N^o d'ordre : 4073

THÈSE

Pour l'obtention du grade de

DOCTEUR DE L'UNIVERSITÉ DE BORDEAUX 1

École doctorale de Mathématiques et Informatique de Bordeaux

DOMAINE DE RECHERCHE : Informatique

Présentée par

Radu TOFAN

Bordures: de la sélection de vues dans un cube de données au calcul parallèle de fréquents maximaux

---

Directeur de thèse : **Nicolas HANUSSE**

Co-direction : **Sofian MAABOUT**

Soutenue le 28 septembre 2010
Devant la Commission d'Examen

JURY

M. Guy	MELANÇON	Président
MME. Rosine	CICCHETTI	Rapporteur
MME. Anne	LAURENT	Rapporteur
MME. Véronique	BENZAKEN	Examinateur
M. Cyril	GAVOILLE	Examinateur
M. Noël	NOVELLI	Examinateur

Remerciements

Je tiens tout d'abord à exprimer toute ma reconnaissance à mes directeurs de thèse Nicolas Hanusse et Sofian Maabout pour leur encadrement, pour leurs nombreux conseils et leur soutien tout au long de ma thèse. J'ai particulièrement apprécié leur patience, pédagogie et disponibilité.

D'un point de vue scientifique, je voudrais rappeler le plaisir et l'honneur que m'ont fait Véronique Benzaken, Rosine Cicchetti, Cyril Gavoille, Anne Laurent, Guy Melançon et Noël Novelli en acceptant d'être membres de mon jury de thèse. Je remercie les rapporteurs Rosine Cicchetti et Anne Laurent pour l'oeil à la fois critique et bienveillant qu'ils ont bien voulu porter sur mes travaux.

Cette thèse a été menée dans le cadre de l'équipe CEPAGE INRIA gérée par Olivier Beaumont, que je remercie pour l'encadrement au sein de cette équipe.

Je souhaite également adresser mes remerciements à mes collègues du laboratoire LaBRI, notamment ceux avec qui j'ai partagé mes bureaux successifs ainsi que de bons moments : Srivathsan Balaguru, Pierre Bourreau, Aurélie Goulielmakis, Florent Le Gac (à quand une jeu d'échec ?), Nicolas Loira, Petru Valicov, Natalia Vinogradova.

Appréciation et amour vont vers mon épouse Gabriela et notre fils Petru qui ont voyagé avec moi dans le bateau de la vie pendant toutes ces dernières trois années.

Finalement, mais le plus important, je voudrais remercier à Dieu de m'avoir donné la chance de continuer mes études à l'Université Bordeaux 1. Il m'a donné le pouvoir et la sagesse de vaincre toutes les difficultés. Sans son amour, cette thèse n'aurait pas pu être terminée.

"La peste de l'homme,
c'est l'opinion de savoir"
Michel de Montaigne

Résumé

La matérialisation de vues est une technique efficace d'optimisation de requêtes.

Dans cette thèse, nous proposons une nouvelle vision "orientée utilisateur" de solutions pour le problème de sélection de vues à matérialiser dans les entrepôt de données : l'utilisateur fixe le temps de réponse maximal. Dans cette vision nous proposons des algorithmes qui s'avèrent compétitifs avec les algorithmes de type "orienté système", dans lesquels les ressources, comme la mémoire, sont considérées comme la contrainte forte.

L'approche "orientée utilisateur" est étudiée avec un contexte dynamique de système d'optimisation de requêtes. Nous analysons la stabilité de ce système par rapport à la dynamique de la charge de requêtes et des données qui sont insérées ou supprimées.

Le concept clé de nos algorithmes de sélection de vues à matérialiser est *la bordure*. Ce concept a été très étudié en fouille de données dans le cadre du calcul des fréquents maximaux. Plusieurs algorithmes séquentiels ont été proposés pour résoudre ce problème. Nous proposons un nouvel algorithme séquentiel MineWithRounds, facilement parallélisable, qui se distingue des autres propositions par une garantie théorique d'accélération dans le cas de machines à plusieurs unités de calcul et à mémoire partagée.

Abstract

The materialization of views is an effective technique for optimizing queries.

In this thesis, we propose a new vision, we qualify it as "user oriented", of the solutions to the problem of selecting views to materialize in data warehouses : the user fixes the maximum response time. In this vision, we propose algorithms that are competitive with the algorithms "oriented system" type, where resources such as memory, are considered as the major constraint.

The "user oriented" approach is studied under a dynamic context. We analyze the stability of this system with respect to the dynamic query workload dynamic as well as data dynamic (insertions and deletions).

The key concept of our algorithms for selecting views to materialize is *the border*. This concept has been widely studied in the data mining community under the maximal frequent itemset extraction setting. Many sequential algorithms have been proposed. We propose a new sequential algorithm **MineWithRounds**, easily parallelizable, which differs from the others in that it guarantees a theoretical speed up in the case of multiprocessors shared memory case.

Table des matières

Table des figures

Table des figures

Liste des tableaux

Chapitre I

Introduction Générale

Contexte

Les systèmes d'information décisionnel (SID) représentent un ensemble de technologies dédié à la transformation de données brutes en information ou en connaissance. Selon Ackoff [Ack89], par "information" nous comprenons des données qui répondent en général à des questions simples (qui, que, où). D'après le même auteur, la connaissance est une collection d'informations qui répond en général à la question "comment".

Shim et al. [SWC+02] distinguent quatre outils SID puissants qui ont émergé dans le début des années 1990 :

- les *entrepôts de données* dont le rôle est de stocker de manière structurée les données et sur lesquels des requêtes d'analyses peuvent être lancées ;
- la *technologie OLAP* (en anglais "On-Line Analytical Processing") représentant les données de manière multi-dimensionnelle et permettant de faire des analyses en fonction de combinaisons d'axes de dimension choisis par un utilisateur ;
- la *fouille de données* dont l'objectif est d'extraire de la connaissance sans a priori sur la nature des résultats attendus, c'est-à-dire sans hypothèse contrairement aux requêtes OLAP ;
- les *applications WEB* dédiées aux données provenant du WEB. Les difficultés d'analyses proviennent essentiellement de l'hétérogénéité des sources de données en terme de structures, contenus et de distribution.

Dans la figure I.1, nous avons décrit une vision personnelle et simplifiée de l'architecture d'un SID afin de présenter notre travail. Pour plus de détails, nous recommandons l'excellent travail sur la conception de l'architecture d'un entrepôt de données mené par Inmon et Kimball dans [Inm07, KRT+07] ou par Han [HK06]. Plus précisément, on peut

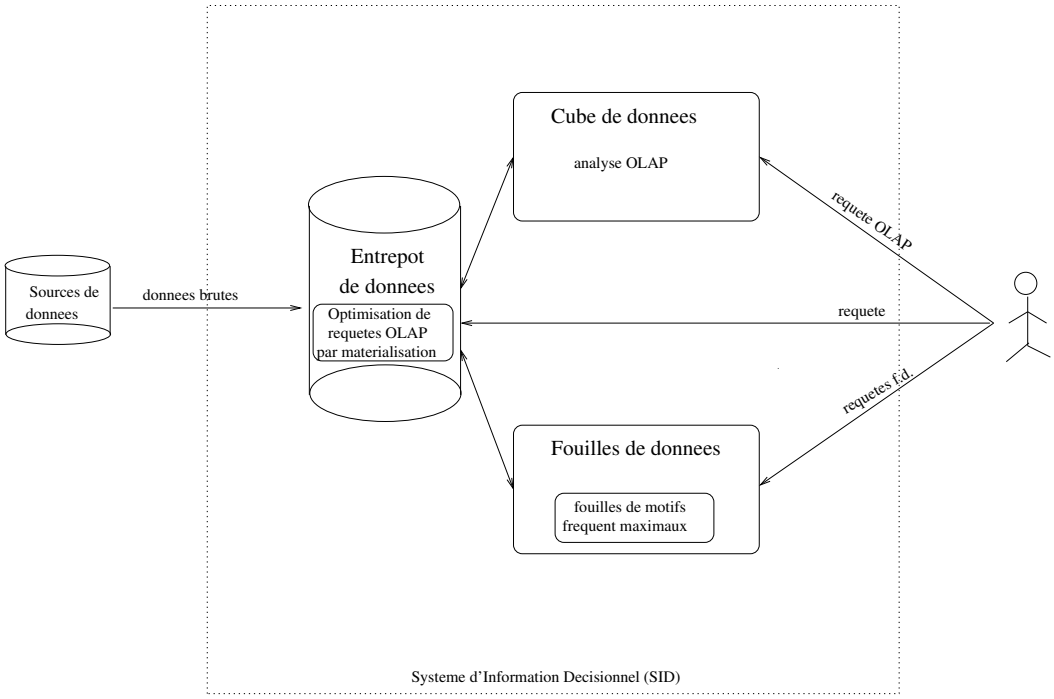


Figure I.1 – L'architecture d'un Système d'Information Décisionnel (SID).

distinguer trois types d'applications d'entrepôt de données, donc trois catégories de requêtes :

- la constitution de rapports basée sur des analyses statistiques ;
- l'analyse multidimensionnelle OLAP basée sur la navigation à l'intérieur des cubes de données ;
- la fouille de données qui permet la découverte de connaissance.

La frontière entre ces trois types de requêtes est un peu artificielle. D'une manière informelle, elles se distinguent les unes des autres en terme de complexité en temps. On peut considérer que les requêtes sont plus complexes en allant de la constitution de rapports (ensembles de requêtes simples prédéterminées) pour laquelle le facteur temps de réponse n'est pas très important à la fouille de données étudiant finement des données ou la dynamique des données en temps réel.

Comme nous le voyons dans la figure I.1, le système a comme entrée les sources de données externes et les requêtes utilisateur. Les sources de données exportent régulièrement de données vers l'entrepôt de données qui les structurent et les stockent.

De façon générale, d'après Inmon, un entrepôt de données est une large collection de données orientée sujet (qui concerne une activité spécifique), intégrée (les données viennent de plusieurs sources), historiée et non-volatile (les données ont un caractère permanent et

non modifiable afin d'assurer leur historique). Comme un entrepôt de données peut contenir des données provenant de plusieurs bases de données opérationnelles sur de longues périodes de temps, il a tendance à prendre beaucoup plus d'espace mémoire qu'une base de données opérationnelle. Les entrepôts de données des entreprises nécessitent un espace mémoire de centaines de gigaoctets ou téraoctets [CD97].

OLAP et cube de données La notion d'OLAP, introduite par Codd dans son article de référence en 1993 [CCS93], se réfère à une technique pour réaliser des analyses complexes sur les informations stockées dans un entrepôt de données. Dans l'OLAP, les données sont modélisées de façon multidimensionnelle, à partir d'une variété de sources et sont stockées dans un entrepôt de données [Pow07]. Le concept clé dans la modélisation multidimensionnelle de données est une structure de données appelée *cube de données*. Afin de faciliter l'analyse multidimensionnelle à l'aide du langage SQL, Gray et al. dans l'article [GCB+97] ont proposé une extension de SQL en introduisant l'opérateur `cube-by`. Même s'il est possible d'exprimer une requête multidimensionnelle à l'aide du langage SQL, il existe des langages spécialisés pour rédiger ce type de requêtes. Le langage MDX qui au départ a été proposé par Microsoft, est devenu *de facto* le standard pour la manipulation des données multidimensionnelles et a été adopté par la plupart des éditeurs de solutions.

Le cube de données permet de voir les données selon plusieurs dimensions. Il est construit à partir de deux types de tables :

- Les *tables de dimension* $D = \{D_1, D_2, \ldots, D_d\}$ représentant la description des données pour un jeu de données d-dimensionnel. Par exemple, la table "date" contient des informations de type "jour", "semaine", "mois", "trimestre" ou "année". Le niveau de granularité choisi correspond à un *attribut*. Pour une dimension donnée, l'ensemble des attributs peut constituer un ordre non nécessairement total. Par exemple, une semaine peut être à cheval sur deux mois consécutifs. Ainsi, une dimension peut être représentée par une hiérarchie d'attributs. Pour simplifier, nous considérons que chaque table de dimension est constituée d'une clé identifiante K_i et du nom de la dimension A_i.
- La *table de faits* qui contient des mesures (ex : "unités vendues") et les clés externes faisant référence à chaque table de dimension.

La table de faits est en général beaucoup plus grande que les tables de dimensions et il s'agit de l'entrée du cube de données. D'un point de vue pratique, une requête utilisateur correspond à un choix d'une combinaison de dimensions et d'attributs pour chaque di-

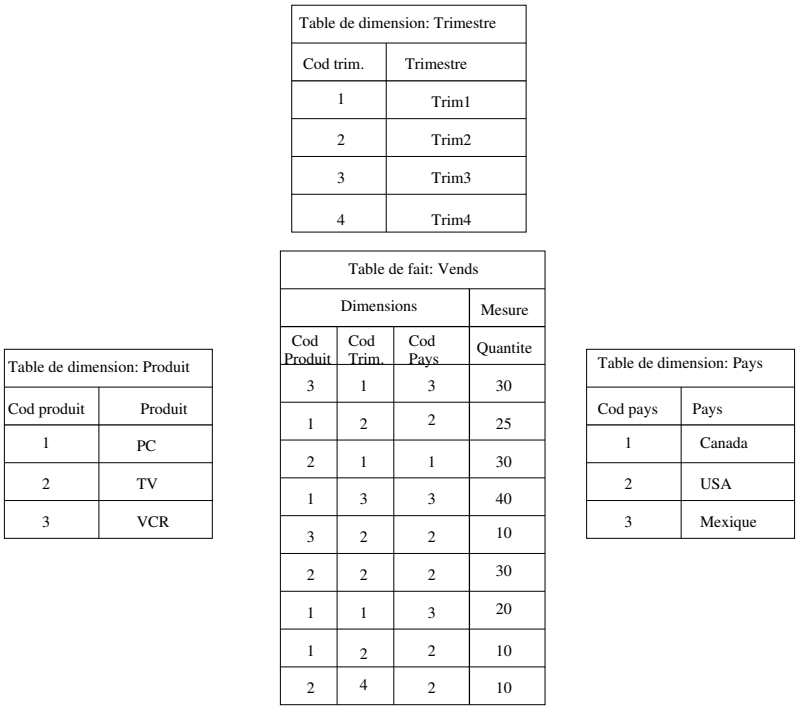


Table de dimension: Trimestre

Cod trim.	Trimestre
1	Trim1
2	Trim2
3	Trim3
4	Trim4

Table de dimension: Produit

Cod produit	Produit
1	PC
2	TV
3	VCR

Table de fait: Vends

Dimensions			Mesure
Cod Produit	Cod Trim.	Cod Pays	Quantite
3	1	3	30
1	2	2	25
2	1	1	30
1	3	3	40
3	2	2	10
2	2	2	30
1	1	3	20
1	2	2	10
2	4	2	10

Table de dimension: Pays

Cod pays	Pays
1	Canada
2	USA
3	Mexique

Figure I.2 – La table de fait "produit,trimestre,pays".

mension, avec éventuellement des sélections sur des valeurs d'attributs. Le résultat d'une requête sans sélection est appelé un *cuboïde*. Par exemple, la figure I.2 représente le cuboïde issu de la table de faits correspondant au choix des dimensions produit, trimestre et pays. La mesure Quantité correspond à la somme des produits vendus regroupés par valeurs des attributs produit, trimestre et pays. Pour un choix d'attributs fixé, il a été proposé de considérer toutes les combinaisons de dimensions possibles. Cette opération a été appelée l'opération "cube", terminologie provenant de la représentation cubique obtenue lorsque trois dimensions sont sélectionnées.

La figure I.3 montre ainsi le cube obtenu à partir des dimensions produit, trimestre et pays. L'ensemble des cellules blanches constitue le cuboïde "produit,trimestre,pays", chaque cellule contenant la mesure associée à une combinaison de 3 valeurs d'attributs. Ainsi, la cellule TV,USA,Trim2 a pour valeur 30. L'ensemble des cellules d'une même couleur correspond à un cuboïde. Les cellules en vert clair (respectivement en vert foncé) correspondent aux valeurs du cuboïde "produit,pays" (respectivement "produit"). Le cuboïde parfois noté "ALL,ALL,ALL" ou "NULL" est appelé le *APEX*. Il contient la mesure associée au regroupement de toutes les dimensions. Dans notre exemple, il doit avoir la valeur 205. Calculer le cube complet est extrêmement coûteux, aussi bien en temps qu'en

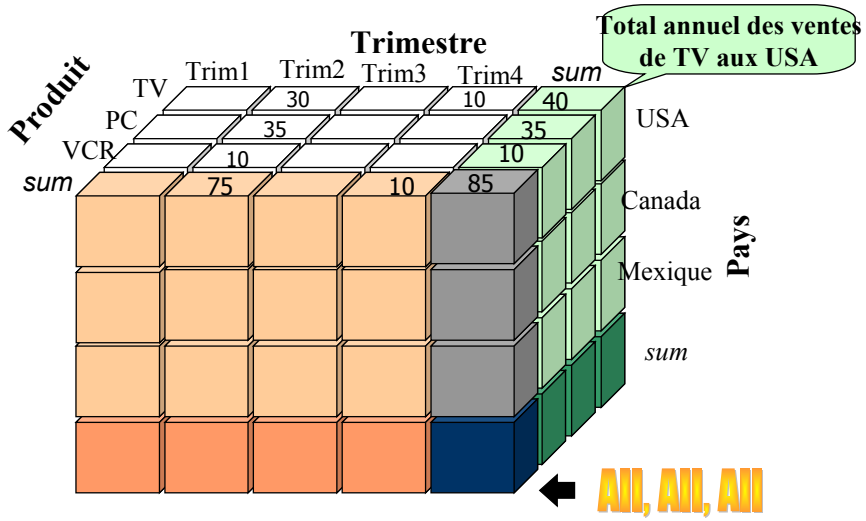


Figure I.3 – Un cube de données de 3 dimensions.

mémoire. Si l'utilisateur manipule d dimensions, le cube est composé de 2^d cuboïdes noté $\mathcal{C}$, hors considération de hiérarchies. En pratique (cf. figure I.4), dans un outil comme `dbminer`, l'utilisateur définit un cube de données qui n'est pas pré-calculé et navigue à l'intérieur en choisissant des dimensions et en affinant son choix en changeant de niveaux dans les hiérarchies des dimensions. Chaque opération de navigation prend du temps (ex : passage du cuboïde "produit,trimestre,pays" à "produit,pays"). Il se pose alors la question des pré-traitement possibles pour accélérer les temps de réponse à ces requêtes appelées requêtes OLAP.

Fouille de données La fouille de données a pour objectif de faire une analyse plus fine et donc plus ambitieuse que l'approche OLAP. De manière générale, l'ambition affichée est l'extraction de connaissances : découverte de corrélations entre dimensions ou attributs, découverte de motifs fréquents, prédiction de comportements, analyse de la dynamique des données. Pour chaque type d'analyse, de nombreux algorithmes existent dans la littérature. Dans cette thèse, nous nous concentrons sur la détermination des ensembles fréquents maximaux dans le cadre des architectures parallèles. En effet, dans un futur proche, la majorité des ordinateurs seront multi-processeurs. Le challenge est d'arriver à

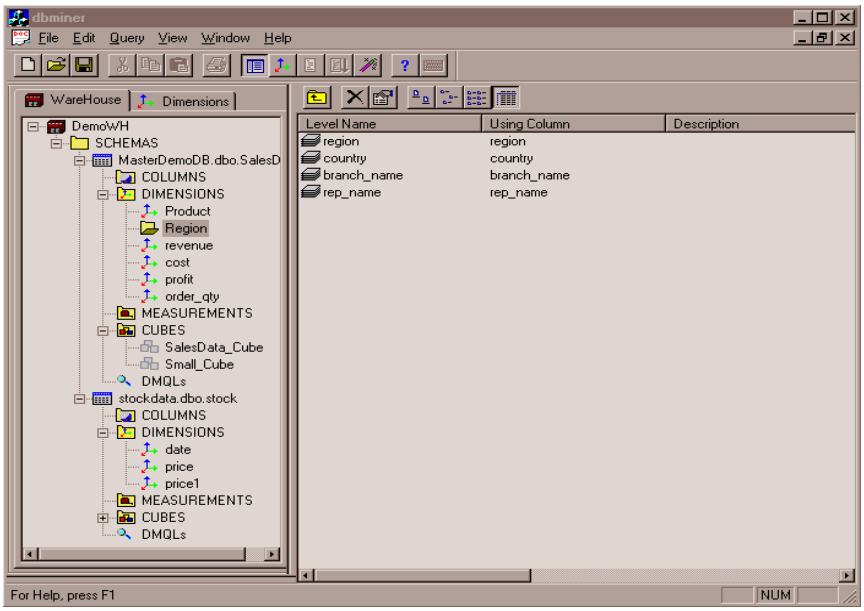


Figure I.4 – DBMINER : Constitution de 2 cubes de données à partir de la base de données MasterDemoDB (sales datacube, small cube).

en tirer réellement parti.

Philosophie générale Dans le cadre des requêtes OLAP ou de la fouille de données et quelle que soit la requête, le facteur temps est primordial. Du côté de l'opérateur ou des ressources (l'entrepôt de données), une réponse rapide nécessite un pré-traitement des données, c'est-à-dire la construction de structures de données adaptées. Indépendamment du temps requis pour ce pré-traitement, le facteur prépondérant devient l'espace mémoire nécessaire au stockage des structures de données. Dans cette thèse, nous revisitons des problèmes classiques mais en donnant la priorité à l'utilisateur, ce qui est original pour les requêtes OLAP. Bien entendu, l'objectif sera de minimiser l'espace mémoire requis mais avec des garanties de performances sur le temps de réponse.

Problèmes et motivations

Les entrepôts de données contiennent de larges volumes de données. Idéalement, l'utilisateur désire que les requêtes OLAP soient satisfaites dans des temps de l'ordre de la

seconde. Les techniques qui ont pour but d'augmenter la vitesse de réponse à une requête sont appelées techniques d'optimisation de requêtes. La technique la plus efficace est la matérialisation des agrégats ou cuboïdes ou vues. Selon Kimbal et al. [KRT+07] : "Les agrégats sont le moyen le plus efficace pour optimiser les performances dans les grands entrepôts de données". Ces agrégats peuvent être soit pré-calculés ou calculés à la volée.

Décrivons brièvement le mécanisme de réponse à une requête. Les requêtes OLAP sur une table de faits T peuvent être présentées en SQL sous la forme :

`SELECT <Attributs>, <Mesures> FROM` $T, D_1, \ldots, D_d$ `WHERE <condition> GROUP BY <Attributs>`.

Les conditions sont des conjonctions de prédicats de la forme soit $T.K_i = D_i.K_i$ (opération de jointure) ou $D_i.A_i =< valeur >$ (opération de sélection). Bien entendu, il est possible de répondre à toute requête directement à partir de la table de fait mais avec un temps proportionnel à la taille de la table de fait. Si tous les cuboïdes [1] sont pré-calculés et stockés, le temps de réponse devient alors proportionnel à la taille de la réponse. Ce modèle de coût simple s'avère réaliste [HRU96] même si les accès disque et à la mémoire cache ne sont pas considérés. Pour résumer, à première vue, il y a deux choix possibles de matérialisation :

- le cube de données est matérialisé en entier. Les inconvénients majeurs sont l'espace de stockage, son temps de pré-calcul ainsi que le temps de mise à jour dans un cadre de données dynamiques. En effet, le nombre des cuboïdes croît exponentiellement par rapport au nombre des dimensions de la table de faits. Par exemple, nous disposons d'un jeu de données issues de fouilles archéologiques [2] où le nombre de lignes de la table de faits est égal à 8300 et le nombre de dimensions est de 13. La taille approximative de cette table est d'1 Mo. La taille du cube de données entier (les 2^{13} cuboïdes) occupe approximatif 2 Go. En pratique les tables de faits sont bien plus grandes. Le cube de données ne tient pas en mémoire vive. L'avantage est que le temps de réponse aux requêtes sera minimal.
- rien à matérialiser : nous n'avons pas besoin de gérer le cube de données, mais la performance de réponse aux requêtes est mauvaise car chaque requête revient à réaliser la jointure de la table de fait avec un ensemble de tables de dimensions et l'on sait très bien que l'opération de jointure est l'une des plus, si ce n'est la plus, coûteuses de l'algèbre relationnelle.

1. augmentés de toutes les fonctions d'index pour les opérations de sélection.
2. Nous remercions N. Novelli pour nous avoir transmis ces données.

Principales mesures de performances En pratique, matérialiser une partie de ce cube peut suffire. De nombreuses solutions proposent de matérialiser seulement un sous-ensemble de cuboïdes $\mathcal{S} \subseteq \mathcal{C}$: les *algorithmes de sélections de vues* (ASV). La nuance entre "vues" et "cuboïdes" vient du fait qu'il est tout à fait envisageable de ne matérialiser que des parties de cuboïdes. Afin de comparer les différents ASV, les principales mesures de performances sont traditionnellement :

- l'*espace mémoire* supplémentaire nécessaire pour stocker le cube de données partiel $\mathcal{S}$;
- le *temps de calcul* de $\mathcal{S}$ et parfois de mise à jour du cube de données partiel dans le cas dynamique ;
- le *temps de réponse* aux requêtes obtenu à partir de $\mathcal{S}$.

Stocker un ensemble $\mathcal{S}$ de petite taille permet d'accélérer le temps de réponse de la manière suivante : s'il existe un cuboïde $s \in \mathcal{S}$ dont les attributs sont un sur-ensemble des attributs de la requête q (l'union des attributs présents dans les clauses `SELECT` et `GROUP BY`), alors il est possible de répondre à q à partir de s. Supposons par exemple que $s =$"produit,trimestre". Si la requête porte sur l'attribut "trimestre", la réponse peut être faite aussi bien à partir du cuboïde de base (coût de réponse 9 car celui-ci contient 9 lignes, c.f figure I.2) que de s (coût de réponse 8 puisqu'il y 8 lignes dans ce cuboïde). Cet exemple illustre les possibilités de réduction du temps de réponse.

Plus formellement, on peut définir à partir du cube de données complet un ensemble partiellement ordonné dont les éléments sont les cuboïdes ordonnés par l'inclusion entre dimensions. On note par $Att(c)$ l'ensemble des attributs de cuboïde c et par $|Att(c)|$ le nombre des éléments de cette ensemble. Pour deux cuboïdes c_1 et c_2, on note $c_1 \preceq c_2$ si et seulement si $Att(c_1) \subseteq Att(c_2)$. Dans ce cas, c_2 est un *ancêtre* de c_1. De plus, si $|Att(c_2)| = |Att(c_1)| + 1$ alors c_2 est un *parent* de c_1. Comme nous ne considérons que des fonctions distributives sur les mesures des cuboïdes, si $c_1 \preceq c_2$ alors c_1 peut être calculé à partir de c_2. Dans notre modèle de coût, le temps de réponse de c_1 à partir de c_2 correspond à la taille de c_2. De manière générale, pour un ensemble fixé $\mathcal{S}$ de cuboïdes matérialisés, le temps de réponse d'une requête q à partir de $\mathcal{S}$ correspond au nombre d'entrées du plus petit cuboïde $q' \in \mathcal{S}$ ancêtre de q. Dans la thèse, nous proposons comme mesure de performance *le facteur de performance* $pf(q, \mathcal{S})$ de $\mathcal{S}$ par rapport à q et qui correspond au ratio de taille entre q et q'. Si q est matérialisé, alors $pf(q, \mathcal{S}) = 1$. Dans l'exemple précédent, $pf(q, q') = 8/7$.

La décision de matérialiser une partie de cube peut se faire en utilisant deux types de contraintes :

– *des contraintes orientées système* :
  – l'espace disponible pour stocker $\mathcal{S}$ est fixé : [HRU96, SDN98, TCFS08, AJD06, MAD06, GM05, LTCF05, BB03]
  – en limitant le temps de maintenance : [LWO01, GM05]
– *des contraintes orientées utilisateur* :
  – en limitant le temps de réponse aux requêtes : [HMT09b, HMT09d, HMT09c]

Par rapport à ces deux types des contraintes nous distinguons respectivement deux types d'approches dans un ASV : **l'approche orientée système** et **l'approche orientée utilisateur**. Notre travail s'inscrit dans la dernière approche et, à notre connaissance, elle constitue le premier travail qui prend comme contrainte utilisateur le temps de réponse aux requêtes.

Bon nombre d'approches considère comme connue la fréquence d'interrogations des cuboïdes. Ainsi, pour les 2^d cuboïdes, on peut associer la charge ou la fréquence des requêtes $w_1, w_2, \ldots, w_{2^d}$. Les fréquences peuvent être observées sur une échelle de temps courte ou longue. Dans certaines conditions, on peut être intéressé par optimiser seulement un sous-ensemble de requêtes, appelé *requêtes cibles* $\mathcal{Q}$.

Limites des solutions actuelles Comme remarqué dans [MKIK07], la sélection des vues à matérialiser est un problème multi-critères : la granularité de la matérialisation (vues correspondant à des fragments ou des cuboïdes complets), les contraintes prises en compte (la mémoire disponible et/ou le temps de mise à jour), la présence ou non de requêtes cibles pouvant être définies à partir des fréquences de requêtes, la complexité des algorithmes et la possibilité d'utiliser les index. La notion de "meilleure solution" est donc relative. Les inconvénients majeurs des solutions actuelles sont les suivants :

– *aucune garantie de performance à court terme* : dans certains contextes (Web, cloud computing, ...), il est très important que le temps de réponse pour n'importe quelle requête soit le plus faible possible. Les utilisateurs sont prêts à payer pour obtenir une qualité de service. Cependant, en pratique, un temps de réponse inférieur à quelques centaines de millisecondes ne peut pas être atteint. En théorie de l'information, une requête ne peut pas prendre un temps inférieur à celui nécessaire pour énumérer la réponse si cette dernière est déjà calculée. La plupart des propositions pour le problème de la sélection de vues retourne des solutions dont l'objectif est de minimiser le *temps moyen des requêtes* en satisfaisant les contraintes de mémoire ou temps de mise à jour. Minimiser le temps moyen des requêtes, ou de manière équivalente le temps total, peut mener à des solutions pour lesquelles le temps de réponse

est optimisé pour certaines requêtes mais pas pour d'autres. Bien que les solutions existantes soient globalement raisonnables, elles ont une mauvaise *performance de crête* : la séquence des requêtes n'est en général pas connue à l'avance et il se peut que des requêtes individuelles mal optimisées soient effectuées. Optimiser chaque requête cible permet d'optimiser également le temps moyen des requêtes alors que l'inverse n'est pas vrai. Dans notre vision, l'administrateur a la possibilité de donner un facteur de performance personnalisé pour chaque requête.

– *l'estimation de la taille des cuboïdes est souvent négligée* : dans bon nombre de solutions de la littérature, l'entrée d'un ASV contient la taille des cuboïdes. Lorsque d devient assez grand, cette hypothèse n'est pas réaliste. D'autre part, l'estimation rapide de la taille d'un cuboïde s'avère très grossière : on considère souvent à tort la taille maximale d'un cuboïde (produit des cardinalités des cuboïdes monodimensionnels) ou que les valeurs d'attributs de données suivent des distributions aléatoires uniformes.

Il en résulte que **la grande majorité des solutions existantes n'offrent pas de garantie de performances en termes d'espace mémoire ou de temps de réponse, y compris sur des jeux de données statiques**. La figure I.5 illustre notre vision sur les systèmes dynamiques d'optimisation de requêtes par matérialisation dans l'approche orientée utilisateur. Nous considérons que l'administrateur a la possibilité de fixer le facteur de performance pour toutes les requêtes.

Le fonctionnement de ce système est le suivant : les requêtes entrant dans le système sont traités par la Table de Faits T avec les Cuboïdes Matérialisées $\mathcal{S}$. En même temps, le système extrait les Cuboïdes Cibles Q de cette requête. Si le facteur de performance f_{out} d'une requête ou de plusieurs requêtes est plus grand que le facteur de performance de référence f_{in}, le Comparateur de Performance lance une exécution de l'Algorithme de Sélection de Cuboïdes à Matérialiser, qui va produire une nouvelle solution $\mathcal{S}$ matérialisée par l'Algorithme de Matérialisation.

La sélection des cuboïdes cibles est laissée à la libre appréciation de l'administrateur. On peut raisonnablement penser que les cuboïdes cibles sont les cuboïdes les plus fréquemment questionnés mais on peut considérer tout autre choix.

Challenges Dans notre vision orientée utilisateur, l'enjeu porte sur les caractéristiques suivantes :

– *temps de pré-calcul* : comment limiter le nombre d'appels à la fonction calcul taille qui s'avère coûteuse si on désire des garanties de performances ?

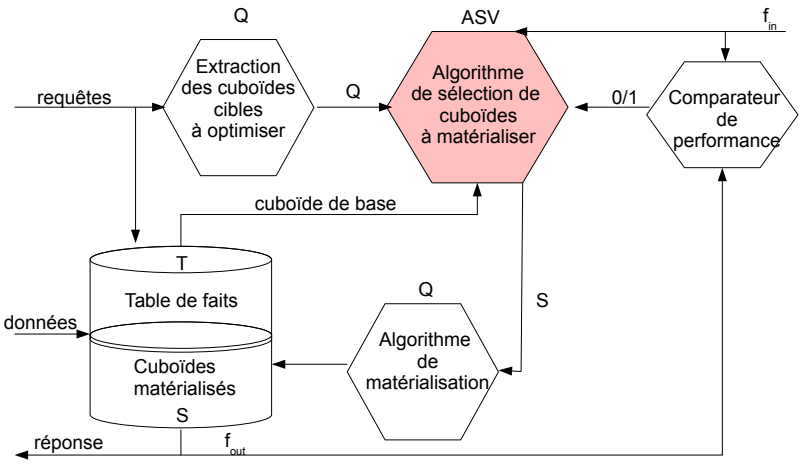


Figure I.5 – Système dynamique d'optimisation de requêtes par matérialisation.

- *espace mémoire* : comment obtenir des garanties en espace mémoire pour $\mathcal{S}$ lorsque les facteurs de performances de cuboïdes cibles sont fixés ?
- *stabilité des solutions* : comment limiter l'impact de la dynamique des données sur le recalcul de solutions $\mathcal{S}$ ou le rafraîchissement des cuboïdes matérialisés. En effet, on ne désire pas qu'un recalcul soit lancé à la moindre mise à jour des données.

Calcul parallèle des fréquents maximaux Nos études sur les ASV ont mené à une revisite sur le calcul rapide de bordures. Les bordures [MT97] correspondent également aux ensembles fréquents maximaux définis par [MT97] dans les bases de transactions [AIS93]. Le temps de pré-calcul de bordures est critique dans nos ASV aussi bien que dans le contexte des fréquents maximaux. Informellement, un ensemble d'items est σ-fréquent s'il apparaît au moins σ fois dans une liste d'ensembles d'items.

L'objectif est de trouver les ensembles d'items dont la fréquence, appelée aussi *support*, est supérieure à σ. Comme le nombre des fréquents maximaux peut être trop grand, de nombreuses restrictions ont été proposées afin de rendre le résultat plus lisible par les utilisateurs. L'une des classes d'items fréquents les plus populaires est celle des fréquents maximaux. Un ensemble d'items est *maximal* si aucun de ses sur-ensembles n'est fréquent. Par la propriété d'antimonotonie, chaque sous-ensemble d'un ensemble d'items fréquents est aussi fréquent. L'ensemble des items fréquents maximaux (IFM) est aussi

appelé *bordure positive*.

Les applications des IFM sont nombreuses. Par exemple, dans la gestion de réseaux, on peut être intéressé par la découverte des longs chemins les plus fréquemment utilisés entre routeurs (correspondant aux items) lorsque pour chaque routage (transaction individuelle) la liste des routeurs traversés (ensembles fréquents) est connue.

Cependant, le calcul des IFM est une tâche très coûteuse en temps et en mémoire. Il est donc naturel d'essayer d'accélérer le calcul des bordures avec l'aide de stations multi-cœurs ou de calcul distribué. Les ordinateurs actuels et futurs comportent de plus en plus de cœurs. Ceci laisse envisager de manipuler de plus grosses quantités de données ou de faire des calculs en temps réel afin d'interagir efficacement avec les données. Cependant, il n'est pas certain que le calcul parallèle de bordures soit plus rapide sur plusieurs cœurs que sur un seul. Les raisons sont nombreuses :

- *le surcoût du calcul parallèle* : la nécessaire synchronicité et délais de communication entre threads, l'usage de la mémoire partagée, ...
- *le découpage en tâches indépendantes* : pour être efficace, un algorithme parallèle est basé sur une décomposition de l'ensemble des tâches en tâches indépendantes. Cela suppose qu'aucun calcul inutile, non effectué en séquentiel, ne soit fait en parallèle. D'autre part, il n'est pas évident d'équilibrer la charge de travail entre les différents entités de calculs, sans quoi des coeurs peuvent être plus chargés que d'autres.

Il existe des solutions de calcul parallèle de bordures qui découpent les tables de transactions en morceaux, répartissent le calcul puis assemblent les résultats. Informellement, si un ensemble d'items n'est fréquent dans aucun morceau, il n'est pas fréquent. Seulement, il n'est absolument pas évident que cette démarche soit toujours efficace et aucune garantie de performance n'est assurée.

Contribution de la thèse

Le concept clé de la thèse est celui de *bordure*. Pour le problème de sélection de cuboïdes à matérialiser, un cuboïde appartient à la bordure si sa taille est inférieure à un seuil fixé et la taille de ses parents dépasse ce seuil. Pour le problème de calcul de fréquents maximaux, un itemset bordure ou itemset fréquent maximal est l'itemset qui a un support plus grand que σ et tous ses sur-ensembles ne sont pas σ-fréquents.

Plus formellement, on peut définir les bordures à partir d'un ensemble partiellement ordonné et d'une fonction de poids monotone. Soit E un ensemble et $\mathbb{P}(E)$ l'ensemble de

ses parties. On considère l'ensemble partiellement ordonné $(\mathbb{P}(E), \subseteq)$. Pour tout $X \in \mathbb{P}(E)$, $w(X)$ est une fonction monotone retournant un nombre réel, c'est-à-dire si $X \subset X'$ alors $w(X) \leq w(X')$. Dans le cas des cubes de données, w désigne la taille d'un cuboïde et dans le contexte des fréquents maximaux, w correspond à l'inverse du support.

Pour f fixé, on note $F_f = \{X \in \mathbb{P}(E) |\ w(X) \leq f\}$. La bordure se définit de la manière suivante :

Définition 1 (Bordure)**.** *La f-bordure (positive) de $(\mathbb{P}(E), \subseteq)$ est notée $\mathcal{B}_f = \{X \in F_f | \forall Y \supset X, Y \notin F_f\}$. La f-bordure négative est notée $\mathcal{B}_f^- = \{X \in \mathbb{P}(E) \setminus F_f |\ \forall Y \subset X, Y \in F_f\}$*

Même si un travail important a été mené pour concevoir des algorithmes de sélection de vues sous contraintes orientées système (d'espace ou de maintenance), nous n'avons pas remarqué l'utilisation de contraintes orientées utilisateur. Pour cette raison, nous proposons dans ce rapport une nouvelle stratégie de sélection de vues qui prend en compte les besoins de l'utilisateur. L'importance d'une stratégie orientée utilisateur a été soulignée dans l'article [FK09].

Toutes nos solutions sont développées en ayant en vue l'idée d'avoir une garantie théorique d'un paramètre choisi. Dans cet esprit, nous donnons les résultats suivants :

- une solution au problème de sélection des vues à matérialiser, dans le cas où l'ensemble de requêtes cibles $\mathcal{Q}$ contient tous les cuboïdes de $\mathcal{C}$, cas que nous appelons *sans cible*. Notre technique est basée sur une construction itérative de bordures. Pour chaque requête de $\mathcal{Q}$, nous fixons le facteur de performance désiré.
- une solution de sélection de vues à matérialiser, dans le cas où $\mathcal{Q}$ contient un sous-ensemble de $\mathcal{C}$, cas que nous appelons *avec cible*. Nous énonçons des propriétés utiles pour le contexte dynamique des données.
- une proposition d'un algorithme séquentiel de calcul de bordures, aisément parallélisable avec des garanties de performances en parallèle. En effet, nous pouvons partitionner l'ensemble des opérations en tâches indépendantes. Nous avons appliqué cet algorithme sur le calcul d'itemsets fréquents maximaux sur une machine multi-cœurs afin de vérifier si notre modèle théorique est effectivement pertinent.

Voici quelques détails sur les différentes contributions.

Sélection de vues à matérialiser avec bordures - cas sans cible

Comme le facteur de performance est fixé pour chaque élément de $\mathcal{Q}$, l'objectif est de minimiser l'espace mémoire en garantissant le facteur de performance. Notre algorithme, nommé *PickBorders*, est une variante de parcours en profondeur dans le treillis du cube de données. Il est économique en mémoire car il ne nécessite qu'une mémoire vive ayant au moins la taille de la bordure que l'on veut calculer. Nous avons comparé *PickBorders* avec deux autres algorithmes classiques conçus dans l'approche traditionnelle : HRU proposé par Harinarayan et al. dans [HRU96] et PBS proposée par Shukla et al. dans [SDN98]. Les expériences montrent le bon comportement de PickBorders par rapport aux mesures de temps de pré-calcul et mémoire d'un côté et facteur de performance de l'autre.

Sélection de vues à matérialiser - cas avec cible

Ce cas est plus général que celui du paragraphe précédent. Nous proposons une solution exacte basée sur la programmation linéaire en nombres entiers et plusieurs solutions approchées. Dans les deux cas, le facteur de performance est toujours l'entrée de nos algorithmes. La solution exacte prend un temps à être trouvée plus grande que celle de la solution approchées et nécessite une estimation précise, donc coûteuse, de la taille des cuboïdes. Cependant, elle permet de faire des comparaisons précises avec les algorithmes approchés. Pour obtenir des algorithmes approchés, nous proposons une technique de réduction de l'espace de recherche en utilisant les bordures et la construction de graphes de recherche. Ainsi, nous obtenons des algorithmes de complexité polynomiale et retournant des ensembles de vues $\mathcal{S}$ dont la taille mémoire est au plus $1 + ln|\mathcal{Q}|$ fois celle de la mémoire minimale. Si f est le facteur de performance maximal fixé, l'approximation augmente seulement à $f(1 + ln|\mathcal{Q}|)$ en réduisant le graphe de recherche partiel. Différentes expérimentations illustrent la pertinence de notre approche. Nous pouvons manipuler les cuboïdes cibles contenant plusieurs milliers d'éléments alors que les approches existantes ne manipulent que quelques dizaines d'éléments.

Nous présentons par la suite la propriété de stabilité de nos solutions. Ceci nous permet de prendre en considération le caractère dynamique des données ainsi que des cibles $\mathcal{Q}$. Pour mesurer la "résistance" du système par rapport aux opérations d'insertion ou de suppression, nous utilisons le concept de stabilité de solution $\mathcal{S}$, c'est-à-dire un re-calcul de $\mathcal{S}$ n'est nécessaire que si le facteur de performance est modifié de plus d'une unité.

Dans cette partie, nous donnons des pistes pour généraliser nos constructions aux

requêtes incluant la sélection. Cette intégration se fait essentiellement par un léger changement du modèle de coût d'une requête.

Des expérimentations viennent compléter notre étude. Elles montrent que nos solutions approchées sont très compétitives en terme de mémoire par rapport aux solutions optimales.

Calcul parallèle des bordures et applications au calcul des IFM

Les stratégies basées sur les parcours en profondeur sont parmi les meilleurs pour calculer les bordures et les ensembles d'items fréquents maximaux.

Cette stratégie est cependant plus difficile à rendre parallèle car elle peut aboutir à un nombre exponentiel d'étapes, appelés aussi *rondes* en algorithmique distribuée, correspondant au nombre de lectures de la table de transactions. Pour une table de transactions définie sur n items, nous proposons une solution pour partitionner les ensembles d'items en $2n-1$ rondes. Chaque ronde se décompose en deux phases : le calcul des fréquences ou des supports d'un ensemble de candidats et la mise à jour des candidats. Nous proposons un nouvel algorithme séquentiel, appelé **MineWithRounds**, qui est basé sur ce partitionnement et qui peut être facilement exécuté dans un environnement multi-threadé. Sous des hypothèses raisonnables (chaque calcul de support prend un temps comparable) et lorsque le temps d'exécution est dominé par le calcul des fréquences, nous pouvons prouver une accélération du temps de calcul d'un facteur p pour une machine à p cœurs. Des expérimentations sur une machine à 8 cœurs hyper-threadé montrent la pertinence de **MineWithRounds**.

Organisation de la thèse

L'organisation du document suit l'ordre des sections précédentes et correspond à l'évolution de notre recherche depuis le début de la thèse. Chaque sujet est détaillé dans un chapitre. Une perspective ouverte par notre travail est présentée à la fin du document. Des versions préliminaires des différents chapitres sont parus dans [HMT09b, HMT09d, HMT09c, HMT09a] et à paraître dans BDA'2010.

Chapitre II

Sélection de vues en absence de cibles

Dans ce chapitre, nous considérons le problème de l'optimisation des requêtes dans le cadre des cubes de données. Plus précisément, nous nous plaçons dans la situation où l'on n'a aucune connaissance sur les requêtes susceptibles d'être posées au cube. Ceci revient donc à considérer que *toutes* les requêtes sont possibles. Cette hypothèse est souvent admise dans le cadre de l'OLAP où il est stipulé que les requêtes d'analyse se font généralement d'une manière imprévisible. Cela étant, nous n'envisageons d'optimiser qu'une famille de requêtes. Plus précisément, nous ne considérons que les requêtes du type JA (Jointure+Agrégation) où chaque table n'apparait qu'une seule fois. Ces requêtes sont donc de la forme :

```
SELECT dimensions, mesures
FROM table de faits T, tables de dimensions Di
WHERE conditions des jointures entre T et les Di's
GROUP BY dimensions
```

Chacune de ces requêtes est équivalente à une requête qui demande à afficher le contenu d'un des cuboïdes du cube de données. Si `Ci` est un cuboïde, alors les requêtes ci-dessous peuvent être réécrites sous la forme :

```
SELECT *
FROM Ci
```

La structure de ce chapitre est la suivante :

- nous présentons le modèle de coût pour les requêtes et la table de faits ainsi que les mesures de performance utilisées

- nous proposons quelques algorithmes (PSC, PTB, PickBorders) dans l'esprit "orienté requête" en présentant d'abord les algorithmes les plus simples
- nous analysons quelques solutions existantes de l'approche "orientée espace" avec un accent particulier sur les algorithmes proposés dans [HRU96, SDN98].
- nous comparons expérimentalement PickBorders à HRU (l'algorithme de [HRU96]) et à PBS (l'algorithme proposée dans [SDN98]) d'un autre côté.

II.1 Préliminaires

Afin de comparer les algorithmes, nous utilisons le même modèle de coût que dans [HRU96]. Il ne tient pas compte des accès disque, mais seulement du nombre d'entrées dans les cuboïdes qui sont utilisées. Dans ce qui suit, nous donnons les définitions nécessaires pour le chapitre.

Définition 2 (Taille d'un cuboïde)**.** *La* taille d'un cuboïde c *est exprimée par le nombre de lignes de ce cuboïde. Nous la notons* $size(c)$.

Définition 3 (Taille d'un ensemble de cuboïdes)**.** *Soit* S *un ensemble de cuboïdes. La* taille de S *est exprimée par les nombres de lignes cumulés de ses cuboïdes. Ainsi,* $size(S) = \sum_{c \in S} size(c)$.

Définition 4 (Dépendance entre cuboïdes, cuboïde parent, cuboïde ancêtre)**.** *Soient deux cuboïdes :* c_1 *qui contient l'ensemble d'attributs* A_1 *avec* $|A_1| = a_1$ *et* c_2 *avec l'ensemble des attributs* A_2 *avec* $|A_2| = a_2$ *attributs.* c_1 dépend *de* c_2*, on note* $c_1 \preceq c_2$*, ssi* c_1 *peut être calculé à partir de* c_2*. Dans le cas des mesures algébriques,* $c_1 \preceq c_2 \Leftrightarrow A_1 \subseteq A_2$*. Nous disons que* c_2 *est* plus détaillé/moins généralisé *que* c_1 *et* c_1 *est* moins détaillé/plus généralisée *que* c_2*. Si* c_1 *dépend de* c_2 *alors* c_2 *est un ancêtre de* c_1*. De plus, si* $a_2 - a_1 = 1$ *alors* c_2 *est un parent de* c_1*.*

Définition 5 (Ancêtres d'un cuboïde)**.** *Soit* S *un ensemble de cuboïdes. On note par* S_c *l'ensemble des ancêtres du cuboïde* c *qui se trouvent dans* S *i.e.,* $S_c = \{c' \in S | c \preceq c'\}$

Exemple 1. *Dans la figure II.2, les cuboïdes bi-dimensionnels (produit,trimestre), (produit, pays) et (trimestre,pays) peuvent être calculés à partir du cuboïde tri-dimensionnel*

Table de dimension: Trimestre

Cod trim.	Trimestre
1	Trim1
2	Trim2
3	Trim3
4	Trim4

Table de fait: Vends

Dimensions			Mesure
Cod Produit	Cod Trim.	Cod Pays	Quantite
3	1	3	30
1	2	2	25
2	1	1	30
1	3	3	40
3	2	2	10
2	2	2	30
1	1	3	20
1	2	2	10
2	4	2	10

Table de dimension: Produit

Cod produit	Produit
1	PC
2	TV
3	VCR

Table de dimension: Pays

Cod pays	Pays
1	Canada
2	USA
3	Mexique

Figure II.1 – Schéma en étoile sans hiérarchie

Group–by par 3 dimensions
Fc d'aggregation: somme

Dimensions			Mesure
Produit	Trim.	Pays	Quantite
VCR	Trim1	Mexique	30
PC	Trim2	USA	35
TV	Trim1	Canada	30
PC	Trim3	Mexique	40
VCR	Trim2	USA	10
TV	Trim2	USA	30
PC	Trim1	Mexique	20
TV	Trim4	USA	10

Group–by par 2 dimensions
Fc d'aggregation: somme

Produit	Trim.	Quantite	Produit	Pays	Quantite	Trim.	Pays	Quantite
VCR	Trim1	30	VCR	Mexique	30	Trim1	Mexique	50
PC	Trim2	35	PC	USA	35	Trim2	USA	75
TV	Trim1	30	TV	Canada	30	Trim1	Canada	30
PC	Trim3	40	PC	Mexique	60	Trim3	Mexique	40
VCR	Trim2	10	VCR	USA	10	Trim4	USA	10
TV	Trim2	30	TV	USA	40			
PC	Trim1	20						
TV	Trim4	10						

Group–by par 1 dimension
Fc d'aggregation: somme

Produit	Quantite	Trim.	Quantite	Pays	Quantite
VCR	40	Trim1	80	Mexique	90
PC	95	Trim2	75	USA	85
TV	70	Trim3	40	Canada	30
		Trim4	10		

Group–by par 0 dimension
Fc d'aggregation: somme

Quantite
205

Figure II.2 – Les tables des cuboïdes pour la table de faits de la figure II.1

(produit, trimestre, pays). Dans ce cas, le cuboïde tri-dimensionnel est parent de tous les cuboïdes bi-dimensionnels et il est ancêtre de tous les cuboïdes. Dans un cube de données, le cuboïde de base est le cuboïde le plus détaillé et le cuboïde apex est le cuboïde le moins détaillée.

Définition 6 (Coût d'un cuboïde)**.** *Le* coût d'un cuboïde c *par rapport à un ensemble matérialisé de cuboïdes* $\mathbb{S}$*, noté par* $cost(c, \mathbb{S})$ *est défini comme suit :*

- *si* $\mathbb{S}$ *ne contient aucun ancêtre de* c *alors* $cost(c, \mathbb{S}) = \infty$ *(cela signifie que l'on ne peut pas calculer* c *à partir de* $\mathbb{S}$*),*
- *sinon* $cost(c, \mathbb{S}) = \min_{w \in \mathbb{S}_c} size(w)$*. Autrement dit, c'est la taille de son plus petit ancêtre se trouvant dans* $\mathbb{S}$*.*

Définition 7 (Coût d'un ensemble de cuboïdes)**.** *Le* coût d'un ensemble de cuboïdes S *par rapport à une autre ensemble de cuboïdes matérialisés* $\mathbb{S}$ *est défini par* $cost(S, \mathbb{S}) = \sum_{c \in S} cost(c, \mathbb{S})$*.*

- *Si* $S = \mathbb{S}$*, alors* $cost(S, \mathbb{S})$ *est le coût minimal que l'on puisse obtenir pour* S *et il est égal à* $size(S)$*.* $MinCost(S) = size(S)$*.*
- *Si* $S = \mathbb{C}$*,* $MinCost(S)$ *est simplement noté* $MinCost$*.*
- *On définit le coût maximal de* S *par* $MaxCost(S) = cost(S, \{c_b\})$*. Ceci revient à calculer tout élément de* S *à partir du cuboïde qui a la plus grande taille.*
- *Si* $S = \mathbb{C}$*,* $MaxCost(S)$ *est simplement noté* $MaxCost$*.*

Étant donné $\mathbb{S}$ un ensemble de cuboïdes matérialisés, pour évaluer la performance de $\mathbb{S}$ par rapport à une requête (ou un cuboïde) c, nous considérons le ratio entre le coût de c relativement à $\mathbb{S}$ et le coût minimal de c qui est en fait la taille de c. Plus précisément,

Définition 8 (facteur de performance)**.** *Soient* $\mathbb{S}$ *un ensemble de cuboïdes matérialisés et* c *un cuboïde. Le facteur de performance de* $\mathbb{S}$ *relativement à* c*, noté* $pf(c, \mathbb{S})$ *est défini par* $\frac{cost(c, \mathbb{S})}{size(c)}$*.*

Exemple 2. *Afin d'illustrer les définitions précédentes nous considérons le graphe de la Figure II.3. Il représente le treillis du cube de données obtenu d'une table de faits* T

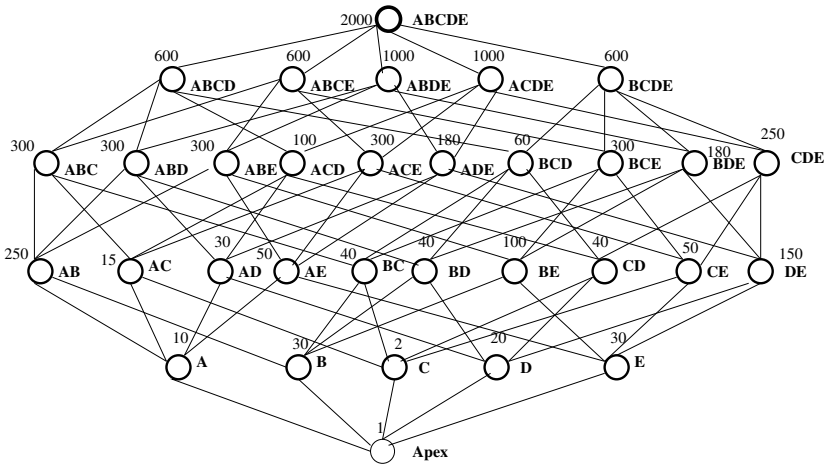


Figure II.3 – Diagramme de Hasse d'un cube de donnée

dont les dimensions sont A, B, C, D et E. Chaque nœud est un cuboïde étiqueté avec ses dimensions et sa taille. Le plus haut cuboïde est le cuboïde de base c_b et le plus bas cuboïde est le cuboïde apex.

Comme nous travaillons, dans le cas "sans cible", $\mathcal{Q} = \mathcal{C}$.

Le coût minimal correspond au cas où l'ensemble des cuboïdes matérialisés $\mathcal{S}$ est égal à $\mathcal{Q}$. Dans l'exemple, $MinCost = \sum_{i=1}^{2^5} size(c_i) = 8928$.

D'un autre côté, le coût maximal correspond à la situation où $\mathcal{S} = \{c_b\}$, c'est-à-dire seul le cuboïde de base est matérialisé. Dans ce cas, chaque cuboïde de $\mathcal{Q}$ est calculé à partir du cuboïde de $ABCDE$. Ainsi, $MaxCost = 2^5 * size(ABCDE) = 32 * 2000 = 64000$. Noter que c'est la quantité minimale de mémoire nécessaire pour pouvoir calculer toutes les requêtes de $\mathcal{Q}$.

Supposons maintenant que $\mathcal{S} = \{ABCDE, BE\}$. Les mesures de performance de $\mathcal{S}$ sont les suivantes : L'espace mémoire utilisé par $\mathcal{S}$ est $size(\mathcal{S}) = size(ABCDE) + size(BE) = 2000 + 100 = 2100$. Le coût pour évaluer toutes les 2^5 requêtes possibles est calculé de la manière suivante. D'abord, nous considérons le cuboïde matérialisé BE. Il est utilisé pour calculer les cuboïdes BE, B, E et apex car c'est leur plus petit ancêtre matérialisé. Toutes les autres requêtes sont calculées à partir de $ABCDE$. Ainsi, $cost(\mathcal{Q}, \mathcal{S}) = 4 * size(BE) + 28 * size(ABCDE) = 56400$.

Nous considérons les requêtes BE et BC. Leurs facteurs de performances respectifs

relativement à $\mathcal{S}$ *sont* $pf(BE, \mathcal{S}) = \frac{cost(BE,\mathcal{S})}{size(BE)} = 100/100 = 1$ *et* $pf(BC, \mathcal{S}) = \frac{cost(BC,\mathcal{S})}{size(BC)} = size(ABCDE)/40 = 2000/40 = 50$. *Ça signifie qu'en stockant l'ensemble des cuboïdes* $\{ABCDE, BE\}$, *le coût d'évaluation de* BE *est exactement le coût minimal, mais pour évaluer* BC, *le coût est 50 fois le coût minimal.*

II.2 Formalisation de notre approche

Dans ce chapitre, nous traitons le problème suivant :

Soit $\mathcal{Q}$ *l'ensemble des requêtes qui correspondent à tous les cuboïdes d'un cube* $\mathcal{C}$. *Étant donné un nombre réel* $f \geq 1$, *trouver un ensemble de cuboïdes* $\mathcal{S}$ *de taille minimale tel que pour tout cuboïde* $c \in \mathcal{Q}$, $cost(c, \mathcal{S}) \leq f * size(c)$.

Autrement dit, nous autorisons l'utilisateur à fixer un seuil pour le facteur de performance relativement aux requêtes puis nous cherchons le plus petit ensemble (celui-ci peut ne pas être unique) de cuboïdes en terme de taille totale permettant de garantir que l'évaluation de chacune des requêtes possibles ne prenne pas plus que f fois le temps minimal pour l'évaluer.

L'algorithme *naïf* pour résoudre ce problème consiste à considérer tous les sous-ensembles $\mathcal{S} \in 2^{\mathcal{C}}$ comme candidats. Pour chacun de ces $\mathcal{S}$ et pour chaque $q \in \mathcal{C}$, si $pf(q, \mathcal{S}) > f$ alors $\mathcal{S}$ n'est pas candidat. Parmi les candidats restants, il reste à prendre un ensemble $\mathcal{S}$ qui ait la taille minimale. Bien sûr, cet algorithme n'est pas efficace de par sa complexité. Dans le chapitre suivant on va voir comment ce problème peut être vue comme une réduction de problème *minimal weighted set cover* (MWSC) qui est NP-complet [Kar72]. Ainsi, une approximation est nécessaire.

II.3 Les algorithmes de notre approche

Dans cette section, nous présentons quelques techniques de sélection d'un sous-ensemble de cuboïdes qui doivent être stockés. Pour chaque technique, nous analysons sa complexité et étudions ses garanties de performance. Nous donnons une première solution à ce problème. Même si elle est plutôt inefficace, c'est un bloc de construction pour la suite. Elle consiste simplement à stocker tous les cuboïdes dont la taille est inférieure à la taille du cuboïde de base divisée par f.

II.3.1 L'algorithme PSC

Étant donné f, un cuboïde est qualifié de *petit* si sa taille est inférieure à $size(c_b)/f$. En matérialisant les *petits* cuboïdes on garantit une bonne performance lors de l'interrogation du cube de données.

Lemme 1. *Soit* $f \geq 1$, $m = size(c_b)$ *et* $\mathcal{S} = \{c \in \mathcal{C} : size(c) \leq m/f\} \cup \{c_b\}$ *alors pour tout* $c \in \mathcal{C}$, $cost(c, \mathcal{S}) \leq f * size(c)$. *Autrement dit,* $pf(c, \mathcal{S}) \leq f$.

Démonstration. La preuve est immédiate. Pour tout $c \in \mathcal{S}$, c est matérialisé donc il est calculé à partir de lui même. Ainsi, on a $cost(c, \mathcal{S}) = size(c)$, d'où $pf(s, \mathcal{S}) = 1$. Pour tout $c \in \mathcal{C} \backslash \mathcal{S}$, c est calculé à partir de c_b. Ainsi, $cost(c, \mathcal{S}) = m$ et $pf(c, \mathcal{S}) = \frac{size(c_b)}{size(c)}$. Comme $size(c_b) = m$ et $size(c) > m/f$, on en déduit que $pf(c, \mathcal{S}) \leq f$. □

Exemple 3. *Soit le cube de données représentées sur la figure II.4. C'est le même cube*

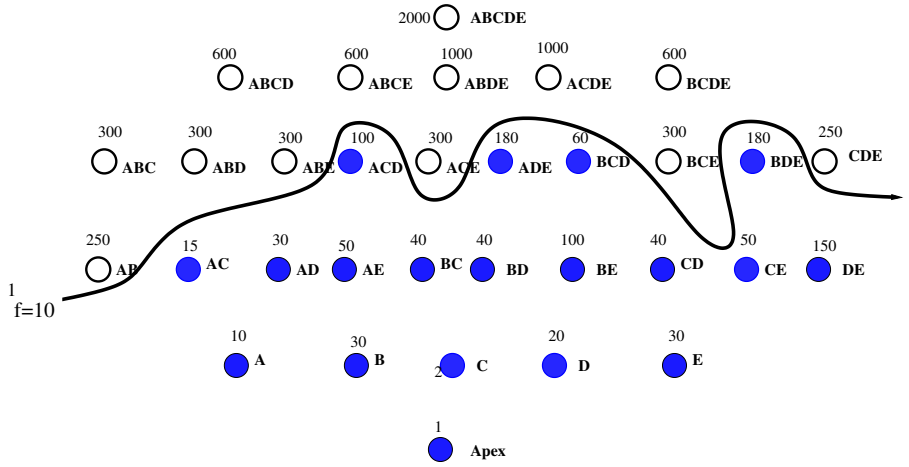


Figure II.4 – Résultat de PSC avec $f = 10$.

de données que celui de la figure II.3. Nous avons supprimé les arêtes entre les nœuds pour le rendre plus compréhensible. En considérant $f = 10$, *tous les cuboïdes en dessous de la courbe sont les cuboïdes dont la taille est inférieure à* $size(ABCDE)/10$. *Il s'agit donc de stocker ces derniers.*

Nous considérons ici deux situations : soit nous disposons de la taille de tous les cuboïdes soit celle-ci nous est inconnue. Dans le premier cas, le calcul de $\mathcal{S}$ est assez simple. En effet, il suffit de trier les cuboïdes selon leur taille, puis choisir ceux dont la taille est

inférieure à m/f. Dans le deuxième cas, on propose d'adopter un algorithme *à la* A priori [AS94, MTV94] pour retrouver les cuboïdes de $\mathcal{S}$. En effet, la condition $size(c) \leq m/f$ est anti-monotone i.e., $c \preceq c'$ et $size(c') \leq m/f$ alors $size(c) \leq m/f$. L'algorithme est décrit par la procédure PSC [1].

Procedure PSC(f, m)

Input: réel f et table de faits T
Output: L'ensemble $\mathcal{S}$ des cuboïdes à matérialiser

```
1  S = ∅;
2  C_1 = {c ∈ C | |Att(c)| = 1};
3  L_1 = {c ∈ C_1 | size(c) ≤ m/f};
4  S = S ∪ L_1;
5  i = 1;
6  while L_i ≠ ∅ do
7      C_{i+1} = { candidats générés de L_i};
8      foreach c ∈ C_{i+1} do
9          size(c)=Calculer_taille(c);
10     L_{i+1} = {c ∈ C_{i+1} | size(c) ≤ m/f};
11     S = S ∪ L_{i+1};
12     i = i + 1;
13 Retourner S;
```

Il s'agit bel et bien de l'algorithme A priori. La procédure *Calculer_size* pourrait être mise en œuvre de deux manières : soit (i) on calcule la taille exacte en utilisant les algorithmes pour le `GROUP BY` qui sont basés sur soit le tri soit le hachage [Gra93] ou bien (ii) on se contente d'une approximation de cette taille en utilisant des techniques telles que celles qui sont décrites dans [AL07] [2]. La complexité maximale de cet algorithme est de 2^d. En effet, si $f = 1$, alors tous les cuboïdes ont une taille inférieure à m/f ainsi la taille de sera calculée. Bien sûr, en pratique f est strictement supérieur à 1 ce qui permet d'avoir une complexité réelle qui est inférieure, voire très inférieure, à 2^d.

Même si $\mathcal{S}$ garantit une qualité pour l'évaluation des requêtes, on voit bien que l'espace mémoire utilisé est loin d'être minimal. Or notre objectif est double : garantir une performance pour les requêtes tout en minimisant l'espace de stockage. Ceci motive notre deuxième solution dans ce qui suit.

1. PSC pour Pick Small Cuboids.
2. Il faut noter ici que l'estimation de la taille d'un cuboïde est un domaine de recherche à part entière qui a fait et qui continue de faire l'objet de nombreux travaux, par exemple le best paper award de la conférence PODS 2010 a été décerné à un article sur ce sujet [KNW10].

II.3.2 L'algorithme PTB

On distingue d'abord entre deux séries de cuboïdes à l'égard de f : ceux pour lesquels nous pouvons réduire le coût maximal en réduisant leur coût par un facteur f et ceux pour lesquels nous y pouvons matérialiser. Il s'avère que, si on stocke les cuboïdes *maximaux* relativement à f, alors nous réduisons le coût maximal de cuboïdes de tous ceux pour qui on applique cette réduction. Donnons d'abord quelques définitions.

Définition 9 (Cuboïde f-réductible)**.** *Soient $c \in \mathcal{C}$, C_c les ancêtres de c dans $\mathcal{C}$, $m = size(c_b)$ et $f \geq 1$. c est* f-réductible *ssi il existe $c' \in C_c$ tel que $size(c') \leq m/f$ et $c' \neq c$.*

En d'autres termes, c est *f-réductible* s'il peut être calculé à partir d'un cuboïde dont la taille est inférieure à celle du cuboïde maximal (i.e., cuboïde de base c_b) divisée par le facteur f.

Exemple 4. *Continuons avec la figure II.6. Le cuboïde $ABDE$ n'est pas 10-réductible car aucun de ses ancêtres n'a une taille inférieure à $size(ABCDE)/10$. Le cuboïde B est quant à lui 10-réductible car au moins l'un de ses ancêtres a une taille inférieure à $size(ABCDE)/10$. En effet, BC est un ancêtre de B et sa taille est 40.*

Maintenant nous définissons les *ensembles f-réductibles*.

Définition 10 (Ensembles de cuboïdes f-réductibles)**.** *Soient $f \geq 1$ et $S \subseteq \mathcal{C}$. On dit que $\mathcal{S}$ est un* ensemble f-réductible *ssi pour tout $c \in \mathcal{S}$, c est* f-réductible.

Exemple 5. *Continuons avec la figure II.6. L'ensemble {A, B, D, AB, BC, BD, CD, DE} est un ensemble 10-réductible puisque chacun de ses éléments a au moins un ancêtre dont la taille est inférieure à 2000/10.*

Ainsi, pour chaque $f \geq 1$, on peut définir *l'ensemble f-réductible maximal*. Il s'agit du sous-ensemble maximal $\mathcal{S}$ de $\mathcal{C}$ tel que $\mathcal{S}$ est *f-réductible*. Étant donné $f \geq 1$, nous cherchons à trouver $\mathcal{S}$ tel que pour chaque *f-réductible* cuboïde c, $cost(c, \mathcal{S}) \leq m/f$. Nous montrons d'abord que *l'ensemble f-réductible maximal* remplit cette condition.

Lemme 2. *Soit $f \geq 1$.* L'ensemble f-réductible maximal *de $2^{\mathcal{C}}$ est $\mathcal{S} = \{c \in \mathcal{C} : size(c) \leq m/f\}$.*

Nous définissons maintenant le concept de cuboïde maximal relativement à un ensemble S.

Définition 11 (Cuboïde maximal)**.** *Soit $f \geq 1$ et $S \subseteq \mathcal{C}$. $c \in \mathcal{S}$ est maximal si il n'y a pas $c' \in \mathcal{S}$ tel que $c \preceq c'$. Nous notons l'ensemble de cuboïdes maximal d'un ensemble $\mathcal{S}$ par $B_{\mathcal{S}} = \{c \in \mathcal{S} | c \text{ est maximal}\}$.*

Autrement dit, c est maximal dans $\mathcal{S}$ s'il n'a pas de parents dans $\mathcal{S}$. La bordure de $\mathcal{S}$ est l'ensemble de ses cuboïdes maximaux. Le lemme suivant montre que si nous ne gardons que la frontière de *l'ensemble f-réductible maximal* alors le coût des cuboïdes *f-réductible* est effectivement réduite.

Lemme 3. *Soit $f \geq 1$ et* $\mathcal{S}$ un ensemble f-réductible maximal. *Alors, pour tout $c \in \mathcal{S}$, $cost(c, B_{\mathcal{S}}) \leq m/f$.*

Démonstration. Chaque $c \in \mathcal{S}$ pourra être calculé à partir d'un élément c' de $B_{\mathcal{S}}$. Comme $size(c') \leq m/f$, alors $cost(c, \mathcal{S}) \leq m/f$. □

Exemple 6. *Continuons avec le cube de données dans la figure II.6. La bordure correspondant à $f = 10$ est l'ensemble $\mathcal{S} = \{ACD, ADE, BCD, BDE, CE\}$. Cet ensemble ainsi que l'ensemble des cuboïdes en dessous de cette frontière est l'ensemble maximal 10-réductible.*

Voici ci-dessous une implémentation possible de l'algorithme PTB [3]. C'est une adaptation de l'algorithme précédent.

La complexité de cet algorithme est $O(d * 2^d)$. En effet, tout au plus, le nombre d'itérations est de d. Dans ce cas, chacun des 2^d cuboïdes est testé s'il est dans la frontière ou non. Comme le nombre maximum de parents de chaque cuboïde est d, on conclut que le nombre maximum de tests est de $d * 2^d$.

La mise en œuvre décrite ci-dessus n'est probablement pas la meilleure. En effet, le problème d'extraction des bordures dans les cubes de données est équivalente à l'extraction des ensembles fréquents maximaux dans les bases de transaction. On pourrait donc utiliser les algorithmes qu'on peut trouver dans la littérature comme par exemple [GZ05, SU03, MT97, GKM+03, Bay98, BCG01].

Avant de clore cette section, nous tenons à faire remarquer au lecteur la différence d'utilisation du facteur f entre les deux propositions précédentes (PSC et PTB). Dans

3. PTB pour Pick The Border.

Procedure PTB(f, m)

Input: réel f et table de faits T
Output: L'ensemble $\mathcal{S}$ des cuboïdes à matérialiser

```
1  S = ∅;
2  C_1 = {c ∈ C | |Att(c)| = 1};
3  L_1 = {c ∈ C_1 | size(c) ≤ m/f};
4  S = S ∪ L_1;
5  i = 1;
6  while L_i ≠ ∅ do
7      C_{i+1} = { candidats générés de L_i};
8      foreach c ∈ C_{i+1} do
9          size(c)=Calculer_taille(c);
10     L_{i+1} = {c ∈ C_{i+1} | size(c) ≤ m/f};
11     foreach c ∈ L_i do
12         E = {c' ∈ L_{i+1} | c ≤ c'};
13         if E = ∅ then
14             S = S ∪ {c};
15     i = i + 1;
16 Retourner S;
```

le premier cas, f est considéré comme "le plus petit" facteur par lequel on augmente le coût minimal, tandis que dans le second cas, il est considéré comme "le plus grand" facteur par lequel le coût maximal est divisé. Même si cette solution semble attrayante car elle est moins consommatrice d'espace que la première, nous n'avons aucune garantie, relativement, au coût minime et donc il ne résout pas notre problème initial. Elle nous a néanmoins permis d'introduire la notion de bordure. Dans la prochaine section, nous présentons notre solution qui permet de garantir une taille de mémoire.

II.3.3 L'algorithme PickBorders

L'idée consiste simplement à maintenir plusieurs bordures chacune étant calculée respectivement à $f^1, f^2, f^3, f^4 \ldots$ en utilisant à chaque fois la même technique que celle de l'algorithme PTB.

Il s'agit donc d'itérer l'algorithme PTB avec toutes les valeurs possibles de f. Or, les valeurs possibles de f sont de la forme f^i où i est un entier allant de 0 à $\lfloor \log_f(m) \rfloor$. Ainsi, une première mise en œuvre serait :

Comme il a été discuté dans la section précédente, la complexité de $PTB(f)$ est en

Procedure PickBorders(f, m)

Input: réel f et table de faits T
Output: L'ensemble $\mathcal{S}$ des cuboïdes à matérialiser

```
1 S = ∅;
2 i = 1;
3 while i ≤ ⌊log_f(m)⌋ do
4 |   S = S ∪ PTB(f);
5 |   i = i + 1;
6 |   f = f^i;
```

$O(d * 2^d)$. Du moment que la boucle **For each** est exécutée $\log(m)$ fois, nous concluons que la complexité de cet algorithme est $O(\log(m) * d * 2^d)$.

Notez que par l'adoption de cette mise en œuvre, on peut utiliser n'importe quel algorithme d'ensemble de fréquents maximaux parmi ceux proposés dans la littérature.

Dans ce qui suit, nous proposons une autre implémentation du même algorithme qui permet d'éviter de calculer plusieurs fois la taille d'un même cuboïde. Elle consiste à utiliser l'algorithme PSC dans le but d'étiqueter chaque visite d'un cuboïde c par un nombre k lorsque $m/f^{k+1} \leq size(c) \leq m/f^k$. Dans ce cas, il devient simple de tester si un cuboïde appartient à une frontière ou non ; il suffit de comparer son label à l'étiquette de ses parents. Si elles sont différentes, alors c appartient à l'ensemble des frontières. La mise en œuvre concrète pourrait être décrite comme suit.

II.3.4 L'analyse théorique

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

PickBorders garantit deux propriétés intéressantes. Premièrement, le coût de sa sortie $\mathcal{S}$ est délimité par le coût minimal fois le facteur f. Cette propriété peut-être qualifiée comme une propriété globale. La seconde, est que la même sortie garantit que si nous prenons chaque cuboïde individuellement, son coût par rapport à $\mathcal{S}$ est également limité par le temps minimal fois f. À notre connaissance, aucune des solutions pour la sélection de cube de données partiels proposées jusqu'à présent ne peut offrir ces garanties. Ce résultat est énoncé dans le théorème suivant.

Théorème 1. *Soit* $f \geq 1$ *et* $\mathcal{S}_i = \{c \in \mathcal{C} : |c| \leq m/f^i\}$ *pour* $i = 1 \ldots \lfloor \log_f(m) \rfloor$. *Soit* $\mathcal{B}_i(\mathcal{S}) =$

Procedure PickBorders(f, m)

Input: réel f et table de faits T
Output: L'ensemble $\mathcal{S}$ des cuboïdes à matérialiser

$\mathcal{S} = \emptyset$;
$C_1 = \{c \in \mathcal{C} \mid d_c = 1\}$;
$L_1 = \{c \in C_1 \text{ s.t } size(c) \leq m/f\}$;
$i = 1$;
while $L_i \neq \emptyset$ **do**
    $C_{i+1} = \{$ candidats générés de $L_i\}$;
    foreach $c \in C_{i+1}$ **do**
        size(c)=Calculer_taille(c);
        Soit k tel que $m/f^{k+1} \leq size(c) \leq m/f^k$;
        $Label(c) = k$;
    foreach $c \in L_i$ **do**
        Soit $E = \{c' \in C_{i+1} \mid c \leq c'\}$;
        if *pour tout* $c' \in E$, $Label(c) \neq Label(c')$ **then**
            $\mathcal{S} = \mathcal{S} \cup \{c\}$;
    $L_{i+1} = \{c \in C_{i+1} \mid size(c) \leq m/f\}$;
    $i = i + 1$;
Retourner $\mathcal{S}$;

$\{c \in \mathcal{S}_i$ *tel que* c *est maximal*$\}$. *Soit* $\mathcal{S} = \bigcup_{i=1}^{\lfloor \log_f(m) \rfloor} \mathcal{B}_{fi}$. *Alors*

1. $cost(\mathcal{S}) \leq MinCost * f$
2. *Pour chaque* $c \in \mathcal{C}$, $cost(c, \mathcal{S}) \leq size(c) * f$.
3. $\mathcal{S}$ *peut-être calculé dans* $O(d * 2^d)$

Démonstration. Il est simple à noter que pour chaque cuboïde c dans $\mathcal{C}$, il existe un ancêtre c' de c dans $\mathcal{S}$ tel que $taille(c') \leq f * taille(c)$. Ainsi, pour calculer chaque cuboïde, nous avons besoin d'un coût au plus égal à la taille du cuboïde (coût minimal) fois le facteur f. Cela prouve le point 2 du théorème. Le premier point est une conséquence directe du second. Enfin, le nombre d'itérations de l'algorithme PickBorders est d. Dans ce cas, le nombre de cuboïdes pour lesquels nous testons s'ils appartiennent à une frontière ou pas est 2^d. Le nombre maximal de parents d'un cuboïde est d. Cela donne le nombre maximal de tests qui est donc borné par $d * 2^d$. □

Un corollaire important de ce théorème est que la solution retournée par l'algorithme PickBorders implique un coût inférieur à la solution optimale fois le facteur f. Définissons d'abord ce qui est une solution optimale.

Définition 12 (Solution optimale)**.** *Soit space la quantité d'espace de stockage. Soit* $S \subseteq \mathcal{C}$. $\mathcal{S}$ *un cube de données partiel* possible *ssi (1)* $size(\mathcal{S}) \leq space$ *et (2)* $cost(\mathcal{C}, \mathcal{S}) \neq \infty$. *L'ensemble des cubes de données partiels possibles en fonction de space est noté* P_{space}. *Soit* $\mathcal{S}^* \in P_{space}$. *On dit que* $\mathcal{S}^*$ *est* optimal *ssi* $cost(\mathcal{Q}, S^*) = \min_{\mathcal{S} \in P_{space}} cost(\mathcal{Q}, \mathcal{S})$.

Autrement dit, une solution optimale est une solution qui (1) respecte l'espace mémoire en ayant une taille qui lui soit inférieure, (2) qui permette de répondre à toutes les requêtes et enfin (3) qui fait que le coût pour évaluer toutes les requêtes soit minimal. Nous avons le résultat suivant :

Corollaire 1. *Soit* $f > 1$. *Soit* $\mathcal{S}$ *la solution de l'algorithm PickBorders. Soit* $space = size(\mathcal{S})$. *Soit* $\mathcal{S}^*$ *le cube de données partiel optimal en fonction de space. Alors* $cost(\mathcal{Q}, \mathcal{S}) \leq f * cost(\mathcal{Q}, \mathcal{S}^*)$.

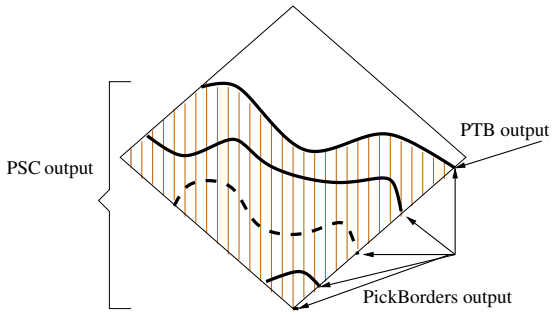


Figure II.5 – Les trois solutions.

En d'autres termes, cela signifie que si l'on considère parmi les ensembles de cuboïdes ceux dont la taille totale est inférieure ou égale à la taille de $\mathcal{S}$, l'ensemble $\mathcal{S}^*$ qui offre le plus faible coût, alors nous garantissons que le coût de $\mathcal{S}$ est au plus f fois le coût de $\mathcal{S}^*$.

Figure II.5 résume les trois algorithmes. Elle montre le treillis du cube de données. La zone hachurée représente la solution retournée par l'algorithme PSC. La courbe supérieure la plus haut est la frontière obtenue en considérant f, c'est à dire la solution de l'algorithme PTB. L'ensemble des courbes représente les frontières par rapport à $f^0, f^1, f^2, f^3 \ldots \lfloor \log_f(M) \rfloor$, soit la solution retournée par PickBorders.

Avant d'illustrer notre proposition, faisons d'abord quelques remarques.

Remarque 1. *La plupart des propositions pour la sélection des cuboïdes à matérialiser vise à réduire le* coût total *de réponse aux requêtes. Elles fournissent ainsi une solution optimale, du moins, elles visent à fournir une telle solution. Il convient de noter que cette notion d'optimalité est une propriété globale. En effet, elle n'offre aucune garantie sur l'évaluation du temps de requête pour des cuboïdes pris individuellement, c'est à dire qu'il se peut que pour certains cuboïdes c, $cost(c, \mathcal{S})/size(c)$ soit très grand. Ainsi, même quand il est possible de calculer la solution optimale, et ceci ne peut être réalisé que lorsque nous avons au plus une dizaine de dimensions, voir par exemple [TCFS08], l'utilisateur peut-être surpris de réaliser que le temps requis pour obtenir une réponse de n lignes soit supérieur à celui nécessaire pour obtenir un autre résultat de m lignes quand $n \ll m$.*

Exemple 7. *Pour illustrer notre proposition, nous poursuivons avec la Figure II.6. Les courbes représentent les frontières par rapport aux différentes puissances du facteur de f*

lorsque $f = 10$. *Les cercles pleins représentent des éléments d'au moins une frontière. Ces cuboïdes sont les seuls à être stockés. La taille totale du cube de données est 8928 ce qui représente également le coût minimal pour calculer tous cuboïdes. Le coût maximal est de 32 * 2000 = 64000. En ne conservant que les éléments de la frontière, la solution occupe une mémoire dont la taille est 2607 et le coût total est de 27554. Il faut ici noter que, même si nous avons fixé* $f = 10$, *le ratio du coût entre le coût de solution de PickBorders et MinCost est* $\frac{27554}{8927} = 3.09$. *Cela signifie qu'en moyenne, le temps de réponse des requêtes en fonction de* $\mathbb{S}$ *est de 3 fois le coût minimal. Si nous exécutons l'algorithme HRU de [HRU96] avec un critère d'espace disponible maximal égal à 2607 (la taille de notre solution) alors seuls ABCDE et BCDE sont retournés. Notez qu'avec cette solution, le coût total est de 41600. Ainsi, le ratio de coût est de* $\frac{41600}{8927} = 4.67$. *Cela est moins bon que la performance de PickBorders. D'un autre côté, si nous exécutons PBS de [SDN98], 16 cuboïdes seront stockés (les premiers seize cuboïdes ordonnés par leur taille respective). Dans ce cas, le coût total est de 34518. Le ratio du coût est donc de* $\frac{34518}{8927} = 3,87$.

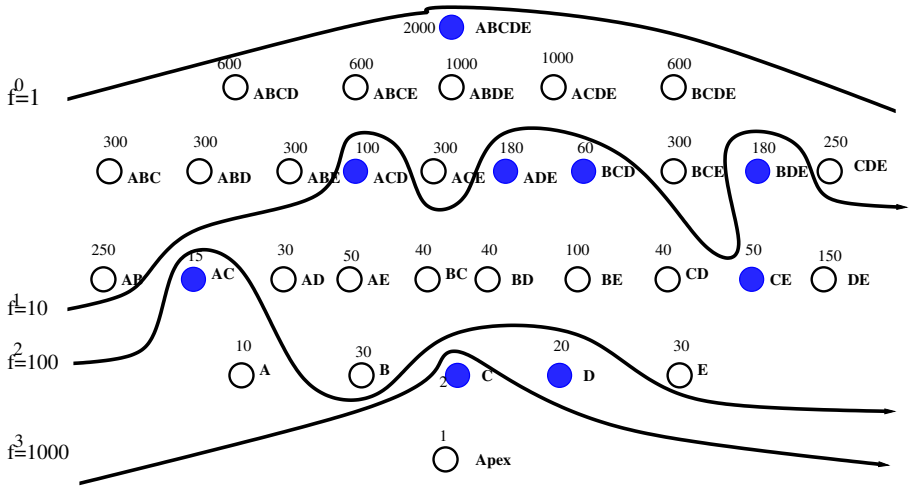


Figure II.6 – PickBorders : les frontières avec $f = 10$

II.3.5 L'approche orientée espace

La formulation classique de la littérature est la suivante :

Étant donnée une limite d'espace mémoire space, trouver un ensemble de cuboïdes $\mathcal{S}$ de taille inférieure à space et de coût minimal *(b)*

Même si notre approche et l'approche existante ne sont pas équivalentes, nous montrons, dans les sections suivantes, comment on peut utiliser notre solution pour résoudre un problème d'approche *(b)*.

L'algorithme HRU

Nous notons $\mathcal{S}^*$ la solution optimale de problème *(b)*, c'est-à-dire :

$$cost(\mathcal{S}^*) = \min_{\mathcal{S} \subseteq \mathcal{C} \text{ s.t } size(\mathcal{S}) \leq space} cost(\mathcal{S})$$

où *space* représente la quantité de mémoire disponible. Dans [HRU96], les auteurs proposent un algorithme glouton qui retourne un sous-ensemble des cuboïdes avec une notion particulière de garantie. Plus précisément, ils utilisent la notion de *gain* :

$$gain(S) = \frac{cost(\{c_b\}, \mathcal{C}) - cost(\mathcal{S}, \mathcal{C})}{cost(\{c_b\}, \mathcal{C}) - cost(\mathcal{S}^*, \mathcal{C})}$$

En résumé, le gain mesure l'amélioration de stockage de $\mathcal{S}$ par rapport au choix de stocker uniquement le cuboïde de base, c'est-à-dire $MaxCost$. En reformulant, on obtient juste que $cost(\mathcal{S}, \mathcal{C}) \leq (1 - gain(\mathcal{S}))MaxCost + gain(\mathcal{S}) * cost(\mathcal{S}^*, \mathcal{C})$. Il se peut donc que la solution retournée soit très éloignée du coût optimal. Dans [HRU96], il est montré que trouver $\mathcal{S}$ qui maximise le gain en respectant la contrainte de mémoire est un problème NP-complet. Les auteurs proposent un algorithme d'approximation, noté HRU, qui garantit que le gain de la solution retournée ne peut pas être inférieur à 63% de l'optimal. En d'autres mots, $\frac{cost(\{c_b\}, \mathcal{C}) - cost(\mathcal{S}, \mathcal{C})}{cost(\{c_b\}, \mathcal{C}) - cost(\mathcal{S}^*, \mathcal{C})} \geq 0.63$.

Il est à noter que cette notion de performance est obtenue par comparaison avec $MaxCost$ alors que notre facteur de performance se définit par rapport $MinCost$. Cette remarque a déjà faite dans [KM99], où il est montré qu'en maximisant le gain on ne peut pas forcément optimiser le temps de réponse à une requête.

La véritable faiblesse de HRU est la complexité en temps. En fait, sa complexité est $O(k * n^2)$ où k est le nombre d'itérations (qui correspond au nombre de cuboïdes sélectionnés) et n est le nombre total des cuboïdes. Comme $n = 2^d$, cet algorithme est trop lent lorsque d est grand.

Procedure HRU(space)

Input: space : espace mémoire disponible
Output: L'ensemble $\mathcal{S}$ des cuboïdes à matérialiser

```
1  S = {c_b};
2  C = C;
3  space = space − size(c_b) ;
4  while space > 0 do
5      c = cuboïde qui n'est pas dans S tel que b(c,S) est maximisé;
6      /* b(c,S) est le bénéfice obtenu en ajoutant c au S*/ ;
7      if space - size(c) > 0 then
8          space = space − size(c) ;
9          S = S ∪ {c};
10         C = C\{c};
11     else
12         space = 0 ;
13 Retourner S;
```

L'algorithme PBS

L'algorithme PBS a été proposé pour battre HRU en terme de temps de calcul. Pour cela, [SDN98] propose une simplification dénommée PBS (Pick By Size). Cet algorithme sélectionne les cuboïdes dans leur ordre croissant de taille jusqu'à ce que il n'y ait plus de mémoire disponible.

Ils montrent que malgré sa simplicité et sa faible complexité en temps, la solution retournée par PBS bat celle de [HRU96] en ce qui concerne le gain pour de nombreux jeux de données. Cependant, aucune notion de garantie de performance, y compris le gain, n'est prouvée pour cet algorithme.

Autres travaux

Il existe des variantes des algorithmes précédents qui considèrent parfois l'ajout d'index à la sélection de cuboïdes : [GHRU97, TCFS08, LTCF05, AJD06, MAD06, GM05, BB03]. Une des techniques utilisées [TCFS08] est basée sur la programmation linéaire en nombre entier. Certaines modélisations produisent des solutions exactes ou partent d'un espace de recherche réduit pour gagner en temps de calcul. Des comparaisons sont faites en utilisant le solveur industriel des contraintes (Ilog CPLEX) pour montrer la pertinence de la réduction de l'espace de recherche.

Procedure PBS(space)

Input: space : espace mémoire disponible
Output: L'ensemble $\mathcal{S}$ des cuboïdes à matérialiser

```
1  S = {c_b}
2  C = C
3  space = space − size(c_b) ;
4  while space > 0 do
5      c = plus petit cuboïde dans(C) ;
6      if space − size(c) < 0 then
7          space = space − size(c) ;
8          S = S ∪ {c};
9          C = C\{c};
10     else
11         space = 0 ;
12 Retourner S;
```

Il est intéressant de noter que toutes ces méthodes suppose une connaissance préalable des tailles des cuboïdes. Cette information est considérée comme faisant partie de leur contribution et elle est soit calculée ou estimée. Cela représente une limitation réelle de l'application de ces méthodes car lorsque le nombre de dimensions est grand, le nombre des cuboïdes augmente de façon exponentielle et les problèmes deviennent rapidement insolubles (lorsque d dépasse 10).

II.3.6 Comparaison expérimentale

Une validation expérimentale est donnée dans cette section. Nous considérons les ensembles de données suivants :

- USData10 : contient des données avec 2,5 millions d'entrées avec 10 attributs correspondant aux onze premiers attributs (à l'exclusion du premier représentant un "row id") de USData fixés. Ce jeu de données traite de données de recensement et est composé de 69 attributs. USData Census 1990 peut-être trouvé à l'adresse `http ://kdd.ics.uci.edu/`
- USData13 : c'est le même ensemble de données que USData10 mais avec 13 attributs (en ajoutant les trois attributs suivants).
- Objets : contient des données avec (seulement) 8000 entrées de 12 attributs portant sur les objets archéologiques trouvés lors de fouilles. Cet exemple constitue un cas

pour lequel le nombre de attributs est relativement grand par rapport à la taille de l'ensemble de données.

– Zipf10 : ce jeu de données est synthétique. Il contient 10^6 entrées de 10 dimensions. De nombreuses observations ont montré que les valeurs des attributs de jeux de données réelles ne suivent pas - en général - une distribution uniforme, mais souvent une distribution de la loi de puissance. Autrement dit, si nous trions les valeurs d'un attribut donné dans l'ordre décroissant, alors la fréquence de la valeur de rang i est proportionnelle à $\frac{1}{i^\alpha}$, $\alpha > 0$. α appartient principalement à la plage $[2,3]$. Dans nos expériences, nous avons considéré une loi de puissance de paramètre $\alpha = 2$.

La table II.1 résume quelques caractéristiques de ces ensembles de données. Elle contient MinCost (chaque fois que le cube de données entier est stocké) et MaxCost (seulement le cuboïde de base est stockée).

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

Table II.1 – MinCost et MaxCost des jeu de données

Les deux algorithmes PBS et HRU prennent en entrée une contrainte sur l'espace disponible de la mémoire. Dans nos expériences, nous avons supprimé cette contrainte afin de les comparer à PickBorders. Plus exactement, nous avons fait varier la contrainte de mémoire comme expliqué ci après. PickBorders n'est pas présenté avec une contrainte de mémoire. Toutefois, il est facile d'en tenir compte. En effet, il suffit d'exécuter PickBorders avec une petite valeur du paramètre f, disons $f = 1.2$ par exemple. Si la taille de la solution retournée est trop grande pour tenir dans la mémoire disponible, alors nous ré-exécutons PickBorders avec le paramètre f^2, f^3 jusqu'à obtenir $\mathcal{S}$ qui peut être stocké dans la mémoire disponible. En fait, il n'est pas nécessaire d'exécuter PickBorders plusieurs fois avec différents paramètres. Il suffit de noter que PickBorders avec le paramètre f^{i+1} possède une sortie incluse dans le PickBorders avec paramètre f^i. Ainsi, PickBorders (f^{i+1}) a déjà été calculé en PickBorders (f^i).

Les trois algorithmes (HRU, PBS et PickBorders) prennent en entrée la liste des cuboïdes et leur taille. Pour pré-calculer la taille des cuboïdes, on peut utiliser le code donné dans [AL07]. Cette première étape peut aisément prendre des heures. Cette étape

n'est nécessaire que pour HRU. Nous ne comptons pas ce temps dans le temps d'exécution des différents algorithmes, ce qui n'est pas très réaliste (cf. conclusion de ce chapitre).

Coût, mémoire et temps

Dans nos expériences, puisque nous ne faisons aucune hypothèse sur la façon dont les cuboïdes sont stockés physiquement, la quantité de mémoire est exprimée en nombre de lignes de l'ensemble des cuboïdes matérialisés. Pour PickBorders, nous exécutons l'algorithme prenant $f = 1.5, f^2 = 2.25, f^3 = 3.38, \ldots$ pour tous les jeux de données. Chaque exécution conduit à une paire Mémoire/Coût. Nous avons ensuite utilisé cette valeur de mémoire en tant que paramètre pour limiter l'espace de HRU et PBS pour obtenir des paires Mémoire/Coût. Cette expérience est représentée dans les figures II.7, II.8, II.9 et II.10. Par exemple, pour USData10, lorsque $f = 1.5$, PickBorders a besoin de 2 160 000 unités de mémoire pour un coût de 4 750 000 alors que pour $f = 3.38$, PickBorders prend 828 000 unités de mémoire et a un coût de 7 100 000.

Dans toutes les expériences, PBS a la plus mauvaise performance en termes de coût et de mémoire. En général, HRU a la meilleure performance, mais PickBorders est un bon challenger. Nous pouvons également remarquer que PickBorders est très concurrentiel lorsque la quantité de mémoire disponible n'est pas trop faible (pour $m > 10^5$ en USData10, pour $m > 10^6$ dans Objects et pour $m > 15.10^6$ en ZIPF10).

PBS et PickBorders ne prennent que quelques secondes alors que HRU va de quelques minutes à plusieurs dizaines d'heures (pour la dimension 13). En raison du temps requis pour exécuter HRU, nous avons stoppé le calcul avant d'ajouter tous cuboïdes à S dès que la fonction de coût est proche de $MinCost$.

Facteur de performance

Nous comparons notre approche avec HRU et PBS en ce qui concerne les performances d'évaluation des requêtes. Un des principaux intérêts de PickBorders est la garantie du facteur de performance. Malgré le fait que ni PBS, ni HRU n'offrent de garantie sur cette mesure, il est intéressant de mesurer la valeur du facteur de performance dans un cadre pratique.

Pour cette expérience, nous avons exécuté PickBorders avec certaines valeurs de f. Pour chaque solution $\mathcal{S}$ obtenue, nous avons exécuté HRU avec la contrainte de mémoire $size(S)$. La figure II.11 montre que PickBorders a un facteur de performance moyen meilleur que PBS et HRU pour USData10 (pour $f = 3,38$ et $f = 11.39$).

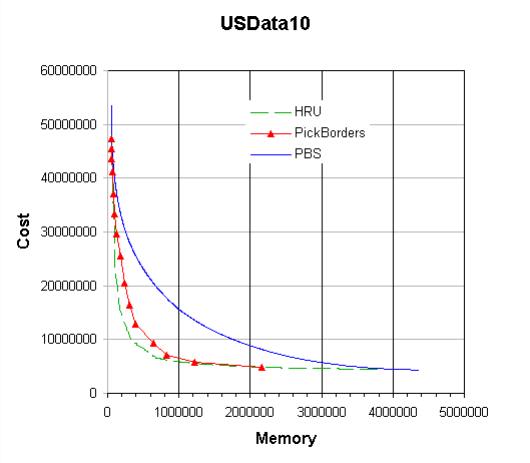


Figure II.7 – Coût/Mémoire pour USData10

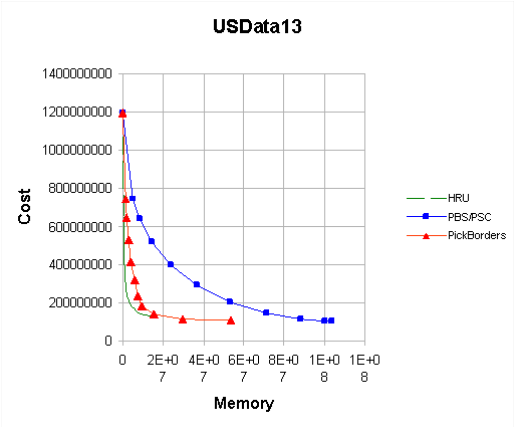


Figure II.8 – Coût/Mémoire pour USData13

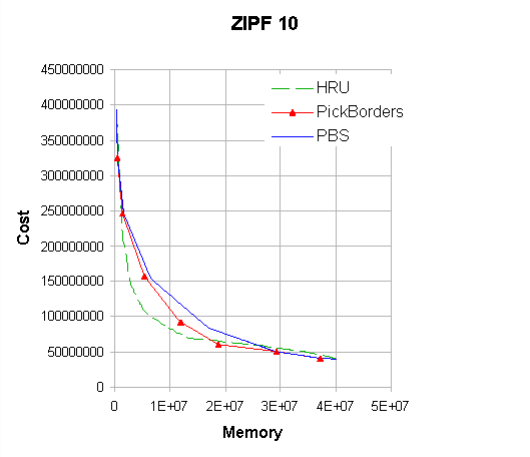


Figure II.9 – Coût/Mémoire pour ZIPF10

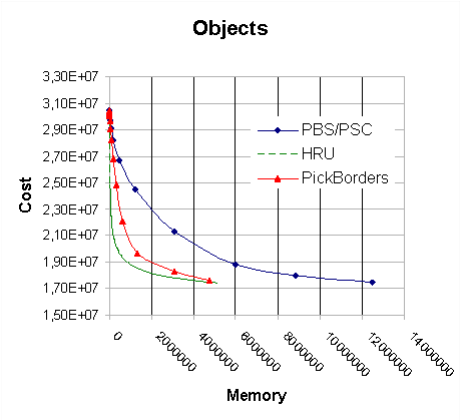


Figure II.10 – Coût/Mémoire pour Objects

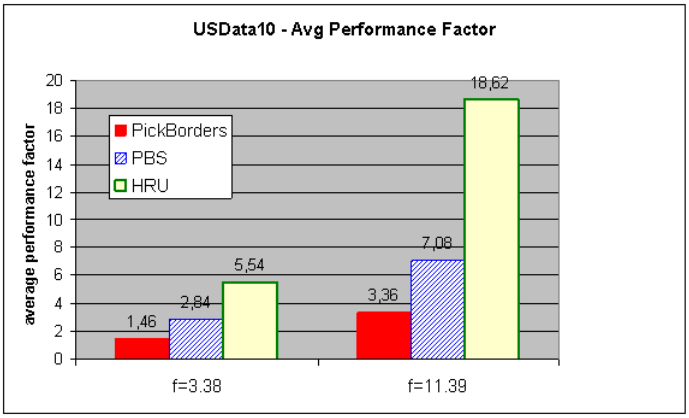


Figure II.11 – Performance facteur avec $f = 3.38$ et $f = 11.39$.

Un regard plus attentif sur la répartition des facteurs de performance dans les figures II.12 et II.13 explique ce fait en montrant que certains cuboïdes devraient être calculés à partir des cuboïdes matérialisés dont la taille est beaucoup plus grande. Par example, l'exécution de HRU avec 830000 d'unités de mémoire, il y a 7 (non matérialisés) cuboïdes dont le plus petit ancêtre matérialisé est de taille au moins 100 fois plus grande. Nous avons exécuté PBS et HRU avec 309000 et 830000 unités de mémoire correspondant respectivement à la quantité de mémoire utilisée par PickBorders avec $f = 3,38$ et $f = 11,39$. La première catégorie représente l'ensemble des cuboïdes avec facteur de performance de 1 (cuboïdes matérialisés et des cuboïdes dont le moindre ancêtre matérialisé ont la même taille). La deuxième catégorie compte les cuboïdes avec un facteur de performance entre 1 et 2. La troisième catégorie compte les cuboïdes avec un facteur de performance dans $]2, 3.5]$ et ainsi de suite. Notez que l'esprit de chaque algorithme est esquissé par la première catégorie : PBS stocke beaucoup de cuboïdes petits, HRU tend à matérialiser quelques cuboïdes de grande taille et PickBorders choisit une combinaison des cuboïdes des différentes tailles.

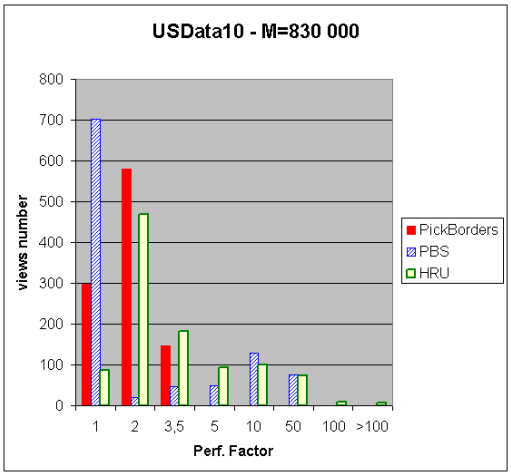


Figure II.12 – Distribution du facteur de performance quand $f = 3.38$

II.4 Conclusion

Nos expériences montrent qu'en fixant un facteur de performance f, le temps moyen de réponse, qui est bien entendu inférieur à f, est très comparable pour PickBorders et HRU. Cela signifie que PickBorders peut également être considéré dans le cadre que l'approche orientée système, c'est-à-dire en fixant une limite mémoire. Dans ce cas, pour trouver le

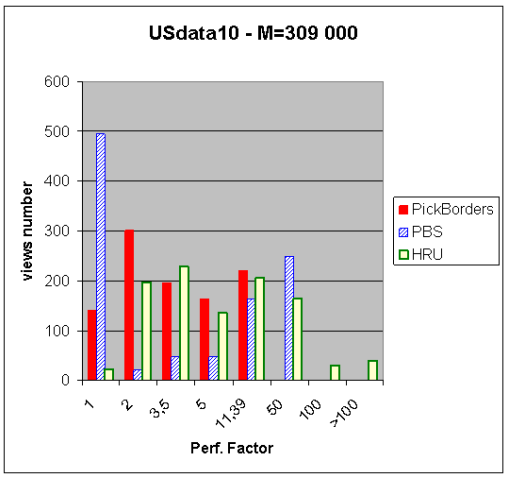


Figure II.13 – Distribution du facteur de performance quand $f = 11.39$

plus petit facteur de performance admissible, plusieurs options sont envisageables : partir de $f_{min} = 1$ et $f_{max} = size(c_b)$ et effectuer une recherche de f en faisant une dichotomie en supposant que si $f > f'$ alors la f-bordure prend moins de mémoire que la f'-bordure. Cette hypothèse n'étant pas totalement certaine, une autre solution consiste à considérer que le facteur de performance minimal est de la forme $f = (1+\epsilon)^i$ et d'incrémenter i tant que la solution ne rentre pas en mémoire. Nous n'avons pas exploré cette piste mais il serait intéressant d'adapter PickBorders dans l'approche orientée système.

Pour conclure, nous pouvons donner des éléments de comparaison entre les différentes solutions recensées pour la sélection de vues, indépendamment de l'approche considérée. Dans le tableau II.2, k désigne le nombre de cuboïdes choisis par les différents algorithmes de sélection de vues. La dénomination "Scénario réaliste" indique si la connaissance préalable de la taille des cuboïdes est nécessaire au début de l'algorithme de sélection de vues. On considère que le scénario est réaliste si cette connaissance préalable n'est pas nécessaire. À titre informatif, le calcul exact de la taille d'un seul cuboïde prend environ 30 secondes pour une taille de 1 million d'entrées. On peut faire une estimation très grossière rapidement en prenant la taille maximale possible (le produit des tailles des cuboïdes monodimensionnels) mais les garanties de performances (temps de réponse et mémoire) deviennent inexistantes. En principe, les algorithmes nécessitent au moins k appels à la procédure de calcul de taille. Le choix de stocker le cube de données complet ou aucun cuboïde en dehors du cuboïde de base ne nécessite aucun appel au calcul de taille.

Ce point de vue sur la complexité indique clairement pourquoi des algorithmes comme HRU ou la programmation linéaire sont inefficaces en terme de temps. D'autre part, la

Type de sélection	Scénario réaliste	Mémoire (sans c_b)	Coût	Garantie de performance facteur	Garantie de gain	Complexité temps
cuboid de base		0	$2^d m$	non	0	0
cube entière		$O(2^d m)$	$MinCost$	1	1	$O(2^d)$
PSC/PBS	oui	$O(km)$	$O(2^d m)$	non	non	$O(kd)$
HRU	non	$O(km)$	$O(2^d m)$	non	$\geq 0,63$	$\Theta(2^d 2^d k)$
PickBorders	oui	$O(km)$	$\leq f * MinCost$	oui	non	$O(d2^d \log(m))$
Program. Linéaire	non	$O(km)$	$MinCost$	1	1	non limitée

Table II.2 – Table de comparaison de performance

complexité en temps dans le pire cas pour PickBorders peut laisser penser que PickBorders est lent. Cependant, nous conjecturons qu'il est possible de proposer une version de PickBorders avec une complexité en temps de l'ordre de $O(dk)$. Nous n'avons pas trop voulu explorer cette piste car il nous a semblé plus important de généraliser l'approche orientée utilisateur, objectif du chapitre suivant.

Chapitre III

Sélection de vues en présence de cibles

Dans le chapitre précédent, nous avons proposé une solution pour évaluer efficacement *toutes* les requêtes possibles (ayant une certaine forme) sur un cube de données. Même si très souvent l'interaction de l'utilisateur avec un cube de données se fait d'une manière ad hoc via des requêtes non prévisibles et non répétitives, il n'en demeure pas moins qu'une partie des interrogations du cube se fait selon un schéma bien établi. Nous pensons notamment à l'activité d'édition de rapports. Ainsi, dans ce chapitre nous étudions le cas où l'on dispose d'un ensemble de requêtes que nous qualifions de *cibles*. Nous proposons des algorithmes permettant de minimiser l'espace de stockage du cube tout en garantissant l'efficacité de l'évaluation des requêtes cibles. De plus, nous abordons le problème de la "dynamique" selon deux volets : l'évolution des données et l'évolution des requêtes cibles. Dans les deux cas, une solution calculée à un instant t peut s'avérer très peu performante à l'instant $t+1$. Pour ce faire, nous introduisons la notion de *stabilité* qui, intuitivement, caractérise le cas des solutions qui sont peu, voire pas, détériorées suite à des mises à jour des données. Nous proposons une analyse théorique dans le cas d'une distribution uniforme des données pour obtenir des conditions suffisantes permettant de garantir la non détérioration des performances. Cette analyse est confirmée par des expérimentations qui portent sur des données qui ne respectent pas une distribution uniforme. Quand les performances ne sont plus assurées, nous proposons une solution pragmatique pour y remédier. Ceci dans le sens où calculer et matérialiser une nouvelle solution peut prendre trop de temps laissant ainsi la base de données indisponible pour les interrogations. Intuitivement, notre solution consiste à améliorer les performances de l'évaluation des requêtes tant que l'on dispose d'assez de temps pour ce faire.

III.1 Sélection de vues en présence de cible

Dans cette section, nous considérons un ensemble $\mathcal{Q}$ qui représente l'ensemble des requêtes que l'on veut optimiser [1]. Les requêtes que l'on considère sont de la forme `SELECT Dimensions, M FROM T WHERE condition GROUP BY Dimensions` où `Dimensions` est un ensemble de dimensions du cube de données, `T` est la table de faits et `condition` est une composition (conjonctions et/ou disjonctions) de conditions simples de la forme $D\,\theta$ *valeur* où D est le nom d'une dimension (ou attribut) et θ est un comparateur dans $\{<,>,=,\leq,\geq,\}$. Nous allons d'abord considérer le cas où l'on ne dispose pas d'index. Nous traiterons le cas des index dans la section III.5. Comme dans le chapitre précédent, le coût minimal pour l'évaluation d'une requête est assimilé à la taille du plus petit cuboïde à partir duquel elle peut être évaluée. Clairement, ces requêtes ne peuvent être évaluées qu'à partir des cuboïdes qui contiennent toutes les dimensions mentionnées dans la requête que ce soit dans la clause `SELECT` ou bien `WHERE`. Comme la taille des cuboïdes est monotone en fonction des dimensions, i.e $Att(c1) \subseteq Att(c_2) \Rightarrow Size(c_1) \leq Size(c_2)$, le plus petit cuboïde à partir duquel la requête peut être évaluée est celui qui contient exactement les attributs figurant dans l'expression de la requête. Ceci va nous permettre de définir le coût minimal d'une requête.

Définition 13. *Soit q une requête de la forme*

```
SELECT    Dimensions, M
FROM      T
WHERE     condition
GROUP BY  Dimensions
```

On note $Dims(q)$ l'ensemble des dimensions mentionnées dans les clauses `SELECT` *ou* `WHERE` *de q. Soit c le cuboïde de $\mathcal{C}$ tel que $Att(c) = Att(q)$. Alors, $MinCost(q) = Size(c)$. Dans la suite, $MinCub(q)$ désignera le plus petit cuboïde à partir duquel q peut être calculé.*

Exemple 8. *Soit $q =$* `Select A, C, COUNT(*) FROM T WHERE D=10 AND B < 25 GROUP BY A, C`*. Les dimensions mentionnées dans q sont A, B, C et D. Ainsi, $MinCost(q) = Size(ABCD)$.*

1. Dans la littérature, on parle de *charge* ou *workload*.

Nous généralisons la relation $\preceq$ et la notion *d'ancêtre* aux requêtes en écrivant abusivement

- $q \preceq c$ si et seulement si $MinCub(q) \preceq c$ et
- c est ancêtre de q si et seulement si $q \preceq c$.

Le problème que l'on veut résoudre consiste donc à trouver un ensemble $\mathcal{S} \subseteq \mathcal{C}$ tel que :

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

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

Nous introduisons d'abord quelques notations. On note par $\mathcal{A}_f(q)$ l'ensemble des cuboïdes qui sont (1) ancêtres d'une requête $q \in \mathcal{Q}$ et (2) dont la taille est inférieure ou égale à $f * MinCost(q)$. $\mathcal{B}_f(q) \subseteq \mathcal{A}_f(q)$ est l'ensemble des éléments maximaux de $\mathcal{A}_f(q)$, c'est-à-dire la f-bordure de q.

Soit $\mathcal{A}_f(\mathcal{Q}) = \bigcup_{q\in\mathcal{Q}} \mathcal{A}_f(q)$. Noter que $MinCub(q) \in \mathcal{A}_f(q)$. On utilise aussi la notation $\mathcal{B}_f(\mathcal{Q}) = \bigcup_{q\in\mathcal{Q}} \mathcal{B}_f(q)$. Aussi, $MinCub(\mathcal{Q}) = \cup_{q\in\mathcal{Q}}\{MinCub(q)\}$.

Il est facile de voir que la solution à notre problème est un sous-ensemble de $\mathcal{A}_f(\mathcal{Q})$. En effet,

Lemme 4. *Soit $\mathcal{S}^* \subseteq \mathcal{C}$ le plus petit ensemble de cuboïdes (en terme de taille mémoire) t.q $cost(q, \mathcal{S}^*) \leq f * size(q)$ pour chaque $q \in \mathcal{Q}$. Alors $\mathcal{S}^* \subseteq \mathcal{A}_f(\mathcal{Q})$.*

III.1.1 Modélisation par un problème de domination dans des graphes orientés et pondérés

Quelle que soit la solution proposée, on peut modéliser notre problème par celui de la recherche d'un ensemble dominant pondéré dans le graphe orienté et pondéré suivant :

Définition 14 (Graphe de recherche)**.** *Soit $G(f) = (V, E, \mathbf{w})$ le* graphe de recherche *défini par :*

- $V = \mathcal{A}_f(\mathcal{Q})$ *et*

– $(v_1, v_2) \in E$ *ssi il existe* $q \in \mathcal{Q}$ *telle que* $v_1 \in \mathcal{A}_f(q)$ *et* $v_2 = MinCub(q)$.

– $\mathbf{w} : V \longrightarrow [1, |\mathcal{C}|]$ *défini par* $\mathbf{w}(c) = size(c)$.

Définition 15 (Graphe de recherche partiel)**.** *Le* graphe de recherche partiel $G_p(f)$ *est le sous-graphe de* $G(f)$ *induit par les sommets* $V' = \mathcal{B}_f(\mathcal{Q}) \cup \mathcal{Q}$.

Exemple 9. *La figure III.1 montre le graphe de recherche et le graphe de recherche partiel pour la valeur* $f = 10$. *On suppose que* $\mathcal{Q}$ *contient trois requêtes qui sont*

– `SELECT B, COUNT(*) FROM T GROUP BY B`

– `SELECT C, COUNT(*) FROM T GROUP BY C`

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

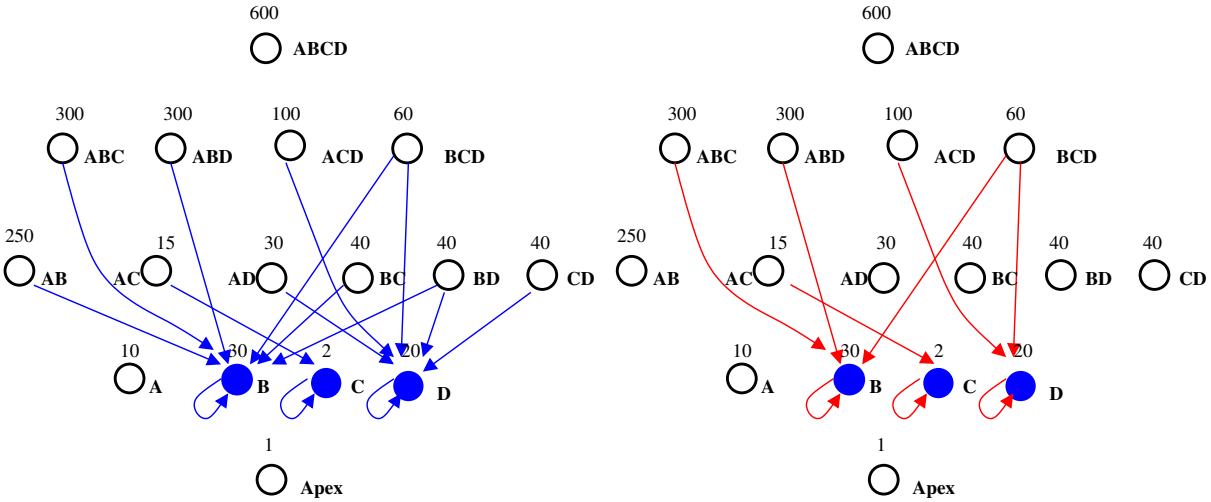


Figure III.1 – Graphes de recherche complet (à gauche) et partiel (à droite) avec $f = 10$.

Le *poids* d'un ensemble de sommets correspond à l'espace mémoire requis pour stocker cet ensemble. Nous allons construire un ensemble dominant $\mathcal{Q}$ dans le graphe de recherche :

Définition 16 (Ensemble dominant)**.** *Soit* G *un* graphe de recherche. $\mathcal{S} \subseteq V$ domine *(ou* couvre*)* $\mathcal{Q}$ *ssi pour tout* $q \in \mathcal{Q}$, *il existe* $s \in \mathcal{S}$ *tel que* $MinCub(q)$ *est voisin (sortant) de* s.

Partant du graphe de recherche, la sélection du plus petit $\mathcal{S}$ revient à chercher l'ensemble de sommets de V de poids minimum qui couvre tous les éléments de $\mathcal{Q}$. Ce problème est une variante du problème bien connu du *minimal weighted set cover* (MWSC) qui est

NP-complet [Kar72]. Cependant, si $\Delta \leq |\mathcal{Q}|$ est le degré sortant maximal du graphe de recherche, il existe un algorithme glouton de complexité en temps $O(\Delta|V||\mathcal{S}|) = O(|V||\mathcal{Q}|^2)$ [2] qui permet de le résoudre avec une $1 + \ln|Q|$-approximation de la solution optimale [Chv79]. Le premier algorithme présenté possède cette approximation et nous en donnons un deuxième qui retourne une $f(1 + \ln\Delta)$ approximation de l'espace mémoire optimal mais avec un gain réel de temps de calcul. Ce deuxième résultat est nouveau. Par ailleurs, il est bien connu que le problème MWSC peut être modélisé par un programme d'optimisation linéaire qui peut être résolu assez rapidement par les solveurs existants lorsque le nombre de variables n'est pas trop élevé.

III.1.2 Solutions approchées

Soit $\Gamma(c)$ l'ensemble des voisins sortant de c. Cet algorithme est une adaptation de l'algorithme classique glouton de construction d'ensemble dominant pour un graphe pondéré.

Algorithm 1: PickFrom$f$Ancestors(G).

```
1 V = sommets de G ;
2 S = ∅ ;
3 while Q ≠ ∅ do
4     c* = arg min_{c∈V(Q)} size(c)/|Γ(c)|;
5     S = S ∪ {c*};
6     V(Q) = V(Q)\({c*} ∪ Γ(c*));
7     Q = Q\Γ(c*);
8 Retourner S;
```

L'algorithme consiste à choisir, lors de chaque itération, l'ancêtre qui a la "charge" $\frac{size(c)}{|\Gamma(c)|}$ minimale. Cette dernière est définie comme étant la taille de l'ancêtre divisée par le nombre de ses voisins (c.à.d le nombre de requêtes de $\mathcal{Q}$ qu'il couvre). Dès qu'un ancêtre est choisi, on le met dans la solution $\mathcal{S}$ et on le supprime du graphe ainsi que les requêtes qu'il couvre. Ceci a pour incidence de modifier le nombre de voisins des sommets restants. En utilisant le résultat de [Chv79] nous montrons le facteur d'approximation obtenu par cet algorithme appliqué à $G(f)$ ou à $G_p(f)$.

2. Même si $|V|$ peut atteindre 2^d, en pratique, V est bien plus petit.

Théorème 2. *Soient $f > 1$ et G un graphe de recherche. Soit $\mathcal{S}$ la solution retournée par PickFromfAncestors(G). Si $\mathcal{S}^*$ est la solution optimale (c.à.d la plus petite en taille mémoire) alors :*

- *pour chaque $q \in \mathcal{Q}$, $cost(q, \mathcal{S}) \leq f * size(q)$;*
- *si $G = G(f)$ alors $size(\mathcal{S}) \leq (1 + \ln \Delta) * size(\mathcal{S}^*)$ où Δ est le degré sortant maximal de G ;*
- *si $G = G_p(f)$ alors $size(\mathcal{S}) \leq f(1 + \ln \Delta) * size(\mathcal{S}^*)$ où Δ est le degré maximal de Gp.*

Démonstration. Seul le troisième point nécessite d'être démontré. Soit $\mathcal{S}^*$ (respective $\mathcal{S}^*_p$) une solution optimale donnée par le graphe de recherche $G(f)$ (respective $G_p(f)$). Soit V' l'ensemble des sommets de $G_p(f)$. Réalisons une solution $\mathcal{S}' \subseteq V'$ à partir de $\mathcal{S}^*$. Pour tout $s \in V' \bigcap \mathcal{S}^*$, s appartient $\mathcal{S}'$. Si $s \notin V' \bigcap \mathcal{S}^*$, alors on ajoute à $\mathcal{S}'$ le plus petit ancêtre s' de s qui appartient à V'. Autrement dit, si s domine $q_1, q_2, \ldots, q_k \in \mathcal{Q}$ avec $size(q_i) \leq size(q_{i+1})$ alors $s' \in \mathcal{B}_f(q_1)$ et $size(s)/f \leq size(s') \leq f * |q_i|$ pour tout $i \leq k$. Par construction, nous obtiens que $\mathcal{S}'$ et $\mathcal{S}^*$ ont même cardinalité et $size(\mathcal{S}') \leq f * size(\mathcal{S}^*)$.

Or, par définition, $size(\mathcal{S}') \geq size(\mathcal{S}^*_p)$. Il s'en suit que $size(\mathcal{S}^*_p) \leq f * size(\mathcal{S}^*)$. Comme l'algorithme PickFrom$f$Ancestors retourne une solution qui est une $(1 + \ln \Delta)$ approximation de l'optimal $\mathcal{S}^*_p$, la solution retournée a une taille au plus $f(1 + \ln \Delta) size(\mathcal{S}^*)$. □

On peut se demander l'intérêt de considérer le graphe de recherche partiel car la garantie sur l'espace mémoire est plus mauvaise qu'en prenant le graphe de recherche. Cependant, ce graphe de recherche partiel est plus petit et il sera plus rapide d'y chercher une solution $\mathcal{S}$.

III.1.3 Solution exacte

Nous donnons maintenant le programme d'optimisation linéaire. Avant cela, nous décrivons les variables utilisées :

- s_i désigne la taille du cuboïde $c_i \in \mathcal{A}_f(\mathcal{Q})$,
- $x_i \in \{0, 1\}$ et signifie que c_i est choisi pour être matérialisé (1) ou non (0),
- $y_{ij} \in \{0, 1\}$ telle que $y_{ij} = 1$ ssi $q_i \in \mathcal{Q}$ utilise c_j pour être calculée

$$\min \sum_{j:c_j \in \mathcal{A}_f(\mathcal{Q})} x_j * s_j \tag{III.1}$$

$$\forall i : q_i \in \mathcal{Q} \sum_{j:c_j \in \mathcal{A}_f(q_i)} y_{ij} = 1 \tag{III.2}$$

$$\forall i : q_i \in \mathcal{Q}, \forall j : c_j \in \mathcal{A}_f(q_i)\ y_{ij} \leq x_j \tag{III.3}$$

$$\forall j : c_j \in \mathcal{A}_f(\mathcal{Q})\ x_j \in \{0,1\} \tag{III.4}$$

$$\forall i : c_i \in \mathcal{Q},\ \forall j : c_j \in \mathcal{A}_f(q_i)\ y_{ij} \in \{0,1\} \tag{III.5}$$

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

La solution "exacte" à notre problème correspond donc à $\mathcal{S} = \{c_j \in \mathcal{A}_f(\mathcal{Q}) : x_j = 1\}$. On peut se demander pourquoi avoir recours à un algorithme approché lorsque l'on peut résoudre exactement un problème. En fait, les solveurs tels que `Cplex`, ou `LP_Solve`[3], sont très puissants jusqu'à un certain nombre de variables. Or, le nombre de x_i et de y_{ij} peut croître assez rapidement avec l'accroissement de $|\mathcal{Q}|$ ou avec l'accroissement de f. Dans les expérimentations que nous avons menées, nous avons utilisé LP_Solve qui est un outil gratuit qui, bien qu'efficace, ne nous a néanmoins pas permis de traiter des cas où $|\mathcal{Q}|$ était "grand".

Remarque 2. *Il est à noter que le nombre de variables y_{ij} correspond au nombre d'arcs dans le graphe de recherche G et que le nombre de variables x_j est égal au nombre de sommets. Il y a donc au moins $2|\mathcal{Q}|$ variables mais peut atteindre $|V| + |V||\mathcal{Q}| \leq (|\mathcal{Q}| +)$. Dans nos expérimentations avec $|\mathcal{Q}| > 200$ et $f > 2$, la taille de $|V| = \mathcal{A}_f(\mathcal{Q})$ dépasse largement les 20000.*

3. Disponible à l'URL `http ://lpsolve.sourceforge.net/5.5/`

III.2 Intégration des dimensions avec hiérarchies

Dans ce qui a été présenté auparavant, nous n'avons pas pris en compte le fait que certaines dimensions puissent être modélisées par des hiérarchies. Ces dernières sont souvent présentes dans les cubes de données et sont utilisées dans le cadre des requêtes du type Roll-Up et Drill-Down.

Roll-Up et Drill-Down sont des opérations de raffinement de requêtes dans OLAP qui permettent, dans une dimension, de passer d'un attribut à son attribut parent, respectivement fils, dans la hiérarchie d'attributs de la dimension. Par example, le fait de passer d'un visualisation de nombre de ventes par ville au nombre de ventes par pays, laquelle appartient ces villes, est appelé Roll-Up. L'opération inverse est appelé Drill-Down.

Dans cette section, nous montrons que, à quelques détails mineurs prés, la prise en compte des hiérarchies ne pose pas de problèmes. Nous définissons d'abord une relation d'ordre entre les niveaux d'une même hiérarchie.

Définition 17 (Hiérarchie d'une dimension)**.** *Soit $d_i \in D$ une dimension du cube de données. Soit $\mathcal{L}_i = \{\ell_1, \ldots, \ell_m\}$ les différents niveaux de d_i. On dit que ℓ_k est* moins spécialisé *que $\ell_{k'}$, et on note $\ell_k \sqsubseteq_i \ell_{k'}$, ssi on peut regrouper les valeurs de $\ell_{k'}$ pour obtenir des éléments de ℓ_k.*

Exemple 10. *Soit la dimension Temps dont les niveaux de la hiérarchie sont* {jour, mois, année}*. On peut regrouper les jours pour former des mois (ou des années) et on peut regrouper des mois pour former des années. Nous avons donc, mois $\sqsubseteq$ jour, année $\sqsubseteq$ mois et année $\sqsubseteq$ jour.*

Dans la suite, nous supposons que pour chaque dimension d_i il existe
- un plus petit élément dans la hiérarchie $\mathcal{L}_i$ et
- un plus grand élément dans $\mathcal{L}_i$ qui est noté *All*

Exemple 11. *Dans l'exemple précédent, le niveau* jour *peut être considéré comme étant le plus petit élément. Le plus grand élément étant* All*.*

Noter qu'en présence de hiérarchies, le nombre de cuboïdes d'un cube de données est égal à $\prod_{i=1}^{|D|} |\mathcal{L}_i|$. Nous définissons maintenant une relation d'ordre entre cuboïdes comme suit.

Définition 18 (Ordre entre cuboïdes)**.** *Supposons que chaque cuboïde c est représenté par un vecteur à $|D|$ dimensions de la forme $[\ell_{1i},\ldots,\ell_{|D|i}]$. Avec ℓ_{ki} qui désigne un élément de la hiérarchie $\mathcal{L}_k$. Soient c et c' deux cuboïdes. c est* plus petit *que c', on note $c \preceq c'$, ssi pour tout $1 \leq k \leq |D|$ on a $c'[k] \sqsubseteq_k c[k]$.*

Exemple 12. *Soient un cube $\mathcal{C}$ avec $D = \{Temps, produit\}$. La dimensions $Temps$ forme la hiérarchie $\mathcal{L}_T = \{jour, semaine, mois, All\}$ et la dimension produit forme la hiérarchie $\mathcal{L}_2 = \{nomP, All\}$. Les relations d'ordre sont les suivantes :*

- *$jour \sqsubseteq_1 mois, jour \sqsubseteq_1 semaine, mois \sqsubseteq_1 All$ et $semaine \sqsubseteq_1 All$*
- *$nomP \sqsubseteq_2 All$*

Soit $c = (mois, nomP)$ et $c' = (jour, nomP)$. Alors, $c \preceq c'$ car (1) $c'[1] = jour \sqsubseteq_1 c[1] = mois$ et $c'[2] = nomP \sqsubseteq c[2] = nomP$.

On peut voir facilement que le cube de données $\mathcal{C}$ muni de la relation d'ordre $\preceq$ définit un treillis. Par ailleurs, nous disposons des mêmes propriétés que celles que nous avons utilisées auparavant, à savoir :

- Si $c \preceq c'$ alors c peut être calculé à partie de c' [4], et
- Si $c \preceq c'$ alors la taille de c est plus petite que celle de c'.

Exemple 13. *Le treillis défini par $(\mathcal{C}, \preceq)$ de l'exemple précédent est décrit dans la figure III.2. Si c est un cuboïde et si $c[i] = All$ alors cela signifie que la dimension i n'est pas présente dans c.*

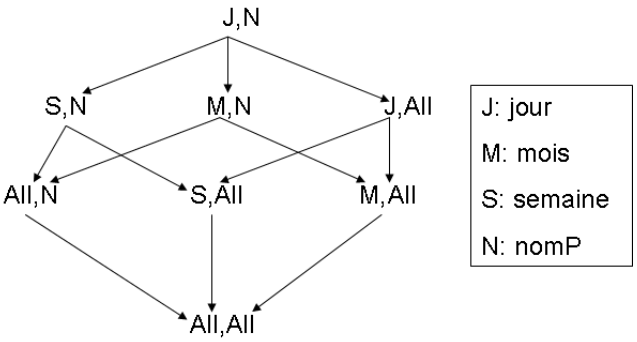


Figure III.2 – Cube de données à 2 dimensions hiérarchisées.

4. Toujours en supposant que les seules mesures considérées sont celles qui sont algébriques [GCB+97].

Le lecteur intéressé peut consulter [HRU96] pour plus de détails sur la prise en compte des hiérarchies et la modélisation sous forme de treillis des cubes de données ainsi obtenus. En conclusion de cette section, nous notons essentiellement que bien que les résultats présentés dans cet article et les expériences effectuées ne tiennent pas compte de la présence de hiérarchies dans les dimensions, ils sont facilement généralisables à ce cadre.

III.3 Stabilité

Dans cette section, nous analysons *la stabilité* des solutions que l'on propose. On entend par *stabilité* le fait qu'une solution $\mathcal{S}$ calculée à un instant t voit ses performances quasiment inchangée à un instant $t+1$ suite aux mises à jour du cube de données.

Considérons un cuboïde q qui appartient à $\mathcal{Q}$ et un cuboïde s qui appartient à la solution calculée à l'instant t tels que s est le plus petit ancêtre de q qui soit matérialisé (c.à.d q est calculé à partir de s). Si le facteur de performance fixé par l'utilisateur est f, alors on sait que $size(q) * f \leq size(s)$. On aimerait savoir quel est le nombre de tuples qu'on doit insérer (ou supprimer) de s pour que le facteur de performance de q dépasse $f+1$. Nous analysons juste les opérations d'insertion et de suppression puisque la modification est une combinaison des deux.

Insertion L'insertion de n tuples dans s engendre l'insertion de m tuples dans q avec $0 \leq m \leq n$. La pire des situations pour le facteur de performance correspond à $m = 0$. En effet, dans ce cas, la taille de s augmente mais pas celle de q. Le lemme suivant donne le nombre minimum de tuples à insérer dans s pour que le facteur de performance de q passe de f à $f+1$.

Lemme 5. *Soient s et q deux cuboïdes tels que s est ancêtre de q et $size(q) * f \leq size(s)$. Alors, il faut insérer au moins $size(q)$ tuples dans s pour que le facteur de performance de q passe à $f+1$.*

Démonstration. La preuve est immédiate. Ajoutons n tuples à s et aucun à q. s peut atteindre une taille $size(s)+n = f * size(q)+n$. Cette valeur dépasse $(f+1) * size(q)$ dès que n dépasse $size(q)$. □

Suppression En utilisant les mêmes arguments qu'auparavant, on peut facilement voir que la suppression de n tuples de s peut provoquer la suppression de m tuples de q avec

$0 \leq m \leq n$. Dans ce cas, la pire des situations pour le facteur de performance correspond à $m = n$. En effet, dans ce cas, le rapport de diminution de taille de q est plus grand que celui de s ce qui a pour effet l'augmentation du facteur de performance.

Lemme 6. *Soient s et q deux cuboïdes tels que s est ancêtre de q et $size(q) * f \leq size(s)$. Alors, il faut supprimer au moins $\frac{size(q)}{f}$ tuples de s pour que le facteur de performance de q passe à $f+1$.*

Démonstration. Enlevons n tuples à s et à q. Les tailles respectives deviennent $size(s) - n \leq f * size(q) - n$ et $size(q) - n$. Or $f * size(q) - n \geq size(s) - n \geq (f+1)(size(q) - n)$ si et seulement si $nf \geq size(q)$. □

Dans les deux types de mise à jour, nous voyons que le nombre de tuples à insérer (respectivement à supprimer) pour détruire la stabilité d'un élément s doit être au moins égal à la taille du cuboïde q (divisé par f dans le cas de la suppression) que l'on veut optimiser. Ainsi, plus ce dernier est petit (en taille) et plus il sera "exposé" aux détériorations de performance. Nous concluons cette partie en faisant remarquer le fait que, généralement, les petits cuboïdes avec peu de dimensions atteignent assez rapidement leurs tailles finales (ex : le cuboïde Apex a une ligne et sa taille n'évolue pas au grès des insertions et des suppressions) si on se limite à des opérations d'ajouts. Ceci bien sûr n'est pas toujours le cas (il suffit de considérer la dimension *Temps* qui est en perpétuelle évolution). Enfin, le fait que ce soit les "petits" cuboïdes qui soient touchés par la détérioration de performance est difficilement perceptible par l'utilisateur. En effet, si le temps d'évaluation d'une requête passe d'1 seconde à 2 secondes, ceci est pratiquement invisible du côté de l'utilisateur, pourtant le facteur de performance a doublé, alors que passer d'1 heure à 2 est bien plus pénalisant. Dans la partie expérimentations, nous rendons compte de ce phénomène en menant une analyse de la stabilité de nos solutions.

Sous réserve de données dont les valeurs sont distribuées uniformément, nous pouvons montrer qu'une solution peut être stable pour de nombreuses instances de $\mathcal{Q}$. Plus précisément, nous pouvons définir la stabilité d'une solution de la manière suivante :

Définition 19. *Soient T une table de faits et T' une table obtenue à partir d'opérations d'ajouts et suppressions de tuples dans T. Soient $\mathcal{Q}$ et $\mathcal{S}$ deux sous-ensembles de cube de données définis à partir de la table de faits $DC(T)$, tels que $\forall q \in \mathcal{Q}$, $f(q, \mathcal{S}) \leq f$. $\mathcal{S}$ est* stable *si et seulement si $\forall q \in \mathcal{Q}$, $f(q, \mathcal{S}) \leq f+1$ pour la table T'.*

Sous réserve de certaines hypothèses, nous pouvons montrer que $\mathcal{S}$ tend à être stable :

Théorème 3. *Soit $\{D_i\}_{i\in[1,D]}$ un multi-ensemble. Soit T une table de faits dont les tuples sont tirés de manière aléatoire et uniforme dans $D_1 \times D_2 \times \dots D_D$. Soient $\mathcal{Q}$ et $\mathcal{S}$ deux sous-ensembles de $DC(T)$ tels que $\forall q \in \mathcal{Q}$, $f(q,\mathcal{S}) \leq f$. Si on ne fait qu'ajouter des tuples à T de manière aléatoire uniforme, avec probabilité $1-2/|\mathcal{Q}|$:*

- *Si $\forall q \in \mathcal{Q}$, $size(q) > \beta \ln|\mathcal{Q}|$, alors $\mathcal{S}$ est* stable.
- *Si $|T| > \beta f \ln|\mathcal{Q}|(\ln\beta + \ln\ln|\mathcal{Q}|)$, alors $\mathcal{S}$ est* stable.
- *Si on effectue moins de $\frac{|T|}{2f\ln 4f}$ insertions de tuples dans T, alors $\mathcal{S}$ est* stable.

Démonstration. Nous ne donnons qu'un schéma très bref de preuve pour étayer notre affirmation. Après un premier calcul pour n'importe quel ensemble $Q \in \mathcal{Q}$, il existe $S \in \mathcal{S}$ tel que $size(S) < f * size(Q)$. Pour des raisons pratiques, s et q représentent $size(S)$ et $size(Q)$. D'après le lemme 5, pour détruire la propriété de stabilité, il faut ajouter au moins q éléments à S. Nous prouvons que cela est impossible car s est proche de sa valeur maximale $m(S) = \prod_{D_j \in Dim(s)} |D_j|$, c'est-à-dire $m(S) - s < q$ avec probabilité $1 - 1/\mathcal{Q}^2$. L'intuition est la suivante : si un cuboïde Q est petit, alors Q et S sont presque saturés. De plus, si T est assez grand, les plus petits cuboïdes (de taille inférieure à $64f^2 \ln|\mathcal{Q}|$) sont saturés. Ces deux affirmations sont vraies avec une grande probabilité.

La première étape consiste à montrer que les valeurs de s et q sont bien estimées. Ensuite, nous montrons que la manière de construire T implique que $m(S)$ et q sont très similaires. Enfin, il se trouve que $m(S) - s < q$ avec une grande probabilité.

Si on ajoute un nouveau tuple $t = (t_1, t_2, \dots, t_D)$ à un cuboïde V de manière aléatoire uniforme dans $Dom(V)$, alors la formule de Cardenas donne l'espérance de la taille de n'importe quel cuboïde en supposant que T soit aussi construit dans le modèle indépendant et aléatoire. Pour tout cuboïde V de taille v, on a :

$$\mathbb{E}(v) = m(V)\left(1 - \left(1 - \frac{1}{m(V)}\right)^{|T|}\right). \tag{III.6}$$

On peut montrer que la variable aléatoire v a une valeur proche de sa moyenne avec

grande probabilité. Plus précisément, en utilisant la borne de Chernoff, pour tout $\delta < 1$, on a :

$$\Pr\left(|v - \mathbb{E}(v)| > \delta\mathbb{E}(v)\right) < 2\exp(-\mathbb{E}(v)\delta^2/2). \tag{III.7}$$

En supposant $\mathbb{E}(v) > 64f^2 \ln|\mathcal{Q}|$ et en prenant $\delta = 1/4f$, on obtient

$$\Pr\left(|v - \mathbb{E}(v)| > \frac{\mathbb{E}(v)}{4f}\right) < 2\exp(-2\ln|\mathcal{Q}|) = \frac{2}{|\mathcal{Q}|^2}. \tag{III.8}$$

Pour $|T|$ assez grand, $\mathbb{E}(v) \approx m(V)(1 - \exp(-|T|/m(V))$.

Dans notre cas, on suppose que pour tout cuboïde V, $Dom(V) = \prod_{D_i \in Dim(V)} Dom(D_i)$. Soit t_V la projection du tuple t sur les dimensions de V, c'est-à-dire $t_V = (t_{i_1}, \ldots, t_{i_{|Dim(V)|}})$ avec $Dom(D_{i_j}) \in Dim(V)$. Si t est choisi de manière aléatoire et uniforme dans $\prod_{i=1}^{D} Dom(D_i)$, alors pour tout cuboïde V et pour tout $V_0 \in Dom(V)$, $\Pr(t_v = V_0) = \prod_{D_i \in Dim(V)} 1/Dom(D_i) = 1/Dom(V)$. Il s'en suit que la formule de Cardenas est valide dans notre cas pour tout cuboïde.

La fin de la preuve n'est pas indiquée ici mais elle consiste en une étude de cas : $m(S) > 2fq$ ou $m(S) \leq 2fq$. Le premier cas s'avère impossible avec nos hypothèses. Dans le deuxième cas, $m(S) - s \leq m(S)(\frac{1}{2f} - \frac{1}{16f^2}) < q$. □

III.4 Maintenance perpétuelle des vues

Nous présentons dans cette section deux algorithmes permettant de tenir compte des deux types d'évolution du système : mise à jour des données et mise à jour des requêtes cibles. Schématiquement, notre vision de la solution que nous proposons peut être décrite par la figure III.3.

$\mathbb{S}$ représente l'ensemble des cuboïdes déjà matérialisés. Chaque requête peut être évaluée soit à partir d'un élément de $\mathbb{S}$ ou bien à partir de c_b (nous supposons que le cuboïde de base est toujours matérialisé). De plus, quand certaines mises à jours sont soumises aux tables sous-jacentes (table de faits et/ou tables de dimensions) soit chaque $s_i \in \mathbb{S}$ est *rafraîchi* en temps réel ou bien cette mise à jour est mémorisée et n'est répercutée sur un cuboïde que quand ce dernier est sollicité pour répondre à une requête. La première alternative, qualifiée de *pro-active* tend à réduire le temps d'évaluation des requêtes quitte à

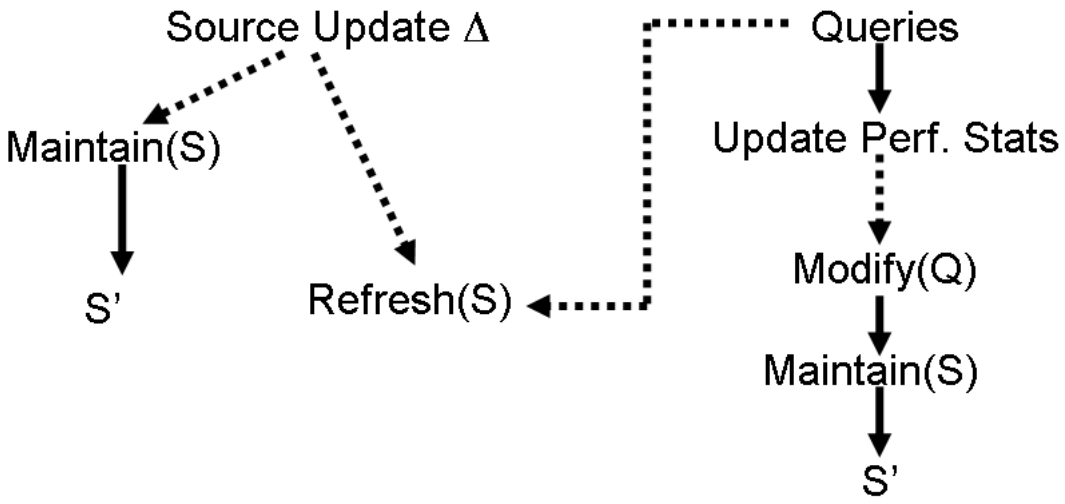


Figure III.3 – Le cycle de vie du cube partiel

"encombrer" le système avec l'activité de rafraîchissement alors que la seconde alternative, qualifiée de *réactive* tend à ne solliciter le système que quand cela est nécessaire ou bien quand ce dernier n'est pas occupé par d'autres tâches (les heures creuses pendant la nuit par exemple).

L'algorithme **maintain**($\mathcal{S}$) est responsable de la mise à jour de $\mathcal{S}$ obtenue précédemment. Cet algorithme est *prudent* dans le sens où il a tendance à améliorer la nouvelle solution $\mathcal{S}'$ tant qu'il lui reste du temps alloué à cette tâche de mise à jour. Le principe peut être décrit comme suit : L'algorithme considère d'abord un sous-ensemble des requêtes cibles qui ont les facteurs de performance les plus élevés (i.e., celles qui sont les moins optimisées). Une fois que celles-ci ont été optimisées grâce à la nouvelle solution, l'algorithme vérifie si le temps imparti n'est pas écoulé. dans ce cas, il reprend un autre sous-ensemble de requêtes de taille plus grande que celle du précédent et il réitère la procédure. A chaque itération, l'algorithme utilise une des procédures $\mathcal{VS}_f$ de sélection de vues à matérialiser parmi ceux présentés dans la section précédente. Ce processus est répété jusqu'à ce que le temps alloué à la mise à jour de la solution soit écoulé ou bien que le nombre de requêtes optimisées atteigne le nombre de requêtes cibles (dans ce deuxième cas, cela signifie que toutes les requêtes ont été optimisées). Le pseudo-code de **maintain**($\mathcal{S}$) est décrit ci-dessous.

L'algorithme commence avec les 2 requêtes qui ont le facteur de performance le plus élevé. Il matérialise la solution partielle puis prend les 4 requêtes les moins optimisées et ainsi de suite. Ce processus peut être stoppé dès que toutes les requêtes ont un facteur

Algorithm 2: maintain.

1 **foreach** $q \in \mathcal{Q}$ **do**
2 mis-à-jour($pf(q, \mathcal{S})$) ;
3 $i = 1$;
4 $Q_i = \emptyset$;
5 **while** *le temps de maintenance n'est pas fini et* $|Q_i| < |Q|$ **do**
6 $Q_i = \min(2^i, |Q|)$-th cuboïdes de plus grand pf;
7 $S_i = \mathcal{VS}_f(Q_i)$;
8 $S' = \{arg\min_{s \in S \cup S_i} pf(q, \mathcal{S}) \mid q \in \mathcal{Q}\}$;
9 Matérialise(S') ;
10 $\mathcal{S} \leftarrow \mathcal{S}'$;
11 $i = i + 1$;
12 Retourne $\mathcal{S}$;

de performance inférieur à f (c'est de toute façon l'exigence de l'utilisateur). Laisser l'exécution continuer permet cependant de réduire la taille de la solution ce qui sera bénéfique lors de la prochaine mise à jour.

L'algorithme **MaintenancePerpétuelle**() utilise **maintain**($\mathcal{S}$). Il gère l'évolution du cube (données et requêtes cibles) durant son cycle de vie. Cet algorithme est exécuté perpétuellement (utilise une boucle **While**). Il reste à l'écoute des événements qui ont lieu sur le cube. Les événements qui l'intéressent sont de trois types : l'arrivée de requêtes, l'arrivée de mises à jours des données et les demandes de l'administrateur de la base de données. Chacun de ces événements peut déclencher l'exécution de la procédure **maintain**($\mathcal{S}$) sous certaines conditions. L'algorithme est décrit comme suit :

L'arrivée des requêtes peut modifier l'ensemble des requêtes cibles $\mathcal{Q}$ ainsi une nouvelle solution $\mathcal{S}'$ doit être calculée. L'arrivée de mises à jours peut détériorer les performances des requêtes ce qui nous oblige à calculer une nouvelle solution. Enfin, l'administrateur peut explicitement demander de lancer **maintain**($\mathcal{S}$). La modification de $\mathcal{Q}$ (ligne 6) dépend de l'algorithme de sélection des requêtes cibles (chose que nous n'avons pas traitée dans ce manuscrit). La ligne 11 fait référence au paramètre λ. Ce dernier représente le ratio entre la quantité de données qui ont été insérées/supprimées par rapport à la taille des données lors du dernier calcul de la solution. Ce paramètre peut être fixé à la lumière du résultat du théorème 3. En effet, celui-ci permet de garantir la stabilité d'une solution si la quantité des données insérées ne dépasse pas un certain seuil. Donc, il n'est pas besoin de calculer une nouvelle solution.

L'algorithme **MaintenancePerpétuelle** présente la propriété *d'auto-stabilisation* que

Algorithm 3: Maintenance Perpétuelle.

```
1  while VRAI do
2      while une requête q est soumis do
3          trouve son ancêtre matérialisé minimaux s ;
4          Mise-à-jour de s si besoin ;
5          Réponde q ;
6          Modifie(Q) si besoin ;
7          maintain(S) si besoin ;
8      if une mise-à-jour δ est soumise then
9          Propagé δ au S ou cumulé ;
10         ΔT = ΔT + δ ;
11         if ΔT > λT_0 then
12             maintain(S) ;
13             T_0 = taille actuelle de T ;
14             ΔT = 0 ;
15     if une requête arrive then
16         maintain(S) ;
```

l'on rencontre dans le domaine des algorithmes distribués. Rappelons d'abord cette propriété : un algorithme distribué est auto-stabilisant s'il converge vers un état correct quel que soit l'état avec lequel il a été initialisé. L'état correct (on parle aussi d'état *légitime*) est atteint après un nombre fini d'étapes. Dans notre cas, la configuration du système peut être modélisée par un vecteur de longueur $|\mathcal{Q}|$. Étant donnée $q_i \in \mathcal{Q}$, le i-ème élément du vecteur est $(pf(q_i), \{s_i\})$ où s_i est le cuboïde ancêtre matérialisé de q_i pour qui $cost(q_i, s_i) = size(s_i)$. Il est clair que quel que soit l'ensemble $\mathcal{S}$, $\forall q_i : pf(q_i) \in [1, |T|]$. Dans notre contexte, un état légitime correspond à une configuration où pour tout i, $pf(q_i) \leq f$. Cependant, parmi les états légitimes seulement certains permettent de garantir la minimalité de l'espace mémoire. Il peut très bien arriver que la solution courante soit un état légitime et malgré ça, on laisse l'algorithme continuer son exécution afin justement de réduire l'espace mémoire de la solution.

Théorème 4. *Pour tout* $\mathcal{S}$, **MaintenancePerpétuelle***() est un algorithme auto-stabilisant permettant d'obtenir un facteur de performance inférieur à* f*. Concernant l'espace mémoire de la solution, il offre la même garantie que celle qui est assurée par l'algorithme de sélection de vues* $\mathcal{VS}_f$ *qu'on choisit pour l'implémenter.*

III.5 Intégration des index

Dans ce que nous avons présenté précédemment, nous avons toujours considéré que les seuls objets que l'on pouvait stocker sont les cuboïdes. Nous avons alors défini le coût minimal d'une requête en fonction de la taille du plus petit cuboïde à partir duquel on peut l'évaluer. Clairement, si les cuboïdes peuvent être indexés, le coût d'une requête peut être moindre. Cependant, nous utiliserons plus de mémoire. Nous devons donc redéfinir notre modèle de coût. Avant de rentrer dans les détails, nous illustrons la nouvelle problématique grâce à l'exemple suivant.

Exemple 14. *Soit la requête* `SELECT A, mesure FROM AB WHERE B=valeur`. *Tel que nous l'avons défini, le coût minimal de cette requête correspond à la taille du cuboïde* `AB`. *Supposons maintenant que les index soient autorisés et que* `AB` *lui même ait un index sur l'attribut* `B`. *Dans ce cas, nous n'avons pas besoin de parcourir tout le cuboïde* `AB` *pour évaluer la requête. Nous avons même la possibilité d'avoir un accès quasi direct aux enregistrements qui satisfont la condition* `B=valeur`. *Dans ce cas, le coût de la requête correspond donc quasiment à la taille du résultat. Noter néanmoins que cette réduction du coût est au prix du stockage (et de maintenance) de l'index. Dans la littérature, on estime la taille d'un index par la taille du cuboïde qui correspond aux clés d'indexation (l'index sur la dimension* `A` *du cuboïde* `AB` *a la taille du cuboïde* `A` *[GHRU97]). Ceci est bien sûr une estimation qu'on peut modifier sans conséquence pour ce qui suit*[5].

Supposons maintenant que le cuboïde `ABC` *soit stocké et que celui-ci soit indexé sur* `B`. *On pourrait utiliser cet index pour évaluer notre requête. On aura cependant plus de lignes à parcourir qu'avec l'index de* `AB`. *Si l'on considère une répartition uniforme, chaque combinaison de valeurs de* `AB` *est associée à toutes les valeurs de* `C`. *Ainsi, le coût pour évaluer notre requête à partir de l'index sur* `ABC` *serait la taille de la réponse multipliée par la taille du cuboïde* `C`. *Cette estimation grossière (c'est en fait la borne supérieure) peut être raffinée de la façon suivante : Si* $\rho = \frac{size(ABC)}{size(AB)}$, *alors* ρ *représente le nombre moyen de valeurs de* C *avec lesquelles chaque combinaison de valeurs de* AB *est associée. Ainsi, le nombre de lignes de* ABC *qui sont consultées pour évaluer notre requête sera égal à son*

5. Il est clair que nous n'allons pas utiliser la même estimation que l'index soit dense ou épars, du type B-arbre ou bitmap.

coût minimal multiplié par $\rho = \frac{size(ABC)}{size(AB)}$.

l'exemple précédent nous a montré la nécessité de redéfinir la notion de coût minimal pour une requête et l'estimation du coût de celle-ci en utilisant un index sur un de ses ancêtres. Nous développons dans ce qui suit plus formellement ces notions.

Virtuellement, à chaque cuboïde ayant d dimensions, on peut associer $\sum_{i=1}^{d} \frac{d!}{(d-i)!}$ index différents, i.e., chaque permutation de chacun des sous ensembles de l'ensemble des dimensions peut former une liste de clés d'indexation.

Nous empruntons à [GHRU97] ses notations. Les requêtes sont de la forme $\gamma_{Dim1}\sigma_{Dim2}$ où $Dim1$ désigne les attributs présents dans la clause `SELECT` et $Dim2$ représente ceux apparaissant dans la clause `WHERE`. Les conditions de la clause `WHERE` sont des conjonctions d'égalités de la forme $Att_i = value$.

Soit $q = \gamma_{Dim1}\sigma_{Dim2}$ et I un index de c sur la séquence d'attributs $\overrightarrow{Dim}$. Soit E le plus grand sous ensemble de $Dim2$ tel que $\overrightarrow{Dim}$ forme un préfixe (pas nécessairement un préfixe propre) de Dim_2. Supposons que $(Dim_1 \cup Dim2) \subseteq Att(c)$, autrement dit, q peut être évaluée à partir de c. Le coût pour évaluer q à partir de c à travers l'index I est

$$Cost(q, c, I) = \frac{size(c)}{size(E)}$$

Le coût minimal de q est obtenu quand q est évaluée à partir de c tel que $Dim1 \cup Dim2 = Att(c)$ (i.e., à partir de c_q) et il existe un index I de c sur $Dim2$. Ainsi,

$$MinCost(q) = \frac{size(Dim1 \cup Dim2)}{size(Dim2)}$$

Noter que si $Dim2 = \emptyset$ alors $size(Dim2) = 1$ puisque ça correspond au cuboïde Apex qui a une taille égale à 1. Nous sommes maintenant prêts à définir le facteur de performance de l'évaluation de q en utilisant l'index I du cuboïde c.

$$pf(q, c, I) = \frac{Cost(q, c, I)}{MinCost(q)}$$

Maintenant nous pouvons étendre l'espace de recherche lors du lancement de l'algorithme de sélection de vues+index à matérialiser : la paire (c, I) est un $f_$ancêtre de q si et seulement si $pf(q, c, I) \leq f$. Les résultats présentés dans les sections précédentes peuvent maintenant directement être généralisés.

Noter cependant que l'espace de recherche va augmenter d'une manière considérable. Des techniques telles que celles présentées dans [GHRU97] (on ne considère que les index sur *tous* les attributs des cuboïdes) ou dans [TCFS08] pourront être utilisées afin de justement réduire l'espace de recherche.

III.6 Expérimentations

Dans cette section nous montrons quelques unes des expérimentations que nous avons entreprises pour valider notre approche.

Jeu de données : Nous avons travaillé sur la table *US Census 1990 data*, disponible à `http ://kdd.ics.uci.edu/`. Nous en avons exclu le premier attribut qui est une clé. La table de faits initiale contient 2.5 millions de tuples et 65 dimensions. Dans nos expériences, nous avons considéré les sous-tables `USData10` et `USData20` correspondant aux 10 et 20 premiers attributs. Un pré-calcul des tailles exactes a été fait pour tous les cuboïdes de `USData10` et `USData20` . Le temps d'exécution des différents algorithmes mesure le temps mis pour construire les graphes de recherche et le temps de calcul de la solution $\mathcal{S}$.

Méthodes de génération de $\mathcal{Q}$ Les différentes mesures de performances d'une solution $\mathcal{S}$ dépend en très grande partie des caractéristiques des requêtes *cibles* $\mathcal{Q}$. En préambule, on peut remarquer que si $\mathcal{Q}$ ne contient que des "gros" cuboïdes, c'est-à-dire $\forall q \in \mathcal{Q}$, $size(q) \geq |T|/f$, la solution $\mathcal{S} = \{c_b\}$ est une solution respectant la contrainte de facteur de performance et dont l'espace mémoire est inférieur à f fois l'espace mémoire de la solution optimale. Nous ne considérons donc pas ce type de requêtes fréquentes. Nous avons donc proposé 3 méthodes de génération aléatoire de $\mathcal{Q}$ qui traduise chacune des caractéristiques précises :

- **génération aléatoire uniforme (UNIF)** : toutes les dimensions ont la même probabilité d'apparaître dans le cuboïde tiré au hasard,
- **génération entre dimension 1 et dimension d_{max} puis aléatoire uniforme (DMAX)** : on se fixe un nombre maximal de dimensions. Pour i allant de 1 à d_{max}, on choisit un cuboïde à i dimensions de manière aléatoire uniforme ;
- **DESC** : on itère le même processus que DMAX mais pour chaque choix aléatoire d'un cuboïde q de niveau i, on ajoute à $\mathcal{Q}$ les 2^i cuboïdes $\{q_j\}_{j\in[1,2^i]}$ descendants de q, c'est-à-dire $q_j \preceq q$. Cette procédure de tirage aléatoire permet de simuler les

navigations du type `Roll_Up` et `Drill_Down` très courantes dans le cadre de l'analyse en ligne.

La méthode de génération UNIF est la plus simple et naturelle. Cependant, on peut montrer facilement qu'une proportion constante des cuboïdes tirés avec cette méthode sont de dimension comprise entre $D/2-\sqrt{D}$ et $D/2+\sqrt{D}$. Les cuboïdes extrêmes, c'est-à-dire petit ou grand en terme de dimension, sont improbables.

Puisque les cuboïdes grands peuvent facilement être résumés par le cuboïde de base, il reste à proposer des méthodes de génération tendant à sélectionner des cuboïdes de petite dimension. C'est l'objectif de DMAX et DESC. La méthode DESC se justifie par le fait qu'en pratique certains attributs sont plus fréquemment utilisés que d'autres.

III.6.1 Étude de USData10

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

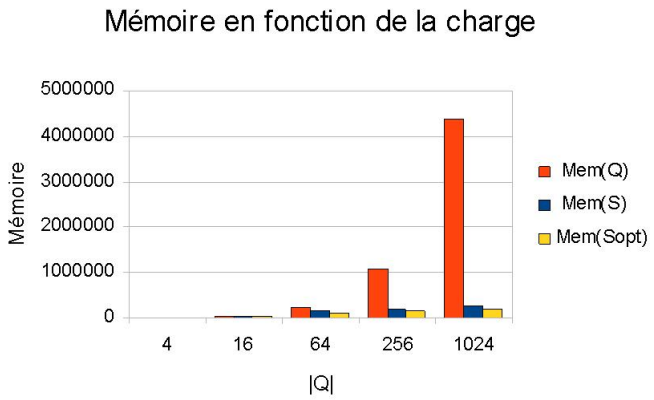


Figure III.4 – Comparaison entre PickFromfAncestors et la solution optimale.

comporte très bien par rapport à la solution optimale et le gain de mémoire par rapport à la matérialisation complète de $\mathcal{Q}$ est extrêmement important. Le véritable apport de PickFrom$f$Ancestors par rapport à la programmation linéaire est le temps d'exécution : pour PickFrom$f$Ancestors, cela va de $0.01s$ ($|\mathcal{Q}|$ =4) à $0.29s$ ($|\mathcal{Q}|$ =1024). Or le programme

linéaire exécuté sur CPLEX met de $0.1s$ ($|\Omega| = 4$) à $3366s$ ($|\Omega| = 1024$) [6]. En dimension supérieure, la programmation linéaire prend beaucoup trop de temps. En dimension ≥ 20, PickFrom$f$Ancestors donne une solution en temps inférieur à la minute. Bien entendu, cela dépend de la valeur de f et de $|\Omega|$. C'est l'objectif de la section suivante.

III.6.2 Étude de USData20

Nous avons mené un certain nombre d'expériences sur ce jeu de données afin de mettre en évidence quelques propriétés. Ce jeu de données a un cuboïde de base qui contient 500K lignes. Pour chaque type de génération de requêtes, nous avons fait varier $|\Omega|$ de 4 à 1024 par puissance de 4. Pour chaque valeur de $|\Omega|$, nous avons fait varier le facteur f de 2 à 16 en le doublant à chaque fois. Enfin, pour chaque combinaison des trois paramètres précédents, nous avons calculé la solution avec le graphe G ainsi qu'avec le graphe partiel G_p.

Par construction, l'espace mémoire de Ω dépend de la méthode de génération. Par exemple, pour 1024 cuboïdes, $|\Omega|$ est de l'ordre de 6.10^7 pour les méthodes UNIF et DMAX pour $d_{max} = 10$ et devient 5.10^6 pour les méthodes DMAX et DESC pour $d_{max} = 8$.

Temps de calcul de $\mathcal{S}$

Les résultats des temps d'exécution (exprimés en secondes) sont illustrés dans les figures III.5 à III.8. Ces courbes illustrent le gain en temps d'exécution lorsque l'on utilise G_p au lieu de G.

Le graphe de recherche partiel offre un très bon compromis en terme de temps de calcul et gain de mémoire. Il mène à une solution avec un temps de calcul environ 4 fois inférieur à celui du graphe de recherche complet avec un facteur de gain en mémoire comparable. Nous n'avons rencontré qu'une exception (cf. Figures III.12 et III.8). Dans ce cas-là, le graphe de recherche partiel "résume mal" car il propose une solution avec beaucoup de cuboïdes. Comme le temps de calcul $O(\Delta|V(G)| \cdot |\mathcal{S}|)$ peut dépendre du nombre de cuboïdes de la réponse, il est peut être plus important pour G_p si la solution qu'il donne a une cardinalité plus grande que celle donné par G. En règle générale, le graphe G_p est bien plus petit que G (degré maximal et nombre de sommets sont plus petits). Ceci explique la plus grande rapidité de l'algorithme approché si on prend en entrée le graphe G_p.

6. LP_Solve était incapable de résoudre des programme d'une taille aussi grande.

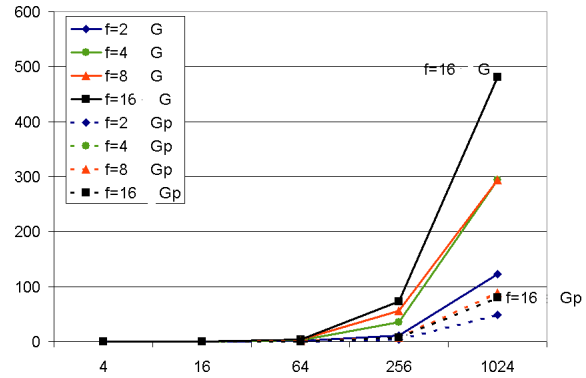


Figure III.5 – Temps de calcul de $\mathcal{S}$ si $\mathcal{Q}$ généré par la méthode UNIF.

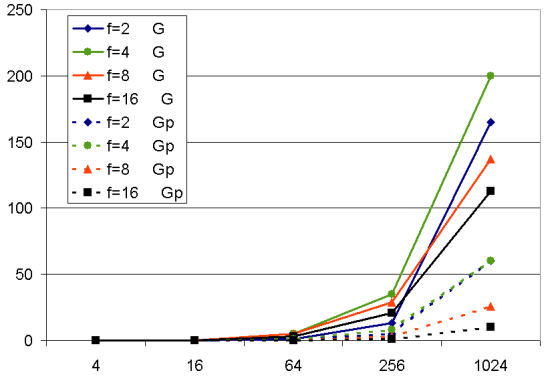


Figure III.6 – Temps de calcul de $\mathcal{S}$ si $\mathcal{Q}$ généré par la méthode DMAX avec $d_{max} = 10$.

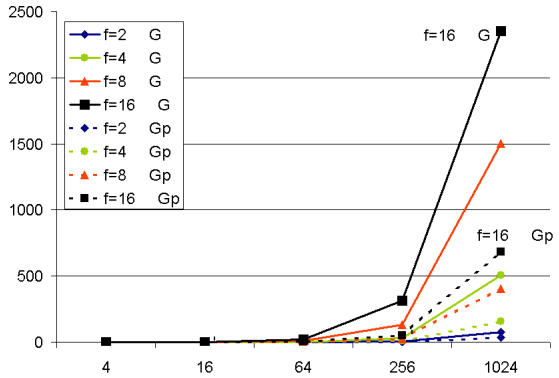


Figure III.7 – Temps de calcul de $\mathcal{S}$ si $\mathcal{Q}$ généré par la méthode DMAX avec $d_{max} = 8$.

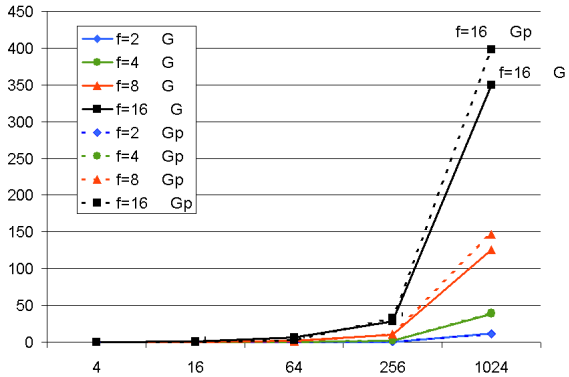


Figure III.8 – Temps de calcul de $\mathcal{S}$ si $\mathcal{Q}$ généré par la méthode DESC avec $d_{max} = 8$.

Gain de mémoire de $\mathcal{S}$ par rapport à $\mathcal{Q}$

Dans cette partie on décrit les gains en espace mémoire qu'on obtient en faisant varier les quatre paramètres utilisés précédemment. Noter que les expériences décrites ici sont celles de la section précédente. Une première remarque consiste à noter le fait qu'avec G on réduit toujours plus l'espace mémoire (cf Théorème 2). On note aussi que dans la plupart des cas (les 3 premiers), les ratios de réduction d'espace mémoire en utilisant G ou G_p sont comparables sauf dans le dernier (figure III.12) où l'on a une nette différence. Rappelons que c'est aussi le cas où le temps d'exécution avec G était meilleur que celui avec G_p (voir III.8). Ce qui est frappant, c'est la différence des ratios de réduction de mémoire en fonction du type de génération de $\mathcal{Q}$. En effet, lorsque *beaucoup* de requêtes de $\mathcal{Q}$ ont *beaucoup* de dimensions, l'ensemble $\mathcal{Q}$ est plus résumable.

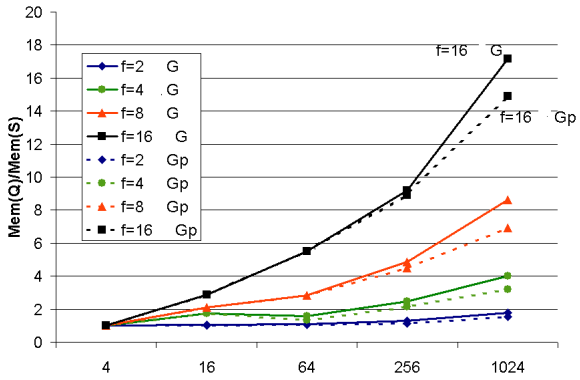


Figure III.9 – Gain de mémoire de $\mathcal{S}$ si $\mathcal{Q}$ généré par la méthode UNIF

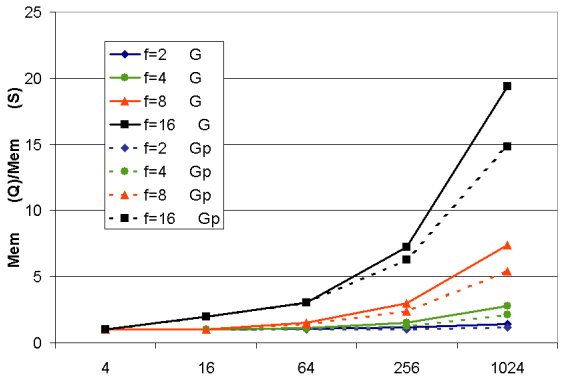


Figure III.10 – Gain de mémoire de $\mathcal{S}$ si $\mathcal{Q}$ généré par la méthode DMAX avec $d_{max} = 10$.

III.6.3 Un premier bilan

La table III.1 dresse un bilan qualitatif des différentes expériences. $\mathcal{Q}$ est considéré comme hétérogène si chaque attribut a la même probabilité d'apparition dans les cuboïdes requêtes. C'est le cas pour toutes les méthodes de génération à l'exception de DESC.

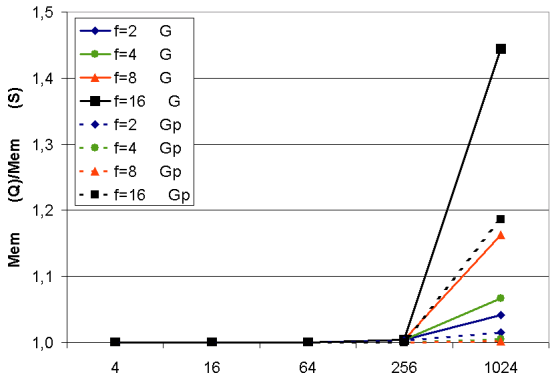


Figure III.11 – Gain de mémoire de $\mathcal{S}$ si $\mathcal{Q}$ généré par la méthode DMAX avec $d_{max} = 8$.

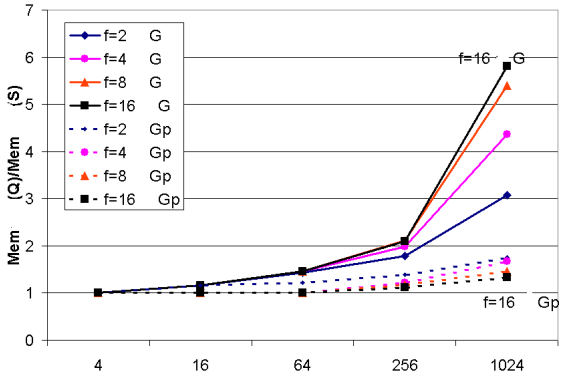


Figure III.12 – Gain de mémoire de $\mathcal{S}$ si $\mathcal{Q}$ généré par la méthode DESC avec $d_{max} = 8$.

génération de $\mathcal{Q}$	Hétérogénéité de $\mathcal{Q}$	$\mathcal{Q}$ Résumable	Temps de calcul de $\mathcal{S}$
UNIF	oui	oui	faible
DMAX8	oui	non	important
DMAX10	oui	oui	faible
DESC8	non	oui	faible

Table III.1 – Comparaison Temps/Mémoire pour les différentes caractéristiques de $\mathcal{Q}$.

Le seul cas pour lequel $\mathcal{Q}$ ne semble pas résumable est lorsque $\mathcal{Q}$ est hétérogène et constitué uniquement de petits cuboïdes en terme de nombre d'attributs.

III.6.4 Analyse de la stabilité

Pour vérifier la stabilité des solutions retournées par notre approche nous avons conduit un certain nombre d'expériences dont le principe est le suivant : on fixe un facteur de performance f, on fixe le nombre de requêtes dans la charge, puis on fait tourner notre algorithme sur différents fichiers de données $fichier_1, \ldots, fichier_n$ tels que $fichier_1 \subset \cdots \subset fichier_n$. Le fichier de départ $fichier_n$ peut représenter une table de faits ou un cuboïde de base. Pour chaque $fichier_i$, on calcule une solution $\mathcal{S}_i$ ainsi que, pour tous les éléments de $\mathcal{Q}$, le facteur de performance $f(q, \mathcal{S}_{i-1})$. Autrement dit, on vérifie dans quelle mesure les performances de $\mathcal{S}_{i-1}$ se *détériorent* quand on insère un certain nombre de tuples pour passer de $fichier_{i-1}$ à $fichier_i$.

Les critères de comparaison que nous avons retenus sont (1) le nombre de requêtes cibles $\mathcal{Q}$ dont le facteur de performance avec $\mathcal{S}_{i-1}$ dépasse le facteur f fixé par l'utilisateur et (2) les dépassements moyen et maximal par rapport à f. Bien sûr, ces deux mesures sont calculées par rapport à $fichier_i$.

Nous avons répété ces expériences en faisant varier les requêtes cibles et les valeurs du facteur de performance. Nous présentons ici juste le cas où $|\mathcal{Q}| = 512$. Les requêtes sont tirées d'une manière aléatoire uniforme dans $DC(T)$ et un facteur de performance maximal fixé par l'utilisateur qui est égal à $f = 4$ et à $f = 8$. Le jeu de données initial est USData-20.

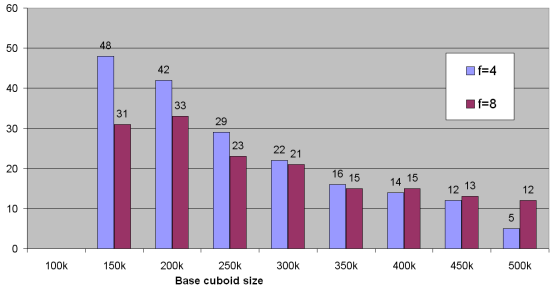


Figure III.13 – Requêtes ne satisfaisant plus le seuil fixé par l'utilisateur.

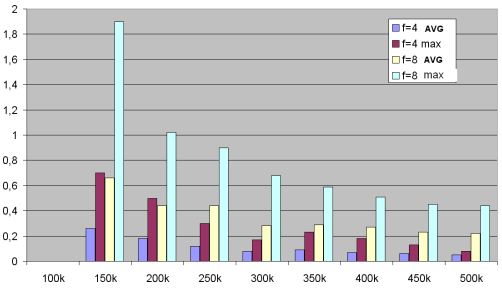


Figure III.14 – Les dépassements.

La figure III.13 montre que le nombre de requêtes qui voient leur facteur de perfor-

mance dépasser le seuil fixé par l'utilisateur quand elles sont évaluées comparativement à une solution calculée précédemment. Par exemple, on a calculé une solution avec le fichier dont le cuboïde de base a une taille de 100k, puis on a compté le nombre de requêtes, parmi les 512 de la charge, dont le facteur de performance dépasse le seuil après avoir inséré 50k tuples dans le cuboïde de base. En passant de 100k à 150k, le nombre de dépassements est égal à à 48 et 31 quand f est égal à respectivement 4 et 8. Il est intéressant de noter que le nombre de dépassements est inversement proportionnel au *ratio* d'augmentation de la taille du cuboïde de base. En effet, pour passer de 100k à 150k, nous avons ajouté la moitié de la taille de 100k, alors que pour passer de 450k à 500k, la ratio d'augmentation est de 1/9. Une autre remarque peut être faite sur l'impact combiné du facteur de performance et de la taille du cuboïde de base : en effet sur le graphe, on voit que pour les "petites" tailles, le nombre de dépassements est inversement proportionnel à la taille (ex : 48 avec $f = 4$ et 31 avec $f = 8$ à 150k) mais cette tendance va s'inverser à partir d'une certaine taille (ex : 5 avec $f = 4$ et 12 avec $f = 8$ à 500k).

Bien que le théorème 3 soit valide sous certaines hypothèses et qu'il ne considère que les dépassements ≥ 1, il permet d'expliquer le fait qu'une solution avec un paramètre $f = 4$ soit plus stable qu'une solution avec un paramètre $f = 8$ si la table de faits est grande (ici, vers 400K). Si la table est plus petite, un certain nombre de cuboïdes requêtes ont une taille bien inférieure à leur taille maximale. La figure III.15 montre l'évolution du rapport de taille $\frac{q'}{q}$ si la taille du cuboïde de base augmente. Nous avons sélectionné arbitrairement quelques couples de cuboïdes (q', q) tels que q' est ancêtre de q. Systématiquement, le rapport de taille augmente fortement quand la taille du cuboïde de base est petite puis tend à se stabiliser car les cuboïdes commencent à approcher leur taille maximale à force d'ajouter des tuples.

La figure III.14 montre les écarts entre le seuil fixé par l'utilisateur et les facteurs de performances des cuboïdes qui dépassent ce seuil suite aux insertions. On note que pour les deux valeurs de f cet écart a tendance à décroître. On note cependant le même phénomène qu'auparavant : à savoir que la différence des écarts moyens s'inverse à partir d'une certaine taille. En effet, au début, l'écart moyen de $f = 4$ est supérieur à celui de $f = 8$, puis la tendance s'inverse à partir de 250k.

La figure III.14 montre les écarts entre le seuil fixé par l'utilisateur et les facteurs de performances des cuboïdes qui dépassent ce seuil suite aux insertions. On note que pour les deux valeurs de f cet écart a tendance à décroître. On note cependant le même phénomène qu'auparavant : à savoir que la différence des écarts moyens s'inverse à partir d'une certaine taille. En effet, au début, l'écart moyen de $f = 4$ est supérieur à celui de

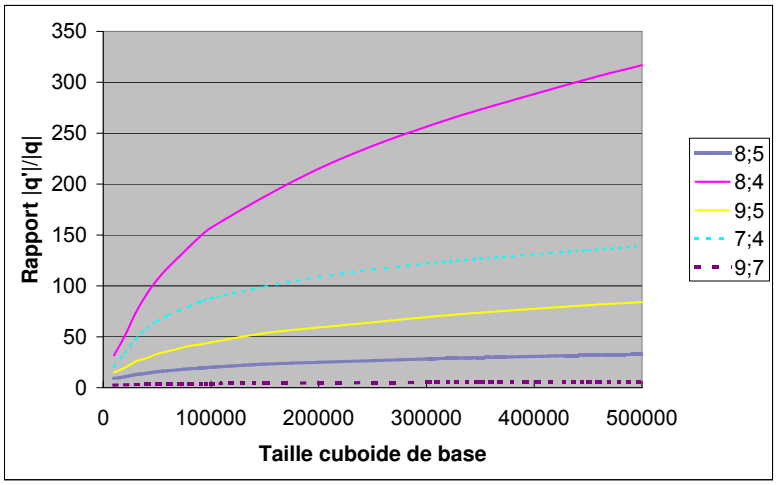


Figure III.15 – Évolution du rapport de tailles de différents paires de cuboïdes (q', q) telles que $q \subset q'$ lors d'ajouts de tuples. Les dimensions d'une paire (q, q') dont notés $Dim(q'); Dim(q)$.

$f = 8$, puis la tendance s'inverse à partir de 250k.

III.7 Conclusion

Nous avons présenté dans ce chapitre des techniques de sélection de vues (et index) pour l'optimisation d'une classe de requêtes sur les cubes de données. Nous sommes efforcés de présenter dans chaque cas les garanties de chaque solution. Dans notre démarche, nous avons tenté d'allier rigueur et pragmatisme. Ainsi, nos résultats présentent une partie théorique ainsi que des expérimentations sur des jeux de données réelles.

Au cours de nos investigations, nous avons été confrontés au problème du calcul ou estimation des tailles des objets manipulés (les cuboïdes). Ce problème est souvent éludé dans les travaux relatifs ou au mieux, juste mentionné en affirmant que c'est un sujet annexe. Or c'est une composante fondamentale pour tout algorithme de sélection de vues à matérialiser. Durant nos recherches bibliographiques, nous avons rencontré deux types de travaux sur l'estimation des tailles : (1) ceux permettant d'estimer correctement la taille d'un cuboïde en utilisant le moins de ressources mémoire possible (exemple : [DF03, KNW10]) permettant ainsi d'estimer la taille de plusieurs cuboïdes lors d'un seul parcours

de la table de faits et (2) les travaux sur le calcul des bordures dans le cadre des ensembles fréquents (exemple : [Bay98]). Nous avons pensé tirer profit des deux types de travaux, i.e., calculer des bordures en utilisant des algorithmes conçus pour les ensembles fréquents et replacer le calcul de support par l'estimation de la taille. L'intégration s'est avérée non pertinente car les algorithmes d'extraction de fréquents maximaux sont fondamentalement séquentiels, du moins les plus efficaces, dans le sens où ils vérifient le support d'un seul candidat à chaque itération. Ceci nous a poussé à concevoir un nouvel algorithme pour l'extraction des fréquents maximaux qui a la propriété d'être facilement parallélisable donc directement utilisable pour le calcul des bordures de cubes de données. C'est l'objet du chapitre suivant.

Chapitre IV

Vers un calcul parallèle des bordures

IV.1 Introduction

L'extraction des motifs fréquents [AS94] est l'un des problèmes les plus étudiés en fouille de données. Dans ce chapitre, nous nous intéressons aux ensembles fréquents (en anglais : frequent itemsets). Nous rappelons brièvement le problème : étant donné un ensemble d'items $\mathcal{I}$ et une table de transactions $\mathcal{T} = \{T_1, \ldots, T_M\}$ un multi-ensemble tel que $T_j \subseteq \mathcal{I}$ est une transaction. Le support d'un ensemble d'items $T \subseteq \mathcal{I}$ est le nombre de transactions $T_j \in \mathcal{T}$ tel que $T \subseteq T_j$. Le problème consiste à trouver tous les ensembles d'items I dont le support, c'est-à-dire la fréquence, est plus grand qu'un seuil minimal σ fixé par l'utilisateur. Les ensembles d'items qui satisfont cette condition sont dits *fréquents*. Comme le nombre d'ensembles des items fréquents peut être très grand (il peut atteindre 2^n si n est le nombre d'items présents dans $\mathcal{T}$), plusieurs classes de ces derniers ont été définies dans la littérature. Parmi ceux-là, on a les fréquents *close* [PBTL99] et les fréquents *maximaux* [Bay98, MT97]. Un ensemble d'items fréquents est close si aucun de ses sur-ensembles n'a le même support et il est maximal si aucun de ses sur-ensembles n'est fréquent. L'avantage des ensembles des items close est qu'ils permettent, si on connait leurs supports respectifs, de dériver tous les fréquents ainsi que leurs supports alors que les fréquents maximaux permettent de déduire les fréquents mais pas leurs supports respectifs. A contrario, les fréquents maximaux peuvent être beaucoup moins nombreux que les fréquents close. Ainsi, dans les applications où ce qui nous intéresse c'est juste le fait de savoir si un ensemble d'items est fréquent ou pas, les fréquents maximaux peuvent être considérés comme des résumés *parfaits* des ensembles fréquents. Il est clair que l'ensemble des fréquents close est inclus dans les fréquents maximaux. La figure IV.1 illustre ces deux concepts. A gauche, nous avons la table de transactions et nous considérons $\sigma = 2$.

A chaque ensemble des items, les numéros de transactions où il apparait sont affichés. Par exemple, à côté de l'itemset AD on voit 24 qui signifie que AD est présent dans les transactions 2 et 4. Il a donc un support égal à 2 et donc il est fréquent.

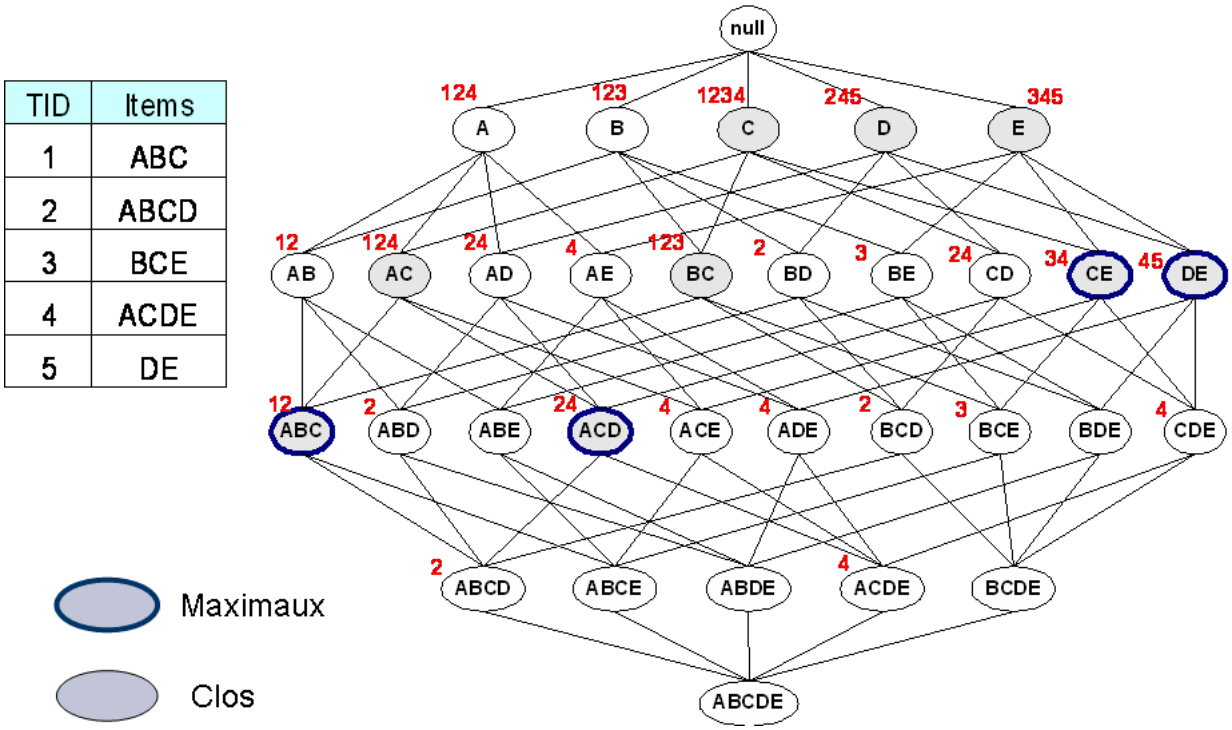


Figure IV.1 – Relations en itemsets fréquents, close et maximaux

Dans la littérature [MT97], l'ensemble des fréquents maximaux est aussi appelé *bordure positive* par opposition à la *bordure négative* qui elle correspond à l'ensemble des itemsets non fréquents minimaux (en termes d'inclusion). Ainsi, un itemset appartient à la bordure négative s'il n'est pas fréquent et si au moins un de ses sous-ensembles (ayant le même nombre d'éléments moins 1) est fréquent.

Bien qu'en pratique, les fréquents maximaux soient bien moins nombreux que les fréquents, en théorie leur nombre peut être du même ordre. Il suffit pour cela de considérer le cas où les fréquents maximaux sont tous les itemsets de taille[1] $n/2$ avec n qui est le nombre d'items. Le nombre de fréquents maximaux est donc de $\binom{n}{\frac{n}{2}}$. Ce nombre est de l'ordre de $O(\frac{2^n}{\sqrt{n}})$. Ceci montre aussi que théoriquement, le calcul des fréquents et le calcul des fréquents maximaux ont quasiment la même complexité dans le pire cas qui est en $O(2^n)$ pour les fréquents. D'autre part, l'exemple ci-dessous, tiré de [Yan04], montre que

1. Nombre d'items qui les composent.

la bordure positive peut être exponentiellement plus grande que la base de transactions elle-même.

Exemple 15. *Soit* $\mathcal{I} = \{I_1, I_2, \ldots I_{2n-1}, I_{2n}\}$ *l'ensemble des* $2n$ *items d'une base de transaction* $\mathcal{T}$ *avec* $2n$ *transactions :* $T_1, T_2, \ldots, T_{2n-1}, T_{2n}$*. Les transactions sont de la forme :* $T_i = \mathcal{I} - \{I_i\}$*, pour* $1 \leq i \leq 2n$*, i.e., la transaction* T_i *a tous les items de* $\mathbb{I}$ *sauf* I_i*. Dans ce cas, le nombre des itemsets n-fréquents maximaux dans* $\mathcal{T}$ *est* $\binom{2n}{n}$*. En effet, nous observons que pour chaque itemset* $J \subseteq \mathcal{I}$ *les transactions qui ont l'itemset* J *sont de ce type :* $\mathcal{T}(\mathcal{J}) = \{\mathcal{I}_i | I_i \notin \mathcal{J}, I_i \in \mathcal{I}\}$*. Cela implique que si* $|\mathcal{J}| = k$*, alors* $f(\mathcal{J}) = 2n - k$*. L'itemset* $\mathcal{J}$ *est n-fréquent maximaux, ssi* $|\mathcal{J}| = n$*. On voit bien, que chaque sur-ensemble d'itemsets n-fréquents maximaux* $\mathcal{J}$ *a le support inférieur à* n*. Donc le nombre des itemsets n-fréquents est* $\binom{2n}{n}$*.* $\binom{2n}{n} = \frac{(2n)!}{n! \times n!} \geq 2^n$*. En résumé, pour une base de transaction de taille* n*, nous avons obtenu un facteur exponentiel de n-fréquents maximaux.*

Même si le problème de calcul des fréquents maximaux est important en lui même, il l'est d'autant plus qu'on le retrouve dans différentes applications. Par example [ACN00, HMT09d] utilise la notion de bordure pour sélectionner les vues à matérialiser dans le cadre de l'optimisation des requêtes, et [NCCL09, DL05] l'utilise pour identifier les motifs émergents entre deux états consécutifs d'un cube de données. La référence [GKM+03] donne d'autres application de ce même problème [2].

Comme le problème du calcul des bordures est NP-difficile [Yan04, GKM+03], les optimisations que l'on peut faire portent sur le bon choix des structures de données, une analyse très fine de l'utilisation de la mémoire et enfin l'utilisation d'heuristiques qui s'avèreront utiles lors de la manipulation de jeux de données réels. Ce sont les approches adoptées par la plupart des propositions faites jusqu'à présent. Une autre direction qui a été peu poursuivie consiste en l'exploitation des nouvelles architectures des ordinateurs. En effet, actuellement, les machines sont de plus en équipées d'un processeur à plusieurs cœurs [3] voire de plusieurs processeurs à plusieurs cœurs. Chaque core est une unité de traitement pouvant exécuter une portion du programme (*thread*). Dans certains cas, un seul core peut même [4] exécuter deux threads simultanément avec la technique de *multithreading*. D'autre part, ces nouvelles architectures ont plusieurs niveaux de mémoire cache

2. Un élément de la bordure y est appelé : *most interesting sentence*.
3. Nous utiliserons par la suite le terme anglais *core*.
4. Voir les caractéristiques des processeurs Xeon d'Intel qui sont construits selon la technologie Nehalem.

selon la *proximité* par rapport au core et selon que le niveau qui est partagé par plusieurs cores. Ainsi, l'accès aux données se trouvant en mémoire centrale doit être analysé comme si cette dernière était l'équivalent de la mémoire secondaire lorsqu'on ne considère que les deux niveaux : mémoire centrale et mémoire secondaire. Les nouvelles machines sont donc capables d'exécuter plusieurs threads en parallèle et cette propriété n'est pas bien exploitée, d'après ce que nous avons pu trouver dans la bibliographie, par les algorithmes d'extraction de bordures. C'est l'objet de ce chapitre dans lequel nous développons un nouvel algorithme se prêtant facilement à une exécution parallèle tirant ainsi profit de la disponibilité de plusieurs processeurs dans une même machine.

Tout algorithme d'extraction d'itemsets fréquents maximaux (IFM) a son temps d'exécution qui est composé de deux parties : *la gestion des candidats* et *le calcul du support des candidats*. Quand la taille des données est importante (nombre d'items, nombre de transactions et la taille moyenne des transactions), le calcul des supports est la partie qui prend le plus de temps. Beaucoup d'efforts a été dévolue à l'optimisation de cette partie. Par exemple, les FP-trees [HPY00] sont une structure de données très efficace pour résumer les tables de transactions tout en gardant l'information nécessaire permettant le calcul des supports. Cependant, cette structure est destinée à résider en mémoire centrale et est difficilement extensible à un contexte où la mémoire disponible n'est pas assez large pour pouvoir y charger ces arbres préfixes comme cela a été discuté dans [HPYM04] par les auteurs qui les ont proposés. D'autres structures comme la représentation verticale [ZG03] ou les bitmaps [BCG01] ont elles aussi montré leur efficacité. Reste que pour les algorithmes travaillant sur des données en mémoire centrale, les FP-trees s'avèrent les plus efficaces.

La parallélisation du calcul des IFM peut se réaliser de deux manières :

- Une solution triviale consisterait à subdiviser les données en plusieurs parts puis à chaque fois qu'on a besoin de calculer le support d'un itemset candidat, on lance autant de processus qu'il y a de parts, chacun calculant un support partiel. Par la suite, il suffit de sommer ces supports partiels pour obtenir le support global. L'efficacité de cette solution n'est pas toujours garantie. En effet, si l'on considère une représentation des données sous forme de liste de transactions, là en effet la subdivision est immédiate. Or cette structure de données est celle qui nécessite le plus de temps de calcul. A contrario, les arbres préfixes (e.g. FPTrees) résument très bien les données par contre leur subdivision n'est pas triviale ; il se peut que l'arbre global ait la même taille que les arbres partiels ce qui rend le parallélisme inefficace.
- La deuxième solution consiste à partitionner les candidats de sorte qu'à chaque

itération on puisse lancer le calcul de plusieurs itemsets simultanément. Ainsi, les processus exécutés en parallèle accèdent tous à la globalité des données. Le partitionnement des candidats est plus ou moins trivial dans le cas des approches par niveau (*level wise*) ; il suffit de subdiviser les candidats d'un niveau en plusieurs parts. Or l'approche par niveaux n'est pas efficace pour le calcul des bordures car elle souffre du nombre important de candidats qu'elle doit générer et donc elle ne permet pas de bien exploiter l'élagage descendant (un sous-ensemble d'un ensemble fréquent n'est pas un IFM, donc il n'est pas nécessaire de calculer son support). Ainsi, notre solution consiste en un partitionnement des candidats qui *imite* le parcours en profondeur dans le sens où il traite exactement les mêmes candidats tout en les regroupant dans des parts.

IV.1.1 Travaux relatifs

Les algorithmes d'extraction des IFM peuvent être répartis essentiellement en trois catégories

1. les algorithmes de recherche en profondeur (ou DFS pour Depth First Search) par exemple Mafia [BCG01], GenMax [GZ05], DepthProject [AAP00], SmartMiner [ZCL02], FPmax* [GZ05] et PADS [ZPWL09],
2. les algorithmes de recherche en largeur (ou BFS pour Breadth First Search) par exemple Pincer search [LK98] et MaxMiner [Bay98] et enfin
3. les algorithmes basés sur la dualisation par exemple Dualize and Advance [GKM+03, SU03]).

Il ressort de la bibliographie que même si le problème d'extraction des IFM n'est pas un problème récent, il continue à susciter de nouvelles contributions. Après la multitude d'algorithmes proposés, la communauté de chercheurs du domaine a mis en place une série de deux workshops (FIMI'03 et FIMI'04) spécialement dédiés à l'analyse et la comparaison de ces algorithmes. En effet, une comparaison analytique de toutes ces propositions s'est avérée difficile. Les expériences sur des jeux de données synthétiques et réels ont confirmé cette difficulté puisque selon le jeu de données et selon le support minimal choisis, tel algorithme peut s'avérer meilleur que l'autre et vice versa. Pire, selon l'implémentation du même algorithme, on pouvait avoir des performances très différentes. Ces rencontres ont aussi permis de mettre en évidence la difficulté de prédire le comportement d'un algorithme selon le *type* de données (la distribution). Ceci a motivé le travail de [FMP05]

qui a proposé une classification des données en fonction de la distribution des bordures négative et positive. Cela a permis aux auteurs d'expliquer quelques unes des raisons de la bonne ou mauvaise performance de certains algorithmes.

Ce qui suit représente les limitations de chacune des trois catégories d'algorithmes citées plus haut :

- pour les algorithmes en DFS, le problème est essentiellement le fait qu'à chaque étape, un seul candidat est pris en compte. Ceci est intéressant d'une point de vue utilisation de la mémoire par contre l'approche ne se prête pas à une parallélisation du calcul de plusieurs candidats simultanément.
- pour les approches BFS, le problème réside dans le nombre important de candidats à générer : Si un IFM est composé de m items alors tous ces sous-ensembles i.e., les 2^m, doivent être générés et leur support calculé.
- l'approche par dualisation consiste à alterner les calculs des bordures négative et positive en fonction des connaissances acquises au cours du traitement. L'estimation d'une bordure positive à partir d'une bordure négative estimée, et vice versa, revient essentiellement à calculer des transversaux d'hypergraphes [EGM03]. A notre connaissance, il n'existe pas actuellement d'algorithme efficace permettant de réaliser ce calcul.

Il convient de mentionner les propositions de [FMP04] qui consistent à d'abord appliquer une méthode BFS puis changer dynamiquement en passant à la méthode par dualisation dès qu'un niveau est atteint. Le choix du niveau à partir duquel on change de méthode reste problématique. [ZEH05, EHZ06] proposent une méthode qui "suppose" un ensemble d'itemsets comme étant des IFM (en fait ce sont les branches du FPtree). Si un de ces itemsets est effectivement fréquent, alors c'est un IFM sinon, on l'intersecte avec d'autres IFM afin d'obtenir de nouveaux candidats. On voit ainsi que cette technique n'adopte pas de parcours particulier (d'où le nom de *leap traversal* pour signifier des sauts).

IV.1.2 DFS versus BFS

DFS et BFS ont la même complexité dans le pire cas même si en pratique DFS s'avère être plus efficace. Pour illustrer notre propos, considérons $\mathcal{I} = \{A, B, C, D, E\}$ comme étant l'ensemble des items. L'espace de recherche peut être représenté par l'arbre lexicographique [Rym92] de $\mathcal{I}$. Cet arbre est illustré par la Figure IV.2.

Supposons que les IFM soient $\{ABCD, E\}$. Comme **DFS** procède en profondeur, il va calculer le support de tous les sur-ensembles de E. Ainsi, pour une bordure de taille

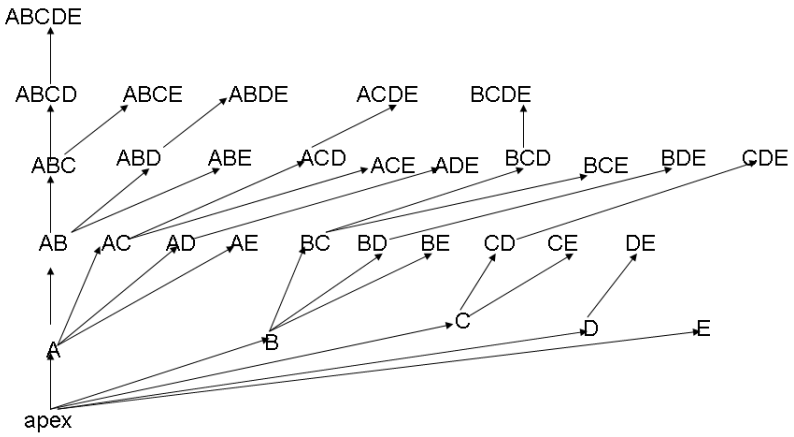


Figure IV.2 – L'arbre lexicographique de $\{A, B, C, D, E\}$

2, il calcule $O(2^{n-1})$ supports où n est le nombre d'items. Noter que le BFS quant à lui ne va pas tester les sur-ensembles de E, du moins seulement les parents directs de E, par contre il teste tous les sous-ensembles de $ABCD$. Cet exemple montre que quand on a de très longs IFM en même temps que de très courts IFM, les deux approches DFS et BFS ne sont pas efficaces.

Supposons maintenant que la bordure positive est composée d'un seul élément qui est $ABCDE$. Dans ce cas, le DFS calcule n supports alors que le BFS en calcule 2^n. Il est facile de vérifier que dans tous les cas, le nombre de calculs de supports effectués par DFS est inférieur ou égal au nombre de calculs effectués par BFS.

Noter qu'à chaque niveau, BFS peut calculer en parallèle le support de tous les candidats du niveau. Ainsi, si nous disposons de p processeurs où p correspond au nombre maximum de candidats que l'on peut avoir dans un niveau i.e., $\binom{n}{n/2}$ (le niveau du milieu), les deux algorithmes ont les mêmes performances.

En conclusion de cette section, nous notons deux remarques essentielles : d'un côté, DFS génère et traite moins de candidats que BFS et de l'autre côté, BFS est facilement parallélisable.

IV.1.3 Organisation du chapitre

Dans la section IV.2, nous introduisons quelques notations et en section IV.3 nous décrivons brièvement l'algorithme DFS qui sera qui sera notre référence en termes de nombres de candidats générés. Dans la section IV.4, nous décrivons l'algorithme que nous

proposons **MineWithRounds**. Cet algorithme est basé sur une partition des candidats en $2n-1$ parts. Cette nouvelle partition offre des propriétés décrites dans le Théorème 6 et permet donc de garantir, sous certaines conditions, une solution parallèle optimale dans le sens où le temps d'exécution est divisé par le nombre de processeurs. En effet, le temps d'exécution de tout algorithme d'extraction des IFM est de la forme $T = T_c + T_s$ où T_c est le temps nécessaire à la génération des candidats et T_s est le temps consacré au calcul des supports. Si l'on suppose que le calcul de tout support, quel que soit le candidat, prend la même unité de temps, alors nous montrons le résultat suivant :

Théorème 5. $T_s(p) \leq \frac{T_s(1)}{p} + 2n - 1$

Bien sûr, cette hypothèse n'est pas toujours vérifiée (le temps de calcul du support peut dépendre de la longueur de l'itemset candidat). Nous n'avons par contre pas la même garantie quant à la partie du temps consacrée à la gestion des candidats, i.e., $T_c(p)$. Cependant, dans la Section IV.5, nos expériences montrent que lorsque les bordures sont larges (i.e., les supports minimaux sont bas), notre algorithme permet d'avoir de bonnes accélérations du temps d'exécution.

IV.2 Préliminaires

Nous définissons d'abord un ordre total sur les items.

Définition 20 (Item Rank)**.** *Soit $\lhd$ un ordre total sur les items apparaissant dans la base de transactions. Si I est un itemset alors $Rank(I) = r$ si et seulement si $|\{J \mid J \lhd I\}| = r - 1$.*

Exemple 16. *Soit $\mathcal{I} = \{A, B, C, D\}$ l'ensemble des items. Supposons que $\lhd$ soit l'ordre alphabétique. Alors, $Rank(A) = 1, Rank(B) = 2, \ldots, Rank(D) = 4$.*

Sans perte de généralité, dans la suite nous allons toujours considérer $\lhd$ comme étant l'ordre alphabétique. Nous allons aussi ne considérer que les itemsets *ordonnés* i.e. si $I = I_1 I_2 \ldots I_k$ est un k-itemset alors $j < \ell \Leftrightarrow I_j \lhd I_\ell$. Par exemple, ACD est un itemset ordonné alors que DAC ne l'est pas. Comme à l'accoutumé, à partir de $\lhd$ nous définissons l'ordre lexicographique sur les itemsets : Soit $I = I_1 I_2 \ldots I_m$ et $J = J_1 J_2 \ldots J_\ell$ deux itemsets ordonnés. On dit que I précède J, noté $I \prec J$, si et seulement s'il existe p tel que $I_k = J_k$

pour $k < p$ et $I_p \lhd J_p$. Par exemple, $ABD \prec B$. De plus, I est un ancêtre de J ssi $I \supseteq J$ et I est un parent de J ssi I est un ancêtre et $|I| = |J| + 1$. Par exemple, ABC est un ancêtre de B et AB est un parent de B. Relativement à un itemset I et à l'ordre $\prec$ nous distinguons deux types d'ancêtres (resp. parents) : ceux qui précèdent I, appelés *ancêtres gauches* et ceux que I précède appelés *ancêtres droits*. Par exemple, AB et BC sont tous les deux parents de B. Comme $AB \prec B \prec BC$, AB est un parent gauche de B et BC en est un parent droit.

On dit qu'un ensemble d'itemsets $\mathcal{S}$ *couvre* un itemset I ssi $\mathcal{S}$ contient un ancêtre de I. Nous définissons la fonction *Pos* qui retourne la position d'un itemset comme suit :

Définition 21. *Soit* $(2^{\mathcal{I}}, \prec)$ *l'ensemble ordonné des itemsets.* $Pos(I) = |\{J \mid J \prec I\}|$.

Par exemple, si $\mathcal{I} = \{I_1, \ldots, I_n\}$ et $I_1 \lhd I_2 \lhd \cdots \lhd I_n$ alors $Pos(\{I_1\}) = 1$ (I_1 suit $\emptyset$), $Pos(\{I_n\}) = 2^n - 1$ et $Pos(\{I_1, I_2\}) = 2$. De ces définitions, la structure de l'arbre lexicographique devient facile à décrire : $\mathcal{T}$ est l'arbre lexicographique de l'ensemble d'items $\mathcal{I}$ ssi la racine de $\mathcal{T} = \emptyset$ (ou apex) et pour chaque itemset $I \subseteq \mathcal{I}$, tous les ancêtres droits de I sont dans le sous-arbre enraciné à I. Les nœuds du même niveau sont quant à eux triés selon l'ordre $\prec$.

Enfin, soit $I = I_1 \ldots I_m$ un itemset ordonné. Le successeur de I, noté $Sucessor(I)$, est l'itemset $J = I_1 \ldots I_m I_k$ tel que $rank(I_k) = rank(I_m) + 1$. Il est clair que tous les itemsets n'ont pas de successeur. Par exemple, si $\mathcal{I} = \{A, B, C, D, E\}$ alors $Sucessor(AC) = ACD$. Noter que AE n'a pas de successeur (ce sont en fait les feuilles de l'arbre lexicographique).

IV.3 DFS simple

Nous commençons par un algorithme DFS simple. Malgré sa simplicité, il calcule moins de support qu'un algorithme BFS. Cet algorithme traverse l'arbre lexicographique en profondeur. Pour les notations, nous supposons que les itemsets sont codés par un vecteur binaire $V(n)$ où $V[j] = 1$ signifie que l'item I_j est présent, sinon $V[j] = 0$. L'ensemble des IFM est noté $\mathcal{B}$ (pour Bordure). Les paramètres n et σ sont respectivement le nombre total d'items et le support minimal. L'entier i est un indice de position dans le vecteur V. Cet algorithme est lancé avec i qui est initialisé à 1, $\mathcal{B}$ à l'ensemble vide et toutes les positions de V à 0.

Il y a trois opérations critiques dans cet algorithme : le test de couverture (ligne 3), le test de support (ligne 7) et la suppression des sous-ensembles (ligne 9). Cette dernière

Procedure DFS(integer i)

Input: integer i
Output: L'ensemble des IFM $\mathcal{B}$

```
1  if i ≤ n then
2      V[i] ← 1;
3      if B couvre V then
4          DFS(i+1);
5      else
6          if support(V) ≥ s then
7              Ajouter V to B;
8              Supprimer de B les sous-ensembles de V;
9              DFS(i+1);
10     V[i] ← 0;
11     DFS(i+1);
```

opération peut être réalisée efficacement. En effet,

Fait 1. *Si I est un nouvel itemset qui doit être inséré dans $\mathcal{B}$ par l'algorithme* **DFS** *alors $\mathcal{B}$ peut contenir au plus un sous-ensemble de I. De plus, si $\mathcal{B}.last$ désigne le dernier élément inséré dans $\mathcal{B}$ alors $\mathcal{B}.last$ est le seul sous-ensemble possible de I.*

Ceci montre que l'opération de suppression peut être réalisée en un temps constant.

Cependant, le test de couverture peut nécessiter la comparaison de V avec tous les éléments de $\mathcal{B}$. [GZ05] a proposé une technique permettant de maintenir une partie *locale* de $\mathcal{B}$ où le test est réalisé permettant ainsi de réduire le nombre de tests d'inclusion. De plus, la plupart des algorithmes *à la* DFS utilise des heuristiques pour réduire ce nombre de tests, par exemple au lieu d'utiliser l'ordre alphabétique, les items sont triés par ordre croissant de leurs supports, aussi les parents d'un itemset sont réordonnés eux aussi en fonction de leur support [5] [GZ05, BCG01], ou bien exploitent la connaissance obtenue durant le parcours pour couper certaines branches [ZPWL09].

Pour le calcul des supports, des structures de données telles que les FP-trees et les FP-arrays [GZ05], diffsets [GZ05] et bitmaps/bitvectors [BCG01] ont été proposées. Enfin, *look-ahead* (vérification du support de la feuille la plus à gauche d'un sous-arbre) a été proposée dans [Bay98] et a été reprise dans la plupart des algorithmes. D'autres techniques

5. Noter que ceci fait que ces algorithmes ne sont pas en DFS pur.

d'optimisations ont été proposées dans la littérature. Le lecteur trouvera dans [Zen07] une très bonne description.

Dans la section suivante, nous introduisons un nouvel algorithme qui traverse l'arbre de recherche de sorte que les avantages de DFS et BFS soient préservés i.e. le moins de candidats possibles et la possibilité de traiter en parallèle plusieurs candidats.

IV.4 Mine with Rounds

Comme nous l'avons décrit dans la section précédente, **DFS** traite *séquentiellement* les itemsets en respectant l'ordre lexicographique $\prec$. De temps en temps, il fait des sauts qui correspondent à de l'élagage (les ancêtres d'un non fréquent sont non fréquents). Dans cette section, nous proposons un nouvel algorithme qui imite **DFS** et dont le principe consiste à déclencher le traitement d'un itemset dès lors qu'aucun des itemsets le précédant dans l'ordre lexicographique n'apporte pas une information supplémentaire sur son statut (fréquent ou non fréquent). Plus précisément, supposons que I vient d'être traité (on vient de calculer son support) et que J est un itemset tel que $I \prec J$. On aimerait déclencher le traitement de J si aucun des itemsets K tels que $I \prec K \prec J$ n'est capable de fournir une information supplémentaire sur I.

Par exemple, considérons les itemsets AB et B de la Figure IV.2. Une fois que AB est traité, on peut commencer le traitement de B. En effet, soit AB est fréquent et dans ce cas B est couvert [6] ou bien AB n'est pas fréquent et dans ce cas, aucun des itemsets compris entre AB and B, i.e, $[ABC \dots AE]$, ne va nous permettre de *deviner* que B est fréquent ou que B n'est pas fréquent. Il faudra de toute façon calculer son support.

D'une manière intuitive, nous partitionnons l'ensemble des itemsets en des *rondes* [7] $\mathcal{R}_1, \dots, \mathcal{R}_d$ telles que un itemset est traité durant la $i^{ème}$ itération s'il appartient à $\mathcal{R}_i$.

Nous formalisons maintenant la procédure de partitionnement.

Définition 22 (Depth First Partitions)**.** *Soit* $(\mathcal{R})_{i\geq 1}$ *une partition de* $2^{\mathcal{I}}$*.* $\mathcal{R}$ *est une* Depth First Partition *(DFP) de* $2^{\mathcal{I}}$ *respectivement à* $\prec$ *si et seulement si pour chaque itemset* I*,*

1. $I \in \mathcal{R}_k$ *et* I *a un successeur* $\Leftrightarrow Successor(I) \in \mathcal{R}_\ell$ *où* $\ell > k$ *et*
2. $I \in \mathcal{R}_k \Leftrightarrow \forall$ *parent gauche* I' *de* I*,* $I' \in \mathcal{R}_l$ *où* $l < k$*.*

6. Le traitement de B dans ce cas signifie simplement le fait de ne pas le considérer comme candidat.
7. Le terme vient de l'algorithmique distribuée et correspond à la notion d'itération.

Pour simplifier la notation, nous parlerons de DFP de $\mathcal{I}$ au lieu $2^{\mathcal{I}}$ sachant que c'est ce dernier qui est partitionné. Intuitivement, les deux conditions ci-dessus disent simplement : (1) la relation de succession doit être respectée par les DFP (un itemset doit être traité avant son successeur et (2) les parents gauches de I doivent satisfaire la relation $\prec$ (ils doivent être traités avant I). La DFP évidente de $\mathcal{I}$ est celle qui contient $2^n - 1$ parts et où $I \in \mathcal{R}_j \Leftrightarrow Pos(I) = j$. Dans le but de maximiser le parallélisme, i.e. mettre le plus d'itemsets dans une même ronde, nous sommes intéressés par les plus petites partitions (en termes de nombres de parts). On dit que la DFP $\mathcal{R}$ est plus petite que la DFP $\mathcal{R}'$ si et seulement si $|\mathcal{R}| < |\mathcal{R}'|$ où $|\mathcal{R}|$ désigne le nombre de parts de $\mathcal{R}$.

Exemple 17. *Soit* $\mathcal{I} = \{A, B, C\}$. *On peut facilement vérifier que* $\mathcal{R} = \{\{A\}, \{AB\}, \{B\}, \{ABC\}, \{AC, BC\}, \{C\}\}$ *et* $\mathcal{R}' = \{\{A\}, \{AB\}, \{B\}, \{ABC\}, \{AC\}, \{BC\}, \{C\}\}$ *sont toutes les deux des DFP.* $\mathcal{R}$ *est plus petite que* $\mathcal{R}'$.

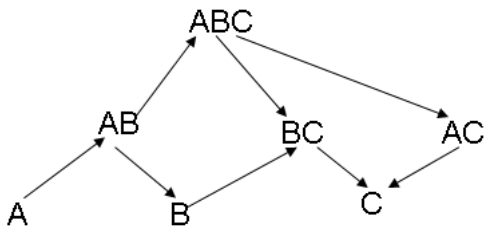
Le théorème suivant montre que la plus petite DFP est unique.

Théorème 6. *Soit* Π *l'ensemble des DFP qu'on peut définir sur* $(\mathcal{I}, \prec)$. *Soit* $\mathcal{R} \in \Pi$ *telle que* $\mathcal{R}$ *a le petit nombre de parts. Alors il n'existe pas une autre DFP* $\mathcal{R}' \in \Pi$ *telle que* $|\mathcal{R}| = |\mathcal{R}'|$ *et* $\mathcal{R} \neq \mathcal{R}'$.

Démonstration. Nous donnons une résumé de la preuve. Considérons le graphe orienté $G(V, E)$ où $V = 2^{\mathcal{I}}$. $(v_1, v_2) \in E$ si et seulement si une des conditions suivantes est vérifiée :

- v_2 est le successeur de v_1 (condition 1 de la définition des DFP) ou
- v_1 est un parent gauche de v_2 (condition 2 de la définition des DFP)

Par exemple, le graphe qu'on obtient à partir de $\mathcal{I} = \{A, B, C\}$ est



De la définition des DFP, on peut montrer que ce graphe n'admet pas de cycle. Ainsi, E définit un ordre partiel parmi les itemsets. Soit $\sqsubset$ l'ordre partiel induit pas E, i.e. $I \sqsubset J$ si et seulement s'il existe un chemin orienté de I vers J. On peut alors regrouper les itemsets quand ils sont incomparables relativement à $\sqsubset$ i.e. ils sont commutables respectivement aux DFP. Pour l'exemple ci-dessus, AC et BC sont incomparables ; $AC \not\sqsubset BC$ et $BC \not\sqsubset AC$. Ils sont donc dans le même groupe (la même chose pour ABC et B).

Pour obtenir les mêmes groupes que précédemment tout en respectant la numérotation des rondes, il suffit de procéder de la manière suivante : supprimer du graphe tous les sommets qui n'ont pas d'arc entrant et mettre ces derniers dans le même groupe i. répéter ce processus en incrémentant i à chaque fois jusqu'à ce que tous les sommets soient supprimés.

Dans l'exemple, les sommets supprimés sont respectivement A puis AB, puis ABC and B simultanément, puis BC et AC simultanément puis enfin C.

Il reste maintenant à montrer que la partition obtenue est bien une DFP (c'est immédiat de par la définition du graphe) et que c'est la plus petite (peut être prouvé par contradiction). □

Notons par $\widehat{\mathfrak{R}}$ la plus petite DFP de $(\mathfrak{I}, \prec)$. Nous donnons maintenant une manière directe pour l'obtention $\widehat{\mathfrak{R}}$.

Théorème 7. *Soit $\widehat{\mathfrak{R}} = \{\mathfrak{R}_1, \ldots, \mathfrak{R}_m\}$ et $I = I_1 \ldots I_k$ un k-itemset ordonné tel que $rank(I_k) = r$. Alors, $I \in \mathfrak{R}_i \Leftrightarrow 2r - k = i$.*

Dans la suite, $Round(I)$ désigne le numéro de la ronde de I. Comme conséquence du théorème précédent, nous avons les propriétés suivantes :

Corollaire 2. *Soit $\widehat{\mathfrak{R}}$ la plus petite DFP de $\mathfrak{I}$ avec $|\mathfrak{I}| = n$. Alors,*

- *le nombre de rondes $|\widehat{\mathfrak{R}}|$ est égal à $2n-1$.*
- *Soient $J^1, \ldots, J^k$ les parents droits de I et $H^1, \ldots, H^\ell$ ses parents gauches. Nous avons*
  - *$Round(H^i) = Round(I) - 1$ pour $1 \leq i \leq \ell$ et*

– $Round(J^i) = Round(I) + (2i-1)$ *pour* $1 \leq i \leq k$

L'exemple suivant illustre les différentes notions que nous avons introduites précédemment

Exemple 18. *Soit* $\mathcal{I} = \{A, B, C, D, E\}$ *l'ensemble des items. la ronde de chaque itemset est illustrée dans la Figure IV.3.*

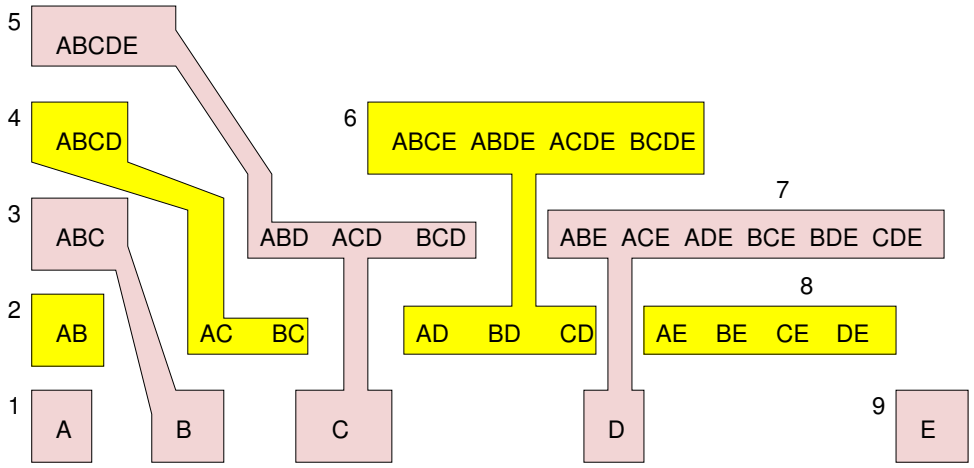


Figure IV.3 – Affectation aux rondes

Nous décrivons maintenant **MineWithRounds** qui utilise le partitionnement défini ci-dessus. Intuitivement, l'algorithme utilise une boucle qui considère à chaque itération les itemsets de la ronde correspondante. Nous utilisons les structures de données suivantes : $\mathcal{C}$ et $\mathcal{B}$ sont des tableaux d'ensembles d'itemsets. $\mathcal{C}[k]$ désigne les candidats devant être traités durant la ronde k et $\mathcal{B}[k]$ est l'ensemble des MFI découverts durant la ronde k. Les fils droits de ABC sont AC et BC (tous les fils sauf celui qui est le préfixe, c'est à dire AB). Le frère droit (right sibling) de ABC est ABD (juste remplacer le dernier item, ici C, par son successeur selon $\lhd$, ici D). Noter que si $Round(I) = i$ et que J le frère droit de I alors $Round(J) = i+2$.

Intuitivement, l'algorithme procède de la manière suivante : Si un candidat $I \in C[k]$ s'avère fréquent, alors son successeur est généré comme candidat pour la prochaine itération $k+1$. Si par contre I n'est pas fréquent, alors (i) ses fils droits sont *potentiellement* pour l'itération suivante $k+1$ et (ii) son frère droit (right sibling) est *potentiellement* candidat pour l'itération $k+2$. les deux derniers types de candidats sont qualifiés de potentiels car ils peuvent s'avérer couverts par des éléments de la bordure. Pour expliquer

Algorithm 4: MineWithRounds.

```
1  C[1] ← {I_1};
2  k ← 1;
3  while k ≤ 2n − 1 et C[k] ≠ ∅ do
4      foreach I ∈ C[k] do
5          // Peut être exécutée en parallèle
6          if support(I) ≥ σ then
7              Ajouter I à B[k];
8              Supprimer le fils gauche I′ de I de B[Round(I′)];
9              Ajouter le successeur I″ de I à C[k+1];
10         else
11             Ajouter les fils droits de I à Children;
12             Ajouter le frère droit de I à C[k+2];
13     foreach I ∈ C[k+1] do
14         // Peut être exécutée en parallèle
15         if I est couvert par B then
16             Supprimer I de C[k+1];
17     foreach I ∈ Children do
18         // Peut être exécutée en parallèle
19         if I n'est par couvert par C[k+1] ∪ B then
20             Ajouter I à C[k+1];
21     i ← k+1;
```

ceci, supposons que AC et BC soient candidats lors de la même ronde. Supposons que AC soit fréquent mais pas BC. Ainsi, AC génère ACD (son successeur) et BC génère C (son fils droit) et BD (son frère droit). Noter que C est couvert par ACD. Ainsi, il n'est pas besoin de considérer C comme un futur candidat.

En ligne 21, nous réalisons le test de couverture relativement à toute la bordure $\mathcal{B}$. Une des optimisations que nous avons adoptées dans notre implémentation consiste à exploiter la remarque suivante : si $I = I_1 \dots I_m$ est un itemset ordonné et $rank(I_m) = r$ alors la première ronde, dans laquelle un ancêtre de I peut apparaître, est la ronde r. Ainsi si la ronde courante est k, le test de couverture de I peut se limiter aux rondes comprises entre r et $k-1$.

Contrairement à ce qu'aurait laissé supposer la première partie où nous avons décrit la manière statique de déterminer la ronde d'un itemset, le lecteur doit avoir noté d'après la description de l'algorithme **MineWithRounds** que les candidats sont générés dynamiquement. En fait, à chaque étape nous avons $C[k] \subseteq \mathcal{R}_k$. $\mathcal{R}_k$ est simplifiée en utilisant les deux élagages : un sur-ensemble d'un non fréquent n'est pas fréquent et un sous-ensemble d'un fréquent n'est pas un IFM.

Dans l'implémentation, les trois boucles **Foreach** des lignes 4, 13 et 17 sont exécutées en parallèle. Nous donnerons plus de détails à ce sujet dans la Section IV.5.

Le théorème suivant compare les algorithmes **MineWithRounds** et **DFS**.

Théorème 8. *Quelle que soit la base de transactions $\mathcal{T}$ et quel que soit le support minimal σ,* **DFS** *calcule le support d'un itemset I si et seulement si* **MineWithRounds** *calcule le support de I.*

Démonstration. Nous donnons une résumé de la preuve. Ce théorème montre donc que les deux algorithmes effectuent les mêmes nombres de calculs de support. Pour le montrer, il suffit de vérifier que **DFS** effectivement calcule le support de I si et seulement si

- le fils gauche de I fréquent et
- aucun des parents gauches de I n'est fréquent.

Ces deux conditions sont, par définition des rondes, celles qui sont satisfaites par **MineWithRounds**. □

Comme **DFS** accède aux données à chaque fois qu'il a besoin de calculer un support et comme **MineWithRounds** traite les candidats par rondes entières, ce dernier accède

donc au plus $2n-1$ fois aux données. Nous pouvons donc conclure que ce dernier est plus performant que **DFS** notamment quand on est en présence de “grandes" bases de transactions.

Une autre remarque concerne la suppression des itemsets de la bordure. Comme nous l'avons vu dans Fait 1, **DFS** exécute cette opération en un temps constant alors que ceci n'est pas cas avec **MineWithRounds** (ligne 8). Nous avons cependant le lemme suivant :

Lemme 7. *Supprimer I' de $\mathcal{B}[Round(I')]$ peut être effectué en $O(\log(|\mathcal{B}[Round(I')]|)$. Comme $|\mathcal{B}[Round(I')]| \leq 2^n$ avec n qui désigne le nombre total d'items, cette opération coûte donc $O(n)$.*

L'exemple suivant illustre l'exécution de **MineWithRounds** lors du calcul de l'ensemble des IFM.

Exemple 19. *Soit $\mathcal{I} = \{A, B, C, D, E\}$ et supposons que la bordure soit $\{ABDE, BC\}$. La Figure IV.4 montre la séquence de génération des candidats. Les arcs montrent la manière dont les candidats sont générés. On commence par A (ronde 1), ce dernier est fréquent. Son successeur AB est donc programmé pour la ronde suivante. ce dernier est fréquent donc à la ronde 3 on traite son successeur ABC. Celui n'est pas fréquent, son frère droit ABD est généré pour être candidat à la ronde 5. Les fils droits AC et BC sont quant à eux programmés pour la ronde 4. Les arcs brisés, représentent les candidats qui sont d'abord générés puis élagués suite aux tests de couverture. Par exemple, à la ronde 4, AC s'avère non fréquent donc son fils droit C est d'abord généré mais celui est couvert par BC lui même fréquent. Il est donc inutile de considérer C à la ronde 5. La même remarque peut être faite pour AD frère droit de AC. Il est d'abord programmé pour la ronde 6, mais à la ronde 5, on trouve que ABD est fréquent ce qui a pour conséquence de supprimer AD. Les itemsets soulignés ne sont pas du tout générés, ils sont là juste pour illustrer la partie de l'arbre élaguée.*

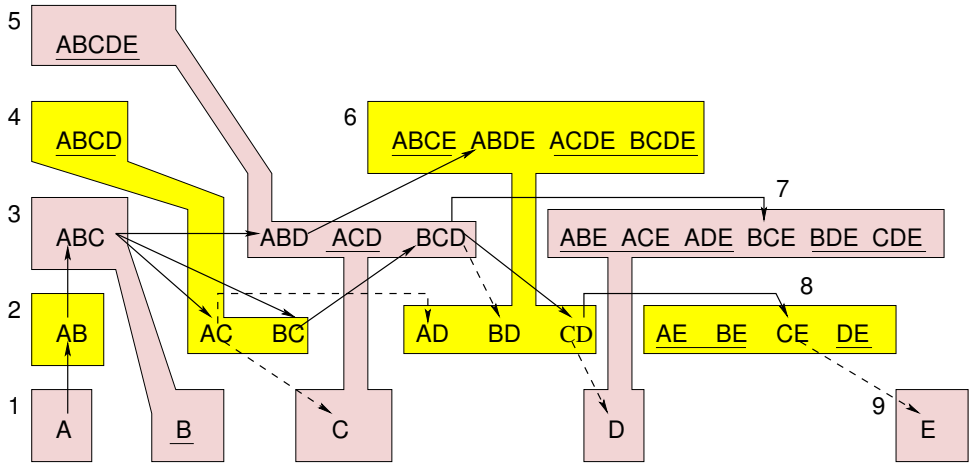


Figure IV.4 – Déroulement d'une exécution de **MineWithRounds**

IV.5 Implémentation parallèle et expériences

Nous avons implanté notre algorithme en C++ avec l'API OpenMP [Ope]. Cette dernière permet d'ajouter des directives aux programmes afin de paralléliser essentiellement les programmes C++ (ou Fortran). Elle est très facile à utiliser et permet d'exploiter pleinement la disponibilité de plusieurs cores. En guise d'illustration, considérons la boucle suivante :

```
for(i=0; i<1000; i++){
   v[i]=i;
   t[i]=i*i;
}
```

Pour l'exécuter en parallèle, il suffit d'utiliser la structure suivante :

```
omp_set_num_threads(8);
#pragma omp parallel {
  #pragma omp for
  for(i=0; i<1000; i++){
     v[i]=i;
     t[i]=i*i;
  }
}
```

Les 1000 valeurs que la variable `i` est censée prendre sont, par défaut, subdivisées en 8 parts égales et chaque thread s'occupera de la mise à jour des parties des vecteurs `v` et `t` qui le concernent. Ainsi, si l'on dispose de 8 cores, le temps séquentiel pour exécuter cette boucle sera en théorie divisé par 8.

Nous avons vérifié la correction de notre programme en comparant ses résultats avec ceux obtenus par des implémentations disponibles comme MAFIA, FPMAX* et PADS [8] et ce pour différents jeux de données et différents seuils de support.

Nos tests ont été réalisés sur une machine équipée de deux processeurs quad-core Intel Xeon sous Redhat Linux enterprise release version 5.4. Grâce à leur capacité de multi-threading, nous avons pu multiplier le nombre de threads jusqu'à 16. La Figure IV.5 montre les caractéristiques de cette machine. Comme notre implémentation n'est pas encore totalement achevée [9] , nous n'allons pas prétendre que notre implémentation est meilleure que les implémentations très optimisées telles que MAFIA, FPMAX* et autres. Les résultats que nous présentons portent sur quatre jeux de données bien connus par la communauté et qui sont : Chess, Mushroom, T10I4D100K and T40I10D100K. Leurs caractéristiques respectives sont

Dataset	# trans.	# Items	Avg. trans. length
Chess	3196	76	37
Mushroom	8124	119	23
T10I4D100K	10^5	10^3	10
T40I10D100K	10^5	10^3	40

Avec chaque jeu de données, nous avons varié le support minimal σ et pour chacune de ces valeurs, nous avons varié le nombre de threads pour les boucles de l'algorithme (le même nombre pour les trois). Le nombre de threads varie de 1 à 16 par puissances de 2. Nos premiers résultats montrent que multiplier le nombre de threads devient intéressant quand le temps d'exécution de l'algorithme séquentiel est assez important de sorte que le surplus de temps pour la gestion des threads soit minime par rapport au temps total. Il y a deux facteurs qui influencent le temps d'exécution : la taille des données qui rend le calcul des supports plus long et la densité des données qui rend la taille de la bordure plus grande donc plus de calculs de supports.

8. A ce propos, nous avons découvert un bug dans PADS sous Linux et pour certains jeux de données. Nous en avons fait part à un des auteurs qui est en train de le fixer.

9. Disponible cependant à `sourceforge.net/projects/minewithrounds`

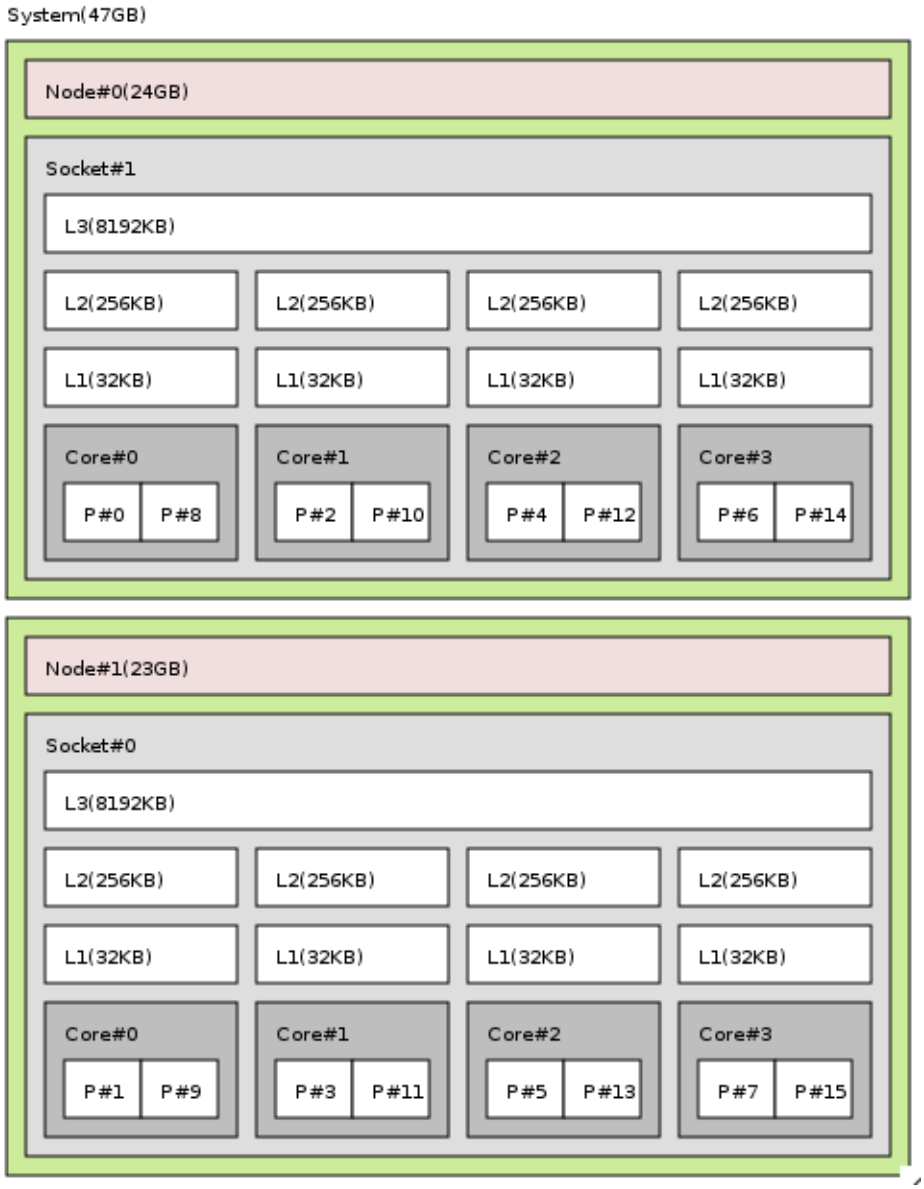


Figure IV.5 – L'architecture de machine

la Figure IV.6 montre les résultats obtenus. Nous montrons les courbes des accélérations (speed up) : le temps d'exécution du programme avec un thread divisé par le temps d'exécution du même programme avec p threads. Nous avons utilisé les FPtrees comme structure des données. Pour le tracé de ces courbes, nous n'avons pas tenu compte du temps nécessaire à la construction des FPtrees du moment que cette partie nous ne l'avons pas parallélisé. En plus des accélérations, la même figure montre le nombre de IFM en fonction de chaque valeur de σ. En général, avec chaque jeu de données, l'accélération est d'autant plus grande que σ est plus petit. Par exemple, avec *Chess* la majorité du temps est consacrée à la gestion des candidats car le jeu de données est dense. Ainsi, quand σ décroît, le nombre de candidats croît rapidement. Avec T10I4D100K et T40I10D100K qui sont moins denses que Chess, nous notons que l'accélération est intéressante même quand les bordures sont de petite taille. Ceci peut s'expliquer par le fait pour ces jeux de données, le temps nécessaire pour le calcul d'un support est plus long que celui pris par Chess à cause de la plus grande taille des données (plus d'items et plus de transactions). Enfin, le jeu de données Mushroom n'est pas dense et a une taille relativement petite. Pour que la parallélisation soit intéressante, nous avons eu à abaisser le support minimal afin d'augmenter la taille des bordures. Analysons plus en détail ce jeu de données lorsque le nombre de threads est égal à 16 et σ est fixé à 100. A première vue, le lecteur peut être surpris de voir une accélération supérieure à 16 ; comment peut on aller plus que 16 fois plus vite lorsque l'on ne dispose que de 16 unités de calcul ? L'explication que nous proposons à ce phénomène est liée justement à l'utilisation des caches. Imaginons que l'algorithme séquentiel accède l fois à un objet mais à chaque fois, ce dernier n'est jamais disponible en cache. Le processus fait donc l accès à la mémoire centrale. Supposons maintenant avec les 16 threads, un seul accès nécessite un chargement de la mémoire centrale et tous les autres (les $l-1$) trouvent l'objet en cache. Dans ce cas, le temps d'exécution est divisé par plus que 16. [LL07] rapporte des accélérations de 24 sur une machine à 8 cores grâce à un placement "intelligent" des données lors de la création des FP-trees.

Un autre facteur qui impacte l'accélération est la distribution des candidats à travers les rondes. Par exemple, la première figure pour *Chess* montre qu'avec 16 threads et $\sigma = 1800$, l'accélération est inférieure à 4. Ceci peut être expliqué, entre autres, par le fait que plusieurs rondes contiennent "très peu" de candidats (moins que 16). Ainsi, le temps de calcul ne pas être divisé par 16 si le nombre de candidats lui même est inférieur à 16. Ceci est exactement ce qui est énoncé dans le Théorème 5. En effet, supposons que le calcul du support de tout itemset prend la même unité de temps. Si s_k est le nombre de candidats de la ronde k, alors le temps d'exécution de cette ronde est estimé par $\lceil \frac{s_k}{p} \rceil \leq \frac{s_k}{p} + 1$ où p

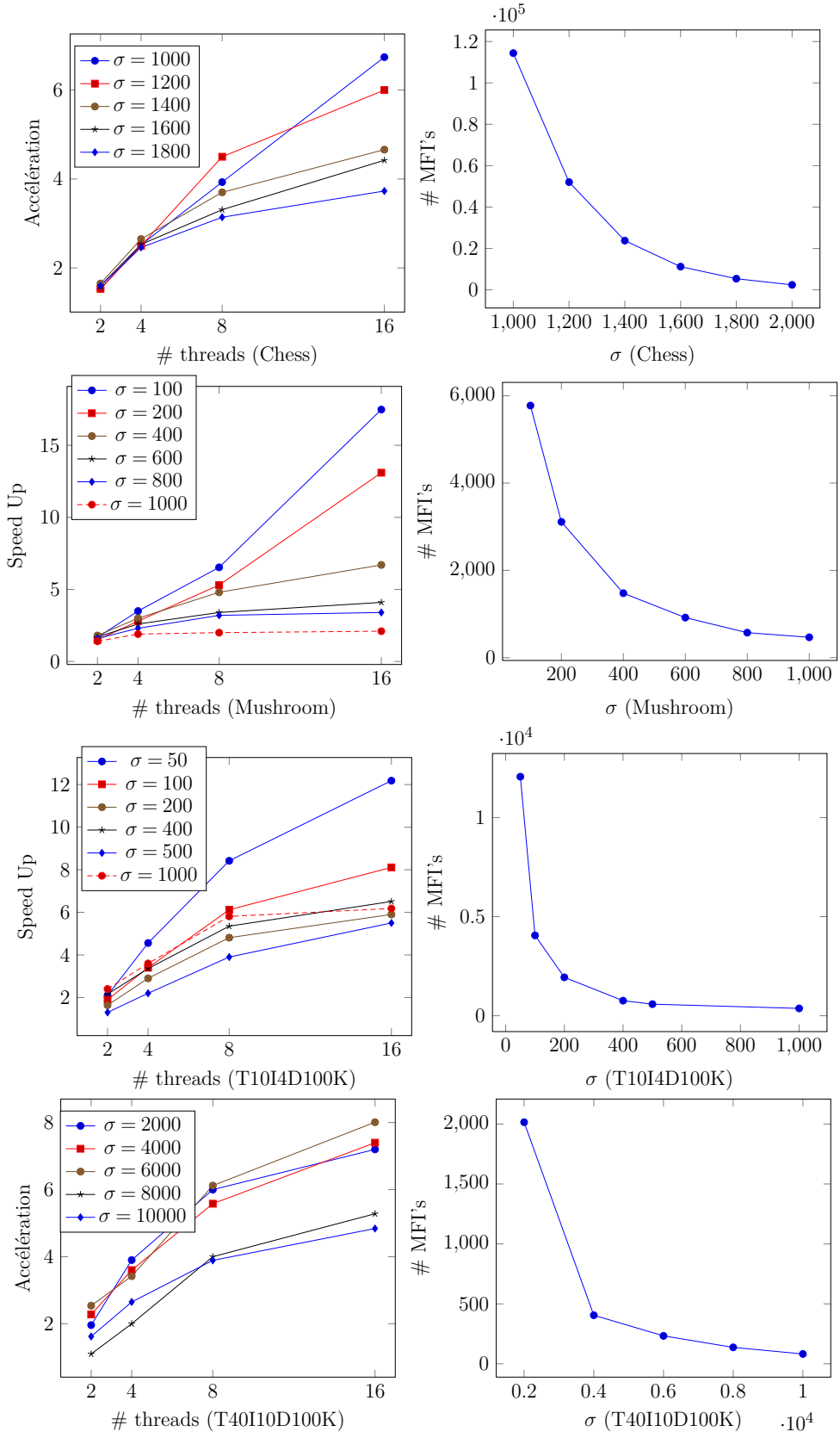


Figure IV.6 – Les accélérations et nombres de MFI's

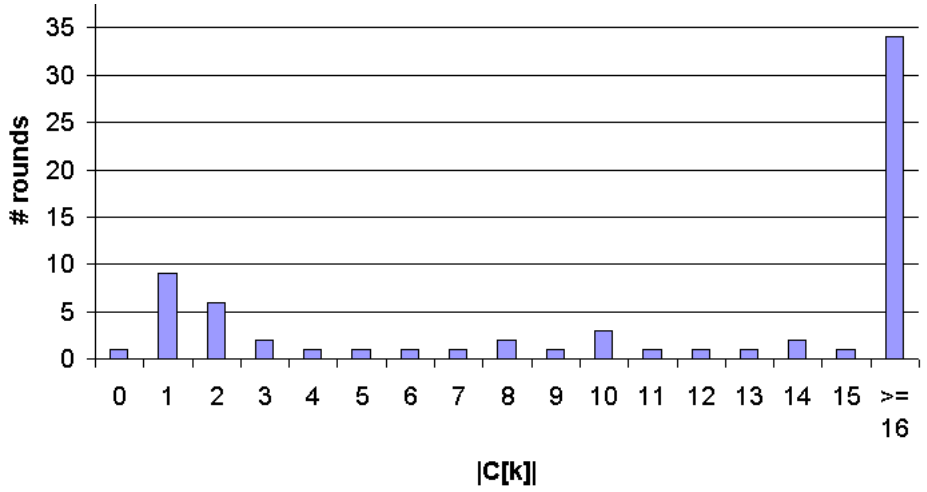
désigne le nombre d'unités de calculs disponibles (cores). La preuve du théorème 5 devient alors immédiate. En pratique néanmoins, le calcul du support d'un itemset est variable. En général, plus l'itemset est long et plus le calcul de son support prend du temps.

La Figure IV.7 montre la distribution du nombre de candidats par rapport aux différentes rondes. Il s'agit toujours du jeu de données Chess. Quand σ est fixé à 1800, le nombre de rondes durant lesquelles on doit traiter plus que 16 candidats est égal à 34 sur un nombre total de rondes qui est 67 [10]. La taille moyenne des rondes est égale à 490. Quand σ est fixé à 1000, le nombre de rondes qui ont plus que 16 candidats est égal à 58 rondes. La taille moyenne des rondes est égale à 9622. Ceci participe à l'explication du fait qu'on gagne plus lorsque $\sigma = 1000$ que quand σ est fixé à 1800.

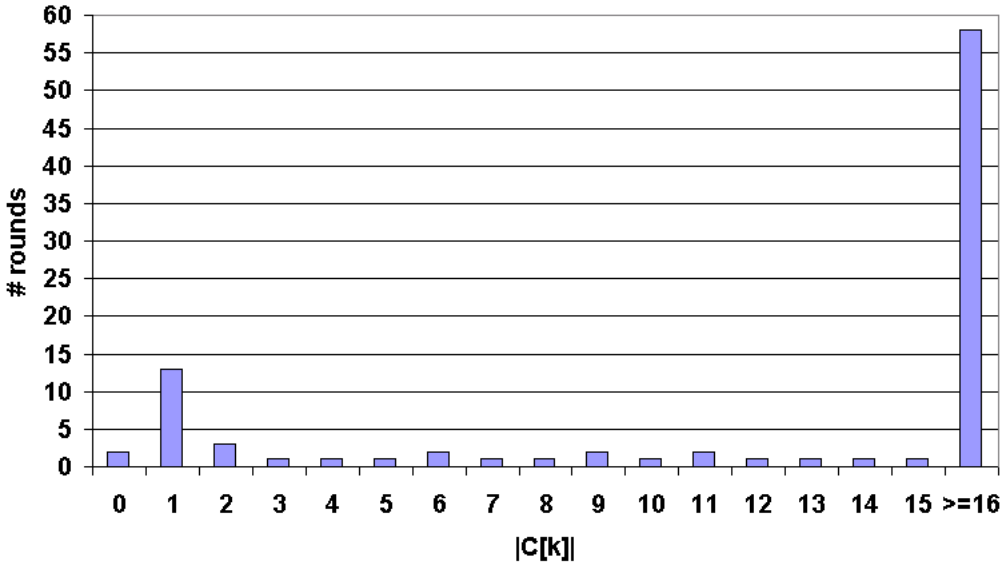
Nous avons aussi mené des expériences pour vérifier le nombre de candidats que notre algorithme génère. Nous avons comparé ce nombre à la borne inférieure prouvée dans [MT97]. Dans cette référence, Mannilla et Toivonnen montrent que tout algorithme calculant l'ensemble des IFM doit calculer le support d'au moins tous les IFM ainsi que tous les éléments de la bordure négative. Plus précisément, si $\mathcal{B}^+$ et $\mathcal{B}^-$ sont respectivement les bordures positive et négative, alors tout algorithme doit calculer au moins $|\mathcal{B}^+ \cup \mathcal{B}^-|$ supports. La Figure IV.8 montre les ratios entre le nombre de supports calculés par **MineWithRounds** sur $|\mathcal{B}^+ \cup \mathcal{B}^-|$. Comme le montre cette figure, le nombre de supports calculés par **DFS** n'est pas très éloigné de la borne inférieure théorique. Il est cependant important de noter qu'à notre connaissance, il n'existe aucun algorithme garantissant un nombre de supports calculés qui soit linéaire ou au moins polynomial par rapport à la taille de $\mathcal{B}^+ \cup \mathcal{B}^-$.

Nous avons aussi comparé **DFS** à PADS. Ce dernier algorithme exploite plusieurs heuristiques et propriétés afin de réduire au maximum le nombre de candidats. A notre connaissance, c'est l'algorithme qui calcule le moins de supports parmi les algorithmes de l'état de l'art. La Figure IV.9 montre quelques uns des résultats obtenus. Comme on pouvait s'y attendre, sans aucune heuristique sophistiquée (mis à part le fait de trier les items selon l'ordre croisant de leur support respectif) **MineWithRounds** teste plus de candidats que PADS (pas beaucoup plus que ça). Ainsi, il est important que dans nos futurs travaux, on prenne en compte ces optimisations. Ceci aura sans doute des répercussions sur la composition des rondes. Par exemple, re-trier à chaque itération les items implique de revoir dynamiquement la fonction *Round*.

10. Parmi les 75 items au total, seuls 34 sont fréquents. On a donc 2*34 - 1 rondes au lieu des 2*75 -1 théoriques.



(a) $\sigma = 1800$



(b) $\sigma = 1000$

Figure IV.7 – Distribution du nombre de candidats (Chess)

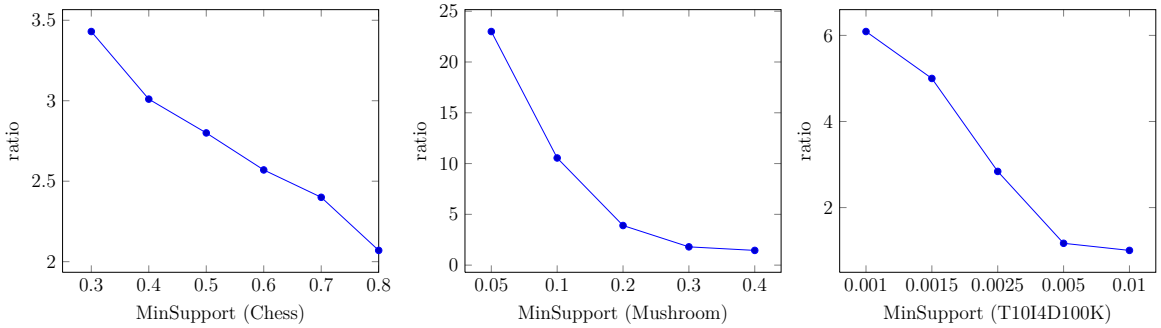


Figure IV.8 – (Nombre de supports calculés)/$|\mathcal{B}^+ \cup \mathcal{B}^-|$

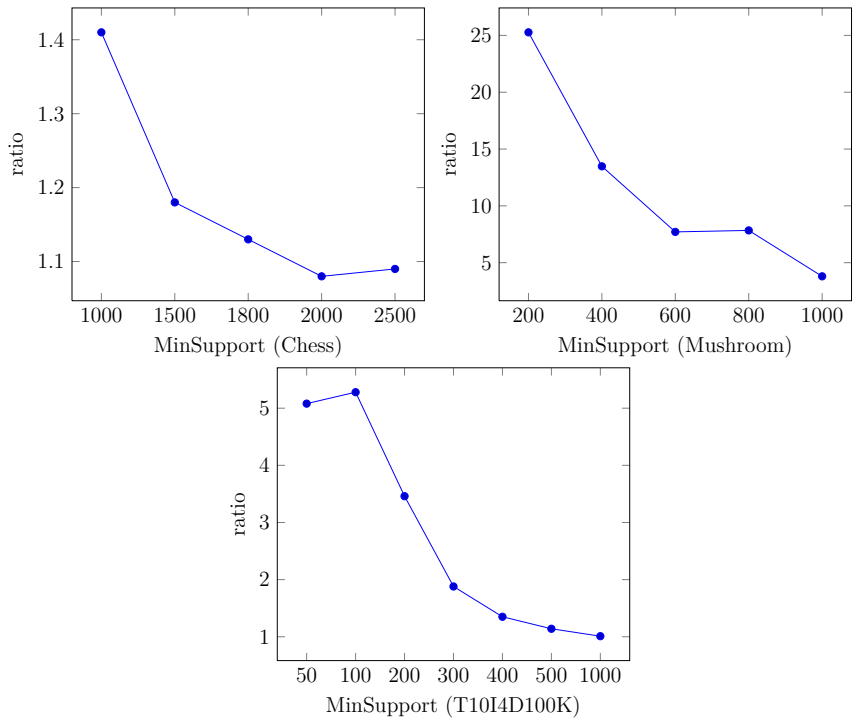


Figure IV.9 – (Ratios des nombres de calculs de supports entre **MineWithRounds** et PADS.

Enfin, nous voulons faire remarquer que même si tout au long de cette section, nous avons insisté sur la parallélisation du calcul des supports, le lecteur doit noter que la gestion des candidats est elle aussi parallélisée. Du moins, partiellement. En effet, les tests de couverture effectuées dans les boucles **Foreach** des lignes 13 et 17 du pseudo-code de **MineWithRounds** nécessitent une synchronisation car toutes les deux mettent à jour une même structure de données (accès concurrents). Nous avons d'abord testé l'utilisation de verrous mais cette solution s'avéra trop pénalisante. Nous avons donc divisé chacune des deux boucles en 2 parties : une première partie parallèle où seules les opérations de lecture sont effectuées (les tests de couverture) et une deuxième partie séquentielle où les candidats couverts sont supprimés.

IV.6 Conclusion

Plusieurs directions ont été poursuivies jusqu'à présent afin d'optimiser le calcul des IFM. Les premières solutions ont essentiellement porté sur la proposition de structures de données compactes permettant de résumer les données et, de ce fait, accélérer le calcul des supports. Aussi, certaines heuristiques se sont avérées très efficaces en pratique. Avec l'avènement des nouvelles architectures de machines avec plusieurs processeurs (ou cores) et leur généralisation, ainsi que leur utilisation des différents niveaux de caches, quelques propositions commencent à émerger [GBP+05]. [LL07] a proposé un algorithme parallèle pour la construction des FPtrees.

Avec notre algorithme, nous réalisons un "bond en avant" dans l'exploitation des architectures multi-cores. En effet, nous avons proposé un nouvel algorithme pour l'extraction des IFM qui est indépendant de la structure de données sous-jacente. Il est basé sur un partitionnement des candidats qui permet de mimer le parcours en profondeur.

Même si nous avons présenté notre travail dans le cadre de la recherche des IFM, nous avons de bonnes raisons de penser que la technique peut être exploitée pour d'autres types de motifs. Par exemple, [YH02] utilise l'ordre lexicographique pour extraire les sous-graphes fréquents. Avec la fonction *Round*, on pourrait subdiviser les sous-graphes de sorte que ces candidats puissent être testés en parallèle. C'est une des directions de recherche que nous comptons poursuivre dans nos travaux futurs.

Aussi, notre implémentation actuelle est loin d'être optimisée. Par exemple, nous avons noté que notre construction des FPtrees est trop lente par rapport aux implémentations concurrentes. Ceci est essentiellement dû au fait que l'allocation dynamique se fait par l'ap-

pel de la fonction `new` à chaque fois que l'on insère un nouveau sommet dans l'arbre. Ces appels multiples pénalisent fortement le temps d'exécution. Les autres implémentations allouent des blocs entiers de mémoire à chaque appel évitant ainsi les appels répétitifs. Ceci a aussi comme avantage le fait que les sommets soient contigus en mémoire, une propriété qui peut être exploitée par l'utilisation intelligente des caches.

Comme l'a montré [GBP+05], les défauts de cache (le fait de devoir chercher la donnée en RAM) est un des défauts majeurs des implémentations actuelles. Ils ont réalisé que moins de 8% des données auxquelles on accède se trouvaient dans le cache. Ainsi, la manière dont les candidats sont attribués aux threads peut avoir un impact important sur la rapidité de l'exécution.

Finalement, nous aimerions explorer le cas des données distribuées. Cette situation permet de traiter le cas de très grandes tailles de données mais elle nécessite des temps de communication entre les différents nœuds du réseau.

Chapitre V

Perspectives

Notre travail ouvre de nouvelles pistes de recherche que ce soit dans le contexte des cubes de données ou bien pour les motifs fréquents.

D'abord, nous avons proposé des solutions au problème de sélection de vues dans le cadre des cubes de données en tenant compte des contraintes de performance d'évaluation des requêtes (facteur fixé par l'utilisateur) et d'espace mémoire (minimiser cet espace). La minimisation de l'espace mémoire requis pour stocker un cube partiel est une heuristique raisonnable pour minimiser en même temps le coût (en terme de temps) de matérialisation et de maintenance. Néanmoins, nous n'avons aucune garantie à ce sujet. Il serait intéressant d'explorer une autre forme du problème de sélection des cuboïdes à stocker en prenant comme contrainte non pas l'espace mémoire de la solution qu'il faut minimiser mais plutôt retenir celle qui a le plus petit coût de maintenance. Cela passe bien sûr par la définition de ce qu'est le coût de matérialisation d'un ensemble de cuboïdes $\mathbb{S}$ voire le coût *minimal* de matérialisation car il est clair que selon l'ordre de matérialisation, le coût est différent. Par exemple, si on a A et AB à matérialiser, commencer d'abord par AB permet d'exploiter ce dernier pour le calcul de A. On pense que l'arbre de Steiner tel qu'il a été utilisé dans [LK06] peut être exploité dans ce cadre. En ce qui concerne la dynamique de notre système d'optimisation des requêtes, nous proposons de modifier le système décrit dans la figure I.5 par la figure V.1, de façon à ce que l'algorithme de sélection de vues prenne en compte le problème de matérialisation.

Une autre piste que nous comptons explorer est *le partitionnement horizontal* des cuboïdes ou d'une manière équivalente les index *épars*[1]. Par exemple, supposons que le coût d'une requête q avec sélection sur l'attribut A évaluée sur un cube c prenne un temps supérieur à f fois le coût minimal. Il se peut que, sans créer d'index *dense* sur c, on puisse évaluer la même requête sur une *partie* de c. Par exemple, subdiviser c en n parties permet d'évaluer la même requête sur une partie de c ce qui a pour conséquence une réduction par

1. Sparse indexes.

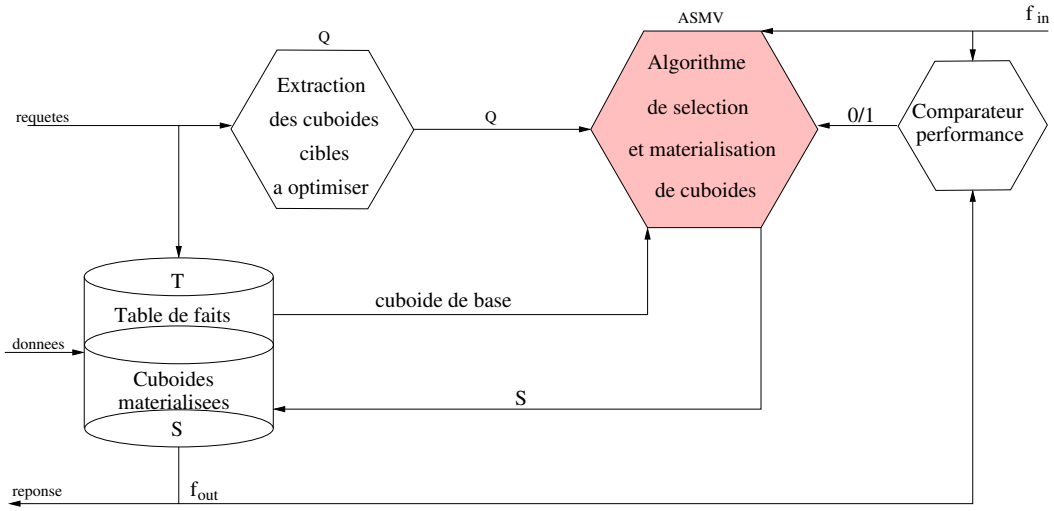


Figure V.1 – Système dynamique d'optimisation de requêtes par matérialisation.

n le coût de l'évaluation. Les SGBD actuels, bien qu'ils proposent pratiquement tous des possibilités de partitionnement, n'offrent pas d'aide sur la façon dont les tables doivent être subdivisées.

Il reste du travail à faire sur l'estimation de la taille des cuboïdes ou des supports des itemset. L'accélération du temps de calcul et le traitement de données massives passent par des estimations rapides (assez fiables) des mesures de support d'itemsets ou de taille de cuboïde. L'estimation de la taille des vues est traitée dans différents articles où l'objectif à atteindre est de limiter la mémoire nécessaire. Dans la littérature, e.g. [Car75, AL07, DF03], l'hypothèse d'indépendance entre attributs et d'uniformité sur la distribution des valeurs est utilisée et donne rapidement des estimations à l'aide de calcul analytiques simples (la formule de Cardenas est souvent avancée comme une "bon estimateur" ce qui est loin d'être le cas quand la distribution n'est pas uniforme). Malheureusement, cette hypothèse n'est pas toujours vérifiée, en fait elle l'est rarement. Par exemple, il est souvent remarqué que la distribution des valeurs d'attributs suit des lois de Zipf. Dans le cas du calcul de supports, on peut espérer réellement gagner du temps en échantillonnant la table de transactions. Certains travaux ont abordé ce sujet, e.g. [Toi96, SDNR96, HNSS95] mais à notre connaissance, sans garantie sur l'approximation.

Par rapport au traitement de grands jeux de données, l'algorithme parallèle MineWithRounds que nous avons proposé, peut être facilement implémenté dans un contexte distribué. Dans ce nouveau contexte, les temps de communication deviennent alors non négligeables, voire prépondérants. Coder efficacement les messages échangés et le bon choix de répartition des données sont des facteurs importants que nous comptons explorer dans

nos futurs travaux.

Notations

Nous donnons ici les notations utilisées dans la thèse avec une description sommaire sans entrer dans les détails. Pour certains d'entre eux, lors de leur première utilisation, nous donnerons des explications supplémentaires.

Dans le cas général, si une lettre majuscule désigne un ensemble d'éléments, alors la lettre minuscule correspondant, suivi d'indice, désigne l'élément d'ensemble. Ex : $D = \{d_1, d_2\}$, $S = \{s_1, s_2\}$.

- T : table de faits
- $G(V, E, w)$: graphe G avec l'ensemble des sommets V, l'ensemble des arcs E et la fonction de poids de sommets w
- $Att(c)$: ensemble d'attributs du cuboïde c
- d : nombre des dimensions dans l'ensemble D
- $\mathcal{C}$: ensemble de tous les cuboïdes d'un cube de données
- $\mathcal{Q}$: ensemble de cuboïdes cibles
- $\mathcal{S}$: ensemble de cuboïdes à matérialiser
- $\mathcal{S}^*$: ensemble optimal de cuboïdes à matérialiser
- $\mathcal{B}$, $\mathcal{B}^+$,$\mathcal{B}^-$: ensemble de bordure, bordure positive, bordure négative
- $\preceq$: relation d'ordre partiel
- f : facteur de performance
- $cost(...)$: le coût d'une requête sur un cuboïde ou d'un ensemble de cuboïdes
- $size(...)$: la taille d'un cuboïde ou d'un ensemble de cuboïdes exprimée en nombre d'entrées
- $MinCost$: minimum d'un fonction de coût
- $MaxCost$: maximum d'un fonction de coût
- c_b : cuboïde de base
- $\mathcal{T}$: table de transactions
- $\mathbb{I}$: ensemble de tous les items

- $\mathcal{I}$: itemset ou ensemble des items
- σ : support d'un itemset
- $DC(T)$ le cube de données construit à partir de la table de faits T

Références bibliographiques

[AAP00] Ramesh C. Agarwal, Charu C. Aggarwal, and V. V. V. Prasad. Depth first generation of long patterns. In *KDD*, pages 108–118, 2000. 75

[Ack89] R.L. Ackoff. From data to wisdom. *Journal of Applied Systems Analysis*, 16 :3–9, 1989. 1

[ACN00] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB '00 : Proceedings of the 26th International Conference on Very Large Data Bases*, pages 496–505, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc. 73

[AIS93] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 207–216. ACM Press, 1993. 11

[AJD06] Kamel Aouiche, Pierre-Emmanuel Jouve, and Jérôme Darmont. Clustering-based materialized view selection in data warehouses. In *ADBIS*, pages 81–95, 2006. 9, 34

[AL07] Kamel Aouiche and Daniel Lemire. A comparison of five probabilistic view-size estimation techniques in olap. In *DOLAP*, pages 17–24, 2007. 24, 36, 100

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*. Morgan Kaufmann, 1994. 24, 71

[Bay98] Roberto J. Bayardo, Jr. Efficiently mining long patterns from databases. In *SIGMOD '98 : Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 85–93, 1998. 26, 70, 71, 75, 80

[BB03] Xavier Baril and Zohra Bellahsene. Selection of materialized views : a cost-based approach. In *BDA*, 2003. 9, 34

[BCG01] Douglas Burdick, Manuel Calimlim, and Johannes Gehrke. Mafia : A maximal frequent itemset algorithm for transactional databases. In *ICDE*, pages 443–452, 2001. 26, 74, 75, 80

[Car75] Alfonso F. Cardenas. Analysis and performance of inverted data base structures. *Commun. ACM*, 18(5) :253–263, 1975. 100

[CCS93] E.F. Codd, S.B. Codd, and C.T. Salley. Providing olap (on-line analytical processing) to user-analysts : an it mandate. Technical report, Arbor Software (now Hyperion Solutions), 1993. 3

[CD97] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and olap technology. *SIGMOD Record*, 26(1) :65–74, 1997. 3

[Chv79] V. Chvàtal. A greedy heuristic for the set covering problem. *Mathematics of operation research*, 4(3) :233–235, 1979. 47

[DF03] Marianne Durand and Philippe Flajolet. Loglog counting of large cardinalities (extended abstract). In *ESA*, pages 605–617, 2003. 69, 100

[DL05] Guozhu Dong and Jinyan Li. Mining border descriptions of emerging patterns from dataset pairs. *Knowl. Inf. Syst.*, 8(2) :178–202, 2005. 73

[EGM03] Thomas Eiter, Georg Gottlob, and Kazuhisa Makino. New results on monotone dualization and generating hypergraph transversals. *SIAM J. Comput.*, 32(2) :514–537, 2003. 76

[EHZ06] Mohammad El-Hajj and Osmar R. Zaïane. Parallel leap : Large-scale maximal pattern mining in a distributed environment. In *ICPADS (1)*, pages 135–142, 2006. 76

[FK09] Daniela Florescu and Donald Kossmann. Rethinking cost and performance of database systems. *SIGMOD Rec.*, 38(1) :43–48, 2009. 13

[FMP04] Frédéric Flouvat, Fabien De Marchi, and Jean-Marc Petit. Abs : Adaptive borders search of frequent itemsets. In *FIMI*, 2004. 76

[FMP05] Frédéric Flouvat, Fabien De Marchi, and Jean-Marc Petit. A thorough experimental study of datasets for frequent itemsets. In *ICDM*, pages 162–169, 2005. 75

[GBP+05] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. D. Nguyen, Y. Chen, and P. Dubey. Cache-conscious frequent pattern mining on a modern processor. In *Proceedings of VLDB conference*, 2005. 96, 97

[GCB+97] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube : A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Min. Knowl. Discov.*, 1(1) :29–53, 1997. 3, 51

[GHRU97] Himanshu Gupta, Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Index selection for olap. In *ICDE '97 : Proceedings of the Thirteenth International Conference on Data Engineering*, pages 208–219, Washington, DC, USA, 1997. IEEE Computer Society. 34, 59, 60, 61

[GKM+03] Dimitrios Gunopulos, Roni Khardon, Heikki Mannila, Sanjeev Saluja, Hannu Toivonen, and Ram Sewak Sharm. Discovering all most specific sentences. *ACM Trans. Database Syst.*, 28(2) :140–174, 2003. 26, 73, 75

[GM05] Himanshu Gupta and Inderpal Singh Mumick. Selection of views to materialize in a data warehouse. *IEEE Trans. Knowl. Data Eng.*, 17(1) :24–43, 2005. 9, 34

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2) :73–170, 1993. 24

[GZ05] Gösta Grahne and Jianfei Zhu. Fast algorithms for frequent itemset mining using fp-trees. *IEEE Trans. Knowl. Data Eng.*, 17(10) :1347–1362, 2005. 26, 75, 80

[HK06] Jiawei Han and Micheline Kamber. *Data mining : Concepts and techniques*. Morgan Kaufmann, 2006. 1

[HMT09a] Nicolas Hanusse, Sofian Maabout, and Radu Tofan. Algorithme distribué pour l'extraction des fréquents maximaux. In Augustin Chaintreau and Clemence Magnien, editors, *Algotel*, Carry-Le-Rouet France, 2009. H. : Information Systems/H.3 : INFORMATION STORAGE AND RETRIEVAL. 15

[HMT09b] Nicolas Hanusse, Sofian Maabout, and Radu Tofan. Algorithmes pour la sélection de vues à matérialiser avec garantie de performance. In *5èmes journées francophones sur les Entrepôts de Données et l'Analyse en ligne (EDA 2009), Montpellier*, volume B-5 of *RNTI*, pages 107–122, Toulouse, Juin 2009. Cépaduès. 9, 15

[HMT09c] Nicolas Hanusse, Sofian Maabout, and Radu Tofan. Matérialisation Partielle des Cubes de Données. In *Actes des 25èmes journées "Bases de Données Avancées" BDA'09 25èmes journées "Bases de Données Avancées" BDA'09*, pages 1–20, Namur Belgique, 10 2009. 9, 15

[HMT09d] Nicolas Hanusse, Sofian Maabout, and Radu Tofan. A view selection algorithm with performance guarantee. In *EDBT*, pages 946–957, 2009. 9, 15, 73

[HNSS95] Peter J. Haas, Jeffrey F. Naughton, S. Seshadri, and Lynne Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *VLDB*, pages 311–322, 1995. 100

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *SIGMOD Conference*, pages 1–12, 2000. 74

[HPYM04] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. Mining frequent patterns without candidate generation : A frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, 8(1) :53–87, 2004. 74

[HRU96] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In *SIGMOD '96 : Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 205–216, New York, NY, USA, 1996. ACM. 7, 9, 14, 18, 32, 33, 34, 52

[Inm07] William H. Inmon. *Building the Data Warehouse, fourth edition.* Wiley Publishing, 2007. 1

[Kar72] R. Karp. Reducibility among combinatorial problems. In R. E Miller and J. W Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972. 22, 47

[KM99] Howard Karloff and Milena Mihail. On the complexity of the view-selection problem. In *PODS '99 : Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 167–173, New York, NY, USA, 1999. ACM. 33

[KNW10] Daniel M. Kane, Jelani Nelson, and David P. Woodruff. An optimal algorithm for the distinct elements problem. In *Proceedings of PODS conference*, 2010. 24, 69

[KRT+07] Ralph Kimball, Margy Ross, Warren Thornthwaite, Joy Mundy, and Bob Becker. *The Data Warehouse Lifecycle Toolkit, second edition.* Wiley Publishing, 2007. 1, 7

[LK98] Dao-I Lin and Zvi M. Kedem. Pincer search : A new algorithm for discovering the maximum frequent set. In *EDBT*, pages 105–119, 1998. 75

[LK06] Ki Yong Lee and Myoung-Ho Kim. Efficient incremental maintenance of data cubes. In *VLDB*, pages 823–833, 2006. 99

[LL07] Eric Li and Li Liu. Optimization of frequent itemset mining on multiple-core processor. In *VLDB*, pages 1275–1285, 2007. 91, 96

[LTCF05] Jingni Li, Zohreh Asgharzadeh Talebi, Rada Chirkova, and Yahya Fathi. A formal model for the problem of view selection for aggregate queries. In *ADBIS*, pages 125–138, 2005. 9, 34

[LWO01] Weifa Liang, Hui Wang, and Maria E. Orlowska. Materialized view selection under the maintenance time constraint. *Data Knowl. Eng.*, 37(2) :203–216, 2001. 9

[MAD06] Nora Maiz, Kamel Aouiche, and Jérôme Darmont. Sélection automatique d'index et de vues matérialisées dans les entrepôts de données. In *2ème journée francophone sur les Entrepôts de Données et l'Analyse en ligne (EDA 2006), Versailles*, volume B-2 of *RNTI*, pages 89–104, Toulouse, Juin 2006. Cépaduès. 9, 34

[MKIK07] Konstantinos Morfonios, Stratis Konakas, Yannis E. Ioannidis, and Nikolaos Kotsis. Rolap implementations of the data cube. *ACM Comput. Surv.*, 39(4), 2007. 9

[MT97] H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Mininig and Knowledge Discovery*, 1(3) :241–258, 1997. 11, 26, 71, 72, 93

[MTV94] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Efficient algorithms for discovering association rules. In *KDD Workshop*, pages 181–192, 1994. 24

[NCCL09] Sébastien Nedjar, Alain Casali, Rosine Cicchetti, and Lotfi Lakhal. Emerging cubes : Borders, size estimations and lossless reductions. *Inf. Syst.*, 34(6) :536–550, 2009. 73

[Ope] OpenMP. www.openmp.org. 88

[PBTL99] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Discovering frequent closed itemsets for association rules. In *Proceedings of ICDT conference*, pages 398–416, 1999. 71

[Pow07] D. J. Power. Decision support systems glossary, dssresources.com, world wide web, 2007. Last updated November 18, 2009. 3

[Rym92] Ron Rymon. Search through systematic set enumeration. In *KR*, pages 539–550, 1992. 76

[SDN98] Amit Shukla, Prasad Deshpande, and Jeffrey F. Naughton. Materialized view selection for multidimensional datasets. In *VLDB '98 : Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 488–499, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc. 9, 14, 18, 32, 34

[SDNR96] Amit Shukla, Prasad Deshpande, Jeffrey F. Naughton, and Karthikeyan Ramasamy. Storage estimation for multidimensional aggregates in the presence of hierarchies. In *VLDB*, pages 522–531, 1996. 100

[SU03] K. Satoh and T. Uno. Enumerating maximal frequent sets using irredundant dualization. In *Discovery Science conference proceedings*, 2003. 26, 75

[SWC+02] J. P. Shim, Merrill Warkentin, James F. Courtney, Daniel J. Power, Ramesh Sharda, and Christer Carlsson. Past, present, and future of decision support technology. *Decision Support Systems*, 33(2) :111–126, 2002. 1

[TCFS08] Zohreh Asgharzadeh Talebi, Rada Chirkova, Yahya Fathi, and Matthias Stallmann. Exact and inexact methods for selecting views and indexes for olap performance improvement. In *EDBT*, pages 311–322, 2008. 9, 31, 34, 61

[Toi96] Hannu Toivonen. Sampling large databases for association rules. In *VLDB*, pages 134–145, 1996. 100

[Yan04] Guizhen Yang. The complexity of mining maximal frequent itemsets and maximal frequent patterns. In *KDD*, pages 344–353, 2004. 72, 73

[YH02] Xifeng Yan and Jiawei Han. gspan : Graph-based substructure pattern mining. In *ICDM*, pages 721–724, 2002. 96

[ZCL02] Qinghua Zou, Wesley W. Chu, and Baojing Lu. Smartminer : A depth first algorithm guided by tail information for mining maximal frequent itemsets. In *ICDM*, pages 570–, 2002. 75

[ZEH05] Osmar R. Zaïane and Mohammad El-Hajj. Pattern lattice traversal by selective jumps. In *KDD*, pages 729–735, 2005. 76

[Zen07] Xinghuo Zeng. Efficient maximal frequent itemset mining by pattern-aware dynamic scheduling. Master's thesis, Simon Fraser University, 2007. 81

[ZG03] Mohammed Javeed Zaki and Karam Gouda. Fast vertical mining using diffsets. In *Proceedings of SIGKDD conference*, 2003. 74

[ZPWL09] Xinghuo Zeng, Jian Pei, Ke Wang, and Jinyan Li. Pads : a simple yet effective pattern-aware dynamic search method for fast maximal frequent pattern mining. *Knowl. Inf. Syst.*, 20(3) :375–391, 2009. 75, 80