
18	515	104	22	20
19	616	104	22	20
20	616	104	55	20
21	212	105	11	10
22	212	105	44	10
23	212	105	55	18
24	212	106	11	18
25	313	105	66	19
26	313	105	22	17
27	313	106	11	15

FIG. 3 – Population d'un entrepôt de données

Client ₁			
C_RID	CID	Nom	Ville
3	616	Gilles	Poitiers
2	212	Eric	Poitiers
1	111	Pascal	Poitiers

Client ₂			
C_RID	CID	Nom	Ville
3	515	Yves	Paris
2	414	Yannick	Nantes
1	313	Didier	Nantes

Produit ₁			
P_RID	PID	Nom	Type
3	106	Sonoflore	Beauté
2	105	Clarins	Beauté
1	101	SlimForm	Fitness

Produit ₂			
P_RID	PID	Nom	Type
3	104	WebCam	Multimédia
2	103	Barbie	Jouet
1	102	Engrais	Jardinage

Temps ₁			
T_RID	TID	Mois	Année
3	11	janvier	2003
2	22	février	2003
1	33	mars	2003

Temps ₂			
T_RID	TID	Mois	Année
3	44	avril	2003
2	55	mai	2003
1	66	juin	2003

FIG. 4 – Tables de dimensions fragmentées

$Dom(Mois) = \{Janvier, Février, Mars, Avril, Mai, Juin\}$

Supposons que les tables de dimension sont fragmentées comme suit :
 La table Client est fragmentée selon l'attribut Ville en deux fragments :

- $Client_1 : \sigma_{Ville=Poitiers}(Client)$
- $Client_2 : \sigma_{(Ville=Paris \vee Ville=Nantes)}(Client)$.

La table Produit est fragmentée sur l'attribut Catégorie (représenté par Cat) en deux fragments :

- $Produit_1 : \sigma_{(at=Beauté \vee Cat=Fitness)}(Produit)$
- $Produit_2 : \sigma_{(Cat=Multimédia \vee Cat=Jouet \vee Cat=Jardinage)}(Produit)$.

La table Temps est fragmentée sur l'attribut Mois en deux fragments :

- $Temps_1 : \sigma_{(Mois=Janvier \vee Mois=Février \vee Mois=Mars)}(Temps)$
- $Temps_2 : \sigma_{(Mois=Avril \vee Mois=Mai \vee Mois=Juin)}(Temps)$.

La figure 4 représente les instances des tables de dimension fragmentées. A partir des schémas de fragmentation des tables Client, Produit et Temps, la table des faits est fragmentée en

Ventes ₁					Ventes ₂				
V_r113	CID	PI3	TID	Ventes	V_r113	CID	PI3	TID	Ventes
5	616	106	11	25	8	616	106	66	28
4	111	101	23	27	7	414	105	66	14
3	212	101	11	100	6	212	106	55	14
2	212	105	11	10	5	111	101	44	20
1	212	106	11	18	4	111	105	66	44
Ventes ₃					Ventes ₄				
V_r113	CID	PI3	TID	Ventes	V_r113	CID	PI3	TID	Ventes
3	616	104	33	30	3	616	104	55	20
2	545	104	11	10	2	212	105	44	10
1	616	106	22	20	1	212	105	55	18
Ventes ₅					Ventes ₆				
V_r113	CID	PI3	TID	Ventes	V_r113	CID	PI3	TID	Ventes
2	313	105	22	17	2	212	103	44	45
1	313	106	11	15	1	212	104	66	40
Ventes ₇					Ventes ₈				
V_r113	CID	PI3	TID	Ventes	V_r113	CID	PI3	TID	Ventes
3	313	102	11	200	3	414	102	55	203
2	414	102	11	102	2	515	102	66	100
1	515	104	22	20	1	515	103	55	17

FIG. 5 – Fragments de faits

huit fragments défini comme suit ⁴ :

$Vente_1 : (Ville = Po) \wedge (Cat = Be \vee Cat = Fi) \wedge (Mois = Ja \vee Mois = Fév \vee Mois = Ma)$
 $Vente_2 : (Ville = Po) \wedge (Cat = Be \vee Cat = Fi) \wedge (Mois = Av \vee Mois = Mai \vee Mois = Ju)$
 $Vente_3 : (Ville = Po) \wedge (Cat = Mu \vee Cat = Jo \vee Cat = Ja) \wedge (Mois = Ja \vee Mois = Fév \vee Mois = Ma)$
 $Vente_4 : (Ville = Po) \wedge (Cat = Mu \vee Cat = Jo \vee Cat = Ja) \wedge (Mois = Av \vee Mois = Mai \vee Mois = Ju)$
 $Vente_5 : (Ville = Pa \vee Ville = Na) \wedge (Cat = Be \vee Cat = Fi) \wedge (Mois = Ja \vee Mois = Fév \vee Mois = Ma)$
 $Vente_6 : (Ville = Pa \vee Ville = Na) \wedge (Cat = Be \vee Cat = Fi) \wedge (Mois = Av \vee Mois = Mai \vee Mois = Ju)$
 $Vente_7 : (Ville = Pa \vee Ville = Na) \wedge (Cat = Mu \vee Cat = Jo \vee Cat = Ja) \wedge (Mois = Ja \vee Mois = Fév \vee Mois = Ma)$
 $Vente_8 : (Ville = Pa \vee Ville = Na) \wedge (Cat = Mu \vee Cat = Jo \vee Cat = Ja) \wedge (Mois = Av \vee Mois = Mai \vee Mois = Ju)$

Les fragments de faits obtenus selon ce schéma de fragmentation sont représentés dans la figure 5.

Un des intérêts de la fragmentation horizontale est le fait qu'elle permet d'éliminer les partitions non pertinentes pour une requête donnée dans le but d'accéder seulement aux partitions valides. Par conséquent, pour calculer le nombre d'E/S nécessaires pour exécuter une requête Q, nous commençons par identifier l'ensemble des sous schémas en étoile valides pour Q (noté $ESCEV(Q)$).

Définition 1 Un sous schéma en étoile sc_i est valide pour une requête Q, si et seulement si, Q accède à au moins un tuple du fragments de faits F_i composant sc_i .

Exemple 2 Supposons la requête Q_1 suivante qui calcule la somme des ventes effectuées par des clients poitevins ayant acheté un produit le mois de janvier :

```

SELECT Sum(ventes)
FROM Ventes V, Client C, Temps T
WHERE V.CID=C.CID AND V.TID=T.TID
AND C.Ville='Poitiers' AND T.Mois='Janvier'

```

⁴Po=Poitiers, Pa=Paris, Na=Nantes, Be=Beauté, Jo=Jouet, Ja=Jardinage, Mu=Multimédia, Fi=Fitness, Jn=Janvier, Fe=Février, Ma=Mars, Av=Avril et Ju=Juin.

Les fragments $Ventes_1$ et $Ventes_3$ sont valides pour Q_1 car ces deux fragments regroupent toutes les ventes effectuées par les clients poitevins durant le premier trimestre. Par contre le fragment $Ventes_2$ n'est pas valide pour Q_1 puisqu'il ne contient que les ventes effectués le deuxième trimestre.

Pour exécuter une requête Q , l'optimiseur effectue les tâches suivantes :

1. Déterminer l'ensemble $ESCEV(Q)$.
2. Exécuter Q sur chaque sous schéma $sc \in ESCEV(Q)$.
3. Faire l'union des résultats obtenus sur chaque sous schéma valide.

Notons que si une requête Q n'utilise que des prédicats de sélection définis sur des attributs ne participant pas dans la fragmentation de l'entrepôt, alors tous les sous schémas en étoiles générés sont valides pour Q .

Pour déterminer l'ensemble des sous schémas valide pour une requête Q , nous avons défini une fonction $Valide(Q, sc_i)$ qui retourne 1 si sc_i est valide pour Q et 0 sinon. Le temps globale d'exécution de Q sur les N sous schémas générés par un schéma de fragmentation SF est donné par : $Cost(Q, SF) = \sum_{i=1}^N Valide(Q, sc_i) \times Cost(Q, sc_i)$, où $Cost(Q, sc_i)$ représente le coût d'exécution de la requête Q sur le sous schéma en étoile valide sc_i .

Notons que la fragmentation horizontale primaire pré-calculent des sélections sur les tables de dimension et la fragmentation dérivée pré-calculent des jointures entre les tables de dimension et la table des faits. Par conséquent, l'exécution d'une requête Q sur un sous schéma en étoile sc_i peut être faite sans effectuer toutes les jointures présentes dans Q si elles sont déjà pré-calculées par le processus de fragmentation. Les jointures à effectuer dépendent du type de matching entre Q et le fragment de faits F_i définissant sc_i . Trois type de matching sont défini, pas de matching, matching total et matching partiel. Soit q la conjonction de prédicats de sélection définissant Q et p la conjonction de prédicats définissant F_i , nous donnons dans ce qui suit la définition des trois types de matching qui peuvent être identifiés entre Q et F_i .

Définition 2 *Une requête Q est en matching partiel avec un fragment de faits F_i si et seulement si, aucun tuple de F_i ne participe dans la réponse à Q . Formellement, il n'y a pas de matching entre Q et F_i , si et seulement si, $p \wedge q$ est non satisfiable.*

Notons que si Q et F_i ne possèdent aucun matching alors F_i n'est pas valide pour Q .

Définition 3 *Une requête Q est en matching partiel avec un fragment de faits F_i , si et seulement si, le sous schéma sc_i contenant F_i est valide pour Q et il existe au moins un tuple dans F_i ne participant pas dans la réponse à Q . Ce cas est constaté lorsque p n'implique pas q et p ne contredit pas q .*

Définition 4 *Une requête Q est en matching total avec un fragment de faits F_i , si et seulement si, tous les tuples dans F_i participent dans la réponse à Q . Formellement, Q est en matching total avec F_i , si et seulement si, $p \rightarrow q$.*

Exemple 3 : *La requête Q_1 défini plus haut n'a aucun matching avec le fragment $Ventes_2$ par exemple, car ce dernier concerne les ventes effectués durant les mois de avril, mai et juin or que Q_1 cherche les ventes effectuées le mois de janvier. Par contre Q_1 est en matching partiel avec le fragment $Ventes_1$ car ce dernier contient toutes les ventes effectuées par les clients*

poitevins durant les mois de janvier, février et mars. Ce matching est partiel car le fragment contient, en plus des ventes effectuées le mois de janvier, également les ventes effectuées les mois de février et mars. Nous constatons que Q_1 n'est en matching total avec aucun fragment car tous les fragments valides pour Q_1 concernent le premier trimestre et non pas le mois de janvier seul. Pour montrer un exemple du matching total, soit la requête Q_2 calculant la somme des ventes effectués par les clients poitevins :

```
SELECT Sum(ventes)
FROM Ventes V, Client C
WHERE V.CID=C.CID
AND C.Ville='Poitiers'
```

Q_2 est en matching total avec $Ventes_1, Ventes_2, Ventes_3$ et $Ventes_4$ car tous ces fragments ne contiennent que des ventes concernant des clients poitevins.

Identification des jointures à effectuer : La fragmentation horizontale pré-calculé des jointures entre la table de faits et les tables de dimension pour générer les fragments de faits. Pour exécuter une requête sur un schéma fragmenté, certaines jointures ne sont pas nécessaires puisque elles ont été déjà pré-calculées. Par conséquent, nous devons déterminer pour chaque requête les jointures à effectuer parmi celles qu'elle contient. Pour montrer l'existence de ce problème, nous considérons la requête Q_1 (défini ci-dessus) qui est en matching partiel avec le fragment $Ventes_1$. Ce dernier contient toutes les ventes effectuées par des clients poitevins durant les mois de janvier, février et mars or que Q_1 cherche les ventes effectuées par des clients poitevins durant le mois de janvier seul. Nous voyons bien que la jointure de $Ventes_1$ avec la table *Client* n'est pas nécessaire car $Ventes_1$ ne concerne que des clients poitevins (cette jointure a été effectuée durant la construction de $Ventes_1$). Par conséquent seule la table *Temps* sera jointe pour sélectionner les tuples de $Ventes_1$ concernant le mois de janvier seulement.

Dans les requêtes que nous considérons, les tables de dimension à joindre sont celles ayant un attribut référencé dans la clause SELECT ou dans la clause WHERE. Sur un sous schéma en étoile valide sc_i , les tables de dimension ayant un attribut référencé dans la clause SELECT sont jointes automatiquement avec le fragment de faits. Par contre la jointure des tables de dimension possédant un attribut de sélection (dans la clause WHERE) n'est pas automatique mais dépend du type de matching entre Q et sc_i .

1. Matching total : dans ce cas aucune table de dimension référencée dans la clause WHERE n'est à joindre, toutes les sélections et les jointures présentes dans Q ont été déjà pré-calculées par le processus de fragmentation. Nous pouvons identifier deux cas possibles : (1) la clause SELECT ne contient aucun attribut de dimension, dans ce cas aucune table de dimension n'est jointe et l'optimiseur charge le fragment de faits seulement pour répondre à Q . Le coût sera alors le coût de chargement du fragment de fait F_i ; (2) la clause SELECT contient des attributs de dimension, dans ce cas les tables contenant ces attributs sont jointes avec le fragment de faits. Nous appliquons le même modèle de coût que celui utilisé dans le cas d'entrepôt non fragmenté en remplaçant la table des faits par le fragment de faits et les tables de dimension par les fragments de dimensions.
2. Matching partiel : dans ce cas, le fragment F_i contient certains tuples qui ne correspondent pas à Q . L'optimiseur doit éliminer ces tuples en effectuant des jointures avec

les fragments de dimension liés à F_i . En plus de ces jointures, l'optimiseur effectue les jointures avec les tables ayant un attribut référencé dans la clause SELECT. Une fois toutes les jointures nécessaires identifiées, nous appliquons alors le même modèle de coût que le cas fragmenté en remplaçant la table des faits par le fragment de faits et les tables de dimensions par les fragments de dimension des tables à joindre.

3.4 Hill Climbing

L'algorithme de Hill Climbing est une méthode de voisinage composée de deux étapes essentielles :

1. Trouver une solution initiale représentant un schéma de fragmentation de l'entrepôt.
2. Améliorer itérativement le schéma initial en utilisant des mouvements locaux tant que la réduction du temps d'exécution des requêtes est possible et que la contrainte de maintenance est satisfaite.

Une solution initiale peut être obtenue aléatoirement (en assignant des numéros choisis de manière aléatoire dans chaque cellule du tableau multidimensionnel). Dans ce travail, nous avons évité la génération aléatoire de la solution initiale. Nous avons utilisé l'algorithme d'affinité (Bellatreche et al., 2000) qui permet de regrouper les sous domaines en se basant sur leur affinité. Une affinité entre deux sous-domaines est la somme des fréquences d'accès des requêtes accédant simultanément à ces deux sous-domaines. Dans la deuxième étape, des mouvements sont effectués sur la solution initiale pour réduire le temps d'exécution des requêtes. Ces mouvements se basent sur l'utilisation de deux fonctions, *Merge* et *Split*. La fonction *Merge* permet de combiner deux partitions de sous domaines en une seule et ainsi diminuer le nombre de fragments générés. La fonction *Split* est duale à la fonction *Merge*. Elle permet d'éclater une partition en deux et augmenter ainsi le nombre de fragments générés. Ces deux fonctions ressemblent aux primitives d'Oracle, MERGE PARTITION et SPLIT PARTITION qui permettent de fusionner deux partitions en une seule et d'éclater une partition en deux respectivement.

- La fonction *Merge* possède la signature suivante : $Merge(P_i^k, P_j^k, A_k, FS) \rightarrow FS'$. Elle prend en entrée deux partitions P_i^k et P_j^k ($P_i^k \neq P_j^k$) de l'attribut A_k et un schéma de fragmentation FS et donne en sortie un autre schéma FS' avec les deux partitions P_i^k et P_j^k fusionnées en une seule partition. La fusion de deux partitions consiste à leur attribuer le même numéro. C'est-à-dire assigner un même numéro à toutes les cellules appartenant aux deux partitions sur le tableau unidimensionnel correspondant à A_k .
- La fonction *Split* possède la signature suivante : $Split(P_i^k, A_k, FS) \rightarrow FS''$. Elle prend en entrée une partition P_i^k de l'attribut A_k et un schéma de fragmentation FS et donne en sortie un schéma FS'' avec la partition P_i^k éclatée en deux partitions P_i^{k1} et P_i^{k2} ($P_i^{k1} \neq P_i^{k2}$). Cela est effectué en donnant deux numéros différents aux cellules représentant P_i^{k1} et celles représentant P_i^{k2} . La fonction *Split* n'est pas appliquée sur une partition élémentaire contenant un seul sous domaine.

Pour illustrer le fonctionnement des deux fonctions, nous considérons l'exemple suivant ;

Exemple 4 Supposant le schéma de fragmentation représenté par la figure 6(a). Dans ce schéma, le domaine de l'attribut Sexe est partitionné en deux partitions, le domaine de l'attribut Saison en trois partitions et le domaine de l'attribut Age en deux partitions. Par exemple

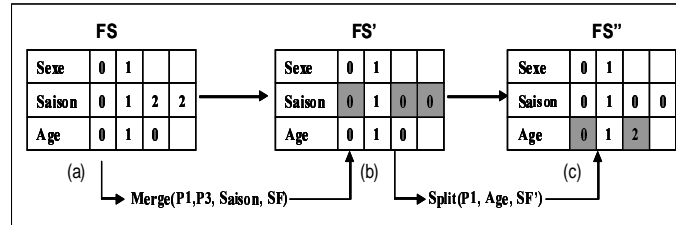


FIG. 6 – Exemple d'application de fonctions Merge et Split

pour l'attribut Saison, la partition P_1 , contient le sous domaine sd_1 , P_2 contient le sous domaine sd_2 et P_3 contient les sous domaines sd_3 et sd_4 . Par conséquent SF génère 12 fragments.

L'application de la fonction Merge sur les partitions P_1 et P_3 de l'attribut Saison ($Merge(P_1, P_3, Saison, SF)$) donne le schéma de fragmentation SF'. Dans SF' l'attribut Saison possède deux partitions après la fusion, la partition P'_1 contient les sous-domaines sd_1 , sd_3 et sd_4 et la partition P'_2 contient le sous-domaine sd_2 (voir figure 6(b)). SF' génère ainsi 8 fragments.

Sur le schéma SF', nous appliquons la fonction Split sur la partition P_1 de l'attribut Age ($Split(P_1, Age, SF')$), nous obtenons le schéma SF''. Dans SF'', la partition P_1 a été scindée en deux partitions, la première contient le sous-domaine sd_1 et la deuxième le sous-domaine sd_3 (voir figure 6(c)). SF'' génère au total 12 fragments.

4 Expérimentations

Dans cette section, nous présentons les résultats expérimentaux pour comparer les différents algorithmes : hill climbing, le génétique et le recuit simulé. Les deux derniers ont été développés dans Bellatreche et al. (2006). Dans ce travail, nous avons réutilisé ces deux algorithmes avec le codage présenté dans cet article en utilisant les fonctions à croissance restreinte. Cette comparaison est effectuée en utilisant un modèle de coût mathématique.

L'entrepôt de données Nous avons utilisé l'entrepôt de données issu du benchmark APB1 (Council, 1998). Le schéma en étoile de ce benchmark possède une table de faits Actvars(24 786 000 tuples), et quatre tables de dimension : Prodlevel (9 000 tuples), Custlevel (900 tuples), Timelevel(24 tuples) et Chanlevel (9 tuples).

Charge de requêtes Nous avons considéré 60 requêtes de recherche composées d'un seul bloc (pas de requêtes imbriquées) avec 40 prédicats de sélection définis sur 12 attributs (ClassLevel, GroupLevel, FamilyLevel, LineLevel, DivisionLevel, YearLevel, MonthLevel, QuarterLevel, RetailerLevel, CityLevel, GenderLevel, AllLevel). Les domaines de ces attributs sont décomposés en 4, 2, 5, 2, 4, 2, 12, 4, 4, 4, 2 et 5 sous domaines. Chaque prédicat possède un facteur de sélectivité calculé sur l'entrepôt réel. Nos algorithmes ont été implémentés en utilisant Visual C++ sur une machine Intel Centrino de 1Go de RAM.

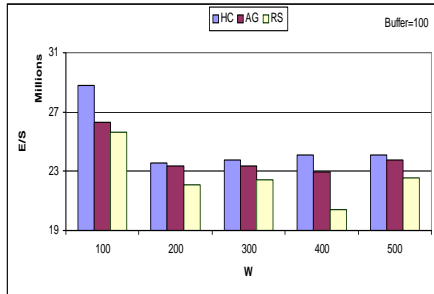


FIG. 7 – Effet de W

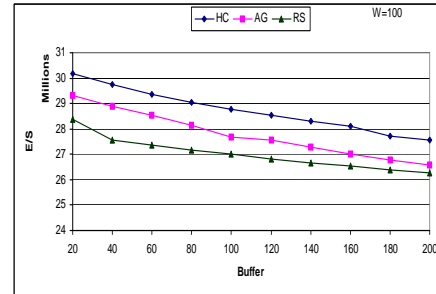


FIG. 8 – Etude de l'impact du buffer

4.1 Comparaison des heuristiques

Les paramètres de l'algorithme génétique sont : nombre d'individus : 20, taux de croisement 80 et taux de mutation 20. Les paramètres du recuit simulé sont : température initiale : 1400, facteur de décroissement de la température 0,9 et l'équilibre 10.

La figure 7 montre les performance de chaque algorithme par rapport au seuil W . Nous avons fait varier W de 100 à 500 en utilisant 40 prédicats et pour chaque valeur de W , nous exécutons chaque algorithme. Le recuit simulé et le génétique donnent de meilleurs résultats. L'algorithme hill climbing est moins performant par le fait qu'il est confronté à une recherche plus fine dans des optimums locaux. L'augmentation du seuil améliore généralement les performances des requêtes car en relâchant W , plus d'attributs sont utilisés pour fragmenter l'entrepôt. Lorsque W est grand les domaines sont décomposés en plus de partitions et donc chaque partition est moins volumineuse. Cela implique moins de données chargées pour exécuter les requêtes utilisant les attributs de fragmentation.

La figure 8 montre l'effet du tampon (buffer) sur la performance des requêtes. Notre modèle de coût est basé sur la gestion du tampon et sa taille. Nous varions la valeur du buffer de 20 à 200 pages et nous exécutons les algorithmes pour chaque valeur. L'augmentation de la taille du tampon implique une amélioration de la performance car un tampon assez large permet de garder les résultats intermédiaires en mémoire où l'accès n'est pas coûteux en temps. Un tampon de petite taille provoque l'écriture des résultats intermédiaires sur le disque et leur rechargement pour les prochaines jointures, ce qui augmente le nombre d'E/S.

La figure 9 montre l'effet de la taille des tables de dimension (en terme de nombre de tuples). Nous avons considéré que toutes les tables de dimension ont le même nombre de tuples et nous avons fait varier ce nombre entre 10 et 100000. Les résultats montrent que pour les petites tailles des tables (de 10 à 1000) les résultats sont presque similaires du fait que pour ces cas les fragments des tables de dimension sont suffisamment petits pour tenir dans un nombre de pages similaires. Lorsque la taille des tables de dimension devient importante, le temps d'exécution augmente car les fragments de ces dernières occupent plus d'espace et provoque plus de données enregistrées sur le disque lors de leur jointure avec les fragments de faits.

Pour voir l'effet du nombre de prédicats utilisés dans les 60 requêtes sur la performance globale, nous faisons varier ce nombre entre 10 et 40. Nous créons quatre classes de requêtes utilisant chacune un nombre de prédicats différents (10, 20, 30 et 40 prédicats). Pour chaque

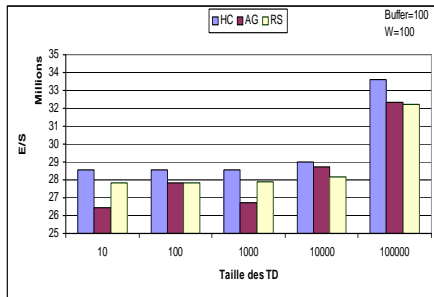


FIG. 9 – Effet de la taille des TD

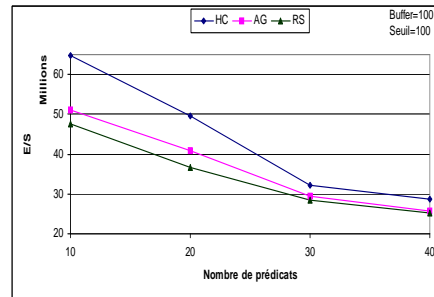


FIG. 10 – Effet du nombre de prédicats

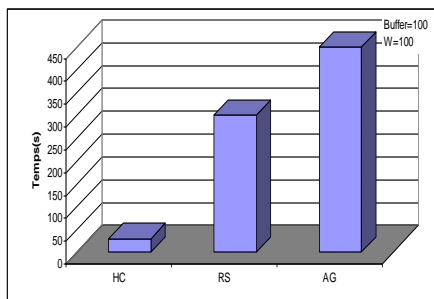


FIG. 11 – Temps d'exécution de chaque algorithme

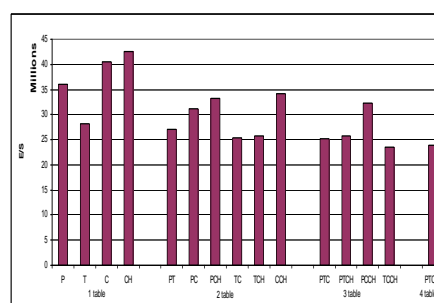


FIG. 12 – Choix et nombre de tables de dimension

instance, nous exécutons nos algorithmes pour $W=100$. Les résultats obtenus montrent que le nombre de prédicats utilisés par les requêtes à un effet considérable sur la performance des requêtes. Lorsque ce nombre est petit, la plupart des requêtes ne bénéficient pas de la fragmentation. Ceci est expliqué par le fait qu'elles accèdent à un nombre important de sous schémas, voire la totalité des sous-schémas si elles ne possèdent pas des prédicats définis sur des attributs de fragmentation. Par conséquent, plusieurs opérations d'union sont nécessaires pour avoir le résultat final. Par contre si le nombre de prédicats est important, le nombre de sous schémas valides pour chaque requête est réduit (surtout pour celles n'utilisant que des attributs de fragmentation) ce qui implique le chargement de moins de données (les sous schémas valides seulement).

La figure 11 montre le temps moyen d'exécution de chaque algorithme pour $W=100$. Le génétique et le recuit simulé consomment plus de temps d'exécution vu qu'ils utilisent plusieurs opérations. L'algorithme génétique prend plus de temps car il manipule plusieurs solutions en même temps. Le hill climbing est l'algorithme le plus rapide puisqu'il est basé sur deux opérations simples à savoir Merge et Split. Si administrateur privilégie la qualité de la solution, il choisira le recuit simulé, sinon le hill climbing s'il privilégie la rapidité d'exécution de son algorithme de sélection.

Pour montrer l'effet du choix des tables de dimension à fragmenter, nous avons mené des expérimentations en utilisant, 1, 2, 3 et 4 tables de dimension pour fragmenter l'entrepôt. Pour

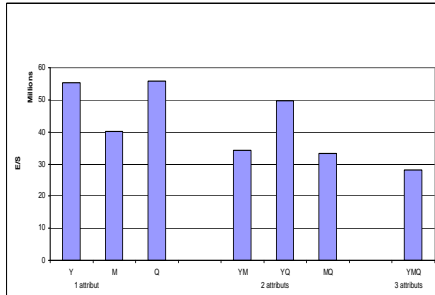


FIG. 13 – Choix des attributs de la table Timelevel

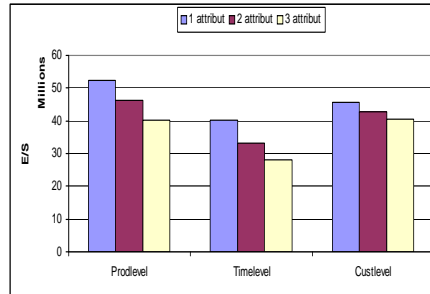


FIG. 14 – Effet du nombre d'attributs de fragmentation de chaque table

chaque cas, nous avons pris en considération toutes les combinaisons possibles. Ces expérimentations ont été effectuées en utilisant le RS puisque il donne les meilleurs résultats. Les tables utilisées sont : Prodlevel(P), Timelevel(T), Custlevel(C) et Chanlevel(CH). La figure 12 montre les résultats obtenus. Ces derniers montrent que le choix des tables de dimension à fragmenter est très important. Par exemple, la table Timelevel est la plus adaptée pour être partitionnée, cela est justifiée que cette table est la table la plus utilisée par la charge de requêtes et par conséquent ces requêtes accéderont à moins de sous schémas en étoile et donc moins de données chargées. Les résultats montrent aussi que choisir plus de tables de dimension pour la fragmentation donne plus de performance à condition de choisir les bonnes tables.

Pour montrer l'effet du choix des attributs, nous avons pris l'exemple de la table Timelevel. Nous avons considéré toutes les combinaisons possibles pour le choix des attributs (voir figure 13). Ces attributs sont (Monthlevel(M), Yearlevel(Y) et Quarterlevel(Q)). L'attribut Monthlevel donne les meilleurs résultats. Ceci s'explique par le fait que Monthlevel est l'attribut le plus utilisé et son utilisation dans la fragmentation permet aux requêtes qui l'utilisent d'accéder à un nombre réduit de sous-schémas et de données.

La même expérience a été menée pour les deux autres tables de dimension. Nous avons pris pour chaque table le meilleur temps d'exécution des requêtes pour le cas de 1, 2 ou 3 attributs choisis pour fragmenter l'entrepôt (voir figure 14). Les résultats montrent que lorsqu'on choisit plus d'attributs pour fragmenter l'entrepôt nous obtenons plus de performance car plus de requêtes sont satisfaites (celles utilisant ces attributs) en accédant à moins de sous schémas en étoile.

5 Validation sous ORACLE10g

Pour valider notre travail, nous avons développé une plate forme sous ORACLE10g pour gérer le partitionnement d'un entrepôt de données. L'architecture de cette plate forme est décrite dans la figure 15.

Cette architecture est composée de trois module principaux : (1) le module de sélection d'un schéma de fragmentation, (2) le module de fragmentation de l'entrepôt et (3) le module de réécriture des requêtes.

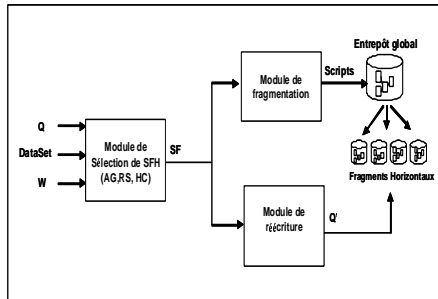


FIG. 15 – Architecture de notre implémentation

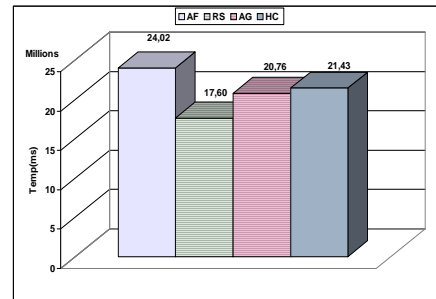


FIG. 16 – Résultats sous Oracle

1. *Module de sélection d'un schéma de fragmentation horizontale (MSSFH)* : à partir d'un ensemble de requêtes, le schéma de l'entrepôt et le nombre de fragments souhaités, MSSFH permet de sélectionner un schéma de fragmentation horizontale en utilisant un des trois algorithmes proposés.
2. *Module de fragmentation (MF)* : Ce module reçoit en entrée un schéma de fragmentation généré par MSSFH et donne en sortie tous les scripts permettant de fragmenter l'entrepôt. Deux types de scripts sont générés. Le premier concerne les scripts pour fragmenter les tables de dimensions en utilisant la fragmentation primaire. Le deuxième type de scripts est pour fragmenter la table des faits. Plusieurs difficultés sont apparues pour réaliser ce module. Elles sont liées au fait qu'ORACLE11G ne supporte pas une fragmentation primaire sur plus de trois attributs et une fragmentation dérivée de la table des faits en utilisant plus de deux tables de dimension (voir Section 1).

Pour remédier à ces difficultés, nous avons développé une technique permettant d'implémenter la fragmentation horizontale sur plusieurs attributs. La technique consiste à ajouter une colonne supplémentaire Col_i dans la table de dimension D_i à fragmenter. Selon le schéma de fragmentation de la table D_i notre module de fragmentation génère un script PLSql permettant de remplir cette colonne pour chaque instance. La valeur de cette colonne pour une instance ins_j d'une table de dimension D_i correspond au numéro de fragment contenant ins_j . Après le remplissage, nous fragmentons la table D_i par liste sur cette colonne.

Pour supporter la fragmentation horizontale dérivée basée sur la fragmentation de plusieurs tables de dimension, nous avons développé la solution suivante : nous créons une colonne supplémentaire Col_F dans la table des faits. Cette colonne contient la concaténation des numéros de fragments des tables de dimension. Pour remplir cette colonne, nous effectuons une jointure entre la table des faits et les tables de dimension fragmentées, et nous concaténons les valeurs des colonnes Col_i utilisées pour fragmenter les tables de dimension. Nous fragmentons ensuite la table des faits par liste sur la colonne Col_F . Pour faire cela, nous créons une vue matérialisée temporaire V comme suit :

```
CREATE MATERIALIZED VIEW V
BUILD IMMEDIATE
AS
SELECT a.customer_level, a.product_level,a.channel_level,
a.time_level, a.unitssold, a.dollarsales,a.dollarcost,
prod_col||'-'||time_col||'-'||cust_col as Col_F
```

Fragmentation: Complexité, Algorithmes de Sélection et Validation sous ORACLE10g

```
FROM actvars a, prodlevel p, custlevel c, timelevel t
WHERE a.customer_level = c.store_level
AND a.product_level = p.code_level
AND a.time_level = t.tid
```

Nous chargeons la vue matérialisée V dans la table des faits et nous supprimons ensuite V.

3. *Module de réécriture (MR)* : ce module reçoit en entrée un schéma de fragmentation *SF* et les requêtes globales, il retourne des requête réécrites sur *SF*. La réécriture d'une requête q_j sur *SF* consiste à identifier les sous schémas valides pour q_j et ensuite la réécrire sur ces derniers.

5.1 Implémentation sous Oracle

Nous avons implémenté les schémas de fragmentation obtenus à partir de chaque algorithme sous Oracle 10g. Nous avons choisi le scénario suivant pour valider les trois algorithmes : L'algorithme génétique et le recuit simulé ont généré un schéma de fragmentation composé de 80 sous schémas en étoile. Le génétique utilise 5 attributs parmi 12 et 3 tables de dimension (Timelevel, Custlevel et Chanlevel) pour fragmenter l'entrepôt. Le recuit simulé utilise 5 attributs mais toutes les tables de dimension ont été utilisées pour fragmenter l'entrepôt. le Hill Climbing a généré 96 sous schémas en étoile en utilisant 4 attributs et 3 tables de dimension (Prodlevel, Timelevel et Chanlevel).

Nous avons implémenté chaque schéma en utilisant notre module de fragmentation. Les requêtes d'origine ont été réécrites par le module de réécriture sur chaque schéma de fragmentation. Nous avons exécuté les requêtes d'origine sur l'entrepôt non fragmenté (AF) et les requêtes réécrites sur l'entrepôt fragmenté correspondant. Le tampon est vidé après chaque exécution de requête. La figure 16 montre les résultats obtenus. Elle illustre deux points importants : (i) que la fragmentation horizontale dans les entrepôts de données est cruciale pour la performance de requêtes et (ii) que le choix de l'algorithme de fragmentation a un impact considérable sur cette performance. Par exemple, l'algorithme de recuit simulé donne de meilleurs résultats (comme dans l'évaluation théorique) que le génétique ou le hill climbing.

6 Conclusion

La fragmentation horizontale a été largement adoptée par la plupart des systèmes de gestion de bases de données commerciaux et les chercheurs, où elle est supportée dans leurs langages de définition de données. Nous avons présenté un état de l'art sur l'évolution de cette technique d'optimisation non redondante au sein de ces systèmes. Cette étude nous a permise de confronter les travaux de recherche avec les travaux industriels et surtout de dégager quelques limites des travaux industriels. Ainsi, nous avons montré la NP-complétude du problème de sélection de schéma de fragmentation d'un entrepôt de données. Le schéma sélectionné comprend deux autres schémas : un schéma de fragmentation des tables de dimension (obtenu par la fragmentation horizontale primaire) et un schéma de fragmentation de la table des faits (obtenu par la fragmentation horizontale dérivée). Pour sélectionner un schéma de fragmentation quasi optimal, nous avons proposé une heuristique de hill climbing. Pour évaluer la qualité de cette heuristique, nous l'avons comparée avec d'autres autres algorithmes : le génétique et le recuit

simulé. Cette comparaison est réalisée avec une étude expérimentale basée sur un modèle de coût calculant le nombre d'entrées/sorties nécessaire pour exécuter un ensemble de requêtes. Pour valider nos algorithmes, nous avons mené une implémentation réelle de tous les schémas de fragmentation générés par les trois algorithmes sous ORACLE10G avec les données du banc d'essai APB1. Cette implémentation comprend la fragmentation horizontale primaire et dérivée. Les résultats obtenus montrent l'intérêt et l'impact de la fragmentation horizontale dans les entrepôts de données relationnels et le choix de l'algorithme de fragmentation.

Il serait intéressant d'adapter les algorithmes proposés pour la fragmentation verticale. Une adaptation directe consiste à remplacer les sous domaines par les attributs de tables à fragmenter.

Références

- Bellatreche, L. et K. Boukhalfa (2005). An evolutionary approach to schema partitioning selection in a data warehouse environment. *Proceeding of the International Conference on Data Warehousing and Knowledge Discovery (DAWAK'2005)*, 115–125.
- Bellatreche, L., K. Boukhalfa, et H. I. Abdalla (2006). Saga : A combination of genetic and simulated annealing algorithms for physical data warehouse design. in *23rd British National Conference on Databases* (212-219).
- Bellatreche, L., K. Karlapalem, et A. Simonet (2000). Algorithms and support for horizontal class partitioning in object-oriented databases. in *the Distributed and Parallel Databases Journal* 8(2), 155–179.
- Ceri, S., M. Negri, et G. Pelagatti (1982). Horizontal data partitioning in database design. *Proceedings of the ACM SIGMOD International Conference on Management of Data. SIGPLAN Notices*, 128–136.
- Corp., O. (2007). Oracle partitioning. *White Paper*, <http://www.oracle.com/technology/products/bi/db/11g/>.
- Council, O. (1998). Apb-1 olap benchmark, release ii. <http://www.olapcouncil.org/research/resrchly.htm>.
- Er, M. C. (1988). A fast algorithm for generating set partitions. *Comput. J.* 31(3), 283–284.
- Garey, M. R. et D. S. Johnson (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA : W. H. Freeman & Co.
- Navathe, S., K. Karlapalem, et M. Ra (1995). A mixed partitioning methodology for distributed database design. *Journal of Computer and Software Engineering* 3(4), 395–426.
- Özsu, M. T. et P. Valduriez (1999). *Principles of Distributed Database Systems : Second Edition*. Prentice Hall.
- Papadomanolakis, S. et A. Ailamaki (2004). Autopart : Automating schema design for large scientific databases using data partitioning. *Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM 2004)*, 383–392.
- Sacca, D. et G. Wiederhold (1985). Database partitioning in a cluster of processors. *ACM Trans. Database Syst.* 10(1), 29–56.

Fragmentation: Complexité, Algorithmes de Sélection et Validation sous ORACLE10g

Sanjay, A., V. R. Narasayya, et B. Yang (2004). Integrating vertical and horizontal partitioning into automated physical database design. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 359–370.

Tucker, A., J. Crampton, et S. Swift (2005). Rgfga : An efficient representation and crossover for grouping genetic algorithms. *Evol. Comput.* 13(4), 477–499.

Summary

Horizontal partitioning has been largely advocated by database community, especially during the physical design phase. Most of today's commercial database systems offer native DDL (data definition language) support for defining horizontal partitions of a table/view using different modes. In this paper, we first present the evolution of horizontal partitioning in commercial database systems during last decade and we point out some limitations. Secondly, we study the problem of selecting horizontal partitioning schema of a relational data warehouse and we show its NP-completeness. Due to its high complexity, we develop a hill climbing heuristic to select a near optimal solution and we propose a mathematical cost model to evaluate the quality of each partitioning schema. We conduct extensive experimental studies to compare using a mathematical cost model the quality of hill climbing with other existing algorithms (genetic and simulated annealing). Finally, we present a real validation of our algorithms on ORACLE10G using data set of APB1 benchmark.