Dépendances fonctionnelles : extraction et exploitation

Eve Garnaud

N^o d'ordre : 4883

UNIVERSITE DE BORDEAUX I

ECOLE DOCTORALE Mathématiques et Informatique

SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

T H È S E

pour obtenir le titre de

Docteur en Sciences

de l'Université de Bordeaux I

Mention : Informatique

Présentée et soutenue par

Eve Garnaud

Dépendances fonctionnelles : extraction et exploitation

Thèse préparée au LaBRI

soutenue le 19 novembre 2013

Jury :

Rapporteurs :	Anne Doucet	-	Université Paris VI
	Jean-Marc Petit	-	Université de Lyon
Co-directeurs :	Sofian Maabout	-	Université Bordeaux IV
	Mohamed Mosbah	-	Institut Polytechnique de Bordeaux
Président :	Olivier Beaumont	-	Université Bordeaux I
Examinateur :	Farid Cerbah	-	Dassault Aviation, Paris

Remerciements

Cette thèse est le fruit de trois ans de travail mais aussi d'échanges avec nombre de personnes qui méritent d'être citées ici.

En premier lieu, je souhaite remercier Sofian Maabout pour tout ce que j'ai appris à ses côtés, son esprit critique, son engagement et son écoute. Un grand merci également à Mohamed Mosbah pour sa disponibilité, ses conseils et son suivi. J'ai eu la chance d'être encadrée par deux directeurs exemplaires tant d'un point de vue scientifique qu'humain.

Je tiens à exprimer toute ma gratitude à Anne Doucet et Jean-Marc Petit d'avoir accepter de relire ma thèse, de m'avoir accordé de leur temps dans la finalisation du mémoire comme pour la soutenance. Je remercie également Olivier Beaumont qui a présidé la soutenance avec sa sérénité et sa bonne humeur. Je tiens à remercier Farid Cerbah pour ses commentaires lors de la soutenance mais aussi tout au long de la thèse lors des réunions du projet SIMID.

Un grand merci également à Loïc Martin pour son regard technique, son soutien et ses encouragements. Je remercie aussi Noël Novelli et Nicolas Hanusse pour nos débats scientifiques animés et fructueux, pour leurs conseils et leur bienveillance.

Je tiens aussi à remercier les personnes que j'ai cotoyé au département informatique de l'IUT de Bordeaux et plus particulièrement Isabelle Dutour, Nicholas Journet, Romain Bourquis, Mickaël Montassier, Patrick Félix, Geneviève Matabos et bien d'autres de m'avoir accueillie aussi chaleureusement et de m'avoir fait confiance pour les enseignements.

L'équipe administrative du LaBRI a également joué un rôle très important. Pour m'avoir guidée et facilitée bien des démarches, un grand merci à Cathy Roubineau, Brigitte Cudeville, Lebna Mizani, Philippe Biais et Véronique Desforges-Bogati. L'équipe technique m'a été également d'une aide très précieuse, je tiens à remercier Benois Capelle, Pierre Héricourt et Béatrice Gazel pour leur disponiblité et leur gentillesse.

La thèse ne serait pas ce qu'elle est sans les amis et toutes les occasions de se réunir. Merci à l'AFoDIB d'avoir créer de bons moments de convivialité et à tous les membres du bureau 123 avec qui j'ai passé 2 années très riches en bonne humeur. Un merci tout particulier à Pierre Halftermeyer pour sa sérénité et ses analyses du monde, à Thomas Morsellino pour son humour et sa générosité, à Lorijn Van Rooijen pour sa simplicité et sa fraicheur, à Allyx Fontaine pour son dévouement et sa bienveillance, à Razanne Issa pour sa douceur et sa légèreté, à Anais Lefeuvre pour sa gentillesse, à Noémie-Fleur Sandillon-Rezer, Benjamin Martin, Samah Bouzidi, Gabriel Renault, Rémi Laplace et tant d'autres...

Je remercie également les permanents qui m'ont guidé, conseillé et fait part de leurs expériences. Je pense notamment à Akka Zemmari, David Janin, Olivier Baudon, David Aubert, Aurélie Bugeau, Pascal Desbarats, Eric Sopéna et Paul Dorbec.

Je tiens finalement à exprimer toute ma gratitude à mes parents qui m'ont toujours encourager, à mes soeurs pour leur soutien et à mes grands-parents qui m'ont suivi de près durant toutes mes études. Un grand merci à mon mari qui a vécu la thèse jour après jour à mes côtes, qui m'a tout donné pour m'aider à réussir.

A tous, merci...

Résumé

Les dépendances fonctionnelles fournissent une information sémantique sur les données d'une table en mettant en lumière les liens de corrélation qui les unient.
Dans cette thèse, nous traitons du problème de l'extraction de ces dépendances en proposant un contexte unifié permettant la découverte de n'importe quel type de dépendances fonctionnelles (dépendances de clé, dépendances fonctionnelles conditionnelles, que la validité soit complète ou approximative). Notre algorithme, **ParaCoDe**, s'exécute en parallèle sur les candidats, réduisant ainsi le temps global de calcul. De ce fait, il est très compétitif vis-à-vis des approches séquentielles connues à ce jour.
Les dépendances satisfaites sur une table nous servent à résoudre le problème de la matérialisation partielle du cube de données. Nous présentons une caractérisation de la solution optimale dans laquelle le coût de chaque requête est borné par un seuil de performance fixé préalablement et dont la taille est minimale. Cette spécification de la solution donne un cadre unique pour décrire et donc comparer formellement les techniques de résumé de cubes de données.

Mots clés : dépendances fonctionnelles, extraction des dépendances, calculs parallèles, cube de données, matérialisation partielle

Abstract

Functional dependancies provide a semantic information over data from a table to exhibit correlation links.
In this thesis, we deal with the dependancy discovery problem by proposing a unified context to extract any type of functional dependencies (key dependencies, conditional functional dependencies, with an exact or an approximate validity). Our algorithm, **ParaCoDe**, runs in parallel on candidates thereby reducing the global time of computations. Hence, it is very competitive comparated to sequential appoaches known today.
Satisfied dependencies on a table are used to solve the problem of partial materialization of data cube. We present a characterization of the optimal solution in which the cost of each query is bounded by a beforehand fixed performance threshold and its size is minimal. This specification of the solution gives a unique framework to describe and formally compare summarization techniques of data cubes.

Keywords : functional dependencies, dependency discovery, parallel computations, data cube, partial materialization

Introduction

Les données stockées par les entreprises tendent à être de plus en plus volumineuses. L'optimisation des requêtes qui s'y rapportent est donc un travail essentiel pour les exploiter de manière la plus efficace. Plusieurs mécanismes s'offrent à nous pour répondre à ce problème :

- les structures non redondantes basées sur la fragmentation horizontale et/ou verticale. Ceci permet de mieux cibler la partie des données à lire sans pour autant surcharger la base de données de meta-données consommatrices de mémoire.
- les traitements parallèles. Ici, il s'agit de décomposer les requêtes en sous-tâches indépendantes pouvant être exécutées simultanément en tirant profit de la disponibilité de plusieurs processeurs.
- l'utilisation d'index. Ces derniers permettent de cibler facilement les parties des données à consulter mais ils sont considérés comme des structures redondantes car ils occupent un espace mémoire non négligeable pour les stocker. De plus, ils doivent être maintenus lors de la mise à jours des données.
- les vues matérialisées. Ce sont des requêtes dont les résultats sont stockés et maintenus. Elles peuvent être utilisées directement si les requêtes les définissant sont soumises à nouveau par l'utilisateur, soit elles sont combinées avec d'autres données afin de réduire les temps d'exécution. De la même façon que les index, les vues matérialisées sont considérées comme des structures redondantes.

Dans cette thèse, nous traitons l'optimisation des requêtes par le biais des vues matérialisées dans un cube de données. Ce dernier peut être vu comme un ensemble de vues (les 2^n vues possibles correspondent aux sous-ensembles de $\mathcal{A} = \{A_1, \ldots, A_n\}$, le schéma de la table T considérée). En pratique, il n'est pas envisageable de calculer toutes ses vues et les stocker car cela prendrait trop de temps et trop d'espace. Le problème qui se pose alors est celui du choix du *meilleur* sous-ensemble de vues.

Nous rencontrons dans la littérature plusieurs formulations de ce problème selon les contraintes considérées et l'objectif à optimiser. Une des formulations qui a attiré le plus de travaux est celle qui consiste à minimiser le coût d'un ensemble de requêtes, considéré comme étant la charge de travail de la base de données, sous une contrainte d'espace disponible pour stocker les vues à matérialiser. Les solutions à ce problème n'offrent aucune garantie quant à l'optimalité vis-à-vis des requêtes prises individuellement. Autrement dit, certaines requêtes seront effectivement optimisées alors que d'autres auront des temps de réponse pouvant être catastrophiques.

Notre travail se positionne d'une manière différente en considérant l'optimalité de l'exécution des requêtes comme étant la contrainte majeure et l'objectif consiste à réduire l'espace mémoire permettant d'y parvenir. En effet, nous partons du principe que le matériel, notamment l'espace de stockage, est une denrée dont le coût est négligeable dès lors que son acquisition nous permet d'avoir des garanties

sur les performances des requêtes. Nous ne voulons pas non plus gaspiller l'argent en acquérant des supports de stockage au delà de ce qui est nécessaire. Cette vision correspond à celle utilisée dans le *cloud computing* où l'utilisateur dispose, en théorie, d'un espace mémoire infini mais il doit payer son utilisation.

Les contributions de notre thèse portent essentiellement sur deux parties.
La première consiste à exploiter les dépendances fonctionnelles satisfaites par les données sous-jacentes afin de caractériser les vues d'un cube à matérialiser. Cette première partie de notre contribution montre l'aspect fondamental que jouent les dépendances dans l'organisation des données. En effet, leur rôle a été identifié par Codd dès que le modèle relationnel a été proposé. Cependant, nous notons que très peu de travaux portent sur l'exploitation des dépendances fonctionnelles dans le cadre des cubes de données. Or, matérialiser partiellement un cube revient, quelque part, à en extraire la partie redondante et quoi de plus puissant que les DFs pour ce type de tâches. En fonction des performances souhaitées par l'utilisateur et les requêtes considérées, nous montrons, via l'utilisation de dépendances fonctionnelles exactes, approximatives et/ou conditionnelles, comment caractériser les solutions au problème de sélection de vues.

D'autre part, certains travaux ont porté sur la construction de *résumés* de cubes. Chacun utilisant des concepts différents rendant la comparaison de ces solutions difficile. Nous montrons que certaines propositions peuvent être exprimées en utilisant les dépendances. Ceci permet non seulement de mieux les cerner mais aussi de les comparer.

Notre deuxième contribution porte sur l'extraction de dépendances. En effet, la caractérisation des solutions pour les cubes de données pré-suppose la disponibilité des dépendances présentes dans les données. Or en pratique, ces dépendances doivent d'abord être extraites. Les méthodes actuelles pour extraire les dépendances satisfaites par une table ne sont pas adaptées pour traiter d'importants jeux de données : la plupart commence par un pré-traitement sur les tuples (soit une complexité en $O(m^2)$ où m est le nombre d'enregistrements de la table T considérée). D'autres, parcourant l'espace de recherche en largeur, sont limitées par l'occupation mémoire des candidats et des calculs lorsque le nombre d'attributs, n, est élevé. Nous proposons une approche permettant de traiter efficacement les jeux de données volumineux.

Le principe générale de notre approche consiste en un partitionnement "astucieux" de l'espace de recherche combinant parallélisme et parcours en profondeur qui est une heuristique largement utilisée dans la littérature. En effet, elle donne plus d'opportunités d'élaguer l'espace de recherche qui est exponentiel.

Nous notons enfin que même si cette contribution est au départ motivée par la matérialisation partielle de cubes de données, nous trouvons dans la littérature plusieurs exploitations possibles des dépendances comme par exemple le nettoyage des données, l'optimisation sémantique de requêtes, sélection d'index et réorganisation physique des données. Ainsi, notre solution peut être considérée comme une première étape avant d'effectuer ces différentes tâches.

Organisation de la thèse : Cette thèse s'articule autour de sept chapitres :

Nous commençons, chapitre 1, par introduire les principales notions permettant de définir les dépendances fonctionnelles. Nous décrivons les différents types de dépendance pour situer les dépendances fonctionnelles et leurs utilisations. Nous détaillons ensuite les catégories de dépendances que nous allons chercher à extraire par la suite.

Le problème de l'extraction des dépendances n'est pas nouveau. Nous présentons les approches les plus performantes et les plus connues à ce jour dans le chapitre 2. Nous mettons ainsi en lumière les meilleures stratégies à adopter.

Dans le chapitre 3, nous proposons un algorithme parallèle permettant d'extraire n'importe quel type de dépendances fonctionnelles. Nous menons quelques expérimentations montrant l'efficacité de notre approche.

Nous présentons la technique d'optimisation des requêtes basée sur les cubes de données ainsi que le problème de sa matérialisation dans le chapitre 4.

La matérialisation partielle des cubes de données a été très étudiée depuis les années 1990. Nous décrivons les principales solutions proposées dans ce domaine au chapitre 5.

Nos contributions, basées sur l'utilisation des dépendances fonctionnelles, sont présentées au chapitre 6. Nous proposons une caractérisation des solutions optimales permettant de comparer formellement les méthodes existantes.

Cette thèse se conclue, au chapitre 7, par les perspectives que nous envisageons d'étudier. Nous distinguons celles permettant d'étendre et de perfectionner nos travaux de celles qui gravitent autour du domaine de recherche.

Table des matières

Chapitre 1

Informations sémantiques sur une base de données

Sommaire

Dans ce premier chapitre, nous définissons précisément le terme de dépendance ainsi que son intérêt dans nos travaux. En effet, il existe de nombreuses manières de l'appréhender en fonction des objets sur lesquels elles portent. Nous nous concentrons sur les dépendances fonctionnelles et définissons les divisions de celles-ci que nous étudions.
Une fois les principales notions détaillées, nous étudions leur utilisation en pratique pour normaliser les bases de données, vérifier la validité des informations qui s'y trouvent et, de manière plus générale, optimiser les requêtes.
Nous concluons ce chapitre par quelques généralités nous permettant de mieux comprendre le problème de l'extraction de ces dépendances. Nous mesurons ainsi les difficultés liés à celui-ci tout en identifiant les raccourcis que nous pourrons prendre pour le résoudre.

L'extraction des dépendances est un problème très étudié depuis les années 1980. Cependant, beaucoup reste à faire dans ce domaine du fait des avancées technologiques et de nouvelles contraintes liées aux données notamment.
Dans ce chapitre, nous posons le cadre théorique des travaux.

Afin de donner un aperçu pratique des méthodes et algorithmes développés dans cette partie, nous allons travailler sur un exemple concret très simple permetant d'illustrer nos propos.
Prenons une table *Clientèle* (table 1.1) définie sur les attributs *Client*, *Métier*,

Produit et *Transporteur*. Pour réduire les notations, nous les désignons seulement par la première lettre.

C	M	P	T
c_1	m_1	p_1	t_1
c_2	m_1	p_1	t_2
c_3	m_2	p_1	t_2
c_4	m_2	p_2	t_3

TABLE 1.1 – *Clientèle*.

Cette table contient quatre clients issus de deux domaines d'activité différents et achetant les produits p_1 ou p_2 qui ont été livrés par trois transporteurs différents.

1.1 Les différents types de dépendances

Le terme *dépendance* est très générique puisqu'il exprime un lien de corrélation entre deux objets. Dans cette section, nous définissons différents types de dépendances afin de préciser le domaine de nos travaux.

Les dépendances, une classe particulière de contraintes d'intégrité décrivant des liens entre différents objets de la base, ont été particulièrement étudiées pour répondre aux problèmes de cohérence de la base de données. [Codd 1971] les a utilisées pour proposer une normalisation de ces bases. Elles ont ensuite été très largement étudiées et développées mais nous verrons cela dans un second temps. Commençons par quelques définitions permettant de mieux situer le contexte.

1.1.1 Les niveaux de dépendances

Plusieurs familles de dépendances sont identifiées en fonction des objets sur lesquels elles portent :
Les dépendances d'inclusion : Elles définissent des relations entre deux tables de la base de données afin de garantir la cohérence des données qui s'y trouvent. Ajoutons à notre exemple une table *Catalogue* donnant, pour chaque produit, ses références et son prix. Un client ne peut pas acheter un produit n'existant pas dans le catalogue. Cela se note ainsi : $Clientèle[Produit] \subseteq Catalogue[Produit]$ pour exprimer le fait que tous les produits présents dans la table *Clientèle* doivent nécessairement être les produits de la table *Catalogue*. Plus formellement, la dépendance d'inclusion est satisfaite (ou vérifiée) par la base si et seulement si $\forall t_i \in Clientèle, \exists t_j \in Catalogue$ tel que $t_i[Produit] = t_j[Produit]$ où $t_i[X]$ est la valeur du tuple t_i sur l'ensemble d'attributs X. Cela peut se transcrire par $\pi_{Produit}(Clientèle) \subseteq \pi_{Produit}(Catalogue)$ où $\pi_X(T)$ donne la projection de la table T sur l'ensemble d'attributs X.
Pour cet exemple, nous n'avons considéré qu'un seul attribut mais ce concept peut

être étendu à des ensembles d'attributs à condition qu'ils soient de même taille et qu'une attention particulière soit portée sur l'ordre des attributs dans ces ensembles afin de les faire correspondre un à un exactement.

Les dépendances de jointure : Elles définissent des indépendances entre attributs afin de garantir la cohérence du schéma de la base de données relativement aux différentes tables qu'elle contient. Supposons que le client c_2 appartenant au corps de métier m_1 achète aussi le produit p_2. Il n'existe donc aucun lien entre le métier et le produit acquis puisque pour chaque métier, le client a besoin indifféremment des produits p_1 et p_2. Il n'y a donc aucun intérêt à stocker ces informations au sein de la même table et il est préférable de les séparer pour assurer une meilleure lisibilité. Le métier et le produit ne sont donc liés que au client. Cette dépendance se note $\bowtie [Client.Métier, Client.Produit]$ ou $*\{Client.Métier, Client.Produit\}$ pour exprimer le fait que toute l'information initiale se retrouve bien dans ces deux tables. Ici, étant donné que la base originale a été scindée uniquement en deux parties, nous parlons plutôt de dépendance multivaluée, notée $Client \twoheadrightarrow Métier$ ou, de manière équivalente $Client \twoheadrightarrow Produit$.

Les dépendances fonctionnelles : Elles définissent des liens entre attributs d'une même table. Nous remarquons sur notre exemple, qu'à chaque client n'est associé qu'un seul corps de métier. Dans ce cas, la valeur du client détermine la valeur du métier. Cela se note $Client \rightarrow Métier$ pour indiquer le fait que si deux tuples portent sur le même client, $t_1[Client] = t_2[Client]$, alors ils ont nécessairement la même valeur pour l'attribut $Métier$. De manière équivalente, si la taille de la projection de la table $Clientèle$ sur l'attribut $Client$ est égale à la taille de sa projection sur les attributs $Client$ et $Métier$, $|\pi_{Client}(Clientèle)| = |\pi_{Client,Métier}(Clientèle)|$, alors la dépendance est satisfaite (proposition 1.1).

Définition 1.1 (Dépendance fonctionnelle) *Soient X et Y deux ensembles d'attributs. X détermine Y ou Y dépend de X, noté $X \rightarrow Y$ si et seulement si $\forall t_1, t_2 \in T$, $t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]$.*
La partie gauche de la DF, X, est sa source et la partie droite, Y, sa cible.

Proposition 1.1 ([Huhtala *et al.* 1999]) *La DF $X \rightarrow Y$ est satisfaite par la table T, noté $T \models X \rightarrow Y$ si et seulement si $|\pi_X(T)| = |\pi_{X \cup Y}(T)|$ où $|\pi_X(T)|$ (resp. $|\pi_{X \cup Y}(T)|$) est la taille de la projection de X (resp. $X \cup Y$) sur T.*

Dans cette thèse, nous nous sommes intéressés plus particulièrement à l'étude des dépendances fonctionnelles. La section suivante définit les différentes catégories de dépendances fonctionnelles que nous avons à notre disposition.

1.1.2 Focus sur les dépendances fonctionnelles

Les dépendances fonctionnelles expriment les liens existant entre plusieurs attributs d'une même table. Cependant, il y a plusieurs manières d'appréhender ces liens.

Dépendance de clé : Intéressons nous à un attribut ou à un ensemble d'attributs vis-à-vis de tous les autres. Il s'agit alors de dépendances de clé :

Définition 1.2 (Dépendance de clé) *X est une clé de T si et seulement si $\forall A_i \in \mathcal{A}, T \models X \rightarrow A_i$.*

La validité d'une dépendance de clé se formalise ainsi :

Proposition 1.2 (Extension de la proposition 1.1) *X est une clé de T si et seulement si $|\pi_X(T)| = |\pi_{X \bigcup_{A_i \in \mathcal{A}} A_i}(T)| = |T|$.*

Dans notre exemple, *Client* est une clé de la table puisque sa valeur détermine toutes les autres et nous avons bien $|\pi_{Client}(Clientèle)| = |Clientèle|$.

Dépendance fonctionnelle conditionnelle : Il peut être intéressant de ne considérer une dépendance fonctionnelle que sur des valeurs données des attributs de la partie gauche et/ou droite. C'est le cas des dépendances fonctionnelles conditionnelles.

Définition 1.3 (Dépendance fonctionnelle conditionnelle) *Soient X et Y deux ensembles d'attributs. Une dépendance fonctionnelle conditionnelle (DFC) entre X et Y est de la forme $(X \rightarrow Y, t_p)$ où t_p est un motif défini sur $X \cup Y$ et prenant ses valeurs dans $(Dom(X) \times Dom(Y)) \cup \{\_\}$ où $\_$ est une valeur anonyme c.-à-d. non fixée à la différence des constantes dans $Dom(X) \times Dom(Y)$. Il permet d'identifier les tuples de la table sur lesquels la DF $X \rightarrow Y$ s'applique.*
Un tuple $t \in T$ satisfait le motif t_p, noté $t \asymp t_p$, si $\forall A \in X \cup Y$, $t[A] = t_p[A]$ ou $t_p[A] = \_$.
La DFC $(X \rightarrow Y, t_p)$ est vérifiée par T, $T \models (X \rightarrow Y, t_p)$, si et seulement si $\forall t_1, t_2 \in T$, avec $t_1 \asymp t_p$ et $t_2 \asymp t_p$, $t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]$.

Sur notre exemple, en ne s'intéressant qu'au corps de métier achetant le produit p_2, nous constatons qu'il n'existe qu'un seul métier associé et donc le produit détermine le métier. Nous avons $Clientèle \models (Produit \rightarrow Métier, p_2|\_)$ (le caractère $'|'$ sert à séparer le motif des attributs de la source de la dépendance de celui de sa cible). La validité d'une telle dépendance se vérifie ainsi :

Proposition 1.3 (Extention de la proposition 1.1) *La DFC $(X \rightarrow Y, t_p)$ est satisfaite par T si et seulement si $|\pi_X(\sigma_{t[X] \asymp t_p[X]}(T))| = |\pi_{X \cup Y}(\sigma_{t[X \cup Y] \asymp t_p[X \cup Y]}(T))|$ où σ est l'opérateur de sélection.*

Il faut distinguer deux sortes de dépendances conditionnelles : les constantes, lorsque tous les attributs (partie droite comprise) ont une valeur constante fixée et les variables pour lesquelles des valeurs anonymes peuvent être rencontrées.

Dépendance fonctionnelle approximative : Nous avons vu que, pour être satisfaite, une dépendance fonctionnelle, quelque soit son type, doit vérifier une égalité entre deux tailles. Cette contrainte peut être relachée en évaluant que la

dépendance est presque vraie ou, plus précisément, vraie à $\alpha\%$. Nous parlons dans ce cas de dépendance approximative. Il existe plusieurs définitions de l'approximation d'une dépendance, nous ne citons que les plus populaires :

- La mesure g_1 donne la proportion de paires de tuples qui violent la dépendance : $g_1(X \rightarrow Y) = |\{(u,v) \mid u,v \in T, u[X] = v[X] \wedge u[Y] \neq v[Y]\}|/|T|^2$,
- La mesure g_2 donne la proportion de tuples appartenant aux paires décrites par la mesure g_1 : $g_2(X \rightarrow Y) = |\{u \mid u \in T, \exists v \in T, u[X] = v[X] \wedge u[Y] \neq v[Y]\}|/|T|$,
- La mesure g_3 donne la proportion minimale de tuples à supprimer pour que la dépendance soit satisfaite : $g_3(X \rightarrow Y) = (|T| - max\{|s| \mid s \subseteq T, s \models X \rightarrow Y\})/|T|$,
- La Force donne le ratio entre les tailles des projections : $force(X \rightarrow Y) = |X|/|XY|$,
- La Confiance donne la proportion maximale de tuples qui satisfont la dépendance : $confiance(X \rightarrow Y) = 1 - g_3(X \rightarrow Y)$.

Les trois premières mesures, notamment g_3, développées par [Kivinen & Mannila 1995] sont les plus utilisées dans la littérature. Nous ne nous sommes intéressés ici qu'au cas des dépendances fonctionnelles "classiques" mais ces mesures peuvent être adaptées au cas des dépendances conditionnelles en ne considérant que la partie de la table qui correspond au motif donné par celle-ci. Si la dépendance $X \rightarrow Y$ est exactement vérifiée par la table, alors $1 \leq i \leq 3, g_i(X \rightarrow Y) = 0$ et $force(X \rightarrow Y) = confiance(X \rightarrow Y) = 1$. Étant donnés une mesure M et un seuil d'approximation α relatif à M (supposons que la dépendance est exacte lorsque $M(X \rightarrow Y) = 1$), la dépendance approximative est satisfaite par la table T, $T \models X \rightarrow_\alpha Y$, si et seulement si $M(X \rightarrow Y) \geq \alpha$.

[Giannella & Robertson 2004] donnent un état de l'art et une étude détaillée de ces différentes mesures. Nous présentons de manière plus approfondie les mesures que nous avons utilisées pour nos travaux dans le prochain chapitre.

Dans le contexte des dépendances conditionnelles, afin de ne retourner que les dépendances les plus pertinentes, il est commun de ne chercher que les dépendances k-fréquentes c.-à-d. si k' tuples sont concernés par la dépendance et $k' \geq k$, alors la dépendance est k-fréquente. Dans tous les travaux proposant d'extraire les dépendances conditionnelles, seules les k-fréquentes sont recherchées puisqu'elles sont considérées comme étant les plus pertinentes pour l'analyse de la table.

Il existe des liens entre ces différents types de dépendances. En effet, la dépendance fonctionnelle $X \rightarrow Y$ peut être vue comme la dépendance conditionnelle $(X \rightarrow Y, t_p)$ pour laquelle toutes les valeurs de t_p sont anonymes. La cible de cette dépendance peut contenir tous les attributs de la table et, dans ce cas, nous traitons une dépendance de clé. Nous avons vu également que les mesures d'approximation peuvent exprimer l'exactitude de la dépendance qui est donc un cas particulier de dépendance approximative. Plus concrètement, leurs pouvoirs d'expression sont décrits par la figure 1.1.

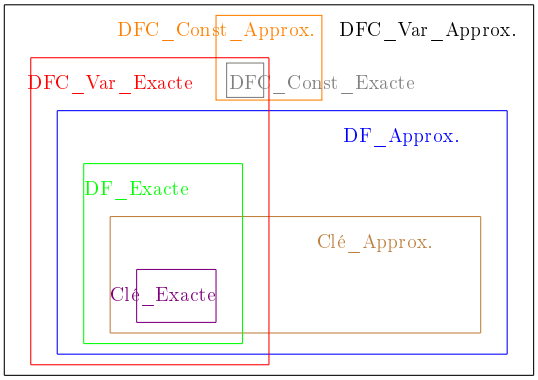


FIGURE 1.1 – Relation d'inclusion entre différents types de dépendances.

Maintenant que nous avons une idée plus précise de ce que sont les dépendances, voyons comment les utiliser en pratique.

1.2 Utilisations

1.2.1 Pour la normalisation

L'étude de ces dépendances a été initialement motivée par un soucis de normalisation de la base de données ([Codd 1971]).

Cette normalisation sert essentiellement à décomposer les tables de la base de données de façon à éviter au maximum les redondances et donc les possibles erreurs tout en assurant qu'aucune information n'est perdue ([Codd 1983]). Les dépendances fonctionnelles, dépendances de clé et dependances de jointure étudiées pour la normalisation sont utilisées pour partitionner les attributs, nous procèdons alors à une décomposition (aussi appelée fragmentation) verticale.

Nous pouvons aller plus loin encore dans ce découpage en s'intéressant maintenant aux tuples qui composent ces tables. Cela se fait grâce aux dépendances conditionnelles ([De Bra & Paredaens 1983]). La table est fragmentée de manière horizontale en séparant par exemple, les clients exerçant le métier m_1 des autres. Ainsi la dépendance fonctionnelle $Métier \rightarrow Produit$ est vraie si nous ne considérons que la table restreinte au métier m_1. De plus, puisqu'une dépendance est satisfaite sur cette nouvelle table, nous pouvons à nouveau procéder à un découpage vertical de celle-ci. Cette décomposition est sans perte d'information puisqu'il suffit de faire l'union des deux tables résultantes pour obtenir la table originale. Si un ensemble de requêtes fréquentes est connu, cela permet également d'isoler le résultat de ces requêtes afin d'optimiser leurs calculs.

1.2.2 Pour la correction des données

Plus récemment, un nouveau besoin est né, celui de certifier la véracité des données. En effet, malgré une normalisation optimale de la base, nous ne sommes pas à

l'abri d'erreurs de saisie ou autre mise à jour douteuse. Il faut donc être en mesure, a minima, de détecter ces erreurs et de les corriger, dans l'idéal, automatiquement. De nombreux travaux ont été menés pour résoudre ce problème à l'aide des dépendances conditionnelles ([Bohannon *et al.* 2007]). L'idée consiste à extraire toutes les dépendances satisfaites sur une table de référence certifiée correcte. Ensuite, il faut vérifier leur validité sur les tables potentiellement impropres pour les corriger.

Prenons l'exemple d'une table gérant des abonnements aux transports en communs. Dans celle-ci, l'âge d'une personne détermine son type d'abonnement. Si un abonné de 5 ans ne dispose pas d'un abonnement junior alors la dépendance $(Age \rightarrow TypeA,\ \_|\_)$ n'est pas vérifiée et nous savons qu'il y a une anomalie sur ce tuple. Une première solution pour rendre à la table sa cohérence est de supprimer ce tuple mais il est facile de voir que ce n'est pas satisfaisant d'ignorer un client sous prétexte qu'un champs est mal renseigné. [Chiang & Miller 2008] proposent alors de nettoyer la table en recherchant les dépendances conditionnelles qui sont presque entièrement vérifiées pour retourner, à un expert, les tuples se présentant comme exceptions pour celles-ci. Ainsi, seuls les enregistrements potentiellement erronés sont soumis à vérification. Nous remarquons alors que, pour espérer corriger ce genre d'anomalies, il faut disposer de dépendances conditionnelles relativement précises. [Diallo *et al.* 2012b] utilisent les dépendances conditionnelles constantes extraites de la table de référence pour rectifier automatiquement les valeurs erronées sur les tables réelles. Le problème de correction des données étant NP-Difficile, les auteurs proposent des heuristiques pour le résoudre.

[Fan *et al.* 2010] proposent d'étendre ces concepts au cas où les données sont distribuées. En effet, dans ce contexte, juger si une DFC est violée ou non n'est pas trivial. Ils s'intéressent donc précisément à la complexité de ce problème en évaluant les coûts de transfert de données d'un site à un autre et les temps de réponse nécessaires à la détection d'une violation. [Fan *et al.* 2012] développent ce problème en proposant une version incrémentale à ce processus de détection.

1.2.3 Pour l'optimisation des requêtes

Pour connaître le type de compte bancaire de M. Martin, par exemple, en supposant que, dans cette table, $Nom \rightarrow TypeCompte$, il suffit de retourner le premier enregistrement correspondant au nom Martin même s'il en existe plusieurs. Nous ne lisons donc qu'une ligne en sachant que l'information qui s'y trouve est complète.

Avant de pouvoir les utiliser, il faut extraire ces dépendances.

1.3 Le problème de l'extraction des dépendances

Dans cette thèse, nous nous intéressons plus particulièrement au cas des dépendances fonctionnelles puisqu'elles offrent déjà un large panel de composantes à étudier. De plus, elles donnent une information sémantique sur la table permettant

leur utilisation dans les problèmes d'optimisation comme nous venons de le voir. Nous parlerons donc de dépendances en omettant les qualificatifs "fonctionnelles" et/ou "exactes" et préciserons le type de dépendance considéré lorsqu'il s'agit de dépendances de clé, de conditionnelles ou d'approximatives.

Dans une base de données, il y a les dépendances connues à la création, celles directement liées aux contraintes d'intégrité, et celles qui sont plus fortuites. Dans une banque parisienne, supposons que tous les clients possedant un compte "gold" habitent dans le 16^{e} arrondissement. Il n'y a pas de règle permettant d'établir un tel lien ni même de critère particulier pour ouvrir ce type de compte mais cette information a tout de même son importance et peut être utile. Il est donc nécessaire de disposer de toutes les dépendances présentes sur une table donnée afin de mieux l'appréhender. Nous allons donc nous pencher sur ce problème d'extraction des dépendances.

1.3.1 Formalisation du problème

Le problème que nous cherchons à résoudre se définit ainsi : étant donnée une table T, trouver l'ensemble $\mathcal{F}$ de toutes les dépendances valides sur T c.-à-d. $\forall \varphi$ telle que $T \models \varphi$, $\varphi \in \mathcal{F}$. Bien entendu, nous pouvons spécifier n'importe quel type particulier de dépendances à extraire.

La première chose à noter est qu'il n'est pas nécessaire d'extraire toutes les dépendances pour toutes les connaître. En effet, certaines dépendances en impliquent d'autres via un système de règles d'inférence. Il s'agit de déduire d'une information initiale, une information complémentaire. Les axiomes d'Armstrong constituent le système d'inférence valide et complet pour l'implication des dépendances fonctionnelles :

Réflexivité si $Y \subseteq X$, alors $X \rightarrow Y$,

Augmentation si $X \rightarrow Y$, alors $\forall Z \subseteq \mathcal{A}, XZ \rightarrow Y$,

Transitivité si $X \rightarrow Y$ et $Y \rightarrow Z$, alors $X \rightarrow Z$.

Pour les dépendances conditionnelles, nous avons besoin de considérer, en plus de l'ensemble d'attributs, les possibles valeurs fixées pour ceux-ci par le biais du motif. [Bohannon *et al.* 2007] proposent un système de 8 règles d'inférence valide et complet pour les dépendances conditionnelles. Les trois premières règles étendent les axiomes d'Armstrong en ajoutant des conditions sur le motif. Dans le contexte des dépendances conditionnelles, ces axiomes ne sont pas suffisants puisqu'il est encore possible de déduire des dépendances à partir d'un ensemble et donc de réduire l'ensemble initial des dépendances. Trois autres règles montrent l'importance du motif dans la définition des dépendances. En effet, les premisses de ces règles sont suffisamment précises pour que la dépendance ne s'applique qu'à très peu de tuples. En étudiant ces tuples, il est donc possible de préciser ou de réduire la dépendance. Une attention particulière doit tout de même être portée sur la cohérence des valeurs entre elles. Naturellement, si un motif ne permet de sélectionner aucun tuple alors les valeurs de celui-ci sont considérées comme étant

incohérentes. Une étude des valeurs cohérentes est donnée par deux dernières règles.

Il n'est donc pas nécessaire d'extraire toutes les dépendances valides sur une table pour connaître entièrement l'ensemble $\mathcal{F}$ recherché. Les dépendances requises sont précisées dans la section suivante.

1.3.2 Quelques définitions

Les systèmes d'inférence nous permettent de réduire nos recherches à des dépendances ayant une partie gauche la plus petite possible et une cible ne contenant qu'un seul attribut. Nous cherchons alors des dépendances minimales définies comme suit :

Définition 1.4 (DF minimale) *Soient X et Y deux sous-ensembles d'attributs de $\mathcal{A}$. La dépendance valide (c.-à-d. satisfaite par T) $T \models X \rightarrow Y$ est minimale si et seulement si Y est un singleton avec $Y \notin X$ et $\nexists X' \subset X$ tel que $T \models X' \rightarrow Y$.*

Un ordre partiel entre les dépendances peut alors être défini formellement :

Définition 1.5 (Ordre entre les DFs) *Soient $\phi_1 = X \rightarrow Y$ et $\phi_2 = X' \rightarrow Y$ deux DFs ayant la même cible (leurs validités n'a pas d'importance ici). ϕ_1 est plus générale que ϕ_2 ou ϕ_2 spécialise ϕ_1, noté $\phi_1 \sqsubset \phi_2$, si et seulement si $X \subset X'$.*
Si les deux dépendances ne partagent pas la même cible ou s'il n'y a pas d'inclusion entre leurs sources, alors elles sont incomparables.

Une dépendance valide est donc minimale s'il n'existe pas de dépendance valide qui soit plus générale relativement à $\sqsubset$.

Il est facile d'étendre ces définitions aux clés minimales en considérant $Y = \mathcal{A} \backslash X$. Pour les dépendances conditionnelles minimales, il faut également prendre en compte le motif. Elles sont définies ainsi :

Définition 1.6 (DFC minimale) *Soient $(X \rightarrow Y,\ t_p)$ une DFC satisfaite par T et $Const(t_p)$ l'ensemble des valeurs constantes de t_p. Cette dépendance est minimale si et seulement si Y est un singleton tel que $Y \notin X$ et ni $(X' \rightarrow Y, t'_p)$ avec $X' \subset X$ et $t'_p = t_p[X']$, ni $(X \rightarrow Y, t''_p)$ avec $Const(t''_p) \subset Const(t_p)$ ne sont satisfaites par la table.*

Définition 1.7 (Ordre entre les DFCs) *Soient $\phi_1 = (X \rightarrow Y,\ t_p)$ et $\phi_2 = (X' \rightarrow Y,\ t'_p)$ deux DFCs ayant la même cible. ϕ_1 généralise ϕ_2, noté $\phi_1 \sqsubset \phi_2$, si et seulement si $(X \subset X' \wedge Const(t_p) \subseteq Const(t'_p)$ ou $(X \subseteq X' \wedge Const(t_p) \subset Const(t'_p))$.*
Elles sont incomparables si elles ne satisfont pas ces conditions.

L'ensemble des dépendances (fonctionnelles, de clé ou conditionnelles) minimales constitue une couverture de l'ensemble $\mathcal{F}$ de toutes les dépendances valides c.-à-d. un ensemble permettant de retrouver $\mathcal{F}$ entièrement. Il s'agit de la couverture canonique de $\mathcal{F}$, notée $\mathcal{F}'$, que nous allons extraire. Pour assurer de la validité de $\mathcal{F}'$ relativement à l'ensemble $\mathcal{F}$ recherché, nous définissons la notion de fermeture :

Définition 1.8 (Fermeture d'un ensemble de DFs) *Soit $\mathcal{F}$ un ensemble de DFs. La fermeture de $\mathcal{F}$, notée $\mathcal{F}^+$, est l'ensemble de toutes les dépendances déduites de $\mathcal{F}$: $\mathcal{F}^+ = \{X \rightarrow Y \mid \mathcal{F} \vdash X \rightarrow Y\}$ où $\vdash$ exprime le fait que la dépendance est impliquée, via le système d'inférence d'Armstrong, par l'ensemble des dépendances.*

Les ensembles de dépendances $\mathcal{F}'$ et $\mathcal{F}$ sont équivalents si et seulement si $\mathcal{F}'^+ = \mathcal{F}^+$. Or, puisque $\mathcal{F}$ représente l'ensemble de toutes les dépendances satisfaites par T, $\mathcal{F}^+ = \mathcal{F}$. Donc $\mathcal{F}'$ est une couverture de $\mathcal{F}$ si et seulement si $\mathcal{F}'^+ = \mathcal{F}$. De la même façon, nous pouvons préciser la fermeture d'un ensemble d'attributs :

Définition 1.9 (Fermeture d'un ensemble d'attributs) *Soient X un ensemble d'attributs et $\mathcal{F}$ l'ensemble des dépendances valides. La fermeture de X relativement à $\mathcal{F}$, notée X^+, est l'ensemble des attributs déterminés par X. Nous avons $Y \in X^+$ si et seulement si $\mathcal{F} \vdash X \rightarrow Y$.*

Dans le prochain chapitre, nous étudions les méthodes existantes pour extraire une couverture de l'ensemble des dépendances valides sur une table. Plus précisément, nous cherchons la couverture canonique de l'ensemble $\mathcal{F}$ puisque toutes les dépendances extraites sont des dépendances minimales. Il est à noter que $\mathcal{F}$ peut admettre plusieurs couvertures minimales mais il n'a qu'une seule couverture canonique.

Chapitre 2

Boîte à outils

Sommaire

Le problème de l'extraction des dépendances est le suivant : étant donnée une table T, trouver un ensemble de dépendances qui permette, via le système d'inférence dédié, de connaître toutes les dépendances valides sur T. Nous cherchons une couverture à partir de laquelle toutes les dépendances peuvent être recalculées.

[Mannila & Räihä 1992] montrent que ce problème, dans le cas des dépendances fonctionnelles, a une complexité exponentielle en nombre d'attributs. En effet, la taille de la couverture est bornée par $2^{n/2}$. De plus, la validation d'une dépendance se fait en $\Omega(m)$. Pour ces raisons, il est difficile d'envisager une recherche sur de grosses bases de données (en terme de nombre d'attributs ou de tuples). Nous devons cependant nous atteler à résoudre ce problème pour mieux répondre aux besoins actuels.

Le problème d'extraction des dépendances a été très étudié depuis les années 1980 du fait de leur utilité dans de nombreux domaines. [Liu *et al.* 2012] proposent un état de l'art des méthodes les plus connues et les plus efficaces. Nous détaillons celles-ci dans ce chapitre. Plus précisément, après avoir étudié les algorithmes retournant les dépendances exactes, nous examinons les mesures qualifiant les dépendances approximatives. Puis, nous analysons quelques techniques permettant de trouver les dépendances de manière approchée en économisant l'espace mémoire nécessaire ou le temps des calculs.

2.1 Les méthodes exactes

Nous l'avons vu, la recherche des dépendances est un problème difficile. Pour faciliter les calculs, il est possible de commencer par récupérer certaines informations de la table. Les dépendances sont alors extraites de ces pré-calculs. Nous com-

mençons par décrire ces méthodes. Nous exposons ensuite les techniques calculant la validité des dépendances directement à partir de la table.

2.1.1 Avec pré-calculs

L'algorithme TANE, proposé par [Huhtala *et al.* 1999], permet de trouver toutes les dépendances minimales (définition 1.4) satisfaites par une table. Pour ce faire, les auteurs partitionnent les tuples relativement à un ensemble d'attributs X en fonction de leurs valeurs : si $t_i[X] = t_j[X]$ alors t_i et t_j appartiennent à la même classe d'équivalence de η_X. Ils prouvent que la dépendance $X \to A_i$ est valide si et seulement si toutes les classes d'équivalence de η_X sont des sous-ensembles de classes de $\eta_{X \cup \{A_i\}}$. Cela revient à voir que si les tuples ayant la même valeur sur X ont également la même valeur sur A_i, alors la dépendance est vérifiée. Pour obtenir ce résultat plus rapidement, ils montrent que si η_X et $\eta_{X \cup \{A_i\}}$ contiennent le même nombre de classes, alors $X \to A_i$ est valide. Cela revient à dire que les projections sur T de X et de $X \cup \{A_i\}$ ont la même taille (proposition 1.1).

L'espace de recherche de ces dépendances est constitué des 2^n sous-ensembles possibles d'attributs. Le candidat X permet de vérifier la dépendance $X \setminus \{A_i\} \to A_i$ pour tout attribut $A_i \in X$ qui n'est pas déterminé par un sous-ensemble de X. TANE procède par niveaux, des candidats à 1 attribut à celui à n attributs. De cette manière, ils garantissent la minimalité des dépendances retournées.

Les auteurs identifient les clés de la table en remarquant que leurs partitions ne contiennent que des singletons. Dans notre exemple, l'attribut *Client* est une clé puisque $\eta_{Client} = \{\{t_1\}, \{t_2\}, \{t_3\}, \{t_4\}\}$.

Reprenant le principe de l'algorithme TANE, [Yao & Hamilton 2008] proposent FD_Mine dans lequel quatre règles permettent de mieux élaguer l'espace de recherche des dépendances :

Règle 1 Si $X \subseteq Y^+$ et $Y \subseteq X^+$, alors X et Y sont équivalents. Dans ce cas, le candidat Y et ses sur-ensembles peuvent être supprimés puisque, dans toutes les dépendances contenant X, nous pouvons le remplacer par Y.

Règle 2 Si $Y^+ \subseteq X$, alors X peut être supprimé car la validité $X \to Z$ est déduite de $X \setminus Y^* \to Z$ où $Y^* = Y^+ \setminus Y$ est la couverture non-triviale de Y, les deux dépendances sont équivalentes.

Règle 3 Seules les dépendances de la forme $XY \to A$ avec $A \in \mathcal{A} \setminus \{X^+ \cup Y^+\}$ nécessitent d'être vérifiées vu que $X^* \cup Y^* \subseteq (X \cup Y)^+$.

Règle 4 Si X est une clé de la table, alors il peut être supprimé.

Les auteurs expliquent que les complexités théoriques de FD_Mine sont les mêmes que pour TANE. En revanche, ils montrent que, en pratique, leur élagage permet de réduire les temps de calcul mais aussi l'occupation mémoire puisque moins de candidats sont traités.

Dep-Miner, l'algorithme proposé par [Lopes *et al.* 2000], utilise les partitions de tuples d'une manière différente : plutôt que de noter, pour chaque ensemble d'attributs, les tuples ayant la même valeur sur ceux-ci, ils repèrent, étant donnés deux

tuples, les attributs pour lesquels ils ont les mêmes valeurs. Ils récupèrent ainsi des ensembles maximaux d'attributs pour lesquels au moins deux tuples ont les mêmes valeurs. Le complémentaire de ces ensembles (le complémentaire de X : $\bar{X} = \mathcal{A} \setminus X$) correspond à l'ensemble des dépendances maximales relativement à $\sqsubset$ (définition 1.5) qui ne sont pas satisfaites. Il faut ensuite en extraire les dépendances minimales valides en cherchant, pour chaque attribut cible, les sous-ensembles d'attributs qui ont une intersection non vide avec tous les sous-ensembles des complémentaires maximaux.

Les auteurs montrent, par l'expérimentation, que Dep-Miner est plus performant que TANE et cela se vérifie d'autant plus lorsque le nombre de tuples ou d'attributs augmente. En effet, les auteurs proposent des optimisations pour chaque étape de calcul et notamment le partitionnement initial des tuples.

Dans ces trois premiers travaux, l'espace de recherche est parcouru en largeur. Cela implique une occupation mémoire importante puisqu'au moins un niveau, potentiellement complet, doit être stocké pour chaque itération des algorithmes. Partis de ce constat, [Wyss *et al.* 2001] proposent l'algorithme FastFDs opérant en profondeur. Ils reprennent l'idée développée pour Dep-Miner mais leur progression dans l'analyse des candidats est plus efficace étant donné qu'ils utilisent une meilleure stratégie d'élagage. En effet, ils peuvent profiter d'un élagage par le bas puisque $X \not\rightarrow A_i \Rightarrow \forall A_j \in X, X \setminus \{A_j\} \not\rightarrow A_i$. De plus, ils ordonnent les attributs de manière à ce que ceux qui couvrent un plus grand ensemble soient considérés en premiers pour ainsi donner une dépendance minimale plus rapidement. Cela leur permet de traiter moins de candidats et donc d'accélérer les temps de calcul de la couverture minimale. Les auteurs constatent effectivement ce gain de temps vis-à-vis de Dep-Miner par quelques expérimentations.

Pour extraire l'ensemble des dépendances conditionnelles, [Chiang & Miller 2008] commencent par partitionner les tuples de la table à la manière de [Huhtala *et al.* 1999]. Ils considèrent également le même treillis constitué des sous-ensembles d'attributs et le parcourent en largeur. Ils construisent ensuite le motif d'un candidat $X \rightarrow A_i$ en fonction des classes d'équivalence communes à η_X et $\eta_{X \cup \{A_i\}}$ et en lui affectant le plus petit nombre de constantes possible relativement aux dépendances valides du niveau précédent.

[Fan *et al.* 2011] proposent trois algorithmes pour extraire les dépendances conditionnelles. CFDMiner ne considère que les conditionnelles constantes. Les auteurs mettent en lumière le lien entre les conditionnelles constantes minimales et les règles d'association. En effet, la règle $(X, t_p) \Rightarrow (A_i, a)$ peut s'exprimer sous la forme $(X \rightarrow A_i, t_p|a)$. CFDMiner se base donc sur des techniques d'extraction de ces règles pour trouver les dépendances conditionnelles constantes minimales. Il commence par construire l'espace de recherche des dépendances à partir des candidats identifiés préalablement comme ayant une taille strictement inférieure à celles des éléments plus spécifiques relativement à $\sqsubset$ (définition 1.7). Il le parcourt ensuite en largeur pour en extraire les règles minimales.
Pour parvenir à des dépendances variables, les auteurs proposent CTANE et

FastCFD, des adaptations des algorithmes TANE et FastFD respectivement, dans lesquels seules des considérations sur le motif ont été ajoutées.

Toutes ces méthodes sont très efficaces tant que le nombre de tuples de la table, m, est relativement petit. En effet, le partitionnement de celle-ci a une complexité en $O(m^2)$ et ce n'est donc pas toujours très judicieux de procéder ainsi même si, par la suite, seule une vision simplifiée de la table est utilisée pour extraire les dépendances.

2.1.2 Sans pré-calculs

Certaines méthodes de calcul des dépendances, pour palier cette complexité fixe, procèdent directement à partir de la table.

L'algorithme FUN, présenté par [Novelli & Cicchetti 2001], ne considère donc aucune connaissance préalable sur la table. L'espace de recherche des dépendances est constitué des 2^n sous-ensembles d'attributs. FUN le parcourt en largeur en commençant par les sous-ensembles de taille 1 et il construit les niveaux supérieurs à l'aide des informations du niveau en cours. Les auteurs définissent la quasi-fermeture d'un ensemble d'attributs qui se calcule grâce aux éléments du niveau précédent : $X^\diamond = X \bigcup_{A_i \in X} (X \setminus \{A_i\})^+$. Puisque $X \subseteq X^\diamond \subseteq X^+$, le calcul de la fermeture et donc de la dépendance est ébauché par les calculs du niveau inférieur. Ils identifient les sources potentielles de dépendances minimales comme étant celles pour lesquelles tous les éléments qui les généralisent ont une taille strictement plus petite. Cela leur permet d'élaguer l'espace de recherche. Ils prouvent que les dépendances minimales sont de la forme $X \rightarrow A_i$ avec $A_i \in X^+ \setminus X^\diamond$ et X tel que $\forall X' \subset X, |X'| < |X|$.
[Diallo *et al.* 2012a] proposent CFUN, une adaptation de FUN permettant d'extraire les dépendances conditionnelles constantes.

Nous constatons par cet algorithme que la connaissance extraite d'un niveau sert à accélérer les calculs du niveau suivant. Cependant, cela implique de stocker deux niveaux simultanément pour une occupation mémoire non négligeable lorsque le nombre d'attributs augmente.

L'ensemble des dépendances minimales satisfaites par une table peut être vu comme une bordure de l'espace de recherche c.-à-d. un ensemble minimal tel que tous ses éléments sont incomparables selon $\sqsubset$ et pour tout candidat n'appartenant pas à la bordure, il existe un élément plus général ou plus spécifique que lui dans la bordure. [Hanusse & Maabout 2011] proposent une approche parallèle pour calculer les bordures. Cependant, l'algorithme MineWithRounds n'est donc pas optimisé spécifiquement pour l'extraction des dépendances. Nous nous basons sur leurs travaux pour proposer un algorithme adapté à la recherche des dépendances.

2.2 Les mesures d'approximation

Les mesures d'approximation servent à identifier les dépendances valides ou presque valides sur une table. En effet, sur une table contenant des milliers de tuples, il peut être intéressant de connaître ainsi des tendances qu'ont certains ensembles d'attributs par rapport à d'autre. De cette façon, deux sortes d'études peuvent être menées sur les données : (i) les dépendances presque satisfaites sont considérées comme reflétant le comportement général de la table, les tuples ne les vérifiant pas sont alors ignorés puisqu'ils ne sont que des exceptions, ou au contraire, (ii) ces tuples méritent d'être analysés précisément puisqu'ils contiennent éventuellement une erreur de saisie ou sont des anomalies qu'il faut identifier.

Dans le premier cas de figure, la mesure de confiance est utilisée puisqu'elle donne la proportion des tuples qui satisfont la dépendance. Nous pouvons également considérer le rapport entre les tailles des projections grâce à la mesure de force. [Giannella & Robertson 2004] donnent une analyse assez complète sur ces mesures d'approximation.

Nous choisissons d'étudier les dépendances presque entièrement satisfaites par la table. Nous utilisons donc dans nos travaux les mesures de confiance et de force.

Monotonie La propriété de monotonie des mesures est très importante pour résoudre le problème de l'extraction des dépendances puisque nous cherchons uniquement des dépendances minimales. Il faut donc être en mesure d'élaguer l'espace de recherche en sachant que les candidats plus généraux ou plus spécifiques respectent ou non le seuil de validité des dépendances. [Lopes *et al.* 2001] prouvent que g_3 est monotone : $X \supseteq Y \Rightarrow g_3(X \rightarrow A_i) \leq g_3(Y \rightarrow A_i)$. Puisque $confiance(X \rightarrow A_i) = 1 - g_3(X \rightarrow A_i)$ nous pouvons en déduire la monotonie de la confiance ainsi : $X \supseteq Y \Rightarrow confiance(X \rightarrow A_i) \geq confiance(Y \rightarrow A_i)$.

Nous ne pouvons pas en dire autant de la mesure de force. En effet, nous savons que $X \subset Y \Rightarrow |X| \leq |Y|$ mais cela ne permet pas de déduire un ordre entre les ratios $|X|/|X \cup \{A_i\}|$ et $|Y|/|Y \cup \{A_i\}|$. Sur notre table *Clientèle*, $X \subset Y \Rightarrow force(X \rightarrow A_i) \leq force(Y \rightarrow A_i)$. Sur la table 2.1, nous observons $\frac{|X|}{|XA|} = \frac{2}{3} > \frac{3}{5} = \frac{|XY|}{|XYA|}$. Cela est lié à la fréquence d'apparition de la valeur x_1 qui est nettement plus importante que les autres. A moins que toutes les valeurs de tous les attributs aient exactement la même fréquence, nous ne pouvons pas prévoir si la mesure de force est monotone ou pas. Cependant, nous avons la propriété suivante ([Page 2009]) : $force(X \rightarrow Y) \leq confiance(X \rightarrow Y) \leq 1$.

2.3 L'estimation des solutions

Dans certains cas d'utilisation, il peut être nécessaire d'obtenir les dépendances très rapidement ou avec que peu d'espace mémoire disponible pour faire les calculs. Alors la qualité de la solution retournée est amoindrie pour répondre à ces critères.

X	Y	A
x_1	y_1	a_1
x_1	y_1	a_2
x_1	y_2	a_1
x_1	y_2	a_2
x_2	y_2	a_1

TABLE 2.1 – Exemple de la non-monotonie de la force.

Il existe plusieurs manières d'estimer les dépendances, nous en donnons ici un rapide panorama.

2.3.1 Méthodes par échantillonnage

Pour savoir si une dépendance est satisfaite ou non par T, il faut lire les tuples de T afin de repérer ceux en accord sur la source mais pas sur la cible. Pour accélérer les temps de calcul, [Kivinen & Mannila 1995] proposent d'utiliser un échantillon S de T construit de manière aléatoire pour valider ou non la dépendance. Une dépendance qui n'est pas satisfaite sur S, n'est pas non plus satisfaite par T. Pour celles qui sont valides sur S, il faut pouvoir borner la probabilité qu'elles soient aussi valides sur T. Pour les mesures g_1, g_2 et g_3 qu'ils présentent, ils donnent les tailles minimales d'échantillon à examiner. Ainsi, une dépendance satisfaite par S a une grande probabilité d'être, au moins, presque vérifiée sur T relativement à la mesure considérée (c.-à-d. étant donnés un paramètre de précision de la dépendance ϵ et un paramètre de fiabilité δ, $g_i(X \to Y) = 0$ sur $S \Rightarrow Proba(g_i(X \to Y) \leq \epsilon$ sur $T) \geq 1 - \delta$).

Nous avons vu que la validation d'une dépendance fonctionnelle $X \to Y$ peut se faire en comparant les tailles des projections $|X|$ et $|XY|$. [Gibbons 2001] présente une méthode pour construire un échantillon de façon à ce que la taille de la projection soit facilement calculable à partir de celui-ci. Il procède en une seule lecture de la table en utilisant une fonction de hachage h pour associer à chaque tuple un identifiant unique. Connaissant la distribution des identifiants donnée par la définition de h, il choisit les tuples à ajouter à l'échantillon en fonction de la valeur $h(t[X])$. Il stocke le nombre de tuples correspondants à ces valeurs et estime le nombre de tuples ignorés. Il peut ainsi en déduire la taille complète de la projection. Il s'agit d'une approximation car $h(t[X]) = h(t'[X]) \not\Rightarrow t[X] = t'[X]$ puisque, en pratique, nous observons des collisions qui ne sont pas différenciées par cette approche.

[Cormode *et al.* 2009] proposent d'estimer la confiance des dépendances conditionnelles en utilisant la méthode d'échantillonage par réservoir ([Vitter 1985]). Ils traitent deux problèmes : soit (i) le motif de la dépendance est donné et il faut en calculer la confiance, soit (ii) étant donnée une dépendance fonctionnelle, ils cherchent un motif ayant une confiance supérieure au seuil fixé. Dans les deux cas, ils donnent

les bornes théoriques des erreurs selon que l'échantillon est construit en une ou deux passes sur les données.

2.3.2 Autres méthodes

L'utilisation d'une fonction de hachage est relativement coûteuse en mémoire puisque, dans l'idéal, à chaque tuple est associé une valeur unique. Nous avons donc une occupation mémoire de l'ordre de $\Theta(|X|)$. La méthode HyperLogLog proposée par [Flajolet *et al.* 2007] nécessite seulement $O(\log|T| + \frac{\log\log|T|}{\epsilon^2})$ bits de mémoire pour faire une $(\epsilon, \epsilon^{-2})$-approximation avec $\epsilon \in]0,1[$. Ils montrent que la taille réelle de la projection sur X, $|X|$, est comprise entre $|X|^* \times (1-\epsilon)$ et $|X|^* \times (1+\epsilon)$ où $|X|^*$ est l'approximation retournée par HLL avec une probabilité supérieure à $1-\epsilon^{-2}$: $Proba(|X| \in [|X|^* \times (1-\epsilon), |X|^* \times (1+\epsilon)]) > 1-\epsilon^{-2}$. HLL fonctionne ainsi :

1. étant donné un entier b (~ 10), un vecteur M de taille 2^b est construit,
2. à chaque tuple $t \in T$ est associé une suite pseudo-aléatoire, $h(t)$, de 0 et de 1,
3. les b premiers bits de $h(t)$ identifient le registre $M[i]$ qui contient la position du bit à 1 le plus à gauche de tous les tuples $t'[X]$ concernés par $M[i]$,
4. une fois que tous les tuples ont été étudiés, une fonction f est appliquée à M pour retourner la taille estimée de X.

Nous avons considéré qu'une dépendance conditionnelle est composée d'une dépendance fonctionnelle $X \to Y$ restreinte à un motif de tuple t_p. Nous pouvons également associer à une dépendance fonctionnelle un tableau T_p de motifs. Un tuple est alors concerné par la dépendance s'il correspond à au moins un motif du tableau et n'entre pas en contradiction avec un autre motif.

[Golab *et al.* 2008] présentent un algorithme approché qui, étant donnée une dépendance fonctionnelle de la forme $X \to Y$, retourne un tableau de motifs qui satisfait la dépendance conditionnelle avec une fréquence globale supérieure au seuil s fixée. Pour ce faire, ils commencent par calculer, à partir de la projection sur X, tous les motifs possibles. Ils déterminent ensuite, pour chaque motif, la fréquence de la dépendance conditionnelle correspondante. Ils insèrent dans le tableau les motifs ayant la plus grande fréquence jusqu'à ce que la dépendance ait une fréquence globale supérieure à s. L'approximation est donc dûe au fait que les motifs du tableau sont ajoutés de manière définitive sur la seule mesure de leur fréquence locale.

2.4 Conclusion

Nous avons étudié les approches permettant d'extraire exactement les dépendances. Pour traiter de gros jeux de données (surtout en terme de nombre de tuples), les pré-calculs en complexité $O(m^2)$ sont préjudiciables et ce d'autant plus que nous ne savons, à priori, rien de la forme des dépendances qui sont satisfaites par la table. Nous préférons donc vérifier la validité des dépendances directement à partir des données de la table. Nous notons également que le parcours de l'espace de recherche en profondeur permet un élagage plus efficace et une occupation

mémoire moindre par rapport au parcours en largeur. Notre approche consiste donc à travailler directement à partir des données de la table en explorant l'arbre de recherche en profondeur.

Il est à noter également que les méthodes connues à ce jour se limitent à extraire soit les dépendances fonctionnelles (en identifiant éventuellement les clés), soit les dépendances conditionnelles en y incluant parfois un dégré d'approximation. Or nous avons vu que les dépendances conditionnelles variables approximatives permettent d'exprimer n'importe quel type de dépendances (figure 1.1). Il nous semble donc judicieux de proposer un contexte unifié permettant d'extraire tous types de dépendances. C'est l'objet de nos travaux présentés au prochain chapitre.

Chapitre 3

Extraction parallèle des dépendances

Sommaire

Dans ce chapitre, nous présentons nos contributions au problème de l'extraction des dépendances. Nous cherchons une couverture de l'ensemble des dépendances satisfaites par une table T. Les dépendances retournées sont des dépendances minimales (définitions 1.4 et 1.6).
Nous avons pour ambition de traiter des jeux de données importants. La méthode la plus appropriée pour ce faire est donc, d'après nos observations du chapitre précédent, (i) d'opérer directement sur les données dispensant ainsi les pré-traitements en $O(m^2)$ et (ii) de parcourir l'espace de recherche en profondeur afin d'éviter les problèmes liés à l'occupation mémoire des candidats et de permettre un meilleur élagage de celui-ci.

Nous l'avons vu au chapitre 1, les dépendances conditionnelles variables approximatives permettent d'exprimer n'importe quel type de dépendances fonctionnelles (figure 1.1). Nous allons donc décrire la manière de les extraire en sachant que toute l'information de la table est représentée par ce biais, seul le formalisme change. Il suffit de préciser, en paramètre, le type de dépendances recherchées pour ne considérer que la partie pertinente de l'espace de recherche. Nous pouvons également fixer le seuil de validité des dépendances à 1 pour n'extraire que celles étant exactement vérifiées par la table.

Pour améliorer les performances de notre algorithme, nous choisissons d'utiliser le parallélisme sur les candidats comme cela a été fait par [Hanusse & Maabout 2011]. La méthode développée par les auteurs nous a donc largement inspiré pour mener à bien nos travaux.

Nous commençons ce chapitre par la définition et les analyses de l'algorithme **ParaCoDe**. Nous citons ensuite les optimisations et extensions que nous avons implémentées. Pour finir, nous mesurons expérimentalement l'efficacité de la méthode proposée.

3.1 ParaCoDe

Dans cette section, nous décrivons l'algorithme **ParaCoDe** qui, étant donnée une table, retourne la couverture canonique des dépendances conditionnelles approximatives qui sont satisfaites. Nous commençons par quelques définitions permettant de préciser l'espace de recherche considéré ainsi que la manière de le parcourir. Nous détaillons ensuite l'algorithme dans sa version la plus simple. Les optimisations sont données dans un second temps.

3.1.1 Préliminaires

Comme dans tous les travaux détaillés au chapitre précédent, les attributs sont ordonnés. Cet ordre est noté $\lhd$. Ainsi, nous ne considérons que des sous-ensembles d'attributs pour lesquels les attributs sont positionnés selon $\lhd$. Nous avons fixé $\mathcal{A} = \{A_1, \dots A_n\}$. L'indice précisé pour chaque attribut donne son rang dans l'ensemble : A_i est le i^e attribut de $\mathcal{A}$ si et seulement si $|\{A_k \in \mathcal{A}, A_k \lhd A_i\}| = i - 1$. Considérons qu'il s'agit de l'ordre alphabétique, nous pouvons ainsi étendre sa définition à des ensembles d'attributs. Par exemple, $A_1A_4A_5$ est un sous-ensemble d'attributs corrects alors que A_7A_2 ne l'est pas puisque $A_2 \lhd A_7$, il doit donc être noté A_2A_7. Nous avons, par extension de l'ordre, $A_1A_4A_5 \lhd A_2A_7$ car $A_1 \lhd A_2$.

3.1.1.1 Construction de l'arbre de recherche

Afin d'assurer la non-trivialité des dépendances retournées, nous fixons l'attribut cible, A_t, et lui affectons un motif anonyme. Les candidats sont de la forme (X, t_p) avec $X \subseteq \Lambda = \mathcal{A} \setminus \{A_t\}$ et t_p le motif portant sur X. La taille d'un candidat, notée $||(X, t_p)||$, correspond au nombre d'attributs de X, $nbAttr(X)$, auquel s'ajoute le nombre de constantes de t_p, $nbConst(t_p)$.

Une fois que nous avons toutes les dépendances minimales pour une cible, il suffit de réitérer jusqu'à ce que $\mathcal{A}$ ait été entièrement exploré. Toutes les notions décrites dans ce qui suit portent sur la même cible. Nous ne la précisons que lorsque cela est nécessaire. Pour simplifier les notations, le rang des attributs est donné relativement à Λ.

Exemple *L'espace de recherche pour la recherche des dépendances minimales de cible Transporteur pour notre table Clientèle est donné par la figure 3.1. Nous*

avons placé les candidats relativement à l'ordre $\sqsubset$ afin d'exhiber leurs liens de généralisation et de spécialisation (définition 1.7). Pour plus de lisibilité, les constantes de l'attribut C sont simplement notées c mais il faut voir que cela cache quatre candidats puisque $|Dom(C)| = 4$. Nous procédons de même pour chaque attribut.

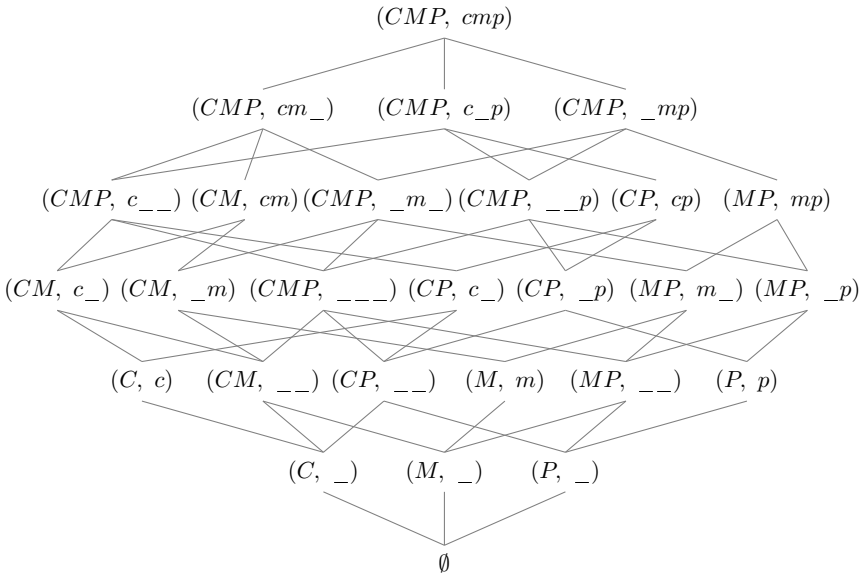


FIGURE 3.1 – Espace de recherche pour la cible *Transporteur*

Pour construire l'arbre de recherche, nous devons ordonner l'ensemble des candidats. Cependant, nous voyons facilement qu'il n'y a aucun intérêt à choisir d'exécuter le candidat (C, c_1) avant (C, c_2) par exemple et le contraire n'a pas plus de sens. De la même façon, nous ne trouvons pas d'avantages à traiter ces candidats avant $(CM, \_\_)$ ou après. Nous commençons alors par identifier les candidats pour lesquels l'ordre d'exécution n'a pas d'importance puisque le fait que l'un d'eux détermine ou non la cible ne donne aucune information sur la validité des autres. Cette classe particulière de candidats incomparables est définie comme suit :

Définition 3.1 (Ensemble de candidats connectés) *Soient (X, t_p) et (Y, s_p) deux candidats. Ils sont connectés, noté $(X, t_p) \sim (Y, s_p)$, si et seulement si*

- *ils sont incomparables relativement à $\sqsubset$ (définition 1.7) avec $||(X, t_p)|| = ||(Y, s_p)||$, et*
- *$X = Y$ ou X est un préfixe de Y (c.-à-d. $A_l \lhd Y \setminus X$ où A_l est le dernier attribut de X) avec $Y \trianglelefteq XZ$ avec $Z = A_{l+1} \dots A_{l+nbConst(t_p)}$ où A_{l+1} est le premier successeur de A_l selon l'ordre $\lhd$.*

Exemple *(C, c_1) et $(CM, \_\_)$ sont connectés puisqu'ils sont incomparables selon $\sqsubset$ ($C \subset CM$ et $Const(C, c_1) = \{c_1\} \supset \{\} = Const(CM, \_\_)$) et $||(C, c_1)|| = 2 = ||(CM, \_\_)||$ avec $CM \trianglelefteq CM$ ($nbConst(c_1) = 1$ et le successeur de C selon $\lhd$ est M). En revanche, (C, c_1) et $(CP, \_\_)$ ne sont pas connectés car la seconde condition n'est pas satisfaite, nous avons $CM \lhd CP$ et le contraire est faux naturellement.*

Les candidats connectés n'ont donc aucun intérêt à être ordonnés les uns par rapport aux autres dans le parcours en profondeur de l'espace de recherche. De plus, l'approche que nous proposons utilise le parallélisme sur les candidats, ils sont donc traités en même temps. En revanche, nous devons définir un ordre entre chaque pair de candidats non connectés :

Définition 3.2 (Ordre DFS) *Soient (X, t_p) et (Y, s_p) deux candidats tels que $(X, t_p) \not\sim (Y, s_p)$. (X, t_p) apparaît avant (Y, s_p) dans l'ordre DFS, noté $(X, t_p) \lessdot (Y, s_p)$, si et seulement si $(X \lhd Y)$ ou $(X = Y$ et $nbConst(t_p) < nbConst(s_p))$.*

Exemple *Pour l'attribut cible Transporteur, l'ordre DFS sur les candidats est le suivant : $(C, \_) \lessdot (C, c_{i\in[1,4]}) \sim (CM, \_\_) \lessdot (CM, c_{i\in[1,4]}\_) \sim (CM, \_\, m_{j\in[1,2]}) \sim (CMP, \_\_\_) \lessdot \cdots \lessdot (P, \_) \lessdot (P, p_1) \sim (P, p_2)$.*

L'arbre DFS de recherche des dépendances se construit au fur et à mesure de l'exploration des candidats. A partir de (X, t_p), nous pouvons générer l'ensemble de ses parents c.-à-d. un ensemble de candidats qui le spécialise immédiatement :

Définition 3.3 (L'ensemble des parents) *(X', t'_p) est un parent de (X, t_p) si et seulement si il respecte une des formes suivantes :*

- *si $nbConst(t_p) < nbAttr(X)$, $X' = X$ et $|Const(t'_p) \setminus Const(t_p)| = 1$;*
- *$X' = XA_{l+i}$[1] avec $t'_p[X] = t_p$ et $t'_p[A_{l+i}] = \_$ où A_{l+i} est un successeur de A_l, le dernier attribut de X selon l'ordre $\lhd$.*

Dans l'ensemble des parents de (X, t_p), $Parents(X, t_p)$, plusieurs peuvent être connectés mais ils suivent l'ordre $\lessdot$. Nous pouvons donc identifier le premier parent (ou le premier ensemble connecté de parents) noté $FParent(X, t_p)$.

Un parent est un candidat tel que $(X, t_p) \sqsubset (X', t'_p) \in Parents(X, t_p)$ puisqu'il spécialise une valeur anonyme de motif ou ajoute un attribut. Les parents d'un candidat apparaissent donc après celui-ci dans l'ordre DFS. Nous pouvons ainsi naviguer dans l'espace de recherche du candidat le plus général au plus spécifique. Définissons maintenant une relation de voisinage horizontale. De cette façon, il n'est pas nécessaire de redescendre dans l'arbre pour atteindre le parent suivant. Il s'agit du frère :

Définition 3.4 (Le frère) *$Sibling(X, t_p) = (X', t'_p)$ est le frère de (X, t_p) si et seulement si*

- *si $nbConst(t_p) = i \neq 0$, $X' = XZ$ où Z est l'ensemble des $i - 1$ successeurs directs de A_l, le dernier attribut de X, relativement à $\lhd$ auquel s'ajoute A_{l+i+1} (c.-à-d. $Z = \{A_{l+1}A_{l+2} \ldots A_{l+i-1}A_{l+i+1}\}$) et $nbConst(t'_p) = 0$;*
- *sinon $X' = X \setminus \{A_l\} \cup \{A_{l+1}\}$ et $nbConst(t'_p) = 0$.*

Exemple *L'arbre DFS de la table Clientèle pour la recherche des dépendances minimales de cible Transporteur est donné par la figure 3.2. Nous avons regroupé les ensembles de candidats connectés de manière explicite.*

1. XA_{l+i} exprime la concaténation de l'attribut A_{l+i} avec l'ensemble X

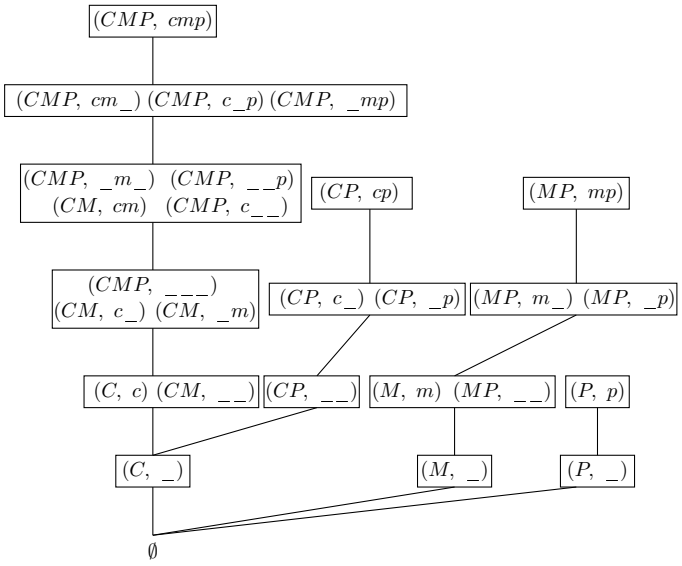


FIGURE 3.2 – Arbre DFS $\mathcal{T}_T$

Intuitivement, l'algorithme monte dans l'arbre jusqu'à trouver une dépendance valide. Ensuite, il faut vérifier sa minimalité et pour cela, nous définissons une relation directe entre un candidat et ses généralisations du niveau inférieur qui n'ont pas encore été traitées puisqu'elles apparaissent après ce candidat dans l'ordre DFS :

Définition 3.5 (L'ensemble des enfants droits) $(X', t'_p) \in RightC(X, t_p)$ *est un enfant droit de* (X, t_p) *si et seulement si il est de la forme* $X' = X \setminus \{A_i\}$ *avec* $A_i \neq A_l \in X$ *où* A_l *est le dernier attribut de* X *avec* $t_p[A_i] = \_$ *et* $t'_p = t_p[X']$.

Exemple $RightC(CMP, \_\_p) = \{(CP, \_p), (MP, \_p)\}$ *avec* $p \in Dom(P)$. (C, c) *pour tout* $c \in Dom(C)$ *n'a pas d'enfant droit.*

Nous savons maintenant comment construire et parcourir l'arbre de recherche. Examinons alors quelques caractéristiques afin de confirmer nos définitions. Elles nous aiderons à proposer des optimisations à l'algorithme.

3.1.1.2 Quelques propriétés

Nous avons regroupé de façon naturelle, sur la figure 3.2, les candidats connectés. Nous devons vérifier formellement la validité des ensembles ainsi formés par le lemme de transitivité suivant :

Lemme 3.1 *Soient* (X, t_p), (Y, s_p) *et* (Z, r_p) *trois candidats.* $(X, t_p) \sim (Y, s_p)$ *et* $(X, t_p) \sim (Z, r_p) \Rightarrow (Y, s_p) \sim (Z, r_p)$.

Preuve *D'après la définition 3.1,* $||(X, t_p)|| = ||(Y, s_p)||$ *et* $||(X, t_p)|| = ||(Z, r_p)||$, *donc* $||(Y, s_p)|| = ||(Z, r_p)||$. *Supposons que* $(Y, s_p) \sqsubset (Z, r_p)$. *Alors, soit* $Y \subset Z$ *et* $Const(s_p) \subseteq Const(r_p)$, *soit* $Y \subseteq Z$ *et* $Const(s_p) \subset Const(r_p)$. *Dans les deux cas,*

c'est impossible puisque (Y, s_p) et (Z, r_p) sont de même taille. Donc, si $Y = Z$, cela est suffisant pour affirmer que (Y, s_p) et (Z, r_p) sont connectés.
Approfondissons le cas où $Y \neq Z$. Supposons que X est préfixe de Y et de Z. Alors $Y \trianglelefteq XX'$ et $Z \trianglelefteq XX'$ où X' est la concaténation des $nbConst(t_p)$ attributs successeur du dernier de X selon $\lhd$. Y et Z sont donc des préfixes de XX'. Ainsi, l'un est naturellement préfixe de l'autre et les candidats sont bien connectés. Supposons maintenant que Y est préfixe de X et X est préfixe de Z. Alors Y est nécessairement préfixe de Z et les candidats sont également connectés. Supposons enfin Y et Z préfixes de X. Alors $X \trianglelefteq YY'$ et $X \trianglelefteq ZZ'$. En suivant le même raisonnement que pour le premier cas, nous aboutissons à la même conclusion, les candidats sont connectés. □

Cela atteste du bien-fondé de l'ordre DFS puisque nous construisons des ensembles de candidats connectés de tailles maximales. Ce sont les nœuds de l'arbre. Ils sont identifiés par la paire $(\mathbb{X}, s)$ où $\mathbb{X}$ est le plus grand sous-ensemble d'attributs contenu dans le nœud et s correspond à son nombre de constantes. Le lemme suivant montre que, construit cette manière, l'identifiant d'un nœud est unique et non équivoque :

Lemme 3.2 *Soit $\mathbb{C}$ un ensemble maximal de candidats connectés. Alors il existe un unique sous-ensemble d'attributs X tel que $(X, t_p) \in \mathbb{C}$ et $\forall (Y, s_p) \in \mathbb{C}$, $nbAttr(Y) < nbAttr(X)$ avec $\forall (X, t'_p) \in \mathbb{C}$, $nbConst(t_p) = nbConst(t'_p)$.*

Preuve *Soient (X, t_p) et (Y, s_p) deux candidats connectés tels que $nbAttr(Y) = nbAttr(X) = max_{(Z,r_p)\in\mathbb{C}} nbAttr(Z)$. D'après la définition 3.1, si $(X', t'_p) \sim (X, t_p)$ avec $X' \subset X$, alors $X \trianglelefteq X' A_{l+1} \dots A_{l+nbConst(t'_p)} = X''$. Étant donné que (X', t'_p) et (X, t_p) sont de même taille et puisque X a la taille la plus grande possible dans l'ensemble, $X = X''$. D'après le lemme 3.1, $(X', t'_p) \sim (Y, s_p)$ et donc $X = Y$. Comme $||(X, t_p)|| = ||(Y, s_p)||$, alors $nbConst(t_p) = nbConst(s_p)$. Donc l'identifiant de $\mathbb{C}$ est unique et non équivoque.* □

Par ce biais, l'espace de recherche des dépendances fonctionnelles est facilement distinguable puisqu'il correspond aux sous-ensembles d'attributs des identifiants $\mathbb{X}$ des nœuds pour lesquels $s = 0$. Sous ce formalisme, l'arbre de recherche de notre exemple est donné par la figure 3.3.

Pour simplifier les notations et ce, sans perte de généralité, nous considérons que l'arbre de recherche est construit à partir d'un ensemble Λ à n attributs (la table sur laquelle les dépendances sont extraites en contient donc $n+1$). Il est bien connu que l'espace de recherche des dépendances fonctionnelles, dans le contexte que nous avons défini, est composé des 2^n sous-ensembles d'attributs. Le nombre de candidats pour la recherche de dépendances conditionnelles est borné par $\prod_{A\in\Lambda}(|Dom(A)| + 1)$ puisque chaque valeur de chaque attribut peut être associée, dans la table, à toutes les valeurs de tous les autres attributs. Il faut donc considérer tous les motifs possibles sans oublier d'y inclure la valeur anonyme pour chaque attribut. Le nombre de nœuds de notre arbre est donné par le lemme suivant :

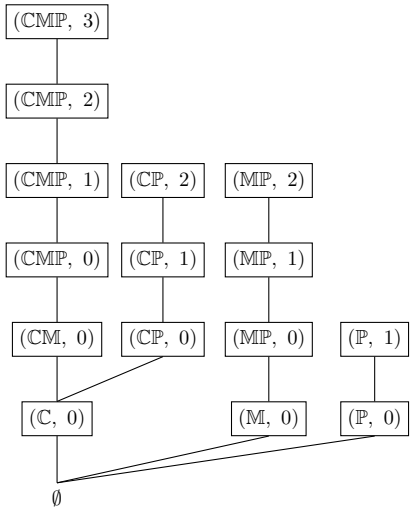


FIGURE 3.3 – Arbre DFS $\mathcal{T}_T$ au niveau d'abstraction des nœuds

Lemme 3.3 *Soit n le nombre d'attributs de l'ensemble Λ sur lequel l'arbre de recherche est construit. Il est composé de $2^n + \sum_{i=0}^{n-1} \binom{n-1}{i} \times (i+1)$ nœuds.*

Preuve *Commençons par considérer les nœuds tels que $s = 0$. Nous pouvons le déduire naturellement du lemme 3.2, cela correspond à l'espace de recherche des dépendances fonctionnelles. Nous avons donc 2^n nœuds de la sorte. Considérons maintenant les nœuds tels que $s \neq 0$. Ce sont ceux qui "étendent" l'espace de recherche précédent par l'ajout de constantes. Nous pouvons ajouter une constante au motif des candidats ayant le nombre d'attributs maximal seulement lorsqu'il ne reste plus d'attributs à considérer. Ainsi l'attribut A_n est contenu dans tous ces nœuds. Il y a 2^{n-1} nœuds de ce type. Le nombre maximal d'attributs de ces nœuds est donné par la ligne $n-1$ des coefficients du triangle de Pascal. Nous avons donc $\binom{n-1}{i}$ de nœuds identifiés par $i+1$ attributs pour $i \in [0, n-1]$. Un nœud peut avoir de 1 à $i+1$ constantes dans le motif, c'est pourquoi nous multiplions le nombre de nœuds par $i+1$.* □

Nous avons maintenant clairement identifié les nœuds de notre espace de recherche. Voyons alors plus précisément comment les candidats connectés qui s'y trouvent sont construits par le biais des deux propositions suivantes :

Proposition 3.1 *Soit $(\mathbb{X}, s)$ un nœud. Alors l'union des premiers parents de chaque candidat de $(\mathbb{X}, s)$ forme intégralement le nœud $(\mathbb{X}', s')$ c.-à-d. $(\mathbb{X}', s') = \bigcup_{(X,t_p)\in(\mathbb{X},s)} FParent(X, t_p)$ est un nœud complet.*

Preuve *Soit (X', t'_p) un premier parent de $(X, t_p) \in (\mathbb{X}, \tau_p)$. D'après la définition 3.3, nous savons que $||(X', t'_p)|| = ||(X, t_p)|| + 1$. Donc, étant donné que tous les candidats de $(\mathbb{X}, \tau_p)$ ont la même taille (définition 3.1), tous leurs parents ont également la même taille.*

Soient (X, t_p) et (Y, r_p) deux candidats connectés de $(\mathbb{X}, \tau_p)$. Étudions leurs parents, (X', t'_p) et (Y', r'_p) respectivement. Puisque (X, t_p) et (Y, r_p) sont incomparables vis-à-vis de $\sqsubset$, leurs attributs sont de la forme (a) $X \subset Y$ ou (b) $X = Y$ et leurs motifs satisfont (i) $Const(t_p) \supset Const(r_p)$, ou (ii) $\exists A_i \in X$ tel que $t_p[A_i] \neq r_p[A_i] \neq \_$, ou (iii) $\exists A_i \in X$ tel que $t_p[A_i] = \_$, $r_p[A_i] \neq \_$ et $\exists A_j \in X$ tel que $t_p[A_j] \neq \_$, $r_p[A_j] = \_$ (naturellement les configurations (b) et (i) ne peuvent pas survenir simultanément pour assurer l'incomparabilité). Nous devons étudier toutes les combinaisons de configurations possibles pour en déduire la connexion éventuelle de leurs parents :

(a) et (i) *Puisque nous cherchons des parents tels que $(X', t'_p) \sqsubset (Y', r'_p)$, supposons que $nbAttr(X) = nbAttr(Y) - 1$ et $nbConst(t_p) = nbConst(r_p) + 1$. Par hypothèse, $(X, t_p) \sim (Y, r_p)$ et donc $XA_l = Y$. Ainsi (X', t'_p) peut être de deux formes (1) (Y, t'_p) où $t'_p[X] = t_p[X]$ et $t'_p[A_l] = \_$. Donc, quelque soit la constante que nous choisissons d'ajouter au motif r_p, nous pouvons obtenir $r'_p = t'_p$ mais jamais $Const(r'_p) \supset Const(t'_p)$ et, si nous ajoutons un attribut à Y, nous revenons à la configuration (i). Sinon, (X', t'_p) peut être de la forme (2) (X, t'_p) où t'_p spécialise une valeur anonyme de t_p et, dans ce cas, quelque soit la forme de (Y', r'_p), ils sont connectés.*

(a) ou (b) et (ii) *D'après la définition 3.3, $\forall A_i \in X$ tel que $t_p[A_i] \neq \_$, $t'_p[A_i] = t_p[A_i]$. Donc la configuration (ii) est maintenue (X', t'_p) et (Y', r'_p) quelque soit leurs constructions et ils sont naturellement connectés.*

(a) et (iii) *La plus petite configuration pour observer ce cas et espérer trouver des parents comparables est $nbAttr(Y) = nbAttr(X) + 1$, $nbConst(t_p) = 2$ et $nbConst(r_p) = 1$. Quelque soit la forme des parents à étudier, nous avons toujours $X' \subseteq Y'$ et $nbConst(t'_p) \leq nbConst(r'_p)$. Donc nous en déduisons qu'ils sont connectés.*

(b) et (iii) *Pour obtenir des parents comparables, supposons que les attributs A_i et A_j sont les seuls à donner ce cas (iii). Alors, nous pouvons générer un parent commun au deux candidats en affectant à $t'_p[A_i]$ la constante $r_p[A_i]$ et à $r'_p[A_j]$ la constante de $t_p[A_j]$. Sinon, les motifs des parents satisfont toujours la (iii) et ils sont donc connectés.*

Donc tous les parents issus de candidats connectés sont également connectés. Supposons qu'il existe $(Z', s'_p) \in (\mathbb{X}', s')$ généré en temps parent de $(Z, s_p) \notin (\mathbb{X}, s)$. Nous avons alors $||(Z, s_p)|| = ||(Z', s'_p)|| - 1$ et nous avons deux cas à considérer pour caractériser (Z, s_p) : $Z = Z'$ et $Const(s_p) \subset Const(s'_p)$ avec $nbConst(s_p) = nbConst(s'_p) - 1$, ou $Z = Z' \setminus \{A_i\}$ avec $s_p = s'_p[Z]$ et $s'_p[A_l] = \_$. D'après les descriptions de chaque cas, il est facile de constater que $(Z, s_p) \in (\mathbb{X}, s)$ si son parent appartient à $(\mathbb{X}', s')$.

Supposons maintenant que $(Z', s'_p) \in RightC(Z, s_p)$. Alors $Z' = Z \setminus \{A_i\}$ avec $A_i \neq A_l$, le dernier attribut de Z et $Const(s'_p) = Const(s_p)$ (définition 3.5). Admettons que (Z', s'_p) est bien incomparable avec tous les candidats de $(\mathbb{X}', s')$ selon l'ordre $\sqsubset$. Si $\exists (X, t_p) \in (\mathbb{X}', s')$ tel que $Z = X$ alors ils sont dans la configuration (ii) ou (iii) et, nous l'avons vu, (Z', s'_p) est alors aussi le parent d'un candidat de $(\mathbb{X}', s')$. Si X est préfixe de Z' alors, nous devrions avoir $Z' \trianglelefteq XX'$. Cependant, ce n'est pas possible puisque X' est une concaténation d'attributs contigus relativement

à $\lhd$ et Z' ne contient pas A_i alors qu'il contient A_l avec $A_i \lhd A_l$. Donc nous avons $XX' \trianglelefteq Z'$ et $(Z', s'_p) \notin (\mathbb{X}', s')$. Si Z' est préfixe de X alors $nbConst(s'_p) \geq 1$ pour assurer de la bonne connexion entre les deux candidats. Donc (Z', s'_p) est également le parent de (Z', s''_p) avec $Const(s''_p) \subset Const(s'_p)$ et, nous l'avons vu, $(Z', s''_p) \in (\mathbb{X}, s)$.
Supposons alors que $(Z', s'_p) \in Sibling(Z, s_p)$. D'après la définition 3.4, $nbConst(s'_p) = 0$. Donc, d'après le lemme 3.2, $Z' = \mathbb{X}'$ et il peut donc être généré à partir du candidat $(\mathbb{X}, t_p) \in (\mathbb{X}, s)$ avec $nbConst(t_p) = s = 0$.
Donc tous les candidats d'un nœud sont bien les premiers parents de candidats connectés. □

Outre le fait que des candidats connectés génèrent un ensemble de premiers parents tous connectés, la proposition suivante montre qu'ils génèrent également le même frère :

Proposition 3.2 *Soient (X, t_p) et (Y, s_p) deux candidats connectés. Alors $Sibling(X, t_p) = Sibling(Y, s_p)$.*

Preuve *Puisque $(X, t_p) \sim (Y, s_p)$ alors, $X.Z = Y.Z' = W$ où Z et Z' sont les ensembles de tailles $nbConst(t_p)$ et $nbConst(s_p)$ respectivement de successeurs relativement à $\lhd$ de A_{l_X} et A_{l_Y} les derniers attributs de X et de Y respectivement. D'après la définition 3.4, $Sibling(X, t_p) = (W \setminus \{A_{l_W}\} \cup \{A_{l_W+1}\}, r_p)$ où A_{l_W} est le dernier attribut de W avec $nbConst(r_p) = 0$. Alors, puisque le frère d'un candidat dépend seulement de l'ensemble W qui est unique pour tous les candidats connectés, nous en déduisons que $Sibling(X, t_p) = Sibling(Y, s_p)$.* □

Maintenant que l'arbre de recherche est formellement caractérisé, nous pouvons étudier le partitionnement des candidats afin de regrouper ceux à considérer en parallèle.

3.1.1.3 Partitionnement des candidats

Naturellement, les candidats connectés peuvent être traités en même temps puisqu'ils ne donnent pas d'information les uns par rapport aux autres. Nous pouvons aller plus loin dans ce découpage des candidats étant donné que plusieurs nœuds peuvent être étudiés en parallèle. Pour ce faire, nous devons assurer que les premiers parents d'un candidat, puisqu'ils donnent une spécialisation directe, mais aussi ses enfants droits qui le généralisent immédiatement soient traités à l'itération suivant celle du candidat. Plus formellement, nous avons le partitionnement suivant :

Définition 3.6 (Partitionnement DFS) *Soit $\mathcal{R} = \{\mathcal{R}_1, \ldots, \mathcal{R}_k\}$ un partitionnement des candidats de l'arbre. $\mathcal{R}$ est un partitionnement DFS si et seulement si $(X, t_p) \in \mathcal{R}_i$, et $(Y, s_p) \in \mathcal{R}_{i+1}$ pour tout $(Y, s_p) \in FParent(X, t_p)$ ou $(Y, s_p) \in RightC(X, t_p)$.*

Exemple *La figure 3.4 donne le partitionnement DFS de l'arbre $\mathcal{T}_T$.*

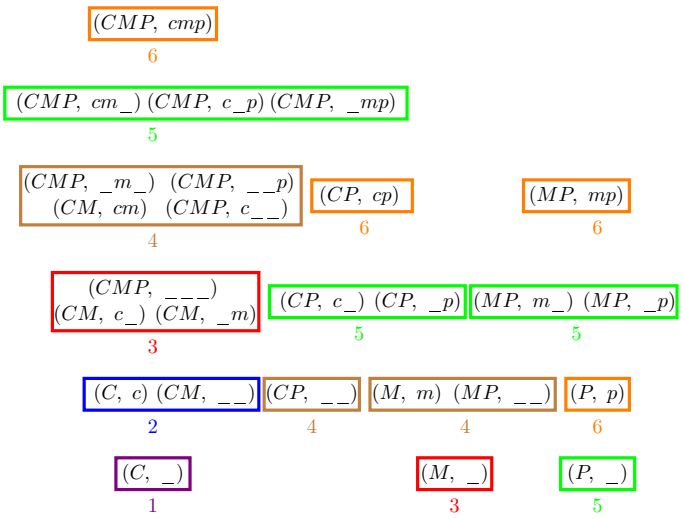


FIGURE 3.4 – Partitionnement DFS de $\mathcal{T}_T$

Ce partitionnement regroupe tous les candidats pouvant être traités en même temps par l'algorithme. Ceux de la partie $\mathcal{R}_i$ seront donc étudiés à l'itération i. Le théorème suivant précise, étant donné un candidat, la partie à laquelle il appartient :

Théorème 3.1 *Soit (X, t_p) un candidat de l'arbre $\mathcal{T}_{A_t}$. Alors, $(X, t_p) \in \mathcal{R}_k \Leftrightarrow k = 2l - nbAttr(X) + nbConst(t_p)$ où l est le rang de A_l, le dernier attribut de X.*

Preuve *Naturellement, $(A_1, \_)$, le premier candidat de l'arbre, doit appartenir à $\mathcal{R}_1$, nous avons $1 = 2 * 1 - 1 + 0$. D'après la définition 3.6, les candidats $(A_1, a) \forall a \in Dom(A_1)$ et $(A_1A_2, \_\_)$ doivent être considérés pour $\mathcal{R}_2$ puisque ce sont les premiers parents de $(A_1, \_)$ et $2 = 2 * 1 - 1 + 1 = 2 * 2 - 2 + 0$. Nous avons $RightC(A_1A_2, \_\_) = (A_2, \_)$, donc $(A_2, \_) \in \mathcal{R}_{3=2+1=2*2-1+0}$ et donc, en suivant un tel chemin en dents de scie, $(A_n, \_)$ doit appartenir à $\mathcal{R}_{2n-1+0}$. Ainsi $(A_l, \_) \in \mathcal{R}_{2l-1+0}$ et tous les candidats dont le dernier attribut est A_l doivent être étudiés avant lui. En effet, $(A_l, \_)$ est alors un enfant droit de ces candidats ou l'enfant droit d'un enfant droit ou ainsi de suite : ceux à deux attributs doivent être traités dans $\mathcal{R}_{2l-2}$, ceux à trois attributs dans $\mathcal{R}_{2l-3}$, etc. et, plus généralement, le candidat (AA_l, t_p) avec $A \subseteq \Lambda_{\lhd A_l}$ ($\Lambda_{\lhd A_l}$ désigne les attributs de Λ ayant un rang strictement inférieur à A_l) et $nbConst(t_p) = 0$ doit être étudié dans $\mathcal{R}_{2l-nbAttr(A)}$. Considérons maintenant le motif. D'après la définition 3.6, (X, t_p) avec $nbConst(t_p) = 1$ doit être traité juste après (X, t'_p) si $nbConst(t'_p) = 0$ et juste avant (X, t''_p) si $nbConst(t''_p) = 2$ puisque $(X, t_p) \in FParent(X, t'_p)$ et $FParent(X, t_p) \ni (X, t''_p)$. Dons, plus généralement, (X, t_p) doit être étudié dans $\mathcal{R}_{i+nbConst(t_p)}$ si (X, t'_p) avec $nbConst(t'_p) = 0$ est traité dans $\mathcal{R}_i$.* □

Chaque candidat est donc affecté à une itération de l'algorithme et tous les candidats de la même partie $\mathcal{R}_i$ sont considérables en même temps. Nous en déduisons le nombre maximal d'itérations de l'algorithme :

Corollaire 3.1 *Soit $\mathcal{R}^*$ le plus petit (en terme de nombre de parties) partitionnement DFS pour un arbre construit sur n attributs. Alors $|\mathcal{R}^*| = 2n$.*

Remarque. Ce partitionnement théorique affecte à la même partie des candidats comparables selon $\sqsubset$ (par ex. $(CM, \_m_1) \in \mathcal{R}_3$ et $(M, \_) \in \mathcal{R}_3$ alors que $(CM, \_m_1) \sqsupset (M, \_)$). Cependant notre stratégie d'élagage permet de ne pas considérer tous les candidats des parties mais seulement ceux étant incomparables (lemme 3.4).

3.1.2 L'algorithme ParaCoDe

Maintenant que les bases sont posées, nous pouvons décrire l'algorithme 1 qui, étant donnés une table T et un seuil de validité α, retourne toutes les dépendances conditionnelles approximatives satisfaites par au moins $\alpha\%$ des tuples.

Algorithme 1: ParaCoDe

Entrée : Table T, seuil α

1 **pour chaque** $A_t \in \mathcal{A}$ **faire**
2 $\quad \mathcal{R}_1 \leftarrow (A_1, \_)$ avec $A_1 \in \Lambda$;
3 $\quad r \leftarrow 1$;
4 $\quad$ **tant que** $\mathcal{R}_r \neq \emptyset$ *ou* $\mathcal{R}_{r+1} \neq \emptyset$ **faire**
5 $\quad\quad$ **pour chaque** $(X, t_p) \in \mathcal{R}_r$ **faire**
6 $\quad\quad\quad$ /* `Boucle parallèle` */;
7 $\quad\quad\quad$ **si** (X, t_p) *n'est pas couvert* $\Gamma^+[t]$ **alors**
8 $\quad\quad\quad\quad$ **si** (X, t_p) *n'est pas couvert* $\Gamma^-[t]$ **alors**
9 $\quad\quad\quad\quad\quad$ **si** $T \models (X \rightarrow_\alpha A_t, t_p)$ **alors**
10 $\quad\quad\quad\quad\quad\quad$ $\Gamma^+[t] \leftarrow \Gamma^+[t] \cup \{(X, t_p)\}$;
11 $\quad\quad\quad\quad\quad\quad$ $\Gamma^+[t] \leftarrow \Gamma^+[t] \setminus \{(Y, s_p) \mid (Y, s_p) \sqsupset (X, t_p)\}$;
12 $\quad\quad\quad\quad\quad\quad$ $\mathcal{R}_{r+1} \leftarrow \mathcal{R}_{r+1} \cup RightC(X, t_p)$;
13 $\quad\quad\quad\quad\quad\quad$ $\mathcal{R}_{r+2} \leftarrow \mathcal{R}_{r+2} \cup Sibling(X, t_p)$;
14 $\quad\quad\quad\quad\quad$ **sinon**
15 $\quad\quad\quad\quad\quad\quad$ $\Gamma^-[t] \leftarrow \Gamma^-[t] \cup \{(X, t_p)\}$;
16 $\quad\quad\quad\quad\quad\quad$ $\Gamma^-[t] \leftarrow \Gamma^-[t] \setminus \{(Y, s_p) \mid (Y, s_p) \sqsubset (X, t_p)\}$;
17 $\quad\quad\quad\quad\quad\quad$ $\mathcal{R}_{r+1} \leftarrow \mathcal{R}_{r+1} \cup FParent(X, t_p)$;
18 $\quad\quad\quad\quad$ **sinon**
19 $\quad\quad\quad\quad\quad$ $\mathcal{R}_{r+1} \leftarrow \mathcal{R}_{r+1} \cup FParent(X, t_p)$;
20 $\quad\quad$ $r \leftarrow r + 1$;
21 **retourner** Γ^+

Principe général Nous fixons, à chaque tour de la boucle `Pour` (lignes 1 à 20), l'attribut cible des dépendances. $\mathcal{R}_1$ reçoit le premier candidat selon l'ordre $\lessdot$ de l'arbre construit sur $\Lambda = \mathcal{A} \setminus \{A_t\}$. Les parties à explorer sont instanciées au fur et

à mesure des besoins selon la définition 3.6. Ainsi, une boucle `Tant que` (lignes 4 à 20) est préférable à un `pour` i `de` 1 `à` $2n$ puisque les candidats sont ajoutés aux parties $r+1$ ou $r+2$ relativement à la partie en cours. Il suffit ensuite d'examiner, en parallèle, tous les candidats $(X, t_p) \in \mathcal{R}_r$. Nous commençons par contrôler leurs possibles liens avec les candidats déjà traités : (i) les candidats déterminant A_t sont stockés dans $\Gamma^+[t]$. (X, t_p) est couvert par $\Gamma^+[t]$ si et seulement si $\exists (Y, s_p) \in \Gamma^+[t]$ tel que $(Y, s_p) \sqsubset (X, t_p)$. Dans ce cas, (X, t_p) détermine la cible mais la dépendance n'est pas minimale, ce n'est donc pas utile de le considérer. (ii) Les candidats ne déterminant pas la cible A_t sont stockés dans $\Gamma^-[t]$. (X, t_p) est couvert par $\Gamma^-[t]$ si et seulement si $\exists (Y, s_p) \in \Gamma^-[t]$ tel que $(Y, s_p) \sqsupset (X, t_p)$. Dans ce cas, nous savons déjà que (X, t_p) ne détermine pas non plus la cible, il n'est donc pas nécessaire de le tester mais nous devons ajouter son premier parent à l'itération suivante. Si le candidat n'est couvert par aucune bordure (lignes 9 à 17), alors nous devons vérifier sa validité relativement à T et à α. Si la dépendance est satisfaite, alors nous l'ajoutons à l'ensemble $\Gamma^+[t]$ auquel nous supprimons les candidats plus spécifiques. Nous devrons ensuite considérer ses enfants droits à l'itération suivante afin d'assurer la minimalité de cette dépendance. Pour ne pas interrompre le parcours des parents, nous ajoutons à $\mathcal{R}_{r+2}$ le frère du candidat. Si en revanche le candidat ne satisfait pas la dépendance, il faut l'ajouter à l'ensemble $\Gamma^-[t]$ auquel sont retirés les candidats les plus généraux. Nous programmons ensuite l'étude de son premier parent à $\mathcal{R}_{r+1}$. Une fois que tous les candidats ont été traité, r est incrémenté pour passer à l'itération suivante.

Exemple *La figure 3.5 illustre l'exécution de l'algorithme sur l'arbre $\mathcal{T}_T$. Nous avons noté en indice des candidats l'itération de l'algorithme à laquelle ils sont exécutés et en exposant le résultat du test de validité : $+$ signifie que la dépendance est satisfaite, $-$ qu'elle ne l'est pas. Nous avons représenté ici seulement les candidats non couverts par une des bordure. En effet, $(MP, m_2p_1) \in FParent(MP, \_\, p_1)$ est couvert par $(MP, m_2\_\,) \in \Gamma^+$ donc il n'apparaît pas sur la figure. La table 3.1 montre l'évolution des ensembles Γ^+ et Γ^-.*

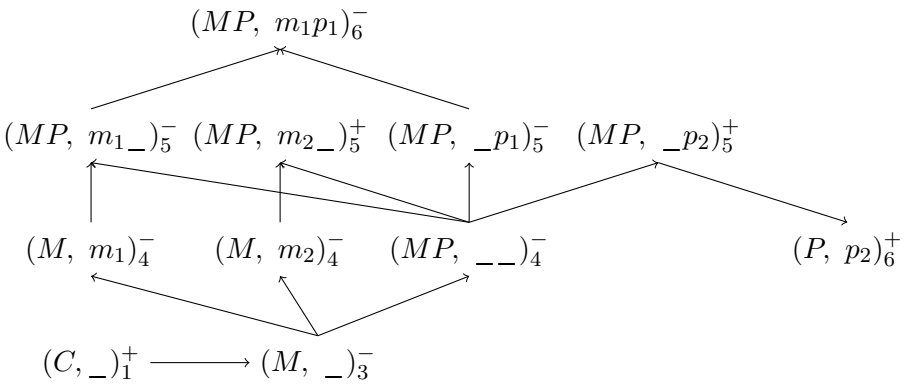


FIGURE 3.5 – Exécution de **ParaCoDe** sur $\mathcal{T}_T$

1	$\Gamma^+ = \{(C, \_)\}$ $\Gamma^- = \{\}$
2	$\Gamma^+ = \{(C, \_)\}$ $\Gamma^- = \{\}$
3	$\Gamma^+ = \{(C, \_)\}$ $\Gamma^- = \{(M, \_)\}$
4	$\Gamma^+ = \{(C, \_)\}$ $\Gamma^- = \{(M, m_1), (M, m_2), (MP, \_\_)\}$
5	$\Gamma^+ = \{(C, \_), (MP, m_2\_), (MP, \_p_2)\}$ $\Gamma^- = \{(MP, m_1\_), (M, m_2), (MP, \_p_1)\}$
6	$\Gamma^+ = \{(C, \_), (MP, m_2\_), (P, p_2)\}$ $\Gamma^- = \{(MP, m_1p_1), (M, m_2)\}$

TABLE 3.1 – Évolution des bordures

Afin d'assurer qu'aucun test n'est redondant, le lemme suivant montre que les candidats traités en parallèle ne sont pas comparables relativement à $\sqsubset$:

Lemme 3.4 *Soient (X, t_p) et (Y, s_p) deux candidats générés pour l'itération $\mathcal{R}_i$. Alors ils sont incomparables relativement à $\sqsubset$.*

Preuve *Soient l_X et l_Y le rang de A_{l_X} et de A_{l_Y}, le dernier attribut de X et de Y respectivement. D'après le théorème 3.1, $2l_X + nbAttr(X) - nbConst(t_p) = 2l_Y + nbAttr(Y) - nbConst(s_p)$. Supposons que $(X, t_p) \sqsubset (Y, s_p)$. Deux cas sont alors à considérer : (i) $X = Y$ et $Const(t_p) \subset Const(s_p)$ ou (ii) $X \subset Y$ et $Const(t_p) \subseteq Const(s_p)$.*
Commençons par examiner le premier. Puisque $X = Y$, nous avons naturellement $2l_X + nbAttr(X) = 2l_Y + nbAttr(Y)$ et donc, comme ils appartiennent à la même partie, nous devons avoir $nbConst(t_p) = nbConst(s_p)$. Dans ce cas, il est facile de voir que nous ne pouvons pas obtenir $Const(t_p) \subset Const(s_p)$. Donc t_p et s_p, et plus généralement (X, t_p) et (Y, s_p), sont soit égaux, soit incomparables.
Considérons alors le second cas (ii) et supposons que $l_X < l_Y$. La situation extrême que nous pouvons rencontrer est $l_X = 1$ et $l_Y = n$. Puisque $0 \leq nbConst(t_p) \leq 1$ and $0 \leq nbConst(s_p) \leq n$, alors $2 \times 1 + 1 - nbConst(t_p) \neq 2 \times n + n - nbConst(s_p)$ et (X, t_p) et (Y, s_p) ne peuvent pas appartenir à la même itération. Fixons alors $l_X = l_Y$. Nous obtenons $nbAttr(X) = nbAttr(Y) - k$ et $nbConst(t_p) = nbConst(s_p) - k$ avec $1 \leq k \leq n - 1$. D'après les définitions 3.3 et 3.5, nous constatons que (X, t_p) est un enfant droit ou l'enfant droit d'un enfant droit ou etc. d'un candidat (Z, r_p) et (Y, s_p) est un parent ou le parent d'un parent ou etc. de ce même candidat (Z, r_p). Ceci est impossible puisque, l'algorithme 1 génère pour un candidat soit ses enfants droits et son frère si la dépendance est satisfaite (lignes 12 et 13), soit son premier parent si elle ne l'est pas (ligne 17). □

Le théorème suivant montre que l'algorithme **ParaCoDe** retourne bien la couverture canonique de l'ensemble des dépendances satisfaites par une table T donnée en entrée :

Théorème 3.2 *L'algorithme* **ParaCoDe** *retourne l'ensemble des dépendances minimales satisfaites par une table T donnée en entrée.*

Preuve *Nous avons vu dans le papier de [Hanusse & Maabout 2011] que l'ensemble des dépendances minimales représente une bordure de l'espace de recherche. Prouvons alors que nous retournons bien une bordure. D'après le lemme 3.4, les candidats traités en même temps sont incomparables selon $\sqsubset$ donc il n'est pas nécessaire de contrôler les ajouts simultanés. Si un candidat est considéré, alors la ligne 7 nous assure qu'il n'existe pas d'éléments plus généraux que lui déjà stockés dans Γ^+. Nous insérons donc dans Γ^+ des éléments minimaux à ce moment de l'algorithme et nous supprimons les plus spécifiques. Nous évaluons ensuite la minimalité des dépendances en considérant les enfants droits.*
Γ^+ contient donc des dépendances minimales incomparables. Lorsqu'une dépendance est valide, nous programmons pour l'itération $r+2$ son frère. Ainsi, nous assurons que tous les candidats pour lesquels nous n'avons pas d'information sont traités donc Γ^+ est complet. □

3.1.3 Optimisations

Nous avons proposé une version relativement naïve de l'algorithme d'extraction des dépendances minimales. Plusieurs étapes méritent une attention particulière : pour commencer, étant donné que tous les candidats générés sont stockés dans Γ^+ ou Γ^-, les tests de couverture deviennent rapidement très coûteux, nous devons donc les optimiser. Ensuite, nous devons chercher à ne générer qu'un minimum de candidats afin de ne pas avoir à faire trop de tests inutiles.

Tests de couverture

Γ^+ contient les candidats les plus généraux déjà traités qui déterminent la cible et Γ^- les candidats les plus spécifiques qui ne la déterminent pas. Savoir si le candidat (X, t_p) est couvert par Γ^+ ou Γ^- revient à parcourir l'ensemble à la recherche d'un élément plus général ou spécifique que (X, t_p) respectivement. Le test d'inclusion entre les ensembles d'attributs et/ou les constantes du motifs est impératif mais le lemme suivant montre qu'il n'est pas nécessaire de considérer tout l'ensemble mais seulement les éléments de taille $||(X, t_p)|| + 1$:

Lemme 3.5 *Soit (X, t_p) un candidat. Si $\exists (Y, s_p) \in \Gamma^-$ tel que $(X, t_p) \sqsubset (Y, s_p)$, alors $||(Y, s_p)|| = ||(X, t_p)|| + 1$.*

Preuve *Puisque (Y, s_p) est déjà stocké dans Γ^- lorsque nous étudions (X, t_p), alors (Y, s_p) apparaît avant (X, t_p) dans l'ordre DFS (c.-à-d. $(Y, s_p) \lessdot (X, t_p)$). De plus, $(X, t_p) \sqsubset (Y, s_p)$ donc $X \subseteq Y$ et, pour respecter ces deux conditions, ils partagent le même dernier attribut. D'après la définition 3.5, (X, t_p) est donc un enfant droit de (Y, s_p) même s'il n'a pas été généré ainsi bien entendu étant donné que (Y, s_p) ne détermine pas la cible. Supposons que $||(Y, s_p)|| = ||(X, t_p)|| + 2$. Nous en déduisons que $(X, t_p) \in RightC(RightC(Y, s_p))$. En suivant l'ordre DFS,*

$RightC(Y, s_p) = (Y', s'_p)$ doit être traité avant (X, t_p) par l'algorithme et il est facile de voir que (Y', s'_p) est également couvert par (Y, s_p). L'itération suivante concerne donc les candidats de $FParent(Y', s'_p)$. Quelque soit le résultat de leurs tests de validité et la continuation des exécutions, il est alors impossible de rencontrer (X, t_p). Le seul test de couverture utile porte donc sur les candidats tels que $||(Y, s_p)|| = ||(X, t_p)|| + 1$. □

Bien entendu, le même raisonnement peut être mené pour la couverture avec l'ensemble Γ^+. Il suffit dans ce cas de considérer les candidats déjà stockés de taille $||(X, t_p)|| - 1$. Nous découpons donc ces ensembles de sorte que la case i ne contienne que des candidats de taille i.
Les candidats de Γ^- couvrant les candidats en cours sont également remarquables par le fait qu'ils y ont été ajouté seulement à l'itération précédente :

Lemme 3.6 *Soit $(X, t_p) \in \mathcal{R}_{i+1}$ un candidat. Si $\exists (Y, s_p) \in \Gamma^-$ tel que $(Y, s_p) \sqsupset (X, t_p)$, alors $(Y, s_p) \in \mathcal{R}_i$.*

Preuve *Supposons que $(Y, s_p) \in \mathcal{R}_i \cap \Gamma^-$ couvre $(X, t_p) \in \mathcal{R}_{i+k}$ avec $k \geq 1$. Puisque $(Y, s_p) \lessdot (X, t_p)$ (c.-à-d. (a) $Y \lhd X$ ou (b) $Y = X \wedge Const(s_p) \subset Const(t_p)$) et, par hypothèse, $(Y, s_p) \sqsupset (X, t_p)$ (c.-à-d. (i) $X \subset Y \wedge Const(t_p) \subseteq Const(s_p)$ ou (ii) $X = Y \wedge Const(t_p) \subset Const(s_p)$), nous savons par le lemme 3.5 que $||(X, t_p)|| = ||(Y, s_p)|| - 1$. Nous en déduisons que seuls les conditions (a) et (i) peuvent survenir simultanément et donc X et Y partagent le même dernier attribut et $nbConst(t_p) \leq nbConst(s_p)$. De ce fait (X, t_p) est un enfant droit de (Y, s_p) (définition 3.5). Cependant, les candidats $(X', t'_p) \in RightC(Y, s_p) \cap \mathcal{R}_{i+1}$ génèrent leurs parents dans l'algorithme 1 et (X, t_p) ne peut alors pas être généré.* □

Il n'est donc pas nécessaire de maintenir l'ensemble Γ^- puisqu'il suffit de ne garder que les candidats de l'itération précédente. Nous économisons donc du temps mais aussi de la mémoire.

Observons maintenant les candidats qui sont couverts. Là encore, nous pouvons nous permettre quelques raccourcis grâce aux deux lemmes suivants :

Lemme 3.7 *Soit (X, t_p) un candidat couvert par Γ^+. Alors (X, t_p) a été généré en tant que premier parent.*

Preuve *Si (X, t_p) est couvert par Γ^+, alors $\exists (Y, s_p) \in \Gamma^+$ tel que $(Y, s_p) \sqsubset (X, t_p)$. Supposons tout d'abord que (X, t_p) est généré en tant qu'enfant droit de (X', t'_p). Alors, d'après la définition 3.5, $(X, t_p) \sqsubset (X', t'_p)$. Donc nous avons également $(Y, s_p) \sqsubset (X', t'_p)$. D'après le lemme 3.4, (X', t'_p) et (Y, s_p) ne peuvent pas être traités dans la même itération donc $(Y, s_p) \lessdot (X', t'_p)$ et (X', t'_p) aurait dû être élaguer puisque ce n'est pas une dépendance minimale. Donc (X, t_p) ne peut pas être généré en tant qu'enfant droit s'il est couvert par Γ^+.*
Supposons maintenant que $(X, t_p) = Sibling(X', t'_p)$. Alors, d'après la définition

3.4, $nbConst(t_p) = 0$, donc $(Y, s_p) \sqsubset (X, t_p) \Rightarrow Y \subset X$. Puisque $(Y, s_p) \lessdot (X, t_p)$ alors, (X, t_p) est nécessairement un parent de (Y, s_p). Étant donné que (X, t_p) est le frère de (X', t'_p), ils appartiennent au même sous-arbre enraciné en (Y, s_p). Donc $(X', t'_p) \in Parents(Y, s_p)$ doit être élagué et (X, t_p) ne peut pas être généré par ce biais. □

Nous devons alors séparer les parents des autres candidats générés pour l'itération suivante. L'instruction des lignes 17 et 19 de l'algorithme 1 doit être remplacée par $parents \leftarrow parents \cup \{(X, t_p)\}$. Avant d'incrémenter r, il faut alors ajouter à $\mathcal{R}_{r+1}$ les candidats de l'ensemble *parents* n'étant pas couverts par Γ^+. Le test de la ligne 7 peut alors être supprimé.
Le lemme suivant montre que ces parents ne peuvent pas être couverts par Γ^- :

Lemme 3.8 *Soit (X, t_p) un candidat généré en temps que premier parent. Alors $\nexists (Y, s_p) \in \Gamma^-$ tel que $(Y, s_p) \sqsupset (X, t_p)$.*

Preuve *Soit (X', t'_p) le candidat ayant généré (X, t_p). Nous avons $(X', t'_p) \sqsubset (X, t_p)$ et donc $(X', t'_p) \sqsubset (Y, s_p)$. Par le lemme 3.6 et la définition 3.6, nous déduisons que (X', t'_p) et (Y, s_p) sont traités lors de la même itération (celle précédent celle de (X, t_p)). Cependant, le lemme 3.4 montre que c'est impossible donc (X, t_p) ne peut pas être couvert par Γ^-.* □

Le test de couverture avec Γ^- est donc à faire à la fin de l'itération r sur les candidats appartenant à $\mathcal{R}_{r+1}$ et ce, avant d'y ajouter les parents, bien entendu. Ainsi, lorsque nous entamons une nouvelle itération, nous pouvons directement tester la validité des candidats en sachant que nous avons un nombre minimal de candidats à considérer.

Génération des candidats

Nous l'avons vu dans le chapitre 2, pour gagner du temps, la première chose à faire est d'élaguer l'espace de recherche au maximum. Même si les candidats inutiles sont supprimés après les tests de couverture avec les bordures positives ou négatives, il est important de ne générer qu'un minimum de candidats. Nous allons détailler ici la manière la plus efficace de générer les frères et les parents.

Génération du frère D'après la définition 3.4 : $Sibling(X, t_p) = (X', t'_p)$ avec $X' = X \cup \{A_{l+1} \dots A_{l+nbConst(t_p)-1} A_{l+nbConst(t_p)+1}\}$ où A_{l+1} est le successeur direct du dernier attribut de X selon l'ordre $\lhd$ et $nbConst(t'_p) = 0$. Nous savons également que chaque élément n'a qu'un seul frère et celui-ci est partagé par tous les candidats d'un ensemble connecté maximal (propriété 3.2).
Il n'est donc pas nécessaire que chaque candidat de l'ensemble génère son frère puisque nous aurions autant de doublons que de candidats et les temps de calcul et de suppression de ces doublons ne sont pas négligeables. Nous voyons facilement qu'un

frère est plus rapide à générer à partir d'un candidat n'ayant aucune constante dans son motif. Le frère ayant également un motif anonyme, le lemme 3.2 nous permet d'affirmer que tous les frères de l'espace de recherche sont des frères de candidats au motif anonyme. Le lemme suivant montre qu'il est suffisant de générer les frères uniquement à partir de candidats au motif anonyme :

Lemme 3.9 *Soit (X, t_p) le candidat au motif anonyme choisi pour générer le frère de son ensemble connecté. Si (X, t_p) ne génère pas le frère, alors ce dernier aurait été élagué.*

Preuve *Soit $(X', t'_p) \sim (X, t_p)$ avec (X', t'_p) qui détermine la cible. Il y a deux raisons qui expliquent que (X, t_p) ne génère pas le frère : (i) $\exists (Y, s_p) \in \Gamma^+$ tel que $(Y, s_p) \sqsubset (X, t_p)$ donc (X, t_p) est élagué, ou (ii) (X, t_p) ne détermine pas la cible et donc il génère son premier parent.*
Commençons par étudier le premier cas de figure (i). Puisque $(Y, s_p) \in \Gamma^+$ lorsque (X, t_p) est traité, $(Y, s_p) \lessdot (X, t_p)$. Par hypothèse, $nbConst(t_p) = 0$ donc, pour avoir $(Y, s_p) \sqsubset (X, t_p)$, il faut que $Y \subset X$ et $nbConst(s_p) = 0$. Donc, Y est un préfixe de X de taille $nbAttr(X) - 1$ (lemme 3.5). Par construction (définition 3.5), le frère de (X, t_p) partage également ce préfixe donc il est aussi couvert par (Y, s_p) et aurait été élagué.
Considérons maintenant le second cas (ii). (X, t_p) génère pour $\mathcal{R}_{r+1}$ un candidat de la forme $(X \cup \{A_{l+1}\}, t'_p)$ où A_{l+1} est le successeur direct de A_l, le dernier attribut de X et $nbConst(t'_p) = 0$. Rappelons que le frère de (X, t_p), à étudier dans $\mathcal{R}_{r+2}$, est de la forme $(X \setminus \{A_l\} \cup \{A_{l+1}\}, t''_p)$ avec $nbConst(t''_p) = 0$. Nous avons donc $(X \setminus \{A_l\} \cup \{A_{l+1}\}, t''_p) \sqsubset (X \cup \{A_{l+1}\}, t'_p)$. Si $(X \cup \{A_{l+1}\}, t'_p)$ ne détermine pas la cible, alors le frère serait élagué puisque $(X \cup \{A_{l+1}\}, t'_p) \in \Gamma^-$. Si $(X \cup \{A_{l+1}\}, t'_p)$ détermine la cible, alors il doit vérifier sa minimalité en générant $(X \setminus \{A_l\} \cup \{A_{l+1}\}, t''_p)$. □

Même si la suppression des doublons se fait relativement rapidement, il est préférable qu'un seul candidat ne consacre du temps à générer son frère.

Génération des parents Du fait de la spécialisation des valeurs anonymes du motif, le nombre de parents générés peut être important et d'autant plus en considérant que plusieurs candidats engendrent les mêmes parents. Nous le voyons sur la figure 3.5, le candidat $(MP, m_1\_)$ est parent de (M, m_1) mais aussi de $(MP, \_\_)$. Il a donc été généré deux fois. Si (M, m_1) détermine la cible, alors $(MP, m_1\_)$ doit être élagué par le test de couverture avec Γ^+. Une solution simple pour éviter les doublons et réduire le nombre de tests de couverture consiste à redéfinir l'ensemble des parents de manière à ce que chaque parent ne soit issu que d'un seul candidat. Nous proposons une nouvelle définition dans laquelle seul le dernier attribut du motif peut être spécialisé et nous pouvons également ajouter un attribut sans modifier le motif :

Définition 3.7 (Ensemble restreint de parents) $(X', t'_p) \in Parents(X, t_p)$ *si et seulement si*

- *Si* $t_p[A_l] = \_$ *où* A_l *est le dernier attribut de* X*, alors* $X' = X$ *avec* $t'_p[X \setminus \{A_l\}] = t_p[X \setminus \{A_l\}]$ *et* $t'_p[A_l] \neq \_$.
- $X' = X \cup \{A_{l+i}\}$ *et* $Const(t'_p) = Const(t_p)$.

Nous devons alors vérifier la validité de la proposition 3.1 pour assurer que tous les premiers parents sont bien générés.

Proposition 3.3 *Soient* (X, t_p) *et* (X, t'_p) *deux candidats tels que* $Const(t_p) \subset Const(t'_p)$ *avec* $nbConst(t_p) = nbConst(t'_p) - 1$ *et* $t_p[A_l] = t'_p[A_l]$*. Alors* (X, t'_p) *est un premier parent de* $(Y, s_p) \sim (X, t_p)$*.*

Preuve *D'après la définition initiale des parents (définition 3.3),* (X, t'_p) *est un parent de* (X, t_p) *puisqu'il spécialise une valeur anonyme de* t_p*. Supposons* $(X, t'_p) \in FParent(Y, s_p)$ *d'après la définition 3.7. Alors* (Y, s_p) *est de la forme (i)* $Y = X$ *et* $s_p[Y \setminus \{A_l\}] = t'_p[Y \setminus \{A_l\}]$ *avec* $s_p[A_l] = \_$ *et* $t'_p[A_l] \neq \_$ *ou (ii)* $Y = X \setminus \{A_l\}$ *et* $s_p = t'_p[Y]$ *avec* $t'_p[A_l] = \_$*. Prouvons alors que, dans les deux cas, nous avons bien* $(Y, s_p) \sim (X, t_p)$*.*
Dans le cas (i), par hypothèse, $\exists! A_i \neq A_l$ *tel que* $t_p[A_i] = \_$ *et* $t'_p[A_i] = a_i = s_p[A_i]$ *avec* $a_i \in Dom(A_i)$*. Nous savons aussi que* $t_p[A_l] = t'_p[A_l] = a_l \in Dom(A_l)$ *et* $s_p[A_l] = \_$*. Donc* $nbConst(t_p) = nbConst(s_p)$ *et, puisque* $X = Y$*, alors* $||(X, t_p)|| = ||(Y, s_p)||$*. Nous en déduisons également que* $Const(t_p) \not\subset Const(s_p)$ *et* $Const(t_p) \not\supset Const(s_p)$*. Ainsi,* (X, t_p) *et* (Y, s_p) *sont bien incomparables selon* $\sqsubset$ *et ils sont aussi connectés.*
Dans le cas (ii), étant donné que, par hypothèse, $nbConst(t_p) = nbConst(t'_p) - 1$ *et* $t_p[A_l] = t'_p[A_l] = \_$*, alors* $nbConst(t_p) = nbConst(s_p) - 1$*. Nous savons que* $Y = X \setminus \{A_l\}$*, donc* $||(X, t_p)|| = ||(Y, s_p)||$*. Puisque* $Y \subset X$ *et* $Const(s_p) \supset Const(t_p)$*, alors* (X, t_p) *et* (Y, s_p) *sont incomparables selon* $\sqsubset$*. Naturellement,* Y *est un préfixe de* X *et* A_l *et le premier successeur du dernier attribut de* Y *car* $(X, t'_p) \in Parent(Y, s_p)$*. Donc nous avons bien* $(Y, s_p) \sim (X, t_p)$*.* □

Nous pouvons donc, sans perdre d'information ni modifier l'espace de recherche, générer, à partir d'un candidat, moins de parents.

En ce qui concerne la génération des enfants droits (définition 3.5), le fait de conserver toutes les valeurs constantes du motif nous assure de la non redondance des candidats engendrés. De plus, l'ensemble généré est à la fois maximal et minimal dans le sens où une autre définition des enfants droits ne permettrait pas de contrôler la minimalité des dépendances trouvées.

3.2 Les paramètres

Nous avons conçu l'algorithme **ParaCoDe** de manière à ce qu'il soit le plus global possible. Nous pouvons ainsi connaître toutes les sortes de dépendances satisfaites par la table. Certains paramètres sont à prendre en compte afin de ne retourner que l'information recherchée par l'utilisateur. En effet, nous donnons la possibilité de fixer le niveau de détail de ces dépendances ainsi que le niveau de vérité.

3.2.1 La granularité

Les dépendances conditionnelles les plus générales sont les dépendances fonctionnelles, celles ayant uniquement des valeurs anonymes dans leurs motifs. C'est le niveau de granularité 0 puisque le motif ne contient aucune constante. Lors de la génération des parents, il suffit d'ignorer les instructions servant à spécialiser les valeurs anonymes.

Supposons une table à 100 attributs, l'analyse des dépendances dont le motif contient 90 constantes n'a peut être pas vraiment de sens, nous pouvons alors stopper l'exploration des parents lorsque i constantes sont déjà instanciées dans le motif. Pour ce faire, nous contrôlons simplement que le niveau de spécialisation des valeurs ne dépasse pas le seuil de granularité fixé.

Si nous souhaitons connaître uniquement les clés minimales, il s'agit du niveau -1. Cette fois, l'arbre de recherche n'est pas construit pour chaque cible mais il contient tous les attributs. Le test de validité consiste simplement à vérifier si le candidat a une taille suffisamment proche de la taille de la table pour être considéré comme clé.

Quelque soit le niveau de granularité fixé, il est facile de vérifier que tous les lemmes et propositions proposés restent valables.

3.2.2 Les mesures d'approximation

Nous avons vu que, du fait de sa non-monotonie, la mesure de force ne peut pas être appliquée lorsque le seuil d'approximation n'est pas égal à 1. Cependant, lorsque nous cherchons des dépendances de granularité 0 ou -1, si $\alpha = 1$, alors il est plus judicieux de procéder en calculant seulement la taille des projections. De plus, la taille d'une projection ne dépend pas de la cible donc nous pouvons la réutiliser, chaque fois que le même candidat se présente, quelque soit la cible.

La manière dont nous calculons la confiance des candidats nous permet de connaître, sans sur-coût supplémentaire, la fréquence d'apparition des valeurs constantes du motif. Ainsi, si une fréquence minimale est requise, nous élaguons efficacement l'espace de recherche en commençant par considérer un domaine restreint pour chaque attribut.

3.3 Expérimentations

Dans cette section, nous détaillons quelques optimisations de programmation puis nous constatons, sur des jeux de données synthétiques mais aussi réels, l'accélération dûe au parallélisme. Enfin, nous comparons nos temps d'exécution avec ce qui se fait de mieux dans la littérature.

Nous avons implémenté l'algorithme **ParaCoDe** en C++ avec OpenMP, une API de programmation parallèle à mémoire partagée. Tous les tests ont été effectués

sur une machine équipée de deux hexa-cœurs Intel Xeon X5680 processeurs 3,33 GHz fonctionnant sous Debian Linux version 3.2.32-1 et 96 Go de RAM, les caches en ont respectivement L1 = 32 Ko, L2 = 256 Ko et L3 = 12Mo.

Nous avons étudié les jeux de données Chess, 11 attributs et 28056 tuples, et Wisconsin Breast Cancer (WBC), 7 attributs et 699 tuples (tirés de http ://archive.ics.uci.edu/ml). Nous avons également traité les jeux de données synthétiques décrits par le tableau 3.2.

	NA	NT	CF
Synth1	7	100K	0.7
Synth2	7	1000K	0.7
Synth3	30	100K	0.7
Synth4	7	100K	0.3

TABLE 3.2 – Caractéristiques des jeux de données synthétiques

NA donne le nombre d'attributs, NT le nombre de tuples et CF est le facteur de corrélation (c.-à-d. les tailles des domaines des attributs sont égales à NT*CF).

3.3.1 Quelques détails de l'implémentation

Nous avons fixé pour nos tests un seuil d'approximation à 1. Nous ne cherchons donc que les dépendances conditionnelles exactes. Pour savoir si un candidat (X, t_p) détermine la cible, nous procédons ainsi :

1. Dans une structure de type `unordered_map`, nous associons à chaque valeur rencontrée de X un vecteur de paires contenant la valeur de la cible ainsi que le nombre d'apparitions.
2. Il ne reste plus qu'à lire cette map pour identifier les valeurs qui correspondent au motif t_p.
3. En comptant le nombre d'apparition de chaque valeur cible en fonction du tuple lu, nous savons si le motif t_p est assez fréquent et si la dépendance est satisfaite relativement au nombre de tuples à conserver.

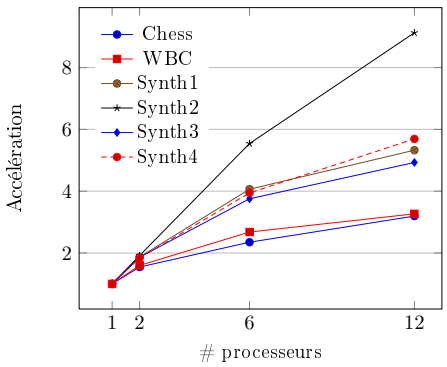
Ainsi, tous les candidats portant sur le même sous-ensemble d'attributs X utilisent cette structure pour déterminer la validité de leur dépendance. Nous devons donc distinguer deux phases dans le calcul de validité : (1) parcours de la table pour initialiser la structure, (2) lecture de la version condensée de la table pour déterminer si le candidat est valide ou pas. Les candidats d'une même partie $\mathcal{R}_i$ sont donc triés afin de commencer la boucle parallèle par le calcul de la structure. Ainsi, nous minimisons les temps d'attente pour synchronisation des processeurs.

Nous avons constaté que quasiment tous les candidats de la forme (A_i, t_p) sont traités à un moment de l'exécution de l'algorithme. Nous commençons alors les traitements pour une cible par le calcul des confiances de tous les attributs A_i. En effet, entamer l'exploration de l'arbre de recherche par les attributs ayant la plus

grande probabilité de déterminer la cible nous permet d'élaguer plus de candidats. Ainsi, l'ordre $\lhd$ se définit comme suit : $A_i \lhd A_j \Leftrightarrow confiance(A_i \rightarrow A_t, \_) > confiance(A_j \rightarrow A_t, \_)$. Nous commençons donc, pour chaque cible, par trier les attributs et les structures produites seront utilisées au cours du calcul.

3.3.2 Mesure de l'accélération

Lorsque le nombre de processeurs alloués au calcul augmente, nous nous attendons à ce que les temps d'exécution diminuent. La figure 3.6 montre l'accélération observée sur nos 6 jeux de données. Le nombre moyen de candidats traités à la même itération est inscrit à droite de la courbe.



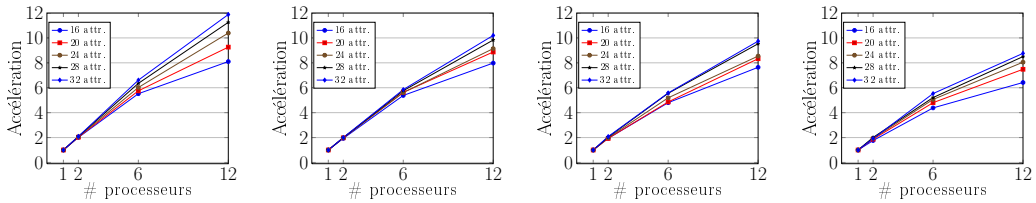
Chess :	7422
BCW :	645
Synth1 :	1840
Synth2 :	12765
Synth3 :	10070
Synth4 :	3967

FIGURE 3.6 – Accélération en fonction du nombre de processeurs

Dans l'idéal, nous aspirons à une accélération linéaire mais ce n'est malheureusement pas le cas ici, analysons alors ce résultat. Le nombre de candidats à traiter en parallèle est relativement important vis-à-vis du nombre de processeurs, cependant, nous devons distinguer deux types de candidats : ceux pour lesquels le calcul de la projection est nécessaire et ceux qui se contentent de parcourir la version résumée de la table. Lorsque l'attribut A_i est spécialisé dans le motif d'un candidat, nous générons $|Dom(A_i)|$ candidats à exécuter simultanément alors qu'une seule projection est requise pour le test de validité. Sur ces jeux de données, la taille de la structure à lire est négligeable vis-à-vis de la taille de la table. L'accélération est donc plus visible sur les candidats calculant la projection. Or, nous avons, en moyenne pour tous les jeux de données, 2 à 3 projections à évaluer par itération. De plus, seul le jeu de données synthétique 2 consacre un temps remarquable pour le calcul de ses projections.

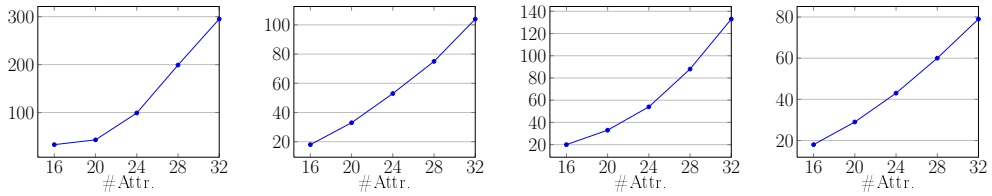
Pour vérifier ce sentiment, nous avons fixé la granularité à 0 afin de n'extraire que les dépendances fonctionnelles. De cette façon, tous les candidats calculent leur projection. Aussi, étant donné que l'espace de recherche de ces dépendances est réduit par rapport à celui des conditionnelles, nous avons créé de nouveaux jeux de données plus volumineux. La figure 3.7 montre les accélérations observées en fonction du nombre d'attributs (indiqué par la légende des courbes), du nombre de tuples (donné par le titre de la figure) et du facteur de corrélation (exprimé en

pourcentage dans le titre de la figure). Nous avons noté sur la figure 3.8 le nombre moyen de projections calculées par itération.



(a) 100K,10% (b) 100K,30% (c) 1M,10% (d) 1M,30%

FIGURE 3.7 – Accélération pour la recherche des DFs



(a) 100K, 10% (b) 100K, 30% (c) 1M, 10% (d) 1M, 30%

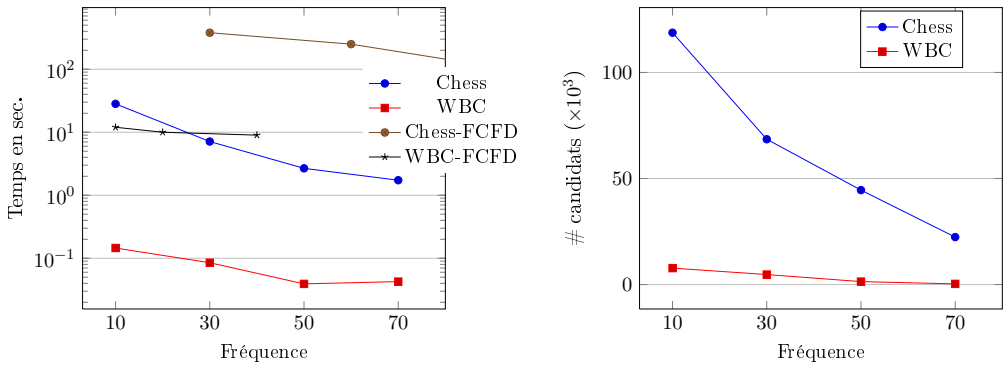
FIGURE 3.8 – Nombre moyen de candidats par itération

Nous remarquons cette fois une bien meilleure accélération. Elle est donc effectivement dûe au nombre de candidats à traiter en parallèle et surtout à la complexité des calculs pour chaque candidat. La même observation est faite lorsque nous cherchons à extraire les clés minimales.

3.3.3 Comparaison avec l'existant

Comparons maintenant notre méthode avec l'algorithme FastCFD développé par [Fan *et al.* 2011] puisque c'est la méthode reconnue comme étant la plus efficace pour extraire les dépendances conditionnelles variables. Puisque nous obtenons de meilleurs temps de calcul avec un maximum de processeurs, nous en utilisons 12 pour ces expérimentations. La figure 3.9(a) donne les temps d'exécution sur les jeux de données réelles lorsque la fréquence minimale des dépendances augmente. Aucun code source n'a pu nous être fourni pour mener notre étude et assurer de la fiabilité de la comparaison dans des conditions similaires, nous nous sommes donc contentés de recopier les temps d'exécution notés dans leur papier. Notre machine est plus performante que celle utilisée par [Fan *et al.* 2011] mais cela donne tout de même une idée de la compétitivité de **ParaCoDe**. Nous avons associé aux temps d'exécution le nombre de candidats traités (figure 3.9(b)). Nous constatons que, même lorsque nous considérons relativement peu de candidats, notre algorithme

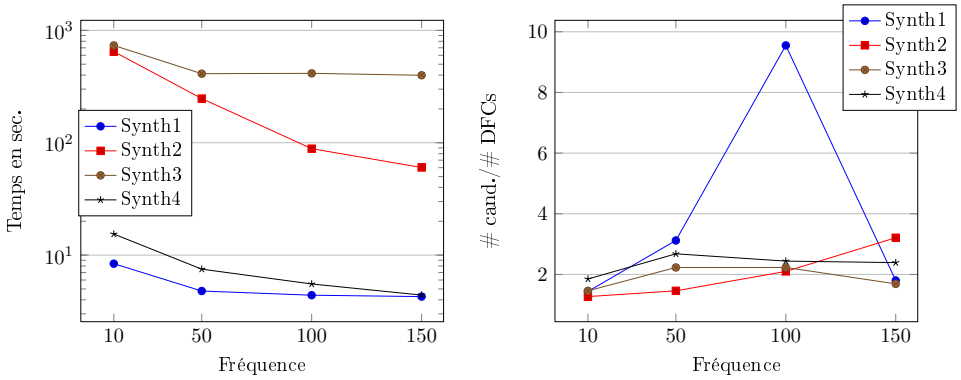
donne de meilleurs temps d'exécution.



(a) Temps d'exécution (b) # candidats

FIGURE 3.9 – Expérimentations sur les jeux de données réelles

Analysons maintenant les jeux de données synthétiques à travers les temps d'exécution (figure 3.10(a)) et le nombre moyen de candidats à considérer avant d'atteindre une dépendance (figure 3.10(b)).
Même si ce ne sont pas les mêmes jeux de données que ceux utilisés par [Fan *et al.* 2011], nous les avons créés en fixant les mêmes paramètres. Nous remarquons alors que, là encore, nos temps d'exécution sont compétitifs vis-à-vis des algorithmes CTANE ou FastCFD. Notons également que le nombre moyen de candidats explorés avant d'obtenir une dépendance minimale est relativement bas sur ces exemples. Cela atteste de la bonne construction de notre arbre de recherche mais aussi de l'efficacité de notre stratégie d'élagage.



(a) Temps d'exécution (b) # candidats/# DFCs minimales

FIGURE 3.10 – Expérimentations sur les jeux de données synthétiques

Étudions maintenant l'extensibilité de notre approche via différents paramètres :

Augmentation du nombre de tuples (jeux Synth1 et Synth2) : La parallélisation de **ParaCoDe** se fait sur les candidats, donc, étant donné qu'il faut 11 fois plus de temps pour calculer une projection sur Synth2 que sur Synth1, nous pouvions envisager cet écart de temps global entre les deux jeux de données. De plus, Synth2 considère environ 10 fois plus de candidats du fait de l'augmentation des tailles des domaines. Cependant, malgré cette complexité supplémentaire, les temps d'exécution pour traiter Synth1 ne sont pas multipliés par 110 mais par 70 seulement en moyenne pour considérer Synth2. Nous l'avons vu, cela est dû à une meilleure utilisation du parallélisme. Nous pouvons donc envisager l'extraction des dépendances sur des jeux de données ayant un nombre important de tuples tout en gardant des temps d'exécution raisonnables relativement à la taille de la table.

Augmentation du nombre d'attributs (jeux Synth1 et Synth3) : L'augmentation du nombre d'attributs va naturellement de pair avec l'augmentation du nombre de candidats. Le nombre de dépendances minimales est également plus élevé pour Synth3 que pour Synth1 (jusqu'à 220 fois plus lorsque la fréquence est fixée à 150). Nous avons vu que, plus il y a de candidats à traiter en parallèle, plus **ParaCoDe** est efficace, mais malheureusement, sur Synth3, le nombre de candidats par itération est très fluctuant. Le temps économisé sur une itération peut être perdu par la suivante, ce qui explique que nous n'observons pas un meilleur résultat.

Réduction du facteur de corrélation (jeux Synth1 et Synth4) : En réduisant le facteur de corrélation, nous réduisons le nombre de valeurs distinctes par attributs et nous augmentons donc le nombre de valeurs fréquentes donc avec lui, le nombre de candidats. Cette fois, puisque nous sommes dans quasiment les mêmes configurations (forme et répartition des candidats, temps de calcul des projections), le parallélisme nous permet obtenir des temps d'exécution comparables.

3.4 Conclusion

Nous avons identifié que toutes les dépendances d'une table peuvent s'exprimer sous la forme de dépendances conditionnelles. Nous avons donc proposé un contexte unifié permettant d'extraire n'importe quel type de dépendances. La recherche des dépendances minimales est un problème de complexité exponentielle en nombre d'attributs et nombre de tuples. Cependant, les machines actuelles permettent de traiter plusieurs tâches à la fois. Nous avons donc proposé **ParaCoDe**, un algorithme parallèle de recherche des dépendances.

Les expérimentations que nous avons menées attestent de l'utilité du parallélisme mais aussi de la bonne construction de notre arbre de recherche ainsi que de l'efficacité de notre stratégie d'élagage. Nous avons comparé nos temps d'exécution avec ceux des méthodes les plus compétitives de nos jours pour constater les bonnes performances de notre algorithme. Nous avons également étudié son extensibilité.

Les résultats obtenus nous encouragent à penser que nous sommes en mesure d'extraire les dépendances sur des jeux de données assez conséquents que ce soit en nombre d'attributs, de tuples ou relativement à la taille des domaines de définitions.

Ces travaux sont l'objet de la publication [Garnaud *et al.* 2013a].

Notre approche semble donc compétitive et adaptée au traitement de gros jeux de données mais il reste des pistes à explorer. Nous avons vu que le parallélisme sur les candidats est très avantageux lorsqu'il y a beaucoup de projections à calculer au cours d'une même itération mais ce n'est pas suffisant. En effet, nous voulons être efficace vis-à-vis des approches séquentielles telles que FUN ou CFUN même lorsqu'il y a peu de candidats. Pour ce faire, un parallélisme sur les données est peut être profitable pour assurer l'accélération. Cela reste à expérimenter pour savoir s'il y a réellement un avantage à préférer cette méthode étant donné le nombre important de candidats relativement au nombre de projection à calculer. Une approche hybride serait sans doute un bon compromis puisque le parallélisme sur les candidats reste préférable pour les tests de couverture.

Dans nos expérimentations, nous avons essentiellement tenu compte des temps d'execution pour juger de l'efficacité de l'algorithme mais une analyse de l'occupation mémoire doit être faite. En effet, nous avons choisi de parcourir l'arbre de recherche en profondeur afin d'élaguer plus efficacement les candidats redondants et d'éviter de surcharger la mémoire. Cependant, nous devons gérer simultanément tous les candidats d'une même itération. Nous stockons des projections de la table, ce qui accélère les temps de calcul mais induit également une occupation mémoire supplémentaire. Nous n'avons pas rencontré dans nos expérimentations de cas critique dans lequel la mémoire serait un problème paralysant mais, puisque nous voulons traiter d'importants jeux de données, une étude approfondie de cette contrainte est nécessaire afin d'identifier les limites de notre méthode relativement à n, m et $\forall A_i$, $|Dom(A_i)|$.

Sur les expérimentations que nous avons menée, le ratio $\#candidats/\#DFCs\ minimales$ est relativement bon mais nous pouvons rencontrer des cas où plus de candidats sont traités avant d'aboutir à une dépendance (notamment lorsque les parties gauches de dépendances contiennent beaucoup d'attributs). Notre stratégie d'élagage peut être améliorée en utilisant les règles proposées par [Yao & Hamilton 2008]. Cependant, cela implique de contrôler des dépendances trouvées pour d'autres cibles et de les analyser pour en déduire les calculs redondants. Nous pouvons alors nous demander si l'étude des dépendances déjà extraites nous permet réellement d'accélérer les temps de calcul. Un examen des stratégies à mettre en place et une analyse théorique des complexités sont à mener pour savoir quelle option est la plus avantageuse.

Le parallélisme a été rendu possible par l'apparition et la popularisation de machines possédant plusieurs cœurs, innovation motivée par un besoin toujours plus grand de performances dans les calculs. Les données à traiter se sont également com-

plexifiées et leur volume augmente de jour en jour. Malheureusement, les avancées technologiques liées au stockage des données semblent précéder l'augmentation des puissances de calcul. Pourquoi alors ne pas anticiper le jour où l'augmentation du nombre de processeurs ne sera plus suffisante en mettant au point une approche quantique d'extraction des dépendances ? Bien entendu, nous devons encore attendre la réalisation physique du qubit mais nous pouvons d'ores et déjà le prévoir.

Même si ce travail mérite d'être approfondi, nous disposons maintenant des dépendances valides sur une table. Nous avons motivé leur extraction par un besoin d'optimisation des requêtes, les prochains chapitres traiteront de ce sujet. Nous allons utiliser ces dépendances comme un outil donnant une connaissance sémantique sur la table.

Chapitre 4

Optimisation des requêtes par les cubes de données

Sommaire

Nous étudions maintenant le problème de la matérialisation partielle des cubes de données. Il n'est pas plus facile à résoudre que celui de l'extraction des dépendances puisqu'il est également NP-Difficile mais la connaissance de ces dépendances nous aide à le résoudre.

Nous commençons ce premier chapitre par une description du besoin d'analyse des données multidimensionnelles pour comprendre l'intérêt des cubes de données pour y répondre. Puisque ces cubes sont utilisés pour optimiser les requêtes, nous détaillons leur construction à différents niveaux de granularité. Nous en déduisons qu'il n'est pas réalisable, tant en temps de calcul qu'en occupation mémoire, de les stocker entièrement. Nous posons alors les bases du problème de leur matérialisation partielle. En effet, il faut choisir les parties à conserver pour optimiser au mieux les requêtes. La définition formelle de ce problème est alors donnée ainsi que quelques notions indispensables à sa résolution.

Comme pour l'extraction des dépendances, nous illustrons nos propos à l'aide de la table *Clientèle*. Afin de mener une étude plus réaliste sur cette table, nous lui ajoutons la colonne $Valeur$ qui donne la mesure sur laquelle porte les analyses. Elle correspond au montant des factures. Nous l'avons vu, l'attribut *Client* est une clé, alors, pour réduire la taille des exemples, nous considérons qu'il donne seulement l'identifiant des tuples. La table ainsi modifiée est décrite par la table 4.1 :

id	*M*	*P*	*T*	*V*
1	m_1	p_1	t_1	10
2	m_1	p_1	t_2	30
3	m_2	p_1	t_2	20
4	m_2	p_2	t_3	50

TABLE 4.1 – Table *Clientèle*.

4.1 Les cubes de données

Les données ne sont pas stockées au hasard ou sans but mais toujours dans l'intention d'en tirer une information utile pour l'aide à la décision notamment. Ces données sont en effet étudiées selon certains critères, ou axes d'analyse, que l'utilisateur cherche à visualiser afin de mettre en avant des tendances ou anomalies particulières en se basant sur une mesure donnée. Alors, pour aider ce besoin d'analyses sur des données de plus en plus complexes, il est essentiel de décomposer l'originale table relationnelle en tables de dimensions décrivant les axes d'analyse. Dans ce contexte, les tables sont appelées tables multidimensionnelles puisque les attributs sont alors ses dimensions. Afin d'examiner plus précisément encore l'objet de l'étude, nous décomposons cette table en pré-calculant tous les résultats des requêtes `GROUP BY`. Cette clause permet de regrouper les valeurs identiques des attributs précisés afin de rendre le résultat plus lisible. Cela revient à une projection sur ces attributs en supprimant les doublons. Pour faciliter la navigation entre ces différentes tables ou résultats de requêtes, [Gray *et al.* 1997] introduisent les formalisations nécessaires autour du cube de données, aussi appelé cube OLAP puisqu'il est surtout utilisé dans ce contexte.

Nous commençons cette section par quelques généralités justifiant l'utilisation des cubes de données pour l'optimisation des requêtes. Nous détaillons ensuite les étapes de sa construction en fonction du point de vue à adopter.

4.1.1 Motivation

Pour calculer une requête, nous avons parfois besoin de regrouper les informations pour donner une vue plus synthétique des dimensions qui nous intéressent. Ceci se fait à l'aide de la clause `GROUP BY` mais que devient alors la mesure ? Elle doit également être agrégée et pour cela, il faut utiliser une fonction d'agrégation (ou de regroupement). Comme l'expliquent les auteurs de [Gray *et al.* 1997], il faut distinguer trois sortes de fonctions d'agrégation : les fonctions distributives telles que `SUM`, `COUNT`, `MIN` ou `MAX`, les fonctions algébriques telles que `AVG` (la moyenne algébrique), `MinN` ou `MaxN` (les N plus petits ou plus grands) et les fonctions holistiques telles que celles donnant la médiane ou le plus fréquent. Nous ne considérons dans cette thèse que des fonctions d'agrégation distributives.

Afin de mener une campagne de publicité ciblée vers les plus mauvais acheteurs, nous devons comparer les ventes de notre table *Clientèle* en fonction du corps de

métier. Dans ce contexte, l'attributs *Métier* de la table est une dimension fixant l'espace de l'analyse et la colonne *Valeur* est la donnée étudiée. La requête SQL à saisir est la suivante :

```
SELECT    Métier, SUM(Valeur) AS Total
FROM      Clientèle
GROUP BY  Métier
```

Nous obtenons pour résultat :

M	$Total$
m_1	40
m_2	70

Pour retourner cette table à deux tuples, nous avons dû parcourir toute la table *Clientèle*. Pour optimiser ces requêtes de type `GROUP BY`, nous pouvons alors envisager de les pré-calculer et de les stocker au sein d'un cube de données.

4.1.2 Construction avec vision cuboïde

Nous avons agrégé ici la dimension *Métier* mais toutes les 2^n combinaisons d'agrégations sont possibles et potentiellement utiles pour l'analyse ce qui implique de considérer toutes les 2^n sous-requêtes d'agrégation. Cette tâche est fastidieuse en pratique et d'autant plus lorsque n est grand. [Gray *et al.* 1997] proposent alors l'opérateur `CUBE BY` pour générer, en une seule requête, toutes les vues possibles sur les données :

```
SELECT    Métier, Produit, Transporteur, SUM(Valeur) AS Total
FROM      Clientèle
CUBE BY   Métier, Produit, Transporteur
```

La figure 4.1 donne le cube de données ainsi obtenu (pour plus de lisibilité, nous avons découpé ce cube en notant sur la droite les niveaux d'agrégation correspondants). Nous le nommons simplement *cube*. Puisque les tuples ainsi créés ne sont pas définis sur tous les attributs de T, nous notons par ALL la valeur générique qui représente l'ensemble des valeurs des domaines.

La taille d'un cube (c.-à-d. son nombre de tuples) est bornée par $\prod_{A_i \in \mathcal{A}}(|Dom(A_i)| + 1)$. En effet, cette taille maximale est atteinte si chaque valeur de chaque attribut est associée, dans la table, à toutes les valeurs de tous les autres attributs et il faut ajouter également au domaine de chaque attribut la valeur générique ALL. Pour notre exemple, nous n'avons que 22 lignes et pourtant, cette représentation sous forme de tableau ne permet pas une analyse rapide dans le sens où toutes les informations sont les unes à la suite des autres. Pour palier ce problème, il faut envisager une autre modélisation.

Un premier partitionnement naturel des tuples est donné par la figure 4.1 : puisqu'un tuple n'agrège qu'un seul sous-ensemble d'attributs, il suffit de regrouper

Métier	*Produit*	*Transp.*	Total	
m_1	p_1	t_1	10	
m_1	p_1	t_2	30	Niveau maximum d'agrégation
m_2	p_1	t_2	20	
m_2	p_2	t_3	50	
m_1	p_1	ALL	40	
m_2	p_1	ALL	20	Agrégation sur *Métier* et *Produit*
m_2	p_2	ALL	50	
m_1	ALL	t_1	10	
m_1	ALL	t_2	30	Agrégation sur *Métier* et *Transporteur*
m_2	ALL	t_2	20	
m_2	ALL	t_3	50	
ALL	p_1	t_1	10	
ALL	p_1	t_2	50	Agrégation sur *Produit* et *Transporteur*
ALL	p_2	t_3	50	
m_1	ALL	ALL	40	Agrégation sur *Métier*
m_2	ALL	ALL	70	
ALL	p_1	ALL	60	Agrégation sur *Produit*
ALL	p_2	ALL	50	
ALL	ALL	t_1	10	
ALL	ALL	t_2	50	Agrégation sur *Transporteur*
ALL	ALL	t_3	50	
ALL	ALL	ALL	110	Aucune agrégation

FIGURE 4.1 – Cube de données de la table *Clientèle*

les tuples correspondant à la même requête d'agrégation. Chaque partie du cube est appelée *cuboïde* ou *vue*, elle est identifiée par le sous-ensemble d'attributs que ses tuples agrègent. Notons $Def(\tau) = X$ l'ensemble d'attributs sur lequel le tuple τ est défini (c.-à-d. $\forall A_i \in X, \tau[A_i] \neq ALL$ où $\tau[A_i]$ est la restriction de τ à l'attribut A_i). L'inclusion entre deux sous-ensembles d'attributs donne un ordre partiel strict sur ceux-ci. Nous choisisssons cet ordre strict mais dans certains cas, nous utilisons l'ordre $\subseteq$ puisque $X \subset Y \Rightarrow X \subseteq Y$ ou encore leurs réciproques $\supset$ et $\supseteq$ étant donné que $X \subset Y \Rightarrow Y \supset X$ ($(2^{\mathcal{A}}, \subset)$ et $(2^{\mathcal{A}}, \subseteq)$ étant des ensembles ordonnés ipsoduaux). Nous pouvons alors dessiner le diagramme de Hasse $D = (P(cube), \subset)$ où $P(cube)$ désigne l'ensemble des parties du cube. Dans la littérature, nous trouvons plus couramment la notion de treillis $P(cube)$ muni de l'ordre $\subset$ puisque ce diagramme admet une borne inférieure, identifiée par $\emptyset$, et une borne supérieure, la table d'origine. Les définitions précises nécessaires à la construction d'un tel diagramme sont largement détaillées dans le premier chapitre du livre de [Caspard *et al.* 2007]. La figure 4.2 décrit le treillis du cube *Clientèle*[1].

Outre le fait que les cuboïdes soient ordonnés partiellement selon $\subset$, cette représentation permet de définir une hiérarchie sur ses nœuds.

1. Comme dans la première partie de cette thèse, les attributs sont ordonnés selon l'ordre alphabétique mais les cuboïdes CM et MC par exemple sont, bien entendu, identiques.

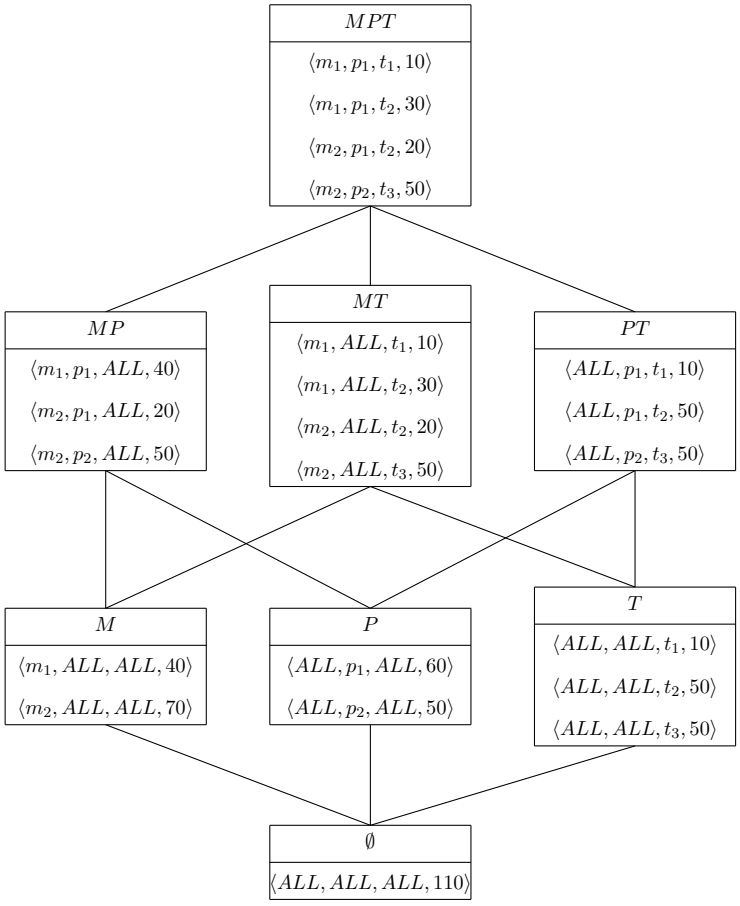


FIGURE 4.2 – Treillis du cube de données *Clientèle*.

Définition 4.1 (Relations entre les cuboïdes) *Soient X et Y deux cuboïdes tels que $Y \subset X$, alors X est un ancêtre de Y et Y est un descendant de X. X est un parent de Y si et seulement si $\nexists Z$ tel que $Y \subset Z \subset X$, réciproquement, Y est un enfant de X (les parents d'un nœud, respectivement ses enfants, sont inclus dans l'ensemble des ancêtres, respectivement des descendants).*
X est alors décrit comme étant plus spécifique que Y ou, de manière équivalente, X spécialise Y et à l'inverse, Y est plus général que X, il le généralise.

Le treillis de la figure 4.3 donne une vision simplifiée du cube de la figure 4.2. Lorsque nous travaillons uniquement sur les cuboïdes du cube sans se préoccuper des tuples qui les composent, cette représentation est alors utilisée.

4.1.3 Construction avec vision tuple

Si maintenant nous souhaitons nous concentrer sur les tuples, il faut dessiner le treillis de manière plus adéquate. Remarquons que, puisqu'un tuple n'est présent

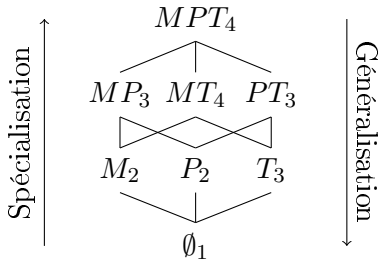


FIGURE 4.3 – Treillis simplifié de la table *Clientèle*.

que dans un seul cuboïde et qu'il existe une relation d'ordre entre les cuboïdes, alors les tuples doivent également être ordonnés :

Définition 4.2 (Relation d'ordre entre les tuples) *Soient τ_1 et τ_2 deux tuples du cube de données. τ_1 est un ancêtre de τ_2 (c.-à-d. il le spécialise) ou, de manière équivalente, τ_2 est un descendant de τ_1 (c.-à-d. il le généralise), noté $\tau_1 \succ \tau_2$ si et seulement si $Const(\tau_1) \supset Const(\tau_2)$ où $Const(\tau_i)$ donne l'ensemble des valeurs (différentes de ALL) de τ_i.*
τ_1 est un parent de τ_2 (τ_2 un enfant de τ_1) si et seulement si $\nexists \tau$ tel que $\tau_1 \succ \tau \succ \tau_2$.

En exploitant directement cette relation d'ordre entre le tuples et donc sans partitionnement préalable en cuboïdes, le treillis peut être visualisé comme sur la figure 4.4 (par soucis de lisibilité, nous avons remplacé la valeur *ALL* par $*$ et la mesure a été omise). L'élément $\langle \emptyset, \emptyset, \emptyset \rangle$ ajouté au sommet du treillis assure que chaque tuple a bien une borne supérieure. Il pourrait être négligé et dans ce cas, nous n'avons qu'un demi-treillis. Cette présentation du treillis de tuples a été donnée par [Casali *et al.* 2003].

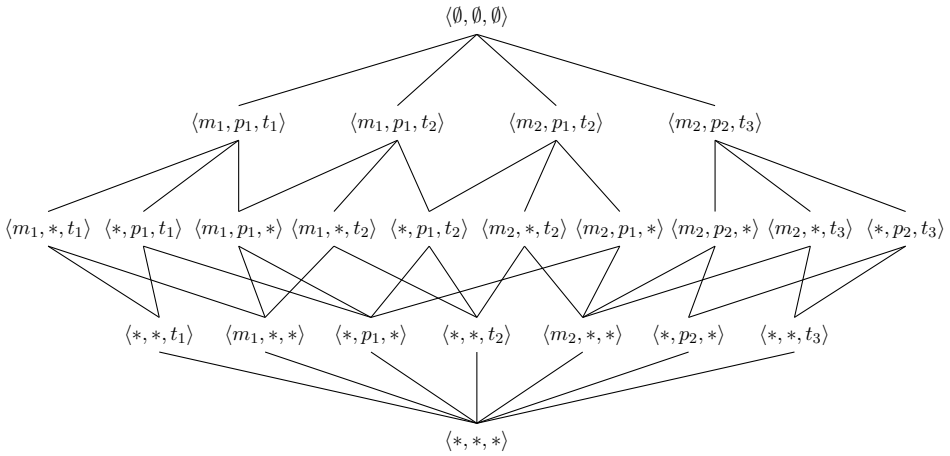


FIGURE 4.4 – Treillis du cube de données *Clientèle*.

La visualisation sous forme d'hypercube est plus répendue et plus efficace pour donner un aperçu général des informations. Les axes de l'hypercube sont

les différentes dimensions de la table et un tuple donne les coordonnées d'une cellule dans laquelle est stockée la valeur mesurée. Cette présentation des données permet de mettre en valeur certaines cellules par rapport à d'autres ou de décrire une masse importante d'information de façon claire et lisible. Tout ceci donne lieu à de nombreux travaux : [Lafon *et al.* 2012], [Messaoud *et al.* 2006] ou [Auber *et al.* 2012] qui aspirent à faciliter l'analyse par regroupement de valeurs ou encore [Lehner *et al.* 1998] ou [Niemi *et al.* 2003] qui cherchent à réduire le nombre de valeurs non définies de l'hypercube.

Pour calculer entièrement un cube de données, nous devons pré-calculer toutes les 2^n requêtes de type `GROUP BY`. Ceci est donc très coûteux en temps et il n'est pas envisageable de garder en mémoire un cube complet. Une sélection des cuboïdes à conserver est donc nécessaire. Ce problème est connu sous le nom de matérialisation partielle des cubes de données. Nous allons l'étudier en détail dans le chapitre 6. Dans la section suivante, nous posons déjà les bases du problème afin de mieux mesurer ses enjeux et difficultés.

4.2 Le problème de la matérialisation partielle

Nous venons de le voir, le cube de données est idéal pour optimiser les requêtes puisqu'elles sont toutes pré-calculées mais encore faut-il que ces pré-calculs soient stockés et il est très difficile, sinon impossible, de tout conserver. Le cuboïde le mieux adapté pour répondre à une requête est naturellement celui qui la traduit exactement mais nous pouvons également y répondre à partir d'un de ses ancêtres puisqu'ils sont plus spécifiques mais donnent au final l'information recherchée. Avant la création du cube de données, toutes les requêtes étaient posées sur la table de base qui représente le cuboïde le plus spécifique de notre cube. C'est pour cela que nous pouvons nous permettre de ne pas matérialiser tous les cuboïdes tout en assurant que chaque requête trouve sa réponse dans l'ensemble stocké.

La difficulté est alors de savoir quels cuboïdes doivent être stockés pour permettre de répondre efficacement (en terme de temps) à toutes les requêtes. Étant donné qu'un cuboïde est une requête pré-calculée, nous parlons indifféremment de cuboïdes, de vues ou de requêtes en fonction de l'usage qui en est fait.

4.2.1 Formalisation du problème

Le problème à résoudre est le suivant : étant donnés une table T, un ensemble de requêtes $\mathcal{Q}$ (pour être le plus général possible, toutes les 2^n requêtes sont autorisées mais, dans la littérature, les auteurs cherchent le plus souvent à n'optimiser qu'un sous-ensemble de requêtes appelé charge de travail ou workload) et un espace mémoire de taille b, trouver l'ensemble des vues S à matérialiser en respectant la contrainte b pour répondre à toutes les requêtes de $\mathcal{Q}$. Bien entendu, la table d'origine doit être stockée puisqu'elle permet de répondre à toutes les requêtes donc $b = |T| + \alpha * (\sum_{q \in \mathcal{Q}} |q|)$ où $|T| = m$ est le nombre de tuples de la table T et $|q|$ est

la taille de la réponse à la requête q (c.-à-d. le nombre de tuples retournés)[2] avec $0 < \alpha < 1$.

4.2.2 Définitions

Pour le moment, les requêtes que nous considèrons consistent simplement à retourner le contenu d'un cuboïde. Lorsque celui-ci est déjà pré-calculé, alors le temps de la réponse est proportionnel à sa taille puisqu'il suffit de le parcourir. Quand par contre il n'est pas stocké, il faut chercher le plus petit (en termes de taille) de ses ancêtres stockés et utiliser ce dernier pour évaluer la requête. Il existe essentiellement deux méthodes pour évaluer les requêtes avec `GROUP BY` ([Graefe 1993, Chaudhuri 1998]), (1) trier la table pour constituer les groupes puis effectuer les opérations d'agrégation pour chaque groupe. Le coût est donc de l'ordre de $m \log m$ avec m la taille de la table ou bien (2) il faut utiliser le hachage ce qui permet d'atteindre un coût linéaire de l'ordre de m. Nous allons supposer l'utilisation du hachage et ainsi, le coût d'une requête est simplement le nombre de lignes parcourues. En supposant qu'il n'y a aucun index sur les cuboïdes, le coût minimal d'une requête est donc le nombre de tuples du cuboïde correspondant. Plus généralement, le modèle de coût que nous utilisons se définit de la façon suivante :

Définition 4.3 (Modèle de coût) *Soit $\mathcal{C}(q, S)$ le coût d'une requête q en sachant que seules les vues de l'ensemble S sont matérialisées. Nous avons $\mathcal{C}(q, S) = min_{w \in A_q \cap S}|w|$ où w est un cuboïde matérialisé appartenant à l'ensemble des ancêtres de q noté A_q.*

Le coût total de $\mathcal{Q}$ relativement à S est donc : $\mathcal{C}(S) = \sum_{q \in \mathcal{Q}} \mathcal{C}(q, S)$. Ainsi, pour tout S, nous avons $\sum_{q \in \mathcal{Q}} |q| \leq \mathcal{C}(S) \leq |\mathcal{Q}| * |T|$.

Nous cherchons alors une solution S (c.-à-d. un ensemble de vues à matérialiser) optimale, comme définit ci-dessous :

Définition 4.4 (Solution optimale) *Une solution $S \subseteq P(cube)$ est optimale si et seulement si :*

1. *nous pouvons répondre à toutes les requêtes de $\mathcal{Q}$ en utilisant seulement les vues de S*
2. *la somme des tailles des éléments de S, notée $|S|$ est inférieure à la borne b*
3. *il n'existe pas de solution S' respectant les deux premières contraintes telle que $\mathcal{C}(S') < \mathcal{C}(S)$.*

Nous avons considéré le problème de la matérialisation partielle avec une vision centrée sur les cuboïdes du cube de données. Nous pouvons examiner de la même façon le problème avec une vision sur les tuples du cube. La solution S est alors un ensemble de tuples. Le coût d'une requête correspond bien au nombre de tuples parcourus pour y répondre et la solution optimale reste inchangée.

2. Nous assimilons volontairement l'occupation mémoire d'une table à son nombre de tuples afin de ne pas alourdir les notations mais, pour être plus précis, il faudrait le multiplier par l'espace réellement occupé par un tuple.

Chapitre 5

Les paramètres de la matérialisation partielle

Sommaire

Nous continuons l'analyse du problème, dans ce second chapitre, à travers un état de l'art. Nous étudions précisément l'algorithme pionnier proposé pour le résoudre puisqu'il a servi de base à d'autres travaux qui ont clairement identifié ses points forts et ses faiblesses.

Nous avons introduit, au chapitre précédent, le problème de la matérialisation partielle des cubes de données avec, pour contrainte, une borne sur l'espace mémoire et, pour objectif, une minimisation des temps de calcul relativement à une solution qui consiste à ne matérialiser que la table de base. Dans ce chapitre, nous détaillons dans un premier temps les méthodes permettant de résoudre ce problème. Il a ensuite été remanié pour considérer, comme contrainte, les temps d'exécution des requêtes avec pour objectif, une minimisation de l'espace mémoire nécessaire au stockage de la solution. Nous étudions alors la résolution de ce nouveau problème. Pour finir, nous donnons un rapide aperçu des autres contraintes et objectifs observés dans la littérature.

5.1 Avec une contrainte de mémoire

Le problème à résoudre est le suivant : soient $\mathcal{Q}$ un ensemble de requêtes posées sur un cube de données et b un espace mémoire de taille fixée. Quel est le sous-ensemble, S, de vues du cube à matérialiser pour répondre à toutes les requêtes de $\mathcal{Q}$ avec un coût minimal en respectant la borne mémoire b ?

5.1.1 Formalisation du problème en programmation linéaire

[Li *et al.* 2005] proposent de trouver exactement la solution optimale à ce problème en utilisant la programmation linéaire. Ils définissent le problème comme suit : Soit $\mathcal{C}(j, \{i\})$ le coût de la requête j sur la vue i, nous avons $\mathcal{C}(j, \{i\}) = +\infty$ si i ne peut pas être utilisée pour répondre à j (c.-à-d. i n'est pas un ancêtre de j), sinon $\mathcal{C}(j, \{i\}) = |i|$ où $|i|$ est la taille de i.
Pour toute vue i, $x_i = 1$ si et seulement si $i \in S$ (c.-à-d. si i est matérialisée), $x_i = 0$ sinon.
Pour toute vue i et pour toute requête j, $y_{i,j} = 1$ si et seulement si i est utilisée pour répondre à j et $y_{i,j} = 0$ sinon.
En programmation linéaire, le problème s'écrit donc ainsi :

minimisons $\sum_i \sum_j \mathcal{C}(j, \{i\}) * y_{i,j}$
en s'assurant que
1. $\sum_i |i| * x_i \leq b$
2. $\forall j \sum_i y_{i,j} = 1$
3. $\forall i, j$ telles que $\mathcal{C}(j, \{i\}) \neq +\infty$, $y_{i,j} \leq x_i$
4. $x_T = 1$

La contrainte 1. garantit le fait que l'espace mémoire utilisé par les vues matérialisées ne dépasse pas la taille b impartie. La contrainte 2. atteste qu'une seule vue doit être utilisée pour répondre à une requête j particulière. Avec la contrainte 3., nous assurons que si une vue i est utilisée pour répondre à une requête j, alors i doit être matérialisée. La dernière contrainte nous impose de matérialiser la table initiale T.

Il est bien connu que le problème de sélection de cuboïdes à matérialiser est NP-Difficile ([Harinarayan *et al.* 1996]) et nous ne pouvons donc pas espérer obtenir une solution optimale dans un temps raisonnable. En effet, le nombre de constantes à considérer est exponentiel en nombre d'attributs.

5.1.2 L'algorithme pionnier

L'algorithme pionnier pour résoudre ce problème est celui proposé par [Harinarayan *et al.* 1996]. Les auteurs proposent un algorithme approché dont le principe est le suivant : la table de base du cube de données est, par défaut, matérialisée, $S = \{T\}$. Ensuite, tant que l'espace mémoire n'est pas saturé, il suffit de matérialiser la meilleure vue relativement à S. Une vue est considérée comme étant la meilleure à un instant donné par rapport au bénéfice qu'elle pourrait apporter en étant matérialisée. Le bénéfice est défini comme suit :

Définition 5.1 (Bénéfice d'une vue) *Soient v une vue du cube et S la sélection déjà matérialisée (S contient au moins T). Le bénéfice de v relativement à S est donné par $B(v, S) = \sum_{w \subseteq v} B_w$ où $B_w = |u| - |v|$ avec $w \subseteq u \in S$ (si $|u| - |v| < 0$, alors $B_w = 0$).*

En d'autres termes, si v permet de réduire les coûts d'évaluation de ses descendants (v compris) par rapport à la solution actuelle, alors son bénéfice est strictement positif et il suffit de sélectionner le cuboïde dont le bénéfice est le plus grand. Pour

simplifier l'explication de l'algorithme, la contrainte mémoire imposée n'est pas exprimée en terme d'occupation en bits ou octets mais en nombre de vues autorisées à être matérialisées quelque soient leurs tailles. L'algorithme 2 permet de sélectionner les k meilleures vues.

Algorithme 2: Sélection des k meilleures vues

```
1 S ← T ;
2 pour chaque i = 1 à k faire
3     S ← S ∪ {v} avec B(v, S) = max_{u∉S}(B(u, S));
4 retourner S
```

Exemple *Sélectionnons seulement 3 cuboïdes (en plus du cuboïde MPT) sur notre exemple pour constituer notre solution. Le tableau suivant donne l'évolution des bénéfices et de la solution aux différentes itérations :*

	Itération 1	*Itération 2*	*Itération 3*
$B(MP,S) =$	$1 \times 4 = 4$	$1 \times 3 = 3$	$1 \times 3 = 3$
$B(MT,S) =$	$0 \times 4 = 0$	$0 \times 3 = 0$	$0 \times 3 = 0$
$B(PT,S) =$	$1 \times 4 = 4$	$1 \times 4 = 4$	
$B(M,S) =$	$2 \times 2 = 4$		
$B(P,S) =$	$2 \times 2 = 4$	$2 \times 1 = 2$	$2 \times 1 = 2$
$B(T,S) =$	$1 \times 2 = 2$	$1 \times 1 = 1$	$0 \times 1 = 0$
$B(\emptyset,S) =$	$3 \times 1 = 3$	$1 \times 1 = 1$	$1 \times 1 = 1$

Arbitrairement, nous avons choisi, à la première itération de sélectionner M, puis, à la seconde, PT qui avait alors le meilleur bénéfice et enfin, MP.

A chaque itération, nous choisissons donc le meilleur candidat mais le caractère définitif de ce choix peut nous faire douter de la qualité de la sélection retournée. Les auteurs définissent alors le gain apporté par une solution par rapport au cas où seule la table d'origine est utilisée pour répondre à toutes les requêtes :

Définition 5.2 (Gain d'une solution) *Le gain apporté par une solution S est donné par $G(S) = |\mathcal{Q}| \times |T| - \mathcal{C}(S)$ où $|\mathcal{Q}| \times |T|$ donne le coût de $\mathcal{Q}$ en utilisant T et $\mathcal{C}(S)$ son coût sur S.*

Exemple *Nous avons $S = \{CMP, M, PT, MP\}$ donc $G(S) = 8 \times 4 - (2 \times 4 + 4 \times 3 + 2 \times 2) = 8$*

Les auteurs montrent que la solution S retournée par l'algorithme approché a un gain d'au moins 63% par rapport à la solution optimale S^* (c.-à-d. $G(S) \div G(S^*) \geq (e-1)/e \approx 63\%$ où e est le nombre d'Euler) mais aussi que cette borne théorique est, en pratique, difficilement atteignable. En effet, ils donnent les cas dans lesquels l'algorithme est moins efficace et la solution trouvée est toutefois très proche de la solution optimale. Dans notre exemple, l'algorithme a

bien retourné une solution optimale puisqu'aucune autre sélection de trois cuboïdes n'a de gain plus grand.

La figure 5.1 (inspirée de l'exemple donné par [Karloff & Mihail 1999]) exhibe les points faibles de la méthode. Si nous souhaitons ne conserver que deux vues (en plus de la vue A), l'algorithme approché commence par conserver la vue C puis, les bénéfices des vues B et D étant identiques, il choisit, par exemple B. Nous avons avec cette sélection, $\mathcal{C}(S) = 1297$. Or, sélectionner les deux vues B et D permet de diminuer davantage la somme des coûts puisque $\mathcal{C}(S^*) = 1000$ mais nous obtenons un gain inférieur.

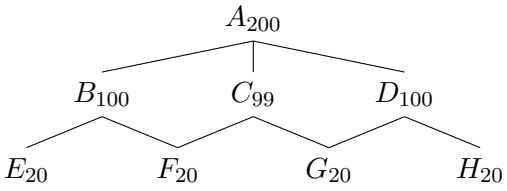


FIGURE 5.1 – Exemple où l'algorithme HRU est inefficace

Que dire alors de cette méthode ? [Karloff & Mihail 1999] montrent que, si $P \neq NP$ alors, la garantie des 63% de gain par rapport à la solution ne contenant que la table de base ne donne aucune réelle garantie sur les coûts des requêtes. La mesure de gain n'est donc pas appropriée pour assurer la bonne performance de chaque requête vis-à-vis d'une solution qui minimise chaque coût. L'optimalité d'une solution doit donc être définie en fonction d'une sélection remplissant au mieux les objectifs et non relativement à la situation de départ où nous n'avons que T pour répondre aux requêtes.

5.1.3 Améliorations et extensions

Plusieurs chercheurs ont alors révisé cette méthode. [Talebi *et al.* 2009] proposent de trouver une solution optimale à ce problème en utilisant la programmation linéaire. L'idée n'est pas nouvelle mais nous avons vu que, en pratique, elle n'était pas envisageable du fait du grand nombre de variables à considérer. Les auteurs proposent alors d'identifier les vues qui n'appartiennent assurément pas à la solution pour réduire au maximum l'espace de recherche :

Définition 5.3 (Espace de recherche réduit) *Soient $P(cube)$ l'ensemble des cuboïdes du cube de données et $P_r(cube) \subseteq P(cube)$ l'espace de recherche réduit. La vue v ne doit pas être matérialisée si elle a une taille supérieure ou égale à la somme des tailles de ses descendants. Plus formellement $v \in P(cube) \setminus P_r(cube) \Rightarrow |v| \geq \sum_{u \subset v} |u|$.*

En effet, si la matérialisation d'une vue ne permet pas de réduire la somme des coûts de toutes les requêtes pouvant s'y ramener, alors, il est préférable de ne pas la stocker au profit de tous ses descendants. Il est à noté cependant que cette règle ne s'applique que lorsque l'espace mémoire est borné en terme d'occupation réelle

et non en nombre de vues. Une fois l'espace de recherche restreint, l'utilisation de méthodes de programmation linéaire peut être légitime.

[Huang *et al.* 2012] proposent une autre manière de définir l'espace de recherche réduit en se basant sur la notion de bénéfice (définition 5.1). Ils montrent que les vues ayant un bénéfice nul ne doivent assurément pas être matérialisées. En effet, sur notre exemple, dès la première itération de l'algorithme 2, la vue MT a un bénéfice nul et, naturellement, il n'augmente pas lorsque la solution en cours S évolue. Il n'est donc pas nécessaire de recalculer le bénéfice de cette vue à chaque itération, ce qui réduit l'espace de recherche.

5.2 Avec garantie de performance

La contrainte est maintenant fixée sur la performance et nous souhaitons réduire au maximum la taille de l'ensemble à matérialiser. Nous devons alors définir formellement cette contrainte :

Définition 5.4 (Facteur de performance ([Hanusse *et al.* 2011])) *Soit q une requête s'exécutant sur le cuboïde c. Alors le facteur de performance de q sur c est donné par $p = \frac{|c|}{|q|} \geq 1$.*
Si $p = 1$ alors l'évaluation de q sur c a un coût optimal.

5.2.1 Définition du problème

Le problème est formulé ainsi : étant donnés une table T et un facteur de performance p, quel est l'ensemble minimal de vues (en terme d'occupation mémoire) à matérialiser en assurant que le coût de chaque requête ne dépasse pas de plus de p fois son coût optimal ?

L'espace de recherche de la solution reste inchangé tout comme le modèle de coût. Seule la solution optimale à ce problème est modifiée :

Définition 5.5 (Solution optimale avec garantie de performance) *Une solution $S \subseteq P(cube)$ est optimale relativement au facteur de performance p si et seulement si*

1. *chaque requête s'exécute sur S avec un facteur de performance $p' \leq p$,*
2. *il n'existe pas de solution plus petite en terme d'occupation mémoire que S qui respecte la première condition.*

Pour répondre au problème, il faut que chaque requête puisse s'exécuter avec un facteur de performance au plus égal à celui fixé préalablement. Une solution naturelle serait alors de matérialiser tout le cube afin de garantir une évaluation optimale de toute les requêtes mais ce n'est pas réalisable en pratique, il faut donc une sélection de taille minimale respectant la première contrainte. Bien entendu, même formulé de cette manière, le problème n'en reste pas moins NP-Difficile.

5.2.2 Les méthodes

[Hanusse *et al.* 2009b] proposent de trouver une solution approchée à ce problème. La solution n'est pas optimale dans le sens où sa taille n'est pas minimale mais la performance des requêtes est assurée. Ils procèdent ainsi :

1. Ils sélectionnent les cuboïdes de $\mathcal{Q}$ qui ne respectent pas la contrainte de performance s'ils sont évalués à partir de T.
2. Ils matérialisent les plus grands relativement à $\subset$.
3. Ils réitèrent cette méthode jusqu'à ce que toutes les requêtes puissent être exécutées sans violer la contrainte p.

Les auteurs montrent que la solution ainsi obtenue est, au plus, p fois plus grande (en terme d'occupation) que la solution optimale.

Exemple *Fixons $p = 1.5$. Les cuboïdes M et P sont matérialisés à la première itération. $\emptyset$ est ensuite matérialisé à la seconde itération. Sur notre exemple, $S = \{MPT, M, P, \emptyset\}$, la solution est donc de taille 9.*

Dans [Hanusse *et al.* 2011], ils étendent leurs travaux en constatant que la sélection des cuboïdes qui ne garantissent pas la performance des requêtes peuvent être assimilés à un espace de recherche restreint suffisant. Ils utilisent donc la programmation linéaire pour trouver exactement la solution optimale.

Nous l'avons évoqué au chapitre précédent, le cube de données peut être vu relativement à ses cuboïdes ou à ses tuples. Le problème de la matérialisation partielle est formulé dans la vision cuboïde du cube mais il peut être adapté à ses tuples. Dans ce contexte, les méthodes de résumé sont également un moyen de garantir les performances des requêtes. En effet, par définition, un résumé donne une forme plus sommaire de toutes les informations sans qu'aucune ne soit perdue. Les techniques de cube condensé, cube quotient et cube clos qui sont, à notre avis, représentatives du domaine sont détaillées dans la section 6.4.

5.3 Autres problèmes de matérialisation partielle

D'autres versions au problème de la matérialisation partielle ont été proposées, faisons alors un rapide tour d'horizon des sujets traités (un état de l'art complet de ces méthodes n'est pas réellement utile pour cette thèse puisque nous n'avons pas traité ces problèmes et c'est pourquoi nous ne citerons que quelques travaux les plus récents ou les plus pertinents pour simplement donner un aperçu de ce qui existe).

Sélection dynamique Nous avons considéré un ensemble $\mathcal{Q}$ de requêtes à optimiser mais cela semble bien utopiste de penser que cet ensemble est fixé une bonne fois pour toute. En effet, il évolue au fur et à mesure des analyses et du temps. Les approches développées par [Kotidis & Roussopoulos 1999] et [Hose *et al.* 2009] proposent alors des solutions révisables pour être au plus près des besoins de l'utilisateur. Un processus de mise à jour de la solution est présenté alors que les travaux, plus statiques, devraient tout recalculer.

Sélection des vues et leurs index Dès lors que l'espace mémoire disponible est occupé par les vues matérialisées, il n'y a plus d'espoir d'optimiser encore davantage les requêtes. C'est le constat qui a été fait par [Talebi *et al.* 2008] et [Kormilitsin *et al.* 2007] notamment et c'est pourquoi ils prévoient le calcul et le stockage des index au même moment qu'ils sélectionnent les vues. Ainsi, ils obtiennent une solution des plus optimisée puisque, sans violer la contrainte d'espace, ils disposent de vues matérialisées et de leurs index.

Contexte distribué Le temps où toutes les données étaient centralisées touche vraisemblablement à sa fin et il faut donc prévoir de nouvelles méthodes pour répondre à ce cas là. [Bauer & Lehner 2003] et [Shukla *et al.* 2000] définissent alors un nouveau modèle de coût en identifiant les nouvelles contraintes et proposent donc une solution au problème.

D'autres travaux concernent la traduction des requêtes lorsque les vues matérialisées sont utilisées. En effet, il faut alors savoir quelles sont les vues les mieux adaptées en terme de réponse mais aussi de temps. Citons l'algorithme développé par [Pottinger & Halevy 2001] qui identifie les variables et objectifs d'une requête pour créer la meilleure combinaison de vues permettant d'y répondre. Plus récemment, [Laurent & Spyratos 2011] utilisent les dépendances fonctionnelles pour répondre le plus efficacement aux requêtes puisqu'elles permettent de facilement identifier la vue à utiliser en spécialisant la requête. Dans le contexte distribué, le paradigme MapReduce détaillé par [Nandi *et al.* 2011] permet de répartir les calculs de requêtes sur les différents sites où sont stocker les informations pour en extraire une réponse global.

Bien d'autres travaux concernant la sélection des vues, leurs utilisations ou leurs maintenances mériteraient d'être cités mais ce chapitre ayant pour vocation à introduire nos contributions, nous nous contenterons de ce bref aperçu. [Halevy 2001] donne un état de l'art plus complet des enjeux autour de la matérialisation partielle. Aussi, pour avoir une idée de la manière dont ces optimisations interviennent dans les SGDBs actuels, [Bruno 2011] propose un état de l'art des différentes implémentations et utilisations.

5.4 Conclusion

Il existe donc plusieurs manières de définir le problème de la matérialisation partielle des cubes de données en fonction de la contrainte et des objectifs à prendre en compte. Dans le prochain chapitre, nous étudions le problème consistant à garantir les performances des requêtes en minimisant l'espace mémoire occupé par la solution.

Nous déduisons de ces précédents travaux un manque évident de précision concernant la nature des éléments à matérialiser. Certains sont sélectionnés mais nous devons en comprendre la raison afin de mieux identifier ce que doit être la solution optimale au problème. Nous allons donc nous attacher à caractériser formellement cette solution. Nous verrons comment la connaissance des dépendances sert à

résoudre le problème de la matérialisation partielle des cubes de données. En effet, elles nous permettent de clairement définir les éléments (cuboïdes ou tuples selon le contexte) à sélectionner.

Nous avons noté qu'un cuboïde est étudié en fonction de ceux qui l'entourent : les ancêtres permettent de garantir la performance et les descendants précisent son bénéfice. Il ne doit pas être choisi relativement à un moment donné de l'exécution de l'algorithme (problème observé sur la figure 5.1) mais plutôt grâce à ses propriétés observées en fonction d'une configuration initiale. De plus, cette analyse ne doit pas être réduite à une comparaison avec la table initiale, tous les cuboïdes peuvent apporter une information les uns par rapport aux autres. Ces enseignements tirés de l'état de l'art guideront nos travaux détaillés au prochain chapitre.

Chapitre 6

Caractérisation de la solution optimale

Sommaire

Nous développons dans ce chapitre une technique basée sur les dépendances fonctionnelles pour caractériser les cuboïdes qui composent la solution optimale au problème de la matérialisation partielle avec garantie de performance. En effet, la dépendance $X \rightarrow A$ exprime le fait que le cuboïde X et son ancêtre $X \cup A$ sont de même taille. Nous en déduisons qu'une requête portant sur X peut être exécutée sur le cuboïde $X \cup A$ avec un facteur de performance (cf. définition 5.4) égal à 1. Cette notion est étendue aux dépendances approximatives pour prendre en compte un facteur de performance supérieur à 1 et également aux dépendances conditionnelles afin de sélectionner les tuples de la solution.

Nous avons considéré différents niveaux de granularité dans l'étude du cube de données. Ceux-ci correspondent à deux types de requêtes à optimiser : celles retournant un cuboïde complet et celles, avec clause `WHERE`, portant sur seulement quelques tuples. Nous allons donc détailler notre solution dans ces deux cas de figure afin de donner une caractérisation des plus précise en fonction des besoins.

Cette caractérisation, donnée par les dépendances fonctionnelles ou les dépendances fonctionnelles conditionnelles, nous assure de l'optimalité de la solution mais elle permet également un calcul efficace de celle-ci. Nous proposons donc quelques algorithmes permettant de trouver une solution optimale et cela dans un temps raisonnable par rapport à la taille du cube de données. Nous donnons des méthodes de calcul pour sélectionner les meilleurs éléments de manière exacte mais aussi approchée puisque, malgré notre caractérisation, le problème a une complexité exponentielle en nombre d'attributs.

Étant donné que nous avons clairement identifié les propriétés des éléments à matérialiser, nous comparons formellement les méthodes de résumé en uniformisant le vocabulaire employé. Ainsi, nous pouvons tirer profit de leurs différents atouts pour améliorer encore le calcul de notre solution.

Nous commençons par le problème de sélection de cuboïdes complets que nous raffinons par la suite pour traiter celui de la sélection de tuples.

6.1 Sélection de cuboïdes complets

Soit p un facteur de performance fixé par l'utilisateur. Les requêtes considérées sont du type `SELECT` X agr(M) `FROM` T `GROUP BY` X avec $X \subseteq \mathcal{A}$ et agr est une fonction d'agrégation sur la mesure M. Quels sont alors les cuboïdes à matérialiser pour que chaque requête de ce type s'exécute avec un coût au plus p fois son coût optimal ?

Nous allons, dans un premier temps, définir les notions nécessaires à la recherche d'une solution, puis nous décomposons le problème : nous étudions d'abord le cas où $p = 1$ et ensuite, l'analyse est étendue au cas $p > 1$.

6.1.1 Définitions

Une requête ne peut être évaluée qu'à partir du cuboïde qui contient exactement sa réponse ou à partir d'un cuboïde plus spécifique (c.-à-d. un de ces ancêtres). Les relations d'ordre que nous avons définies au chapitre 4 dans le treillis sont précisées pour prendre en compte le facteur de performance.

Définition 6.1 (p_Ancêtre) *Le cuboïde c' est un p_ ancêtre du cuboïde c si et seulement si le coût d'évaluation d'une requête posée sur c' ne dépasse pas de plus de p fois le coût de cette même requête sur c : $\frac{|c'|}{|c|} \leq p$.*

Exemple *Sur notre exemple, le cuboïde PT est un 1_ ancêtre de T et un 1.5_ ancêtre de P puisque $|PT| = |T| = 3$ et $|P| = 2$.*

Intuitivement, la solution que nous cherchons permet d'évaluer chaque requête à partir d'un ancêtre matérialisé dont la taille ne dépasse pas p fois la taille du résultat de la requête. Nous cherchons donc une solution p_optimale comme définie ci-dessous :

Définition 6.2 (Solution $p\_$ optimale) *Une solution S est $p\_$ optimale si et seulement si :*

1. *S est $p\_$ correcte : $\forall q \in \mathcal{Q}, \mathcal{C}(q,S) \leq p \times |q|$ (c.-à-d. $q \in S$ ou S contient un $p\_$ ancêtre de q).*
2. *S est de taille minimale parmi toutes les solutions S' $p\_$ correctes.*

La première condition nous assure que la contrainte de performance est bien respectée. La seconde condition précise l'optimalité en terme d'espace mémoire.

Pour calculer le facteur de performance d'une requête sur un de ses ancêtres, la taille de l'ancêtre est divisée par la taille du résultat de la requête. Ce ratio de tailles rappelle la condition de validité des dépendances fonctionnelles. C'est pourquoi elles sont un outil fondamental pour la recherche de la solution. Nous utilisons la mesure de force comme définie au chapitre 1 pour pouvoir envisager le cas $p > 1$:

Définition 6.3 (DF approximative) *Soient T une table définie sur l'ensemble d'attributs $\mathcal{A}$, X et Y deux sous-ensembles de $\mathcal{A}$. T satisfait la dépendance fonctionnelle $X \rightarrow Y$ avec une force $f \leq 1$, noté $T \models X \rightarrow_f Y$, si et seulement si $\frac{|X|}{|XY|} = f$ où $|X|$ est le nombre de tuples (ou la taille) de X et $|XY|$ la taille de XY. La DF $X \rightarrow Y$ est exacte si $f = 1$, sinon, elle est approximative.*

Exemple *$T \rightarrow_1 P$ est la seule dépendance exacte de notre table. Les dépendances approximatives $M \rightarrow_{2/3} P$ et $P \rightarrow_{0.5} T$ entre autres sont également vérifiées.*

Nous pouvons vérifier que pour tout X et tout Y ensembles d'attributs, $T \models X \rightarrow_f Y$ avec $\frac{1}{m} \leq f \leq 1$ où m est le nombre de tuples de T. Aussi, si $T \models X \rightarrow_f Y$ alors le cuboïde portant sur les attributs $X \cup Y$ est un $p_$ancêtre du cuboïde portant sur X avec $p \leq 1/f$. Ce lien entre p et f est prouvé par le lemme suivant :

Lemme 6.1 *Y est un $p\_$ ancêtre de X si et seulement si $T \models X \rightarrow_f Y'$ avec $Y' = Y \setminus X$ et $f \geq \frac{1}{p}$.*

Preuve

$$
\begin{aligned}
Y\ p\_\ \text{ancêtre de}\ X \quad &\Leftrightarrow \quad \frac{|Y|}{|X|} \leq p \ \text{(d'après la définition 6.1)} \\
&\Leftrightarrow \quad \frac{|X|}{|Y|} \geq \frac{1}{p} \\
&\qquad = \frac{|X|}{|XY|} \ \text{puisque}\ X \subset Y\ \text{donc}\ Y = Y \cup X \\
&\Leftrightarrow \quad X \rightarrow_f Y\ \text{avec}\ f = \frac{|X|}{|XY|} \geq \frac{1}{p} \\
&\Leftrightarrow \quad X \rightarrow_f Y'\ \text{car}\ Y = X \cup Y'\ \text{donc}\ X \cup Y = X \cup Y'.
\end{aligned}
$$

□

L'ensemble des dépendances fonctionnelles permet donc d'identifier les $p_$ancêtres de chaque cuboïde.

6.1.2 Solution 1_optimale

Pour qu'une solution soit 1_correcte, il faut que chaque requête soit évaluée avec son coût minimal. Pour autant, tous les cuboïdes ne doivent pas nécessairement appartenir à la solution. De la même façon que nous avons qualifié les p_ancêtres d'un cuboïdes, les dépendances fonctionnelles servent à préciser la nature des cuboïdes à matérialiser :

Définition 6.4 (Cuboïde clos) *X est clos si et seulement si $\forall Y, X \subset Y \Rightarrow |X| < |Y|$. Autrement dit, tous les ancêtres de X ont une taille strictement supérieure à celle de X, c.-à-d. X n'a pas de 1_ ancêtre.*

Exemple *Dans notre exemple, $|MT| = |MPT|$ et $|T| = |PT|$. Donc les cuboïdes MT et T ne sont pas clos alors que tous les autres le sont.*

Naturellement, tous les cuboïdes clos, ceux qui n'ont pas de 1_ancêtre, doivent être matérialisés puisqu'ils ne peuvent pas être évalués à partir d'un ancêtre sans violer la contrainte de performance. Cela est formalisé par la proposition suivante :

Proposition 6.1 *Soient F_1 l'ensemble des cuboïdes clos et S la solution 1_ optimale au problème de sélection de vues. Alors $S = F_1$.*

Preuve *Supposons qu'il existe un cuboïde $c \in S$ non clos. c est non clos donc il existe un ancêtre c' de c tel que $|c| = |c'|$. Le coût d'évaluation d'une requête sur c est donc le même que le coût d'évaluation de la même requête sur c'. Il est donc préférable de stocker uniquement c' et non c puisque c' permet de répondre à un plus grand nombre de requêtes. Procédons de la même façon sur c' s'il n'est pas clos non plus et ainsi de suite. c n'appartient donc pas à S et tous les cuboïdes de S sont des cuboïdes clos.*
Supposons maintenant qu'il existe un cuboïde c clos qui n'appartient pas à S. Pour évaluer une requête q posée sur c, il va donc falloir utiliser un ancêtre c' de c appartenant à S. Or, étant donné que c est clos, nous savons que $\forall c'$ ancêtre de c, $|c| < |c'|$ donc $\mathcal{C}(q, \{c'\}) > \mathcal{C}(q, \{c\})$. Nous en déduisons naturellement que S n'est pas la solution 1_ correcte si $c \notin S$.
Donc S contient tous les cuboïdes clos et uniquement ceux là. □

Exemple *Nous avons encadré sur la figure 6.1 les cuboïdes de la solution 1_ optimale. Tout en garantissant un coût minimal d'évaluation de chaque requête, nous avons une sélection matérialisée de taille 15. Nous avons donc réduit de plus d'un tier la taille du cube sans perdre en performance.*

Voyons alors comment les dépendances fonctionnelles peuvent nous aider dans la recherche des cuboïdes clos. Le lemme suivant montre qu'un cuboïde clos se définit par sa fermeture (définition 1.9) :

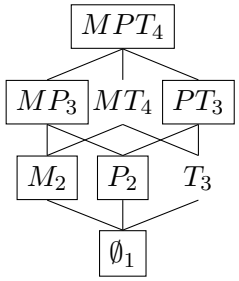


FIGURE 6.1 – Solution 1_ optimale.

Lemme 6.2 *X est clos si et seulement si $X^+ = X$.*

Preuve *$\Rightarrow$: si X est clos alors, par définition, $\forall Y$ tel que $X \subset Y$ on a $|X| < |Y|$ (X n'a pas de 1_ ancêtre). Ainsi, $\frac{|X|}{|Y|} < 1$. D'où, $X \not\rightarrow_1 Y$ et donc $X^+ = X$.*
$\Leftarrow$: Si $X^+ = X$ alors $\nexists A_i$ tel que $\mathcal{F} \models X \rightarrow_1 A_i$. Ainsi, $\forall A_i$, $\frac{|X|}{|XA_i|} < 1$ et donc, $|X| < |XA_i|$. D'où X qui est clos par définition. □

Exemple *Ceci confirme donc que T n'est pas clos puisque nous savons que $Client\grave{e}le \models T \rightarrow_1 P$ et effectivement, nous avons vu que $|T| = |PT|$.*

Le théorème suivant présente la complexité de cette caractérisation :

Théorème 6.1 *Le problème de la sélection d'une solution 1_ optimale est NP-Difficile.*

Preuve *La proposition 6.1 montre que la solution 1_ optimale correspond exactement à l'ensemble des cuboïdes clos. Or, d'après le lemme 6.2, identifier les cuboïdes clos revient à extraire les dépendances fonctionnelles. Nous l'avons vu au chapitre 2, ce problème est NP-Difficile.* □

La proposition suivante montre que la taille de la solution diminue lorsque p augmente. De cette façon, si l'espace mémoire est borné, comme c'est la cas dans la plupart des travaux antérieurs, nous pouvons augmenter progressivement la valeur de p afin de trouver une solution de taille raisonnable qui a le meilleur facteur de performance possible relativement à celle-ci.

Proposition 6.2 *Soient S_p une solution p_ optimale et $S_{p'}$ une solution p'_ optimale avec $p' < p$. Alors $|S_p| \leq |S_{p'}|$.*

Preuve *Soient $Q^p_{S_p}$ et $Q^{p'}_{S_{p'}}$ l'ensemble des requêtes auxquelles il est possible de répondre à partir de S_p sans violer la contrainte de performance p et à partir de $S_{p'}$ avec une performance p' respectivement. Puisque $S_{p'}$ et S_p sont des solutions optimale, alors $Q^{p'}_{S_{p'}}$ et $Q^p_{S_p}$ permettent de répondre à toutes les requêtes sur le*

cube. S_p étant une solution p_ optimale, elle ne permet pas de répondre à toutes les requêtes lorsque l'exigence sur p est plus stricte. Donc $Q^{p'}_{S_p} \subseteq Q^{p}_{S_p}$ mais, à l'inverse, $Q^{p'}_{S_{p'}} = P(cube) = Q^{p}_{S_{p'}}$ et donc $S_{p'}$ est une solution p_ correcte. Étant donné que S_p est une solution p_ optimale, il n'existe pas de solution p_ correcte plus petite. Ainsi, nous obtenons bien $|S_p| \leq |S_{p'}|$. □

Dans la section suivante, nous développons les outils définis jusque là pour caractériser une solution p_optimale.

6.1.3 Solution p_optimale

Pour réduire la taille de la sélection, le seul moyen est de relaxer la contrainte de performance. Grâce aux dépendances fonctionnelles approximatives définies à l'aide de la mesure de force (définition 6.3), nous pouvons étendre les notions de cuboïdes clos et de fermetures comme suit :

Définition 6.5 (Cuboïde p_clos) *Le cuboïde X est p_ clos si et seulement si $\forall Y \supset X$, $p \times |X| < |Y|$ c.-à-d. X n'a pas de p'_ ancêtre avec $p' \leq p$.*

Définition 6.6 (p_Fermeture) *Soit X un ensemble d'attributs. La p_ fermeture de X, notée X^+_p, est l'ensemble des attributs A_i tels que $X \rightarrow_f A_i$ et $f \geq \frac{1}{p}$. X est p_ clos si et seulement si $X^+_p = X$.*

Exemple *Fixons dans nos exemples $p = 3/2$. Nous avons entre autres $M^+_{3/2} = MP$, $P^+_{3/2} = MP, PT$ et $T^+_{3/2} = MP^+_{3/2} = MT^+_{3/2} = PT^+_{3/2} = MPT$. Le cuboïde $\emptyset$ est 3/2_ clos.*

Pour garantir la p_correction de la solution, tous les cuboïdes p_clos doivent être matérialisés comme le prouve la proposition suivante :

Proposition 6.3 *Soit S une solution p_ optimale, alors $F_p \subseteq S$ où F_p est l'ensemble des cuboïdes p_ clos.*

Preuve *Par définition, un cuboïde p_ clos $c \in F_p$ n'a pas d'ancêtre dont la taille est moins de p fois supérieure à la sienne. Pour ne pas violer la contrainte de performance, il est donc indispensable de matérialiser tous les cuboïdes p_ clos.* □

Exemple *$F_{3/2} = \{\emptyset, MPT\}$ est l'ensemble des cuboïdes qui doivent nécessairement être matérialisés mais nous constatons rapidement que ce n'est pas suffisant. En effet, le plus petit ancêtre matérialisé de M est MPT mais l'évaluation d'une requête portant sur M aurait alors un coût 2 fois plus élevé que le coût minimal. La contrainte de performance est donc violer si seuls les cuboïdes p_ clos sont matérialisés.*

La proposition suivante montre les liens qui existent entre les ensembles d'attributs $p_$clos.

Proposition 6.4 *$\forall p, p'$ tels que $p' \leq p$, $F_p \subseteq F_{p'}$ où F_p (resp. $F_{p'}$) est l'ensemble des cuboïdes $p\_$ clos (resp. $p'\_$ clos).*

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

En effet, il est facile de constater que si un cuboïde n'a pas d'ancêtre moins de 4 fois plus grand que lui par exemple, alors il n'a pas non plus d'ancêtre moins de 2 fois plus grand que lui. Alors, puisqu'un cuboïdes $p_$clos est aussi un cuboïde $1_$clos, nous pouvons trouver une solution dans cet ensemble :

Proposition 6.5 *Soit S une solution $p\_$ optimale. Alors il existe une solution S' $p\_$ optimale telle que $S' \subseteq F_1$.*

Preuve *Soient c un cuboïde de S tel que $c \notin F_1$ et c' sa fermeture. Par définition, c' est un $1\_$ ancêtre de c avec $c' \in F_1$ et donc $|c| = |c'|$. Soit Q_c^p l'ensemble des requêtes auxquelles il est possible de répondre à partir de c sans violer la contrainte de performance p (c.-à-d. c est un $p\_$ ancêtre de tous les cuboïdes de Q_c^p). Nous avons $q \in Q_c^p \Rightarrow q \subseteq c \wedge |q| * p \geq |c|$ et donc naturellement $q \subseteq c' \wedge |q| * p \geq |c'|$ ce qui implique $q \in Q_{c'}^p$ et plus généralement $Q_c^p \subset Q_{c'}^p$. Puisque S est $p\_$ optimale, alors $c' \notin S$ puisque, $S \backslash \{c\}$ est une solution $p\_$ correcte plus petite. Soit $S' = S \backslash \{c\} \cup \{c'\}$. Alors $\forall q, \mathcal{C}(q, S) = \mathcal{C}(q, S')$ et $|S| = |S'|$. En remplaçant ainsi tous les cuboïdes non $1\_$ clos de S par leurs $1\_$ fermetures, nous obtenons donc une solution $p\_$ optimale contenant uniquement des cuboïdes $1\_$ clos.* □

Nous avons vu dans le chapitre précédent que, la première chose à faire pour obtenir une solution optimale, est de réduire l'espace de recherche de la solution. Pour trouver une solution $p_$optimale, il peut donc être réduit à l'ensemble des cuboïdes $1_$clos et la proposition 6.4 nous assure de la cohérence de la double inclusion $F_p \subseteq S \subseteq F_1$. Ainsi l'utilisation de la programmation linéaire peut être envisagée pour trouver une solution $p_$optimale.

Exemple *Pour obtenir une solution $3/2\_$ correcte, il faut choisir entre les ensembles minimaux $\{MP\}$, $\{M, PT\}$ et $\{M, P\}$ puisque tous ces cuboïdes sont $1\_$ clos et permettent de répondre à toutes les requêtes en respectant le facteur de performance fixé. $\{MP\}$ ajouté à $F_{3/2}$ donne une solution de taille 8, la taille minimale pour respecter la contrainte de performance.*

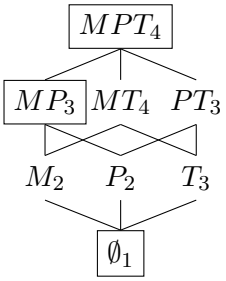


FIGURE 6.2 – Solution 3/2_optimale.

Avec la méthode proposée par [Hanusse et al. 2009a], nous avons trouvé $S = \{MPT, M, P, \emptyset\}$. *Les auteurs avaient bien identifié que les requêtes portant sur les cuboïdes* M *et* P *ne pouvaient pas être évaluées à partir* MPT *mais ils ont choisi alors de les matérialiser sans considérer leur fermeture commune* MP. *Nous constatons donc par cet exemple l'intérêt d'utiliser les dépendances fonctionnelles pour assurer la minimalité en terme de taille de la solution.*

Une solution p_optimale contient donc tous les cuboïdes p_clos et d'autres à choisir parmi les 1_clos. Une question reste alors ouverte : comment caractériser les cuboïdes 1_clos d'une solution p_optimale ?

6.2 Sélection de tuples

Dans la section précédente, nous avons choisi de matérialiser ou non un cuboïde dans son intégralité, c.-à-d. sans tenir compte des tuples qui les composent. Nous allons affiner notre analyse à l'étude des tuples en examinant cette fois les dépendances fonctionnelles conditionnelles. Considérons maintenant des requêtes contenant une clause `WHERE`. Leurs réponses est donc le sous-ensemble d'un cuboïde et la performance de la requête doit alors être réévaluée.

Exemple *Avec* $p = 3/2$, *sur notre cube partiellement matérialisé de la figure 6.2, intéressons nous au corps de métier* m_2 *par la requête suivante :*

```
SELECT    Métier, SUM(Valeur) AS Total
FROM      Clientèle
WHERE     Métier = m2
GROUP BY  Métier
```

Le résultat de cette requête correspond au tuple $\langle m_2, ALL, ALL, 70\rangle$ *de la figure 4.2, soit un coût optimal de 1. A partir du cuboïde matérialisé* MP, *il faut lire les tuples* $\langle m_2, p_1, ALL, 20\rangle$ *et* $\langle m_2, p_2, ALL, 50\rangle$ *pour sommer les valeurs* 20 *et* 50 *et ainsi obtenir le résultat. Cette requête a donc un coût de 2 sur notre sélection matérialisée et la contrainte de performance n'est pas respectée.*

L'étude des tuples à matérialiser est donc une réelle nécessité pour garantir le respect de la contrainte de performance pour répondre à des requêtes comportant

une clause `WHERE`. Nous allons dans un premier temps adapter les définitions détaillées dans le contexte des cuboïdes pour mieux appréhender les enjeux et difficultés liés à l'étude des tuples. Nous pourrons ensuite caractériser les solutions optimales.

6.2.1 Définitions

La solution au problème de la matérialisation partielle en se concentrant sur l'analyse des tuples se définit comme la précédente : trouver un sous-ensemble de tuples du cube de données tel que, quelque soit la requête q, son coût ne dépasse pas de plus de p fois le coût minimale (c.-à-d. le nombre de tuples de la réponse) et ce sous-ensemble contient le minimum de tuples pour respecter la première contrainte.

Nous faisons la distinction entre une solution $p_$optimale contenant uniquement des cuboïdes complets et une autre pour laquelle des tuples sont sélectionnés. La première est notée S alors que la seconde est notée $\mathcal{S}$.

Nous l'avons vu dans le contexte des cuboïdes, pour pouvoir donner une caractérisation à la solution, il faut étudier les ancêtres. $\langle m_2, p_1, ALL, 20\rangle$ est un ancêtre de $\langle m_2, ALL, ALL, 70\rangle$ puisque nous avons bien une inclusion des domaines de définition $Def(\langle m_2, ALL, ALL, 70\rangle) = M \subset MP = Def(\langle m_2, p_1, ALL, 20\rangle)$ mais ici, cet ancêtre seul n'est pas suffisant pour retrouver le résultat de la requête. Nous avons donc besoin de définir un ensemble d'ancêtres pour préciser la performance de la requête. Nous commençons par regrouper les tuples ancêtres qui appartiennent à un même cuboïde :

Définition 6.7 ($p_X_$Ancêtre) *Soient τ un tuple et $\mathcal{T}$ l'ensemble de ses ancêtres appartenant à un cuboïde $X \supset Def(\tau)$. $\mathcal{T}$ est un $p_X_$ancêtre de τ si et seulement si $|\mathcal{T}| \leq p$.*

Exemple *$\{\langle m_2, p_1, ALL, 20\rangle, \langle m_2, p_2, ALL, 50\rangle\}$ est un $2_MP_$ancêtre de $\langle m_2, ALL, ALL, 70\rangle$.*

Pour évaluer une requête q sans clause `WHERE`, il suffit de sélectionner le plus petit ancêtre de q qui soit matérialisé ou q lui-même s'il l'est et de lire complètement ce cuboïde. Lorsque certaines conditions sont données pour q, il faut alors choisir le plus petit $X_$ancêtre, $\mathcal{T}$, mais cela suppose que tous les tuples de $\mathcal{T}$ sont matérialisés pour obtenir le résultat global. Cependant, il serait restrictif de considérer un tuple par rapport aux ensembles auxquels il appartient pour choisir de le matérialiser ou non. De plus, nous constatons qu'il n'est pas nécessaire que tous les tuples d'un ensemble d'ancêtres $\mathcal{T}'$ soient dans le même cuboïde pour répondre exactement à la requête. L'ensemble $\{\langle m_2, p_1, ALL, 20\rangle, \langle m_2, ALL, t_3, 50\rangle\}$, par exemple, donne le même résultat que le $MP_$ancêtre considéré précédemment. Nous parlons alors plus généralement d'un ensemble ancêtre pour désigner un ensemble de tuples répondant complètement à une requête. Cet ensemble est caractérisé par sa taille comme formellement défini ci-dessous :

Définition 6.8 (p_Ensemble_ancêtre) *Soient τ un tuple et $\mathcal{T}$ un ensemble de tuples tel que $\forall \tau' \in \mathcal{T}, \tau \prec \tau'$. $\mathcal{T}$ est un ensemble_ancêtre de τ si et seulement si l'union des T_ancêtres de chaque $\tau' \in \mathcal{T}$ correspond exactement aux T_ancêtres de τ où T est la table de base du cube de données. En d'autres termes, τ peut être correctement calculé à partir de $\mathcal{T}$ si et seulement si τ et les tuples de $\mathcal{T}$ sont des descendants du même ensemble de tuples de la table de base.*
Si $|\mathcal{T}| \leq p$, alors $\mathcal{T}$ est un p_ensemble_ancêtre de τ.

Naturellement, chaque X_ancêtre est un ensemble_ancêtre. Nous n'étudions que des ensembles_ancêtres minimaux, c.-à-d. $\forall \tau' \in \mathcal{T}, \mathcal{T} \setminus \{\tau'\}$ n'est pas un ensemble_ancêtre.

Exemple *$\{\langle m_2, p_1, t_2, 20\rangle, \langle m_2, p_2, t_3, 50\rangle\}$ est le MPT_ancêtre de $\tau = \langle m_2, ALL, ALL, 70\rangle$. Prenons $\tau_1 = \langle m_2, p_1, ALL, 20\rangle \succ \tau$ avec $\{\langle m_2, p_1, t_2, 20\rangle\}$ son MPT_ancêtre et $\tau_2 = \langle m_2, ALL, t_3, 50\rangle \succ \tau$ avec $\{\langle m_2, p_2, t_3, 50\rangle\}$ son MPT_ancêtre. Donc $\{\tau_1, \tau_2\}$ est bien un 2_ensemble_ancêtre de τ.*

La définition de p_ensemble_ancêtre relative à un seul tuple peut être étendue à un ensemble de tuples : l'ensemble_ancêtre de q, une requête retournant plusieurs tuples, est l'union des ensembles_ancêtres des tuples de q. Bien entendu, pour chaque $\tau_i \in q$, nous utilisons un seul, le plus petit, ensemble_ancêtre $\mathcal{T}_i$. Donc $\mathcal{T}_q = \bigcup_{\tau_i \in q} \mathcal{T}_i$ et $\mathcal{T}_q$ est un p_ensemble_ancêtre de q si et seulement si $\frac{|\mathcal{T}_q|}{|q|} \leq p$. Comme précédemment, le coût d'une requête correspond au nombre minimum de tuples matérialisés qu'il faut lire pour y répondre entièrement.

Nous avons utilisé les dépendances fonctionnelles pour caractériser les cuboïdes à matérialiser. Dans le cas du choix des tuples à matérialiser, nous nous servons des dépendances conditionnelles définies au chapitre 1. Nous définissons ces DFCs approximatives comme suit :

Définition 6.9 (DFC approximative) *Soient T une table définie sur l'ensemble d'attributs $\mathcal{A}$, X et Y deux sous-ensembles de $\mathcal{A}$ et τ un tuple défini sur X. T satisfait la dépendance fonctionnelle conditionnelle $X_\tau \rightarrow Y$ [1] avec une force f, noté $T \models X_\tau \rightarrow_f Y$, si et seulement si $\frac{1}{|\mathcal{T}|} = f$ où $\mathcal{T}$ est le XY_ancêtre de τ.*
Si $f = 1$ alors la DFC est exacte, sinon, elle est approximative.

Exemple *Sur notre table Clientèle, nous avons, entre autres, les DFCs exactes suivantes (comme sur la figure 4.4, pour réduire les notations des tuples, la valeur ALL est remplacée par $*$ et nous avons omis la mesure) :*

$$\begin{aligned} M_{\langle m_1,*,*\rangle} &\rightarrow_1 P \\ P_{\langle *,p_2,*\rangle} &\rightarrow_1 MT \\ T_{\langle *,*,t_1\rangle} &\rightarrow_1 MP \\ T_{\langle *,*,t_2\rangle} &\rightarrow_1 P \\ T_{\langle *,*,t_3\rangle} &\rightarrow_1 MP \end{aligned}$$

1. Nous n'utilisons pas la notation décrite dans la première partie car celle-ci nous parait plus parlante pour l'usage qui en sera fait.

Malgré un contexte légèrement différent, l'intuition que les mêmes outils peuvent être utilisés dans la recherche d'une sélection de cuboïdes ou de tuples est palpable. Définissons alors la $p_$fermeture d'un tuple :

Définition 6.10 ($p_$Fermeture d'un tuple) *La $p_$fermeture du tuple τ, notée τ_p^+, est l'ensemble de ses $p_$ensembles$_$ ancêtres.*

Il est à noter que nous considérons seulement les DFCs constantes. Avec $\mathcal{F}_f^c$ l'ensemble des DFCs valides sur T ayant une force supérieure ou égal à f, si $X_\tau \rightarrow_f Y \in \mathcal{F}_f^c$ alors τ a un $p_XY_$ancêtre avec $p = 1/f$. Ces DFCs permettent donc de facilement préciser les performances des $X_$ancêtres de chaque tuple et de calculer sa $1_$fermeture.

Comme dans le contexte des cuboïdes, la $p_$fermeture d'un tuple permet de le caractériser :

Définition 6.11 (Tuple $p_$clos) *Un tuple τ est $p_$clos si et seulement si $\nexists \mathcal{T}$ tel que $\mathcal{T}$ est un $p'_$ensemble$_$ ancêtre de τ avec $p' \leq p$ c.-à-d. $\tau_p^+ = \{\tau\}$.*

Exemple *Soit F_1^t l'ensemble des tuples $1_$clos. Pour notre table Clientèle, sans compter les tuples de la table MPT qui sont naturellement $1_$clos puisqu'ils n'ont aucun ancêtre, nous avons $F_1^t = \{\langle ALL, ALL, ALL, 110\rangle, \langle m_2, ALL, ALL, 70\rangle, \langle ALL, p_1, ALL, 60\rangle, \langle m_1, p_1, ALL, 40\rangle, \langle ALL, p_1, t_2, 50\rangle\}$.*

Maintenant que les bases sont posées, comme précédemment, nous commençons par étudier la solution $1_$optimale afin d'en donner une caractérisation puis, nous étendrons ces résultats à la recherche d'une solution $p_$optimale.

6.2.2 Solution $1_$optimale

Sans surprise, étant donnée l'analyse qui a été faite dans le contexte de la matérialisation partielle de cuboïdes, la solution $1_$optimale est unique et elle correspond exactement à l'ensemble des tuples $1_$clos, F_1^t.

Proposition 6.6 *Soient F_1^t l'ensemble des tuples $1_$clos et $\mathcal{S}$ la solution $1_$optimale. Nous avons $\mathcal{S} = F_1^t$.*

Preuve *Montrons dans un premier temps l'inclusion $F_1^t \subseteq \mathcal{S}$. Soit $\tau \in F_1^t$ et $\tau \notin \mathcal{S}$. Par définition, puisque τ est $1_$clos, il n'a pas de $1_$ensemble$_$ ancêtre. Nous ne pouvons donc pas évaluer une requête contenant τ à partir de $\mathcal{S}$ sans violer la contrainte de performance. Ainsi, $\mathcal{S}$ n'est pas $1_$correcte si elle ne contient pas tous les tuples de F_1^t.*
Prouvons maintenant l'inclusion inverse $\mathcal{S} \subseteq F_1^t$. Soit $\tau \in \mathcal{S}$ et $\tau \notin F_1^t$. Puisque τ n'est pas $1_$clos, alors il a au moins un $1_$ensemble$_$ ancêtre $\mathcal{T} = \{\tau'\}$. Si $\tau' \in \mathcal{S}$ alors cette solution n'est pas optimale étant donné que $\mathcal{S} \setminus \{\tau\}$ est $1_$correcte et a une taille plus petite. Supposons alors que τ' n'est pas matérialisé. Alors, pour

garantir la 1_ *correction de* $\mathcal{S}$, τ' *doit avoir un* 1_ *ensemble_ ancêtre* $\mathcal{T}' = \{\tau''\}$ *qui est donc également un* 1_ *ensemble_ ancêtre de* τ. *Là encore,* $\mathcal{S}$ *n'est pas de taille minimale si elle contient un tuple non* 1_ *clos.* □

Exemple *La sélection optimale de tuples de notre table Clientèle est donnée par la figure 6.3, ce sont les tuples en gras. Nous avons également encadré en gras les cuboïdes complets de la solution* 1_ *optimale que nous avions choisie de matérialiser précédemment.*

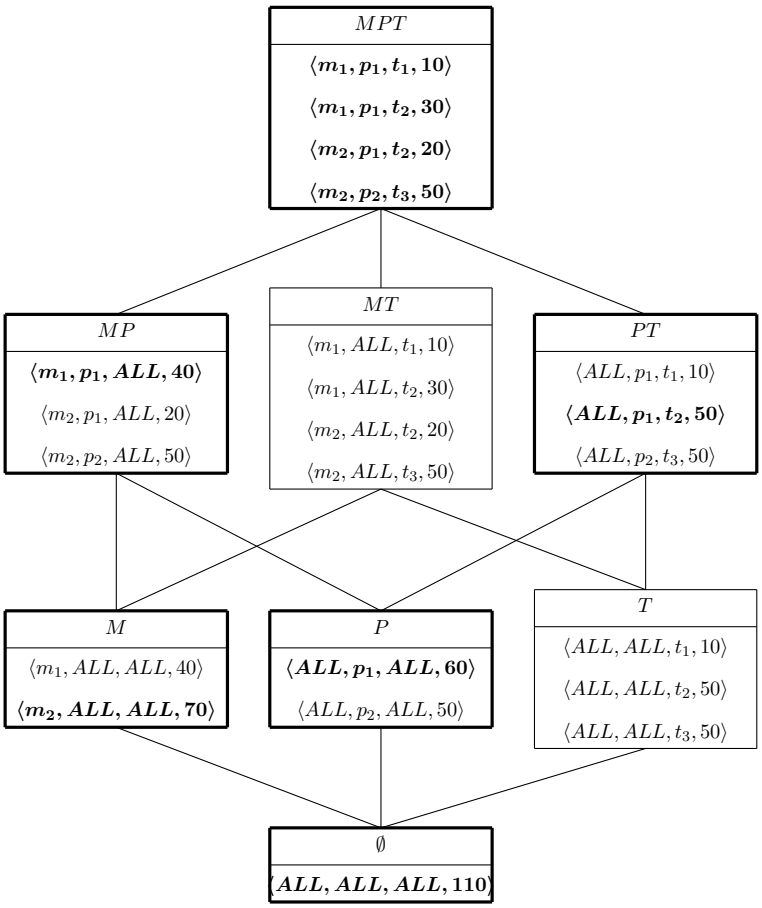


FIGURE 6.3 – Solutions 1_optimales.

Sur l'exemple précédent, les tuples 1_clos appartiennent tous aux cuboïdes 1_clos. La proposition suivante exprime formellement l'étroite relation qui existe entre la solution 1_optimale S contenant des cuboïdes complets et la solution 1_optimale $\mathcal{S}$ pour laquelle nous avons sélectionné les tuples.

Proposition 6.7 *Soient* $\mathcal{S}$ *la solution* 1_ *optimale trouvée en sélectionnant des tuples et* S *les tuples de la solution* 1_ *optimale trouvée en sélectionnant des cuboïdes complets. Alors* $\mathcal{S} \subseteq S$.

Preuve *Soit $\tau \notin S$ un tuple du cuboïde c. D'après la proposition 6.1, $S = F_1$, donc c n'est pas 1_clos et, par définition, $\exists c' \supset c$ tel que $|c| = |c'|$. Donc le c'_ancêtre de τ ne contient qu'un seul tuple et donc, par définition, τ n'est pas 1_clos et en conséquence, $\tau \notin \mathcal{S}$.* □

Notons alors que $\mathcal{S}$, en plus de réduire la taille de la solution consistant à matérialiser les cuboïdes complets, permet de répondre aux requêtes avec clause `WHERE` ce qui n'était pas le cas de la précédente. Il est donc préférable, en terme d'espace mémoire, de sélectionner les tuples et non les cuboïdes lorsque p est fixé à 1 et cela quelque soit le type de requête à traiter.

En revanche, nous avons vu que la solution 3/2_optimale composée de cuboïdes complets n'est pas une solution 3/2_correcte lorsque nous considérons les requêtes avec clause `WHERE`. Nous devons alors chercher à caractériser précisément cette solution 3/2_optimale.

6.2.3 Solution p_optimale

Naturellement, pour avoir une solution p_correcte, nous devons matérialiser tous les tuples qui n'ont pas de p_ensemble_ancêtre, c.-à-d. les tuples p_clos.

Proposition 6.8 *Soit $\mathcal{S}$ une solution p_ optimale, alors $F_p^t \subseteq \mathcal{S}$ avec F_p^t l'ensemble des tuples p_ clos.*

Preuve *Soit τ un tuple p_ clos ($\tau \in F_p^t$). τ est nécessairement matérialisé pour assurer la p_ correction de $\mathcal{S}$ puisqu'il n'a pas de p_ ensemble_ ancêtre.* □

Exemple *Dans le contexte des tuples, il est absurde de rechercher les tuples p_ clos si p n'est pas un entier naturel. En effet, à un tuple, ne peut être associé que des ensembles_ ancêtres ayant une taille $t \in \mathbb{N}$. Notre exemple étant trop minimaliste, seuls les tuples de la table Clientèle sont p_ clos avec $p \geq 2$.*

Bien entendu, la proposition d'inclusion des ensembles clos donnée dans le contexte des cuboïdes (proposition 6.4) est vraie dans l'analyse des tuples clos, nous avons bien $F_{p'}^t \subseteq F_p^t$ si $p' > p$. Nous aboutissons donc à la même conclusion que dans le contexte de la matérialisation de cuboïdes complets, à savoir : il existe une solution p_optimale composée uniquement de tuples 1_clos :

Lemme 6.3 *Soient $\mathcal{S}$ une solution p_ optimale et $\tau \in \mathcal{S}$ avec $\tau \notin F_1^t$. Alors $\mathcal{S}' = \mathcal{S} \setminus \{\tau\} \cup \tau_1^+$ est une solution p_ optimale.*

Preuve *Soient $\tau \in \mathcal{S}$ un tuple tel que $\tau \notin F_1^t$ et $\tau' \in F_1^t$ un élément de sa 1_fermeture. Puisque $\mathcal{S}$ est p_ optimale, $\tau' \notin \mathcal{S}$. Donc si τ' est matérialisé à la place de τ dans $\mathcal{S}'$, alors naturellement $|\mathcal{S}'| = |\mathcal{S}|$.*
Soit q une requête qui utilise τ, nous avons $\mathcal{C}(q, \mathcal{S}') = \mathcal{C}(q, \mathcal{S})$. Donc, puisque la

performance des requêtes est garantie et que nous assurons que la taille n'a pas ainsi augmenté, $\mathcal{S}' \subseteq F_1^t$ est bien une solution p_ optimale. □

Exemple *Nous avons vu que notre table Clientèle ne contient aucun tuple* 2_ *clos autre que ceux de la table de base. Pour autant, leur matérialisation ne suffit pas à donner une solution* 2_ *correcte. La figure 6.4 montre les solutions* 2_ *optimales avec, en gras, les tuples matérialisés et nous avons encadré d'un trait plus épais la solution* 2_ *optimale calculée sur les cuboïdes complets.*

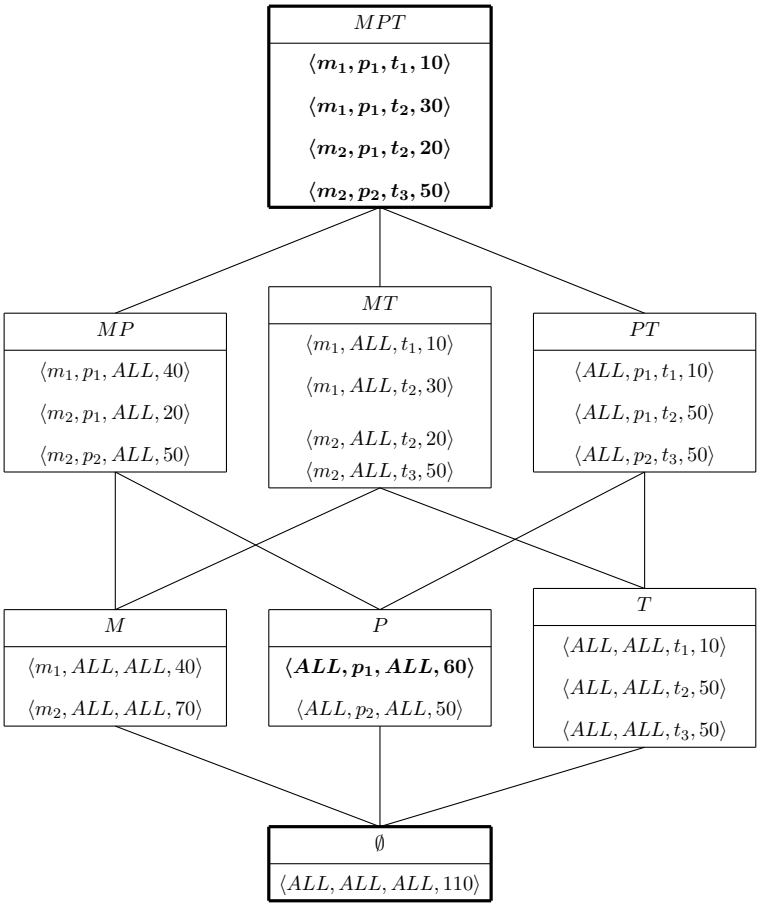


FIGURE 6.4 – Solutions 2_optimales.

La preuve de la proposition 6.2 concernant les tailles des solutions lorsque p est modifié s'adapte naturellement aux cas des solutions construites en sélectionnant les tuples et nous avons bien $|\mathcal{S}_p| \leq |\mathcal{S}_{p'}|$ où $\mathcal{S}_p$ et $\mathcal{S}_{p'}$ sont respectivement des solutions p et p'_optimales avec $p' < p$.

Maintenant que nous avons caractérisé les solutions et que nous savons exactement ce que nous cherchons, nous sommes en mesure de proposer des algorithmes permettant de trouver ces solutions.

6.3 Algorithmes

Nous avons fixé les fondements théoriques permettant de caractériser les solutions optimales au problème de la matérialisation partielle en sélectionnant des cuboïdes mais aussi des tuples. Voyons alors concrètement comment trouver cette solution. Nous n'allons parler ici que de cuboïdes mais, étant donné que les cuboïdes ou les tuples des solutions ont les mêmes caractéristiques, les mêmes méthodes peuvent être utilisées en considérant les tuples plutôt que les cuboïdes.

6.3.1 Solution exacte

Nous l'avons vu, une solution 1_optimale est composée de tous les cuboïdes 1_clos et l'espace de recherche d'une solution p_optimale peut être restreint à ces cuboïdes. Commençons alors par les identifier grâce à l'algorithme 3 en fixant $p = 1$ puis, il suffira de faire varier p.

Algorithme 3: Sélection des cuboïdes p_clos

```
   Entrée : p
1  F_p ← T ;
2  pour chaque cuboïde c ∈ Q faire
3  |  clos ← vrai ;
4  |  pour chaque c' ∈ P(cube) parent de c et clos = vrai faire
5  |  |  si |c| × p ≥ |c'| alors
6  |  |  |  clos ← faux ;
7  |  si clos = vrai alors
8  |  |  F_p ← F_p ∪ {c} ;
9  retourner F_p
```

Description de l'algorithme 3 :
Puisque la table de base, T, n'a pas d'ancêtre, elle est ajoutée directement à l'ensemble des cuboïdes p_clos, F_p (ligne 1). Nous avons considéré que toutes les requêtes sont équiprobables, l'ensemble des requêtes à optimiser, $\mathcal{Q}$, est donc égal à l'ensemble des 2^n cuboïdes du cube, ici $\mathcal{Q} = P(cube)$ mais cela ne coûtait rien d'ajouter cette distinction pour anticiper les besoins. Nous savons que tout cuboïde a une taille inférieure ou égale à celle de ses parents donc il n'est pas nécessaire de regarder la taille de tous les ancêtres plus haut dans le treillis pour savoir si le cuboïde en question est clos ou non. Il est ajouté à F_p si il n'existe aucun p_parent.

Nous l'avons vu avec le théorème 6.1, le problème de sélection des cuboïdes clos est NP-Difficile et nous n'avons donc aucun espoir de le résoudre en temps polynômial. Cet algorithme a une complexité en $O((2^n - 1) \times (m + n))$ puisque, pour chaque cuboïde autre que la table d'origine, nous devons calculer sa taille (c.-à-d. parcourir les m lignes de la table T pour en supprimer les doublons relativement aux attributs du cuboïde considéré) puis la comparer avec celles de ses parents ($\emptyset$ a n parents). En pratique, nous calculons les tailles des n enfants de T (c.-à-d.

ceux du niveau inférieur) en parcourant ses m lignes mais chacun d'eux n'a qu'un seul parent avec qui comparer sa taille. Ensuite, nous nous intéressons à leurs descendants directs mais leurs tailles peuvent être calculées à partir des projections de leurs parents à condition que celles-ci soient stockées et il suffit de ne lire que $m' \leq m$ lignes. Chacun d'eux n'a que 2 parents et ainsi de suite. La complexité donnée est donc une borne très largement supérieure à la complexité réelle, au moins en ce qui concerne le nombre de parents mais quoi qu'il en soit, les $2^n - 1$ cuboïdes du treillis doivent être considérés.

Rappelons que, grâce au lemme 6.2 et à la définition 1.9, les cuboïdes clos peuvent être obtenus avec l'aide des dépendances fonctionnelles. Nous proposons alors l'algorithme 4 pour identifier ces cuboïdes à partir de l'ensemble $\mathcal{F}_f$ de toutes les DFs de force $f' \geq f = 1/p$ valides et non triviales sur la table ($\mathcal{F}_f$ est construit de sorte que chaque partie gauche soit unique, c.-à-d. $\varphi \in \mathcal{F}_f$ si et seulement si φ est de la forme $X \rightarrow X_p^+ \setminus X$).

Algorithme 4: Sélection des cuboïdes clos à partir de $\mathcal{F}_f$

Entrée : $\mathcal{F}_f$
1 $F_p \leftarrow P(cube)$;
2 **pour chaque** $X \rightarrow Y \in \mathcal{F}_f$ **faire**
3 $\quad F_p \leftarrow F_p \setminus \{X\}$;
4 **retourner** F_p

Description de l'algorithme 4 :
Le principe de l'algorithme est très simple : si l'ensemble d'attributs d'un cuboïde intervient dans la partie gauche d'une DF, alors il n'est pas $p_$clos. Une fois que toutes les DFs sont passées en revue, il ne reste, dans F_p, que les cuboïdes $p_$clos. En construisant $\mathcal{F}_f$ de cette manière, nous avons au maximum $2^n - 2$ DFs non triviales (seuls les cuboïdes aux sommets du treillis, $\mathcal{A}$ et $\emptyset$, sont clos).
Il est naturel de penser que, en pratique, les risques sont faibles de rencontrer ce cas extrême et que, de ce fait, la complexité réelle est plutôt linéaire en $|\mathcal{F}_f|$ puisque cela revient à tester si une dépendance est impliquée par $\mathcal{F}_f$ comme l'expliquent [Abiteboul *et al.* 1995]. Cependant, nous avons considéré $\mathcal{F}_f$ comme acquis, or, sa préparation est coûteuse. En effet, nous avons vu dans la partie précédente que la mesure de force n'est pas monotone donc, seul un algorithme naïf est en mesure de calculer cet ensemble et sa complexité est en $O(2^n \times (m + n))$. En revanche, pour trouver une solution $1_$optimale, cette méthode est préférable puisque nous avons mis en place, précédemment, des algorithmes efficaces pour calculer $\mathcal{F}_1$.

Nous avons maintenant trouvé la solution $1_$optimale, F_1. Nous avons également identifié les cuboïdes $p_$clos, F_p, et, puisque l'espace de recherche peut être réduit à F_1, nous pouvons envisager de trouver une solution $p_$optimale de manière exacte en utilisant les techniques de programmation linéaire décrites par [Li *et al.* 2005]. Rappelons tout d'abord quelques notations pour décrire formellement le problème :
$\mathcal{C}(i, \{j\})$ est le coût de la requête i sur la vue j,

x_j est égal à 1 si le cuboïde j est matérialisé, et 0 sinon,

y_{ij} dénote le fait que le cuboïde j est utilisé pour évaluer la requête i, il est égal à 1 dans ce cas et à 0 sinon.

$$\begin{array}{ll} \text{Minimisons} & \sum_i |i| \times x_i \\ \text{en s'assurant que} & \forall i,j, \mathcal{C}(i,\{j\}) \times y_{ij} \times x_j \leq p \times |i| \quad (1) \\ & \forall i, \sum_j y_{ij} \times x_j \geq 1 \quad (2) \end{array}$$

Pour qu'une solution soit optimale, elle doit être la plus petite possible, c'est notre paramètre à minimiser. La première contrainte concerne la performance des requêtes : si une vue j est utilisée pour répondre à la requête i, $y_{ij} = 1$, alors j doit être un $p_$ancêtre matérialisé de i. Pour garantir ensuite la cohérence de la solution, il est établi que toute requête doit être évaluée à partir d'un cuboïde matérialisé (contrainte 2). Nous obtenons ainsi la solution $p_$optimale recherchée.

Exemple *Rappelons que* $F_1 = \{MPT, MP, PT, M, P, \emptyset\}$. *Pour plus de lisibilité, notons* 1 *le cuboïde* MPT, 2 *le cuboïde* MP, *etc. et* 6 *le cuboïde* $\emptyset$. *Voici le code donné au solveur lp-solve*[2] *pour trouver une solution* $3/2\_$ *optimale :*

/* `Minimisation de l'occupation mémoire :` */
Min : $4 \times x_1 + 3 \times x_2 + 3 \times x_3 + 2 \times x_4 + 2 \times x_5 + 1 \times x_6$;
/* `Contrainte (2) :` */
$y_{11} \times x_1 \geq 1$;
$y_{21} \times x_1 + y_{22} \times x_2 \geq 1$;
$y_{31} \times x_1 + y_{33} \times x_3 \geq 1$;
$y_{42} \times x_2 + y_{44} \times x_4 \geq 1$;
$y_{52} \times x_2 + y_{55} \times x_5 \geq 1$;
$y_{66} \times x_6 \geq 1$;
/* `Déclaration des variables :` */
bool $x_1, x_2, x_3, x_4, x_5, x_6$;
bool $y_{11}, y_{21}, y_{22}, y_{31}, y_{33}, y_{42}, y_{44}, y_{52}, y_{55}, y_{66}$;

La contrainte (1) a été combiné à la seconde puisque, au lieu de considérer tous les y_{ij} *possibles, pour une requête* i *donnée, les cuboïdes* j *pour y répondre sont uniquement ses* $p\_$ *ancêtres. Cela nécessite donc un pré-calcul de notre part mais réduit le nombre de variables à considérer.*
Bien évidemment, la solution retournée est la solution $3/2\_$ *optimale* $S = \{CMP, MP, \emptyset\}$.

6.3.2 Solution approchée

Malgré notre caractérisation et les pré-calculs facilitant le travail du solveur en réduisant l'espace de recherche et les contraintes et donc, diminuant le nombre de variables, trouver une solution $p_$optimale peut être laborieux. Nous voyons bien que le nombre de variables à considérer (16 sur notre exemple pourtant très simple) est rapidement trop important pour espérer obtenir une solution $p_$optimale dans un temps raisonnable. Nous proposons alors l'algorithme 5 qui trouve une

2. www.lpsolve.com

solution $p_$correcte même si sa taille n'est pas minimale. Il se base sur l'algorithme de recherche de l'ensemble couvrant de poids maximal. En effet, ces deux problèmes sont liés : il suffit de considérer chaque cuboïde 1_clos du treillis comme étant un nœud d'un graphe biparti orienté qui couvre l'ensemble des requêtes trouvant leurs réponses sur celui-ci. Nous cherchons donc à matérialiser un ensemble de taille minimale pour couvrir toutes les requêtes. Ce problème est également NP-Difficile mais [Chvatàl 1979] propose une solution approchée dont la taille est au plus $log(d)$ fois plus grande que la solution $p_$optimale où d est le degré sortant maximal.

Exemple *La figure 6.5 donne le graphe biparti sur lequel nous cherchons un ensemble couvrant de poids maximal.*

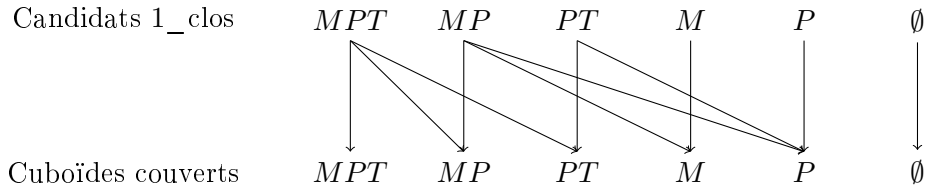


FIGURE 6.5 – Graphe permettant de trouver une solution approchée.

Nous fixons le poids d'un candidat c au nombre de cuboïdes couverts en respectant p divisé par la taille de c. Ainsi, un candidat de petite taille qui permet de répondre à beaucoup de requêtes a un poids relativement grand (il a donc de grande chance d'être sélectionné) par rapport à un candidat couvrant peu de requêtes ou dont la taille est plus grande.

Algorithme 5: Ensemble couvrant de poids maximal : une solution approchée

```
1  S ← F_f ;
2  candidats ← F_1 \ F_p ;
3  requetes ← F_1 \ Q^p_{F_p};
4  tant que requetes ≠ ∅ faire
5      p_max ← 0 ;
6      pour chaque cuboïde c ∈ candidats faire
7          poids ← |Q^p_c ∩ requetes| ÷ |c| ;
8          si poids > p_max alors
9              p_max ← poids ;
10             C ← c ;
11     S ← S ∪ {C} ;
12     candidats ← candidats \ {C} ;
13     requete ← requetes \ Q^p_C
14 retourner S
```

Description de l'algorithme 5 :
Initialement, la solution S contient tous les cuboïdes $p_$clos (ligne 1). Les candidats

à la matérialisation sont les cuboïdes de l'ensemble $F_1 \setminus F_p$ (ligne 2) et l'ensemble des requêtes à couvrir correspond à $F_1 \setminus \mathcal{Q}^p_{F_p}$ (ligne 3). Il n'est pas nécessaire de considérer toutes les requêtes mais seulement celles-ci puisque les cuboïdes qui ne sont pas 1_clos sont couverts de la même manière que leurs fermetures dans F_1 et celles auxquelles nous pouvons répondre à partir d'un cuboïde p_clos sont déjà couverte par la solution.
Tant qu'il reste des requêtes à couvrir, nous calculons le poids de chaque candidat et celui qui a un poids maximal est sélectionné (ligne 10). Il est ensuite ajouté à S (ligne 11) et supprimé de l'ensemble *candidats* (ligne 12). Nous supprimons également les requêtes ainsi couvertes (ligne 13).

Exemple *Pour trouver une solution* $3/2$_ *optimale, nous avons initialement* $S = \{MPT, \emptyset\}$, *candidats* $= \{MP, PT, M, P\}$ *et requetes* $= \{M, P\}$. *A la première itération de la boucle* `tant que`, *nous avons* $poids(MP) = 2/3$, $poids(PT) = 1/3$ *et* $poids(M) = poids(P) = 1/2$. MP *est donc ajouté à* S *vidant ainsi l'ensemble des requêtes restant à couvrir et mettant fin au programme. Ici, la solution retournée est une solution optimale mais nous pouvions seulement garantir que sa taille ne dépasserait pas* $log(3)$ *fois la taille de la solution optimale.*

Si tous les cuboïdes sont 1_clos et aucun n'est p_clos (sauf le cuboïde de base bien entendu) alors, à la ligne 2, *candidats* contient $2^n - 1$ éléments. À chaque tour de la boucle `tant que`(lignes 4 à 13), nous supprimons entre 1 et $n - 2$ éléments de l'ensemble *requetes*, soit en moyenne, $n \div 2$ (si le candidat est au niveau 1 du treillis, il n'a qu'un seul descendant, $\emptyset$, et s'il est au niveau $n - 1$, il a $n - 2$ descendants). Il y a donc environ $(2^n - 1) \div n/2 = (2^{n+1} - 2) \div n$ tours de boucle. Pour chacun, tous les éléments restants ainsi que leurs descendants sont considérés, nous avons donc une boucle `pour` de taille inférieure à $|candidats| \times (n - 2)$ où $n - 2$ est le nombre maximum de descendants dans l'espace de recherche.

Cet algorithme a donc une complexité exponentielle en nombre d'attributs mais il est facile de voir que l'utilisation de cette méthode est tout de même judicieuse étant donné le nombre de variables à considérer pour trouver une solution exacte.

6.3.3 Expérimentations

Afin d'évaluer la qualité de notre algorithme approché, nous devons comparer la solution retournée avec la solution exacte. Pour ce faire, nous avons utilisé le solveur lp-solve. Il s'est avéré (et ce n'est pas surprenant étant donné le grand nombre de variables à considérer) qu'il est impossible de terminer l'exécution dans des temps raisonnables (inférieurs à 4h) quand le nombre de dimensions est important. Nous avons donc restreint leur nombre à 7 et, dans ses conditions, notre algorithme s'exécute de façon quasi instantanée.

Soit S^* la solution optimale retournée par lp-solve. Notre caractérisation nous permet d'affirmer que $S^* = F_p \cup s$ où F_p est l'ensemble des éléments p_clos et s un ensemble d'éléments 1_clos. Notre approximation, trouvée grâce à l'algorithme 5 de recherche de l'ensemble couvrant de poids maximal ne porte que sur s. Rappelons

que d est le degré maximal du graphe décrit plus haut. Nous trouvons donc une $log(d)$-approximation de s.

Pour nos tests, nous comparons la taille de notre solution approchée à celle de la solution $p_$optimale lorsque le facteur de performance, p, augmente. Nous notons également, la taille occupée par les $p_$clos afin d'interpréter au mieux nos résultats et pour apprécier l'utilité de notre caractérisation dans l'élagage de l'espace de recherche.

Cas d'étude 1 :
Nous étudions un premier cube dans lequel il y a très peu de dépendances fonctionnelles à utiliser. Il s'agit du jeu de données US Census 1990 (kdd.ics.uci.edu) que nous avons tronqué afin d'obtenir un cube à 7 dimensions. Le cube a 362275 lignes et tous les cuboïdes sont 1_clos.
Nous constatons (figure 6.6) que ses 128 cuboïdes sont également 1.5_clos, c'est pourquoi, lorsque $p = 1.5$, nous trouvons la solution 1.5_optimale. Dans cet exemple, notre solution approchée se révèle être la solution optimale lorsque $p < 3$. En effet, dans ces cas là, l'espace de recherche comporte au maximum 10 cuboïdes à couvrir.
Ensuite cependant, nous notons une importante chute du nombre de $p_$clos ce qui explique pourquoi les courbes représentant notre solution et la solution optimale commencent à se distinguer plus clairement.

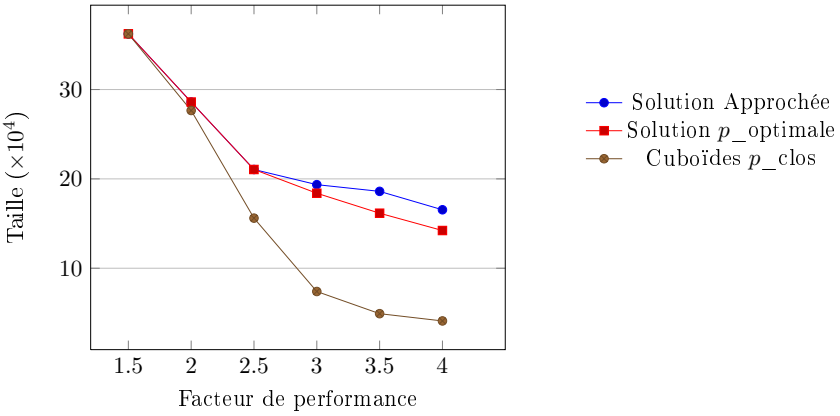


FIGURE 6.6 – Expérimentations sur le premier cube à 7 dimensions

Cas d'étude 2 :
Dans cet exemple tiré du jeu de données wisconsin breadth cancer data (archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/)
auquel 4 dimensions ont été supprimées afin de n'en avoir plus que 7, l'ensemble des dépendances fonctionnelles approximatives à exploiter est plus riche que sur l'exemple précédent. Le nombre de $p_$clos est donc beaucoup moins significatif ce qui implique que nous avons un espace de recherche plus important. Cela explique le fait que nos résultats sont moins proches, en terme d'occupation mémoire, de la solution optimale que précédemment. Le cube a 22494 lignes et 126 1_clos.

L'espace de recherche étant naturellement plus grand, le solveur lp-solve considère un grand nombre de variables et prend donc plusieurs heures pour s'exécuter notamment lorsque $p \geq 2$ alors que notre algorithme retourne une solution après seulement quelques secondes.
Il est à noter cependant, sur la figure 6.7, que la somme des tailles des $p_$clos et celle de la solution globale décroissent en suivant sensiblement le même coefficient lorsque p augmente. Ceci n'est pas surprenant puisque, lorsque la valeur de p augmente, nous pouvons nous permettre d'utiliser des cuboïdes situés plus haut dans le treillis donc d'en stocker moins. Cela illustre donc la proposition 6.2. De plus, les trois courbes restent relativement "parallèles". Le fait d'avoir plus de dépendances fonctionnelles à considérer implique que le nombre de descendants de chaque cuboïde devient considérable (déjà un maximum de 45 descendants $1_$clos lorsque $p = 1.5$ et jusqu'à 92 pour $p = 4$). Notre approximation se basant sur ce nombre, il est tout à fait prévisible d'observer de moins bons résultats sur cet exemple que dans le précédent.

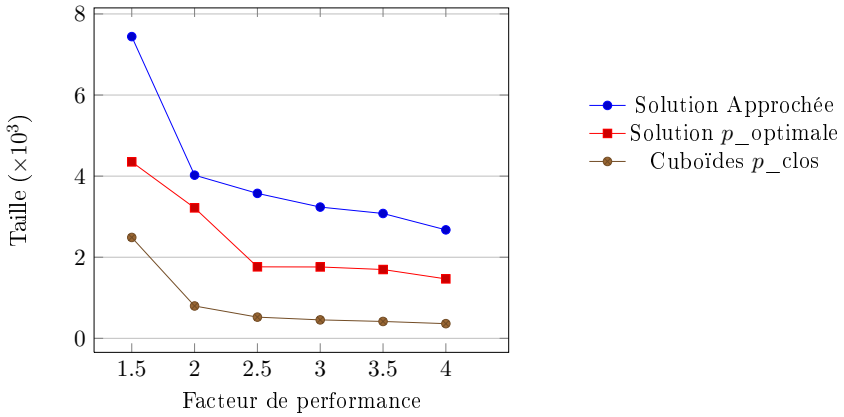


FIGURE 6.7 – Expérimentations sur le second cube à 7 dimensions

Nous voyons donc sur ces deux exemples que essentiellement deux facteurs sont à prendre en compte pour évaluer l'efficacité de cette approximation :

- Le temps d'exécution : quelle que soit la taille de l'espace de recherche, il est de l'ordre de la seconde pour un cube à 7 dimensions avec l'algorithme 5. En revanche, il faut parfois attendre plusieurs heures avant d'avoir la solution optimale avec lp-solve puisque le nombre de variables influe sur la difficulté du calcul.
- Le nombre de dépendances fonctionnelles à considérer : lorsque ce nombre est faible, notre résultat est très proche de l'optimal puisque l'espace de recherche et le nombre de descendants sont négligeables. Cependant, lorsque ce nombre augmente, notre solution est plus "éloignée" de l'optimale même si elle reste très acceptable dans le sens où nous savons borner sa déviation par rapport à la solution optimale.

6.4 Caractérisation des résumés

Les méthodes de résumé des cubes de données proposent une sélection de tuples représentatifs permettant de répondre à n'importe quelle requête sans perte de performance. Nous les qualifions donc de solutions 1_correctes. Dans cette section, nous étudions l'optimalité de ces techniques puisque, nous l'avons vu, la solution 1_optimale est unique. Il est donc intéressant de voir quelle méthode s'en rapproche le plus. Le problème majeur rencontré lorsque nous souhaitons comparer plusieurs méthodes réside dans le fait que chacune d'elle utilise un vocabulaire et des notions bien précis. Nous allons alors exprimer chaque technique avec les terminologies décrites précédemment. Comme l'a dit Antoine de Saint-Exupéry : *"Unifier, c'est nouer même les diversités particulières, non les affecter pour un ordre vain."* Notre but ici n'est donc pas de désigner la meilleure méthode mais plutôt d'étendre notre champs de vision et d'appréhender le problème sous un angle différent.

6.4.1 Cube condensé

Le concept du cube condensé a été proposé par [Wang *et al.* 2002]. Les auteurs assurent que toutes les requêtes trouvent leur réponse sans avoir besoin d'effectuer d'opération de décompression sur le cube ni aucune agrégation. Le cube condensé est donc bien une solution 1_correcte. Les définitions et propriétés données sont celles tirées de l'article [3] et les explications qui s'y rapportent sont des interprétations personnelles permettant de les comprendre avec notre vocabulaire.

Pour donner l'intuition de leur méthode, les auteurs identifient des tuples particuliers en commençant par établir un lien entre chaque tuple, nommé τ_i, du cube et un ensemble de tuples de la relation de base, notre table T, $\{t_j, \dots t_k\}$.

Dans leur représentation du cube de données, les tuples sont partitionnés en fonction du tuple de la table d'origine à partir duquel ils sont calculés. Lorsque plusieurs tuples de la table sont nécessaires pour donner un tuple agrégé, il est placé dans une dernière partition. Nous avons donc un partitionnement en $m+1$ parties. A partir de ces tuples de la table, le partitionnement est donné à l'aide de la définition suivante :

Définition 6.12 (Tuple de base unique sur dimensions propres) *Soit $DP \subset \mathcal{A}$ un sous-ensemble d'attributs. Si le tuple t de T est le seul représentant de ses valeurs dans la projection de T sur DP, alors t est un tuple unique de base sur ses dimensions propres DP.*

Exemple *t_1 et t_4 sont uniques sur la dimension T mais également, par augmentation, sur les dimensions MT et PT alors que t_2 n'est unique que sur MT, t_3 sur MP et MT, t_4 est aussi unique sur P et donc, par augmentation, sur MP également.*

3. Afin d'assurer une certaine cohérence avec ce qui précède, nous utilisons nos notations même si ce ne sont pas celles de l'article.

La figure 6.8 donne le cube de données reprenant le formalisme du papier. Nous pouvons ainsi y ajouter l'identifiant des tuples ainsi que la colonne Cuboïde qui précise l'ensemble de définition de chaque tuple. Pour associer plus facilement les tuples

ID	*Cuboïde*	*Métier*	*Produit*	*Transp.*	*Total*
1	$\emptyset$	ALL	ALL	ALL	110
2	MPT	m_1	p_1	t_1	10
3	T	ALL	ALL	t_1	10
4	MT	m_1	ALL	t_1	10
5	PT	ALL	p_1	t_1	10
6	MPT	m_1	p_1	t_2	30
7	MT	m_1	ALL	t_2	30
8	MPT	m_2	p_1	t_2	20
9	MP	m_2	p_1	ALL	20
10	MT	m_2	ALL	t_2	20
11	MPT	m_2	p_2	t_3	50
12	P	ALL	p_2	ALL	50
13	T	ALL	ALL	t_3	50
14	MP	m_2	p_2	ALL	50
15	MT	m_2	ALL	t_3	50
16	PT	ALL	p_2	t_3	50
17	M	m_1	ALL	ALL	40
18	M	m_2	ALL	ALL	70
19	P	ALL	p_1	ALL	60
20	T	ALL	ALL	t_2	50
21	MP	m_1	p_1	ALL	40
22	PT	ALL	p_1	t_2	50

FIGURE 6.8 – Cube de données de la table *Clientèle*

du cube aux tuples de la table Clientèle, il est bon de rappeler l'ordre d'apparition des tuples et de les numéroter (figure 6.9). Ainsi, τ_2 de cube est calculé à partir du

ID	*Métier*	*Produit*	*Transp.*	*Valeur*
1	m_1	p_1	t_1	10
2	m_1	p_1	t_2	30
3	m_2	p_1	t_2	20
4	m_2	p_2	t_3	50

FIGURE 6.9 – Table *Clientèle*

premier tuple, t_1, de la table Clientèle et τ_{18}, par exemple, à partir des tuples t_1 et t_2.

Les auteurs s'intéressent aux tuples de la table qui sont à l'origine d'un seul tuple agrégé dans des cuboïdes identifiés par l'ensemble DP. Nous avons abordé ce sujet

dans l'autre sens en remarquant qu'un tuple du cuboïde DP a un $1_T_$ancêtre. Leur manière de représenter le cube complet a donc le mérite d'être clair sur ce point puisque les tuples agrégés sont dans la même partition que le tuple de T dont ils sont issus. Les auteurs définissent alors une fonction *Expand* qui retourne un tuple du cube (ou un ensemble de tuples du cube) à partir d'un tuple de T et d'un ensemble d'attributs DP (ou d'un ensemble d'ensembles appelé $DPSET$), plus formellement :

Définition 6.13 (Fonction *Expand*) *Expand prend en entrée un tuple de T avec ses dimensions propres DP ou un ensemble de dimensions propres $DPSET$ et retourne un tuple du cube complet ou un ensemble de tuples respectivement.*

- $Expand(t, DP) = \tau$ *avec* $\forall A_i \in \mathcal{A}, \tau[A_i] = t[A_i]$ *si* $A_i \in DP$ *et* $\tau[A_i] = ALL$ *sinon, et* $agr(\tau) = agr(t)$ *où* $agr(\tau)$ *retourne la valeur du tuple* τ *relativement à la fonction* agr.
- $Expand(t, DPSET) = \{\tau, \tau = Expand(t, DP_i) \wedge DP_i \in DPSET\} = \bigcup_{DP_i \in DPSET} Expand(t, DP_i)$.

Exemple *Nous avons vu que t_1 est unique sur T donc $Expand(t_1, T) = \langle ALL, ALL, t_1, 10\rangle = \tau_3$. Par extension, $DPSET = \{T, MT, PT\}$, alors $Expand(t_1, DPSET) = \{\tau_3, \tau_4, \tau_5\}$.*

La fonction *Expand* donne donc les tuples définis sur DP_i pour lesquels le tuple de T considéré est un $1_$ancêtre. Les auteurs peuvent ainsi décrire formellement la composition d'un cube condensé :

Définition 6.14 (Cube condensé) *Un cube condensé CC est défini sur le schéma $(\mathcal{A}, M, DPSET)$ et ses tuples sont notés $cc(cc.t, cc.DPSET)$ où $cc.t = (a_1, \ldots a_n, M(cc))$. Pour les tuples n'appartenant pas à T (c.-à-d. tels que $Def(cc) \neq \mathcal{A}$), $DPSET = \{\}$. Tous les tuples de CC respectent les conditions suivantes :*

1. $\forall cc \in CC$ *avec* $Def(cc) \neq \mathcal{A}$, $\nexists cc_T$ *tel que* $cc \in Expand(cc_T.t, cc_T.DPSET)$
2. $CC \cup \bigcup_{cc \in CC} Expand(cc.t, cc.DPSET) = Cube$.

Le première condition traduit le fait qu'un tuple agrégé appartenant à l'ensemble des tuples donnés par la fonction *Expand* à partir d'un tuple de T et de son ensemble de dimensions propres n'appartient pas au cube condensé. La seconde assure que, par extension, un cube condensé contient exactement tous les tuples du cube complet. Un cube condensé, pour être utile, se doit d'avoir une taille la plus petite possible. Les auteurs définissent alors le cube condensé minimal :

Définition 6.15 (Cube condensé minimal) *Un cube condensé minimal CCM est un cube condensé ne contenant aucun tuple pouvant être généré directement à partir d'un tuple de T.*

En d'autres termes, l'ensemble d'attributs propres de chaque tuple de T est le plus complet possible afin que $Expand(cc_T.t, cc_T.DPSET)$ retourne tous les tuples pour lesquels cc_T est un $1_T_$ancêtre et qui n'appartiennent donc pas à CCM.

ID	M	P	T	$Total$	DPSET
1	m_1	p_1	t_1	10	$\{\{T\},\{MT\},\{PT\},\{MPT\}\}$
2	m_1	p_1	t_2	30	$\{\{MT\},\{MPT\}\}$
3	m_2	p_1	t_2	20	$\{\{MP\},\{MT\},\{MPT\}\}$
4	m_2	p_2	t_3	50	$\{\{P\},\{T\},\{MP\},\{MT\},\{PT\},\{MPT\}\}$
5	m_1	ALL	ALL	40	$\{\}$
6	m_2	ALL	ALL	70	$\{\}$
7	ALL	p_1	ALL	60	$\{\}$
8	ALL	ALL	t_2	50	$\{\}$
9	m_1	p_1	ALL	40	$\{\}$
10	ALL	p_1	t_2	50	$\{\}$
11	ALL	ALL	ALL	70	$\{\}$

FIGURE 6.10 – Cube condensé minimal de la table *Clientèle*

Exemple *La figure 6.10 donne le cube condensé minimal de la table Clientèle. Les 4 premiers tuples correspondent aux tuples de la table d'origine et les 7 suivants sont ceux qui n'ont pas de* $1\_MPT\_$ *ancêtre.*

Tous les tuples du cube complet qui n'appartiennent pas au cube condensé minimal sont donc des tuples agrégés issus d'un seul tuple de la table d'origine. Le lemme suivant donne une première caractérisation du cube condensé minimal :

Lemme 6.4 *Soient* CCM *le cube condensé minimal de cube et* $\tau \in cube \setminus CCM$. *Alors* $\tau_1^+ = \mathcal{A}$.

Preuve *Par définition de la fermeture d'un tuple (définition 6.10),* $\tau_1^+ = \mathcal{A}$ *est équivalent à dire que* τ *a un* $1\_T\_$ *ancêtre,* t, *puisque* T *est défini sur* $\mathcal{A}$. *Par conséquent, d'après la définition 6.7,* $\tau \prec t$ *et le résultat de la fonction d'agrégation sur* τ *est le même que celui sur* t. *Donc, conformément à la définition 6.13,* $\tau \in Expand(t, Def(\tau))$ *et donc* $\tau \notin CCM$. □

Rappelons qu'un tuple $1_$clos est un tuple qui n'a pas d'ensemble_ancêtre de taille 1 (définition 6.11). Un cube condensé minimal ne contient pas les tuples qui ont un ancêtre dans T mais il contient les autres tuples et notamment certains qui ne sont pas $1_$clos. Notre solution $1_$optimale ne contient aucun tuple $1_$clos et nous voyons, par la proposition suivante, qu'elle est donc incluse dans un cube condensé minimal.

Proposition 6.9 *Soient* CCM *un cube condensé minimal et* F_1^t *l'ensemble des tuples* $1\_$ *clos. Alors* $F_1^t \subseteq MCC$.

Preuve *Soit* $\tau \in cube \setminus CCM$. *D'après le lemme 6.4,* τ *a un* $1\_T\_$ *ancêtre et donc* τ *n'est pas* $1\_$ *clos, c.-à-d.* $\tau \notin F_1^t$. □

La principale différence entre notre approche et le cube condensé minimal réside donc dans le fait que nous considérons toutes les relations qui existent entre les tuples du cube alors que les auteurs du CCM n'étudient les tuples qu'en fonction de la table d'origine.

Exemple *Notre solution* 1_ *optimale, transposée selon leur formaliste, est donnée par la figure 6.11. Le tuple 5 du CCM, puisqu'il a un* 1_ *ensemble_ ancêtre, n'est pas matérialisé mais, en contrepartie, le DPSET de notre tuple 7 a été étendu pour prendre en compte cette suppression. Il en est de même pour le tuple 8 du CCM puisqu'il peut être calculé à partir de notre tuple 8 puisque le DFSET a été redéfini.*

ID	M	P	T	$Total$	DPSET
1	m_1	p_1	t_1	10	$\{\{T\},\{MT\},\{PT\},\{MPT\}\}$
2	m_1	p_1	t_2	30	$\{\{MT\},\{MPT\}\}$
3	m_2	p_1	t_2	20	$\{\{MP\},\{MT\},\{MPT\}\}$
4	m_2	p_2	t_3	50	$\{\{P\},\{T\},\{MP\},\{MT\},\{PT\},\{MPT\}\}$
5	m_2	ALL	ALL	70	$\{\}$
6	ALL	p_1	ALL	60	$\{\}$
7	m_1	p_1	ALL	40	$\{\{M\},\{MP\}\}$
8	ALL	p_1	t_2	50	$\{\{T\},\{PT\}\}$
9	ALL	ALL	ALL	110	$\{\}$

FIGURE 6.11 – Notre solution 1_optimale

Cette méthode n'est donc pas optimale dans le sens où elle n'est pas de taille minimale mais elle présente cependant un intérêt majeur. En effet, lorsque le besoin ne s'était pas encore fait sentir de pré-calculer les requêtes, elles étaient évaluées à partir de T et les auteurs ont gardé cette manière naturelle de procéder en l'optimisant à l'aide de l'ensemble $DPSET$. Ainsi, si la requête q n'a pu alors trouver de réponse (c.-à-d. $t \in T$ ancêtre de q tel que $X \in t.DPSET$ où X est l'ensemble d'attributs sur lequel la requête est définie), elle est pré-calculée et son résultat se trouve dans la dernière partie du cube condensé minimal. Cet ensemble permet donc de répondre plus facilement aux requêtes en identifiant les descendants des tuples de T. Cependant, une telle précision est difficilement adaptable au cas où le facteur de performance est étendu.

6.4.2 Cube quotient

Le concept du cube quotient, présenté par [Lakshmanan *et al.* 2002], semble beaucoup proche de nos travaux. En effet, dès le résumé du papier, les auteurs précisent leurs principaux objectifs : (i) donner un résumé le plus petit possible, (ii) préserver la structure sous forme de treillis (telle qu'elle est donnée par la figure 4.4) afin de garantir une navigation aisée dans le cube, et (iii) ne perdre aucune information. Ils proposent pour cela de partitionner le cube et de caractériser les propriétés de ses partitions.

La définition suivante donne le type de partitions requis pour atteindre les trois buts fixés :

Définition 6.16 (Partition convexe) *Soit π une partition du cube. La partie $\pi_i \in \pi$ est convexe dans la mesure où $\forall \tau, \tau' \in \pi_i$ tels que $\tau \prec \tau'$ alors $\forall \tau''$ avec $\tau \prec \tau'' \prec \tau'$, $\tau'' \in \pi_i$. Si toutes les parties de π sont convexes alors π est convexe.*

L'analogie entre cette définition et celle de l'ensemble des dimensions propres du cube condensé est assez naturelle. En effet, comme cela a été prouvé par [Wang *et al.* 2002], si deux éléments du cube appartiennent à un ensemble alors, par augmentation, tous les éléments se trouvant sur les chemins permettant d'aller de l'un à l'autre sont nécessairement dans ce même ensemble. Nous devons alors identifier les deux extrémités du chemin. Pour cela, les auteurs définissent la notion d'équivalence entre tuples comme suit :

Définition 6.17 (Tuples équivalents) *Soient agr une fonction d'agrégation*[4] *et τ_1, τ_2 deux tuples du cube. τ_1 et τ_2 sont équivalents selon agr, noté $\tau_1 \equiv_{agr} \tau_2$, si et seulement si $agr(\tau_1) = agr(\tau_2)$ et $\tau_1 \prec \tau_2 \vee \tau_1 \prec \tau_2$ où $agr(\tau_i)$ donne la valeur de mesure selon agr du tuple τ_i.*

Nous en déduisons que deux tuples sont équivalents si et seulement si nous pouvons répondre exactement à une requête portant sur l'un en utilisant uniquement l'autre. Ce qui signifie que l'un est un 1_ensemble_ancêtre de l'autre. Quelques précautions doivent cependant être prises puisque cette remarque n'est vraie que pour les fonctions strictement monotones `SUM` et `COUNT`. Nous le verrons plus tard, l'analyse est différente en ce qui concerne les fonctions `MIN` et `MAX`. Les auteurs en revanche ne font pas cette distinction et considèrent indifféremment ces quatre fonctions.

Le cube peut donc être partitionné en affectant à une même partie tous les tuples équivalents. Les auteurs montrent qu'un partitionnement construit selon $\equiv_{agr}$ est nécessairement convexe.

Exemple *Sur la figure 6.12, nous avons lié d'un trait épais les tuples équivalents. Le cube peut donc être partitionné en 9 parties (le tuple $\langle \emptyset, \emptyset, \emptyset \rangle$ n'étant qu'un tuple fictif, il n'appartient à aucune partie).*

Les auteurs ajoutent une précision quant à la qualité de la relation d'équivalence :

Définition 6.18 (Congruence faible) *Soient $(cube, \preceq)$ le treillis du cube et $\equiv$ une relation d'équivalence sur les tuples du cube. $\equiv$ est une congruence faible si $\forall \tau_1, \tau_1', \tau_2, \tau_2' \in cube$ avec $\tau_1 \equiv \tau_1'$, $\tau_2 \equiv \tau_2'$, $\tau_1 \preceq \tau_2$ et $\tau_2' \preceq \tau_1'$ alors $\tau_1 \equiv \tau_2$.*

Ils re-formulent cette définition ainsi : une relation d'équivalence est une congruence faible si et seulement si le partitionnement π en résultant est tel que

4. Les auteurs précisent que ces fonctions doivent être monotones et donc, pour la fonction `SUM`, les valeurs de la table doivent être soit toutes positives soit toutes négatives.

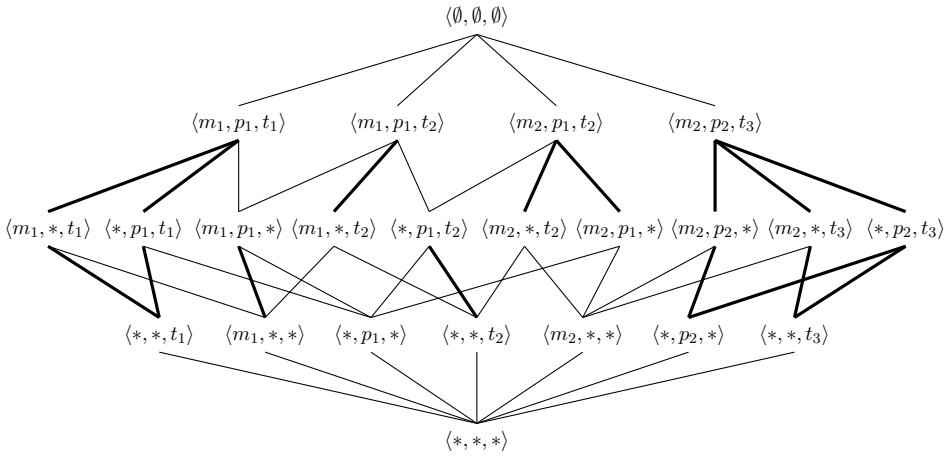


FIGURE 6.12 – Liens d'équivalence entre les tuples du treillis.

$\forall \pi_i, \pi_j \in \pi$, $\forall \tau_i \in \pi_i$ et $\forall \tau_j \in \pi_j$ avec $\tau_i \prec \tau_j$ alors $\forall \tau'_i \in \pi_i$ et $\forall \tau'_j \in \pi_j$, nous avons $\tau'_j \not\prec \tau'_i$.

Nous sommes ainsi assurés de la bonne interprétation que nous avions faite de la relation d'équivalence. En effet, si $\tau_j \in \pi_j$ est un ancêtre de $\tau_i \in \pi_i$, en sachant que τ_j seul ne permet pas de calculer τ_i puisqu'ils sont dans des parties différentes, alors les descendants de τ_i, ni même de n'importe quel tuple de π_i, ne peuvent pas admettre τ_j comme 1_ensemble_ancêtre et être dans π_j.

Les auteurs prouvent que de telles relations d'équivalence ne peuvent donner lieu qu'à des partitions convexes. Le contexte est maintenant clairement défini, il est temps alors de préciser la construction du cube quotient et plus particulièrement sa structure en treillis :

Définition 6.19 (Treillis d'un cube quotient) *Soient* $(cube, \prec)$ *le treillis d'un cube de données et* $\equiv$ *une congruence faible sur les tuples du cube. Les éléments du cube quotient sont les parties données par* $\equiv$ *et* $\forall \pi_i, \pi_j \in \pi$, $\pi_i < \pi_j$ *si et seulement si* $\exists \tau_i \in \pi_i$, $\exists \tau_j \in \pi_j$ *tels que* $\tau_i \prec \tau_j$. *Ce treillis est noté* $(cube/\equiv, <)$.

Pour répondre aux requêtes, il faut alors savoir à quelles parties ses tuples appartiennent. Pour cela, il suffit de conserver la borne supérieure, $\pi_i^{sup} = \{\tau \in \pi_i \mid \nexists \tau' \in \pi_i : \tau' \prec \tau\}$, et inférieure, $\pi_i^{inf} = \{\tau \in \pi_i \mid \nexists \tau' \in \pi_i : \tau' \prec \tau\}$, de chaque partie. Ensuite, il faut comparer les tuples recherchés avec ces bornes pour connaître leurs valeurs.

Exemple *Le treillis de la figure 6.13 représente le cube quotient de la table Clientèle. 17 tuples suffisent donc pour répondre exactement à toutes les requêtes.*

Il est à noter que notre solution 1_optimale (figure 6.3) coïncide exactement avec les bornes supérieures de chaque partie. En effet, si un tuple est 1_clos, alors il n'a pas d'ancêtre ayant la même mesure que lui et donc il est le plus spécifique

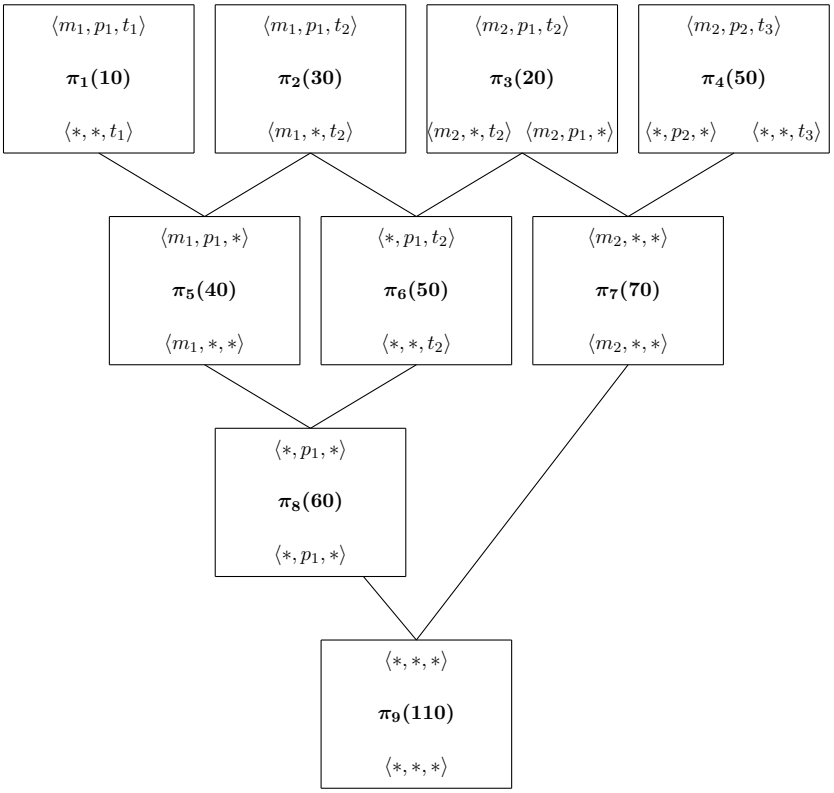


FIGURE 6.13 – Treillis du cube quotient défini sur `SUM`.

(c.-à-d. le plus grand relativement à $\prec$) d'un ensemble de tuples descendants ayant la même valeur de mesure. La proposition suivante le prouve en re-formulant la définition du cube quotient en terme de fermeture de tuple :

Proposition 6.10 *Soient τ et τ' deux tuples. Ils appartiennent à la même partie si et seulement si $\tau_1^+ = \tau_1'^+$.*

Preuve *($\Rightarrow$) Si $\tau \equiv \tau'$ (c.-à-d. ils appartiennent à la même partie) alors, d'après la définition 6.17, il existe un ancêtre commun τ'' tel que $agr(\tau'') = agr(\tau) = agr(\tau'')$ où $agr(\tau)$ donne la valeur de la mesure du tuple τ relativement à la fonction d'agrégation agr (sans perte de généralité, τ'' peut être égal à τ ou τ'). Donc τ et τ' peuvent être calculés à partir de τ''. Donc τ'' est un 1_ensemble_ancêtre de τ et τ' et, par conséquent, il appartient à leur 1_fermeture (définition 6.10). D'après la définition 6.8, τ et τ' ne peuvent pas être évalués entièrement à partir d'un tuple plus spécifique que τ'' donc τ'' est la borne supérieure des parties contenant respectivement τ et τ'. Puisqu'un tuple n'appartient qu'à une seule partie et que nous avons $\tau'' \in \tau_1^+ \wedge \tau'' \in \tau_1'^+$, donc $\tau_1^+ = \tau_1'^+$.*
($\Leftarrow$) Soit $\tau'' = \tau_1^+ = \tau_1'^+$ un tuple. Nous savons alors que $agr(\tau) = agr(\tau') = agr(\tau'')$ et τ'' est un ancêtre commun à τ et τ'. Donc τ et τ' appartiennent à la même partie. □

Nous savons donc caractériser les tuples des bornes supérieures des parties du cube quotient mais seulement pour les fonctions d'agrégation `SUM` et `COUNT`. Qu'en est il alors pour les fonctions `MIN` ou `MAX` ? Pour donner l'intuition d'un résultat, nous allons commencer par étudier un exemple simple.

Exemple *La figure 6.14 présente le cube quotient de la table Clientèle construit avec la fonction d'agrégation* `MIN`.

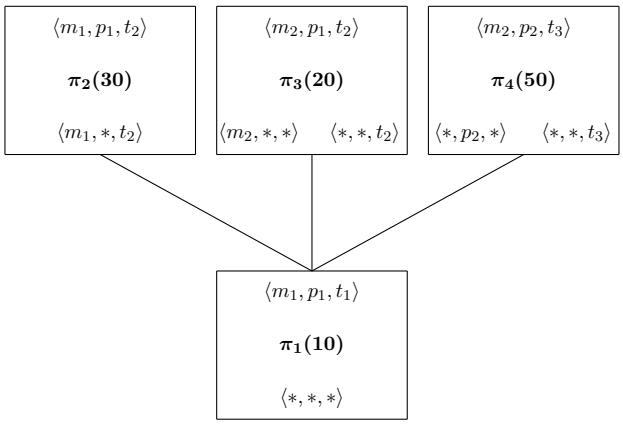


FIGURE 6.14 – Cube quotient avec `MIN`

Nous voyons bien par cet exemple que le cube quotient est très fortement lié à la fonction d'agrégation utilisée alors que nous n'en avons pas réellement tenu compte. En effet, pour connaître la valeur minimale de mesure d'un tuple nous avons besoin de parcourir un ensemble d'ancêtres alors que les auteurs du cube quotient ne conservent que le tuple ayant la valeur minimale. Ainsi, la borne inférieure d'une partie devient déterminante pour savoir à quelle classe appartient un tuple alors que nous n'avons observé que les ancêtres sans jamais vraiment s'intéresser aux descendants. Ceci ouvre donc vers de nouvelles perspectives pour caractériser plus précisément les solutions $p_$optimales. En effet, nous avons utilisé l'ensemble des requêtes couvertes pour choisir de matérialiser ou non un cuboïde afin d'obtenir une solution approchée (algorithme 5) mais une étude plus approfondie nous permettrait sans doute de dégager certaines propriétés concernant la solution optimale.

Nous avons donc donné une caractérisation partielle du cube quotient puisque la proposition 6.10 n'identifie que les bornes supérieures des parties pour les cubes construits avec des fonctions strictement monotones seulement. Cela est dû principalement au fait que les tuples n'ont été étudiés que sur les attributs de dimensions et non sur la mesure. Une modification simple à apporter consiste à considérer la mesure de la même manière que les dimensions et ainsi, nous obtenons une caractérisation des bornes supérieures pour les fonctions `MIN` et `MAX`. Un problème se pose alors lorsque plusieurs tuples de la table de base ont la même valeur puisque nous n'avons considéré qu'une seule borne supérieure alors que l'ensemble π_i^{sup} peut

contenir plusieurs tuples. Devant la multitude de cas à analyser et puisque les problèmes initiaux sont différents, nous nous contentons de la caractérisation donnée en sachant qu'elle peut être étendue afin de prendre en compte les différents cas de figure.

6.4.3 Cube clos

L'idée de matérialiser les tuples clos afin de ne perdre aucune information et d'obtenir un résumé le plus petit possible n'est pas nouvelle. [Casali *et al.* 2009] proposent le concept du cube clos qui, comme son nom l'indique, ne contient que des tuples 1_clos. Cependant, les auteurs n'utilisent pas les dépendances fonctionnelles conditionnelles pour le calculer mais ils s'appuient sur la correspondance de Galois puisqu'elle est intimement liée à la notion de fermeture. Nous n'allons pas entré dans le détail de l'analyse du treillis de Galois puisque les résultats qu'ils obtiennent correspondent à notre solution 1_optimale composée de tuples.

6.5 Conclusion

Nous avons proposé une caractérisation de la solution optimale à l'aide des dépendances fonctionnelles. Nous avons étudié, dans un premier temps les cuboïdes pour trouver une sélection minimale (en terme d'occupation mémoire) respectant strictement un seuil de performance fixé par l'utilisateur. Ces travaux sont l'objet de la publication [Garnaud *et al.* 2011]. Nous avons défini un cuboïde en fonction de ses ancêtres pour savoir s'il était pertinent ou non de le matérialiser. Nous avons prouvé que toute solution p_optimale est incluse dans la solution 1_optimale, ce qui permet de réduire l'espace de recherche. Dans une version étendue de ce papier, [Garnaud *et al.* 2012a], nous avons présenté des algorithmes pour trouver ces solutions. La solution 1_optimale peut être trouvée en temps linéaire en nombre de dépendances fonctionnelles valides sur la table. Ensuite, si l'espace de recherche est suffisamment réduit, nous pouvons envisager l'utilisation d'un solveur pour en extraire la solution p_optimale. Sinon, nous avons présenté un algorithme d'approximation de cette solution qui, en pratique donne de bons résultats.

Dans un second temps, nous avons adapté cette caractérisation en étudiant chaque tuple du cube de données. Ces résultats, présentés dans [Garnaud *et al.* 2012b] [5], nous ont permis d'étudier précisément les techniques de résumé de cube. Nous avons ainsi caractérisé, sous un formalisme commun, les tuples matérialisés dans chacunes des méthodes.

Ce travail présente néanmoins quelques lacunes puisque nous ne sommes pas parvenu à caractériser entièrement une solution p_optimale. En effet, nous l'avons défini comme étant composée d'éléments 1_clos et p_clos mais nous n'avons pas précisé la nature des 1_clos à matérialiser. Nous avons contourner ce problème en

5. Une version plus détaillée de ce papier est à paraître dans le journal Annals of Mathematics and Artificial Intelligence [Garnaud *et al.* 2013b].

réduisant l'espace de recherche afin d'utiliser la programmation linéaire et ainsi aboutir à une solution $p_$optimale. Pour savoir précisément quels sont ces éléments $1_$clos à matérialiser, nous avons identifié plusieurs axes à étudier : (i) analyse des éléments $p'_$clos $\forall p', p > p' > 1$ puisque nous savons que $F_p \subseteq F_{p'} \subseteq F_1$ où F_i est l'ensemble des éléments $i_$clos (proposition 6.4), (ii) caractérisation d'un élément vis-à-vis de ses descendants couverts puisqu'ils nous ont permis de trouver une solution approchée avec l'algorithme 5, (iii) identification des éléments restant à couvrir et des sous-ensembles permettant de le faire via les opérateurs Sum et Product définis par [Casali *et al.* 2003] mais nous devons les étendre pour prendre en compte le facteur de performance.

Pour ce qui est de la sélection de tuples, il reste un point supplémentaire à éclaircir : le calcul des ensembles_ancêtres. En effet, grâce aux dépendances conditionnelles, nous calculons facilement la $p_$fermeture d'un tuple. Nous en déduisons ses $p_$ensembles_ancêtres contenus dans un même cuboïde mais ce n'est pas suffisant pour identifier tous ses $p_$ensembles_ancêtres si $p > 1$. Ici encore, l'utilisation des opérateurs Sum et Product proposés par [Casali *et al.* 2003] semble être indiqué pour répondre à ce problème. Une habile combinaison de ces deux opérateurs permettrait peut être d'aboutir à l'ensemble_ancêtre recherché mais nous ne sommes pas parvenus à un tel résultat.

Chapitre 7

Conclusion et perspectives

Cette thèse présente nos contributions dans le domaine de l'extraction des dépendances ainsi que sur la résolution du problème de la matérialisation partielle des cubes de données.

Nous avons présenté **ParaCoDe**, un algorithme générique d'extraction des dépendances traitant les candidats en parallèle. Nous avons constaté son efficacité sur d'importants jeux de données. Cependant, pour être compétitif vis-à-vis des méthodes séquentielles lorsqu'il y a peu de candidats à considérer, nous envisageons un parallélisme sur les données permettant un gain de temps sur chaque calcul de projection. Pour améliorer encore les performances, l'intégration des techniques d'élagage de l'espace de recherche utilisées dans la littérature doit être programmée. Une analyse de la consommation mémoire doit également être menée afin d'identifier les limites de notre méthode.

Concernant la matérialisation partielle du cube de données, que ce soit en terme de cuboïdes ou de tuples, nous avons complètement caractérisé la solution 1_optimale. Celle-ci nous a permis de réduire l'espace de recherche d'une solution p_optimale. Cependant, nous ne sommes pas parvenus à caractériser précisément cette dernière. De plus, l'identification des ensembles ancêtres d'un tuple n'est pas donnée entièrement par sa p_fermeture comme c'est le cas pour les cuboïdes. Un examen approfondi de ces ancêtres est donc encore nécessaire pour préciser nos solutions.

Ouverture du sujet

Une fois extraites les dépendances minimales satisfaites par une table doivent être analysées par un expert. Or, nous nous sommes contentés de retourner une liste (potentiellement longue) des dépendances trouvées. Comment rendre leur exploitation, par un humain, plus facile ? Dans le contexte des dépendances fonctionnelles, la couverture retournée peut être représentée sous forme de matrice ou de manière graphique. Les dépendances fonctionnelles sont un cas particulier de dépendances conditionnelles, or, à notre connaissance, rien n'a été proposé pour visualiser l'ensemble des dépendances conditionnelles.

Toujours dans cet esprit d'analyse facilitée par une bonne visualisation, nous avons évoqué (section 4.1) la possibilité de décrire le cube de données sous forme d'hypercube. Nous connaissons, grâce aux dépendances conditionnelles, les tuples équivalents comme définis par [Lakshmanan *et al.* 2002]. Nous pouvons ainsi choisir les axes et les parties de domaine à rapprocher dans l'espace afin de n'avoir que des

groupes de même valeur à analyser. L'article de [Choong *et al.* 2003] définissant la bonne qualité de l'ordonnancement des axes et celui de [Niemi *et al.* 2003] proposant de réduire le nombre de cases nulles (c.-à-d. sans valeurs définies) serviront de base pour développer cette piste.

Nous avons considéré le problème de la matérialisation partielle de cube de données. Dans le contexte des requêtes de préférence, les skylines (par ex. quel est le meilleur hôtel, c.-à-d. le moins cher et le plus proche de la plage ?), un skycube est également trop volumineux et trop coûteux à calculer pour être produit entièrement. Nous envisageons donc de caractériser, à l'aide des dépendances, les skycuboïdes à matérialiser. Notons $Sky(AB)$ le skyline de T sur les critères A et B et $Sky(A)$ le skyline sur le critère A. Rien ne nous permet d'affirmer à l'avance, comme c'est le cas pour les requêtes de sélection, que $Sky(A) \subseteq Sky(AB)$. Cependant, si $A \rightarrow B$, alors cette inclusion est vérifiée. Les mêmes notions que celles utilisées dans le contexte des cubes de données peuvent donc certainement être appliquées aux skycubes.

Actuellement, la meilleure stratégie pour stocker une masse importante de données consiste à les répartir sur différents serveurs. La matérialisation que nous avons proposée ne tient compte que du contexte locale, comment en déduire une solution globale ? [Bauer & Lehner 2003] et [Shukla *et al.* 2000] traitent déjà du problème de la matérialisation partielle des cubes de données dans un cadre distribué. Nous pouvons alors nous baser sur leurs travaux pour étendre notre caractérisation. Devons nous pour cela extraire les dépendances satisfaites par l'ensemble des données du système ? L'article de [Fan *et al.* 2010] traite de ce problème. Nous devons l'étendre pour considérer également l'extraction des dépendances conditionnelles.

Bibliographie

[Abiteboul *et al.* 1995] Serge Abiteboul, Richard Hull et Victor Vianu. Foundations of databases. Addison-Wesley, 1995. (Cité en page 76.)

[Auber *et al.* 2012] David Auber, Daniel Archambault, Romain Bourqui, Antoine Lambert, Morgan Mathiaut, Patrick Mary, Maylis Delest, Jonathan Dubois et Guy Mélançon. *The tulip 3 framework : A scalable software library for information visualization applications based on relational data.* Rapport technique, INRIA, 2012. (Cité en page 51.)

[Bauer & Lehner 2003] Andreas Bauer et Wolfgang Lehner. *On Solving the View Selection Problem in Distributed Data Warehouse Architectures.* In Proceedings of SSDBM conference, pages 43–54. IEEE Computer Society, 2003. (Cité en pages 59 et 94.)

[Bohannon *et al.* 2007] Philip Bohannon, Wenfei Fan, Floris Geerts, Xibei Jia et Anastasios Kementsietsidis. *Conditional Functional Dependencies for Data Cleaning.* In Proceedings of ICDE conference, pages 746–755. IEEE, 2007. (Cité en pages 7 et 8.)

[Bruno 2011] Nicolas Bruno. Automated physical database design and tuning. CRC Press inc, 2011. (Cité en page 59.)

[Casali *et al.* 2003] Alain Casali, Rosine Cicchetti et Lotfi Lakhal. *Cube Lattices : A Framework for Multidimensional Data Mining.* In Proceedings of Conference on Data Mining, SDM. SIAM, 2003. (Cité en pages 50 et 92.)

[Casali *et al.* 2009] Alain Casali, Sébastien Nedjar, Rosine Cicchetti et Lotfi Lakhal. *Closed Cube Lattices.* In New Trends in Data Warehousing and Data Analysis, volume 3 of *Annals of Information Systems*, pages 1–20. Springer, 2009. (Cité en page 91.)

[Caspard *et al.* 2007] Nathalie Caspard, Bruno Leclerc et Bernard Monjardet. Ensembles ordonnés finis : concepts, résultats et usages, volume 60. Springer, 2007. (Cité en page 48.)

[Chaudhuri 1998] Surajit Chaudhuri. *An Overview of Query Optimization in Relational Systems.* In Proceedings of PODS conference, pages 34–43. ACM, 1998. (Cité en page 52.)

[Chiang & Miller 2008] Fei Chiang et Renée J. Miller. *Discovering Data Quality Rules.* In Proceedings of PVLDB conference, pages 1166–1177, 2008. (Cité en pages 7 et 13.)

[Choong *et al.* 2003] Yeow Wei Choong, Dominique Laurent et Patrick Marcel. *Computing appropriate representations for multidimensional data.* Data Knowl. Eng., vol. 45, no. 2, pages 181–203, 2003. (Cité en page 94.)

[Chvatàl 1979] Vasek Chvatàl. *A greedy heuristic for the set-covering problem.* Mathematics of Operations Research, vol. 4, no. 3, pages 233–235, 1979. (Cité en page 78.)

[Codd 1971] Edgar F. Codd. *Normalized Data Base Structure : A Brief Tutorial.* IBM Research Report, San Jose, California, vol. RJ935, 1971. (Cité en pages 2 et 6.)

[Codd 1983] Edgar F. Codd. *A Relational Model of Data for Large Shared Data Banks (Reprint).* Communications of ACM, vol. 26, no. 1, pages 64–69, 1983. (Cité en page 6.)

[Cormode *et al.* 2009] Graham Cormode, Lukasz Golab, Flip Korn, Andrew McGregor, Divesh Srivastava et Xi Zhang. *Estimating the confidence of conditional functional dependencies.* In Proceedings of SIGMOD Conference, pages 469–482. ACM, 2009. (Cité en page 16.)

[De Bra & Paredaens 1983] Paul De Bra et Jan Paredaens. *Conditional Dependencies for Horizontal Decompositions.* In Proceedings of ICALP conference, volume 154 of *LNCS*, pages 67–82. Springer, 1983. (Cité en page 6.)

[Diallo *et al.* 2012a] Thierno Diallo, Noel Novelli et Jean Marc Petit. *Discovering (frequent) constant conditional functional dependencies.* International Journal of Data Mining, Modelling and Management, vol. 4, no. 3, pages 205–223, 2012. (Cité en page 14.)

[Diallo *et al.* 2012b] Thierno Diallo, Jean-Marc Petit et Sylvie Servigne. *Discovering Editing Rules For Data Cleaning.* In Proceedings of AQB conference, pages 40–48. Purdue University, 2012. (Cité en page 7.)

[Fan *et al.* 2010] Wenfei Fan, Floris Geerts, Shuai Ma et Heiko Müller. *Detecting inconsistencies in distributed data.* In Proceedings of ICDE conference, pages 64–75. IEEE, 2010. (Cité en pages 7 et 94.)

[Fan *et al.* 2011] Wenfei Fan, Floris Geerts, Jianzhong Li et Ming Xiong. *Discovering Conditional Functional Dependencies.* IEEE Transactions on Knowledge and Data Engineering, vol. 23, no. 5, pages 683–698, 2011. (Cité en pages 13, 40 et 41.)

[Fan *et al.* 2012] Wenfei Fan, Jianzhong Li, Nan Tang et Wenyuan Yu. *Incremental Detection of Inconsistencies in Distributed Data.* In Proceedings of ICDE conference, pages 318–329. IEEE Computer Society, 2012. (Cité en page 7.)

[Flajolet *et al.* 2007] Philippe Flajolet, Éric Fusy, Olivier Gandouet et Frédéric Meunier. *HyperLogLog : the analysis of a near-optimal cardinality estimation algorithm.* In Proceedings of AofA conference, page 127–146. DMTCS, 2007. (Cité en page 17.)

[Garnaud *et al.* 2011] Eve Garnaud, Sofian Maabout et Mohamed Mosbah. *Dépendances fonctionnelles et matérialisation partielle des cubes de données.* In Proceedings of EDA conference, volume B-7 of *RNTI*, pages 155–169. Hermann, 2011. (Cité en page 91.)

[Garnaud *et al.* 2012a] Eve Garnaud, Sofian Maabout et Mohamed Mosbah. *Les dépendances fonctionnelles pour la sélection de vues dans les cubes de données.* Journal of Decision Systems, vol. 21, pages 71–91, 2012. (Cité en page 91.)

[Garnaud *et al.* 2012b] Eve Garnaud, Sofian Maabout et Mohamed Mosbah. *Using Functional Dependencies for Reducing the Size of a Data Cube.* In Proceedings of FoIKS conference, pages 144–163. Springer, 2012. (Cité en page 91.)

[Garnaud *et al.* 2013a] Eve Garnaud, Nicolas Hanusse, Sofian Maabout et Noel Novelli. *Calcul parallèle de dépendances.* In Proceedings of BDA conférence, pages 1–20, 2013. (Cité en page 43.)

[Garnaud *et al.* 2013b] Eve Garnaud, Sofian Maabout et Mohamed Mosbah. *Functional dependencies are helpful for partial materialization of data cubes.* Annals of Mathematics and Artificial Intelligence, pages 1–30, 2013. (Cité en page 91.)

[Giannella & Robertson 2004] Chris Giannella et Edward L. Robertson. *On approximation measures for functional dependencies.* Information Systems, vol. 29, no. 6, pages 483–507, 2004. (Cité en pages 5 et 15.)

[Gibbons 2001] Phillip B. Gibbons. *Distinct Sampling for Highly-Accurate Answers to Distinct Values Queries and Event Reports.* In Proceedings of VLDB conference, pages 541–550. Morgan Kaufmann, 2001. (Cité en page 16.)

[Golab *et al.* 2008] Lukasz Golab, Howard J. Karloff, Flip Korn, Divesh Srivastava et Bei Yu. *On Generating Near-Optimal Tableaux for Conditional Functional Dependencies.* Proceedings VLDB conference, vol. 1, no. 1, pages 376–390, 2008. (Cité en page 17.)

[Graefe 1993] Goetz Graefe. *Query Evaluation Techniques for Large Databases.* ACM Computing Surveys, vol. 25, no. 2, pages 73–170, 1993. (Cité en page 52.)

[Gray *et al.* 1997] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow et Hamid Pirahesh. *Data Cube : A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals.* Data Mining and Knowledge Discovery, vol. 1, no. 1, pages 29–53, 1997. (Cité en pages 46 et 47.)

[Halevy 2001] Alon Halevy. *Answering queries using views : A survey.* VLDB J., vol. 10, no. 4, pages 270–294, 2001. (Cité en page 59.)

[Hanusse & Maabout 2011] Nicolas Hanusse et Sofian Maabout. *A parallel algorithm for computing borders.* In Proceedings of CIKM conference, pages 1639–1648. ACM, 2011. (Cité en pages 14, 20 et 32.)

[Hanusse *et al.* 2009a] Nicolas Hanusse, Sofian Maabout et Radu Tofan. *Algorithmes pour la sélection des vues à matérialiser avec garantie de performance.* In Proceedings of EDA conference, pages 107–122. Cépaduès, 2009. (Cité en page 68.)

[Hanusse *et al.* 2009b] Nicolas Hanusse, Sofian Maabout et Radu Tofan. *A view selection algorithm with performance guarantee.* In Proceedings of EDBT conference, volume 360, pages 946–957. ACM, 2009. (Cité en page 58.)

[Hanusse *et al.* 2011] Nicolas Hanusse, Sofian Maabout et Radu Tofan. *Revisiting the Partial Data Cube Materialization.* In Proceedings of ADBIS conference, pages 70–83. Springer, 2011. (Cité en pages 57 et 58.)

[Harinarayan *et al.* 1996] Venky Harinarayan, Anand Rajaraman et Jeffrey D. Ullman. *Implementing data cubes efficiently.* In Proceedings of SIGMOD conference, pages 205–216. ACM Press, 1996. (Cité en page 54.)

[Hose *et al.* 2009] Katja Hose, Daniel Klan et Kai-Uwe Sattler. *Online Tuning of Aggregation Tables for OLAP.* In Proceedings of ICDE conference, pages 1679–1686. IEEE, 2009. (Cité en page 58.)

[Huang *et al.* 2012] Rong Huang, Rada Chirkova et Yahya Fathi. *Deterministic View Selection for Data-Analysis Queries : Properties and Algorithms.* In Proceedings of ADBIS conference. Springer, 2012. (Cité en page 57.)

[Huhtala *et al.* 1999] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka et Hannu Toivonen. *TANE : An Efficient Algorithm for Discovering Functional and Approximate Dependencies.* Computer Journal, vol. 42, no. 2, pages 100–111, 1999. (Cité en pages 3, 12 et 13.)

[Karloff & Mihail 1999] Howard J. Karloff et Milena Mihail. *On the complexity of the view-selection problem.* In Proceedings of PODS conference, pages 167–173. ACM, 1999. (Cité en page 56.)

[Kivinen & Mannila 1995] Jyrki Kivinen et Heikki Mannila. *Approximate Inference of Functional Dependencies from Relations.* Theoretical Computer Science, vol. 149, no. 1, pages 129–149, 1995. (Cité en pages 5 et 16.)

[Kormilitsin *et al.* 2007] Maxim Kormilitsin, Rada Chirkova, Yahya Fathi et Matthias Stallmann. *View and index selection for query-performance improvement : algorithms, heuristics and complexity.* Rapport technique, Technical report, NC State University, 2007. (Cité en page 59.)

[Kotidis & Roussopoulos 1999] Yannis Kotidis et Nick Roussopoulos. *DynaMat : a dynamic view management system for data warehouses.* ACM SIGMOD Record, vol. 28, no. 2, pages 371–382, 1999. (Cité en page 58.)

[Lafon *et al.* 2012] Sébastien Lafon, Fatma Bouali, Christiane Guinot et Gilles Venturini. *Réorganisation hiérarchique de visualisations dans OLAP.* In Proceedings of EGC conference, pages 411–422. Hermann, 2012. (Cité en page 51.)

[Lakshmanan *et al.* 2002] Laks V. S. Lakshmanan, Jian Pei et Jiawei Han. *Quotient Cube : How to Summarize the Semantics of a Data Cube.* In Proceedings of VLDB conference, pages 778–789. VLDB Endowment, 2002. (Cité en pages 86 et 93.)

[Laurent & Spyratos 2011] Dominique Laurent et Nicolas Spyratos. *Rewriting aggregate queries using functional dependencies.* In Proceedings of MEDES conference, pages 40–47. ACM, 2011. (Cité en page 59.)

[Lehner *et al.* 1998] Wolfgang Lehner, Jens Albrecht et Hartmut Wedekind. *Normal Forms for Multidimensional Databases.* In Proceedings of SSDBM conference, pages 63–72. IEEE Computer Society, 1998. (Cité en page 51.)

[Li *et al.* 2005] Jingni Li, Zohreh A. Talebi, Rada Chirkova et Yahya Fathi. *A formal model for the problem of view selection for aggregate queries.* In Proceedings of ADBIS conference, pages 125–138. Springer, 2005. (Cité en pages 54 et 76.)

[Liu *et al.* 2012] Jixue Liu, Jiuyong Li, Chengfei Liu et Yongfeng Chen. *Discover Dependencies from Data - A Review.* IEEE Trans. Knowl. Data Eng., vol. 24, no. 2, pages 251–264, 2012. (Cité en page 11.)

[Lopes *et al.* 2000] Stéphane Lopes, Jean-Marc Petit et Lotfi Lakhal. *Efficient Discovery of Functional Dependencies and Armstrong Relations.* In Proceedings of EDBT conference, pages 350–364. Springer, 2000. (Cité en page 12.)

[Lopes *et al.* 2001] Stéphane Lopes, J-M Petit et Lotfi Lakhal. *A framework for understanding existing databases.* In Proceedings of Database Engineering & Applications conference, pages 330–336. IEEE, 2001. (Cité en page 15.)

[Mannila & Räihä 1992] Heikki Mannila et Kari-Jouko Räihä. *On the Complexity of Inferring Functional Dependencies.* Discrete Applied Mathematics, vol. 40, no. 2, pages 237–243, 1992. (Cité en page 11.)

[Messaoud *et al.* 2006] Riadh Ben Messaoud, Omar Boussaid et Sabine Loudcher Rabaséda. *A Multiple Correspondence Analysis to Organize Data Cubes.* In Proceedings of DB&IS conference, pages 133–146. IOS Press, 2006. (Cité en page 51.)

[Nandi *et al.* 2011] Arnab Nandi, Cong Yu, Philip Bohannon et Raghu Ramakrishnan. *Distributed cube materialization on holistic measures.* In Proceedings of ICDE conference. IEEE Computer Society, 2011. (Cité en page 59.)

[Niemi *et al.* 2003] Tapio Niemi, Jyrki Nummenmaa et Peter Thanisch. *Normalising OLAP Cubes for Controlling Sparsity.* Data & Knowledge Engineering, vol. 46, no. 3, pages 317–343, 2003. (Cité en pages 51 et 94.)

[Novelli & Cicchetti 2001] Noel Novelli et Rosine Cicchetti. *FUN : An Efficient Algorithm for Mining Functional and Embedded Dependencies.* In Proceedings of ICDT conference, pages 189–203. Springer, 2001. (Cité en page 14.)

[Page 2009] Wim Le Page. *Mining Patterns in Relational Databases.* PhD thesis, University of Antwerp, 2009. (Cité en page 15.)

[Pottinger & Halevy 2001] Rachel Pottinger et Alon Halevy. *MiniCon : A scalable algorithm for answering queries using views.* VLDB J., vol. 10, no. 2-3, pages 182–198, 2001. (Cité en page 59.)

[Shukla *et al.* 2000] Amit Shukla, Prasad Deshpande et Jeffrey F. Naughton. *Materialized View Selection for Multi-Cube Data Models.* In Proceedings of EDBT conference, volume 1777 of *LNCS*, pages 269–284. Springer, 2000. (Cité en pages 59 et 94.)

[Talebi *et al.* 2008] Zohreh A. Talebi, Rada Chirkova, Yahya Fathi et Matthias Stallmann. *Exact and inexact methods for selecting views and indexes for OLAP performance improvement.* In Proceedings of EDBT conference, pages 311–322. ACM, 2008. (Cité en page 59.)

[Talebi *et al.* 2009] Zohreh Asgharzadeh Talebi, Rada Chirkova et Yahya Fathi. *Exact and inexact methods for solving the problem of view selection for aggregate queries.* International Journal of Business Intelligence and Data Mining, vol. 4, no. 3, pages 391–415, 2009. (Cité en page 56.)

[Vitter 1985] Jeffrey Scott Vitter. *Random Sampling with a Reservoir*. ACM Trans. Math. Softw., vol. 11, no. 1, pages 37–57, 1985. (Cité en page 16.)

[Wang *et al.* 2002] Wei Wang, Hongjun Lu, Jianlin Feng et Jeffrey Xu Yu. *Condensed Cube : An Effective Approach to Reducing Data Cube Size*. In Proceedings of ICDE conference, pages 155–165. IEEE, 2002. (Cité en pages 82 et 87.)

[Wyss *et al.* 2001] Catharine M. Wyss, Chris Giannella et Edward L. Robertson. *FastFDs : A Heuristic-Driven, Depth-First Algorithm for Mining Functional Dependencies from Relation Instances - Extended Abstract*. In Proceedings of DaWaK conference, pages 101–110. Springer, 2001. (Cité en page 13.)

[Yao & Hamilton 2008] H. Yao et H.J. Hamilton. *Mining functional dependencies from data*. Data Mining and Knowledge Discovery, vol. 16, no. 2, pages 197–219, 2008. (Cité en pages 12 et 43.)