École doctorale de Mathématiques et Informatique de Marseille

Cubes Émergents pour l'analyse des renversements de tendances dans les bases de données multidimensionnelles

THÈSE

présentée et soutenue publiquement le 01 décembre 2009

pour l'obtention du

Doctorat Aix-Marseille Université
délivré par l'Université de la Méditerranée
Spécialité Informatique

par

Sébastien NEDJAR

Composition du jury

Président :	Le Président	
Rapporteurs :	Nicolas Spyratos,	Professeur à l'Université Paris-Sud (Orsay)
	Jérôme Darmont,	Professeur à l'Université Lumière Lyon 2
Examinateurs :	L'examinateur 1	???
	Rosine Cicchetti,	Professeur à l'Université de la Méditerranée
Directeur :	Lotfi Lakhal,	Professeur à l'Université de la Méditerranée

Laboratoire d'Informatique Fondamentale de Marseille – CNRS UMR 6166
Faculté des Sciences de Luminy – Université de la Méditérannée

Mis en page avec la classe thloria.

Remerciements

Merci !

Je dédie cette thèse

Table des matières

Table des figures

Liste des tableaux

1

Introduction

Sommaire

À partir d'énormes volumes de données collectées ou « population », l'utilisateur d'un entrepôt de données souhaite dégager les tendances qui s'y dessinent, selon tels ou tels critères d'analyse, afin de mieux étayer ses prises de décision. Les requêtes Olap permettant de calculer de telles tendances sont complexes pour plusieurs raisons. Tout d'abord, elles mettent en jeu des opérations coûteuses, visant à exhiber des tendances générales dans la population étudiée et qui sont fondamentalement des opérations d'agrégation. Ensuite, elles s'appliquent sur des ensembles massifs de données qu'il s'agit de lire à partir d'une mémoire secondaire, vraisemblablement en procédant par fragmentation, puis de traiter. Enfin ces requêtes génèrent des résultats incomparablement plus volumineux que les données en entrée si bien que l'écriture sur disque des résultats représente un facteur important de coût. Une autre facette du problème est que l'utilisateur d'un entrepôt exige un temps de réponse rapide à ses requêtes. Pour résoudre ce dilemme, le concept de cube de données a été introduit (Gray *et al.*, 1997). À partir d'une relation, l'idée sous-jacente est de pré-calculer tous les agrégats qu'il est possible de calculer en combinant un ensemble d'attributs qui sont les critères de décision pour l'utilisateur. Ces attributs sont baptisés dimensions. Originellement, le cube de données a été proposé comme le résultat d'une requête combinant les Group-By selon toutes les combinaisons de dimensions (Gray *et al.*, 1997). Le résultat de chaque Group-By est appelé un cuboïde. À partir des partitions générées par les Group-By, les agrégats sont obtenus en appliquant des fonctions statistiques sur certains attributs mesures ou en effectuant le dénombrement des tuples. Ainsi, à partir d'une relation initiale dotée de dimensions et de mesures, son cube regroupe les mêmes informations mais à tous les niveaux de granularité possibles, du plus détaillé (la relation elle-même) au plus agrégé, lorsque la totalité de cette relation est résumée en un unique tuple. Disposant d'un tel cube, il est alors possible de répondre efficacement à toute requête de l'utilisateur, bien qu'elle soit ad hoc, concernant des tendances générales se dessinant dans les données originelles. Effectivement, tous les pré-calculs ayant été réalisés, le résultat des requêtes Olap peut être obtenu en mettant en œuvre des opérations simples et peu coûteuses. Le cube correspond à la table des faits, centrale dans le schéma Star d'un entrepôt.

Nos travaux de recherche se situent dans ce contexte général de l'Olap. Nous proposons une

méthode d'extraction des renversements de tendances en adoptant une approche data cube. Le résultat est un Cube Émergent (Nedjar *et al.*, 2009) qui permet, en comparant deux cubes différents, d'exhiber de nouvelles connaissances. La démarche que nous développons se veut la plus complète possible. Elle englobe différentes représentations du Cube Émergent permettant une réduction notable de l'espace de stockage requis et dédiées à différents usages. Elle intègre plusieurs algorithmes de calcul de ces représentations et inclut une approche d'évaluation de la taille des Cubes Émergents. Avant de décrire plus en détail nos contributions, nous proposons un survol des travaux antérieurs.

1.1 Survol des travaux existants

Les approches existantes, que nous nous sommes proposés d'étudier et de présenter, relèvent de trois problématiques différentes et retracent l'évolution chronologique des recherches dans le domaine de la gestion des entrepôts de données. Ces trois problématiques sont les suivantes : le calcul efficace de cubes de données, leur représentation solidement fondée et une approche d'analyse multi-critère issue des bases de données.

Dans le contexte OLAP, il est primordial de proposer des approches de calcul de cubes performantes (Agarwal *et al.*, 1996). De nombreux algorithmes ont été conçus pour optimiser les temps de calcul nécessaires. Ils doivent réaliser les agrégations requises pour chaque combinaison de dimensions. Or toutes ces combinaisons constituent l'ensemble des parties de l'ensemble des dimensions qui est classiquement représenté sous forme d'un treillis : le treillis des cuboïdes.
Pour les algorithmes de la première génération, cette structure est un outil permettant d'organiser au mieux les différentes étapes du calcul et d'exploiter, au mieux, les résultats intermédaires. Cette ré-utilisation induit évidemment un « parcours » de bas en haut de la structure, i.e. des cuboïdes les plus détaillés vers les plus agrégés, de manière à pouvoir obtenir un cuboïde, non plus à partir de la relation originelle, mais à partir des résultats forcément de taille plus réduite d'un autre cuboïde. Différents algorithmes ont été proposés s'appuyant sur des techniques déjà utilisées pour implémenter le GROUP-BY, telles que les tris ou le hachage (Gray *et al.*, 1997; Agarwal *et al.*, 1996)

Après ces approches, l'algorithme BUC (Beyer et Ramakrishnan, 1999) a réellement innové et introduit une deuxième génération d'approches. La première nouveauté de BUC est d'inverser totalement le calcul des cuboïdes en commençant par le plus agrégé possible. Dans ces conditions, il est impossible de ré-utiliser des résultats intermédaires puisqu'il est impossible de « desagréger » des données. Néanmoins, BUC est plus performant que toutes les démarches précédentes. De plus, en combinant les concepts de cube et de motif fréquent, BUC introduit un nouveau type de data cube : le cube iceberg. À partir du cube de données complet d'une relation, le cube iceberg ne retient que les tendances suffisamment pertinentes pour l'utilisateur, i.e. caractérisées par une valeur de la mesure au delà d'un certain seuil. Soulignons que, parmi tous les algorithmes efficaces, BUC est le seul à être parfaitement intégrable au sein d'un SGBD (il n'utilise pas de structure de données trop sophistiquée) tout en restant relativement simple.
Une attention particulière doit également être apportée à l'algorithme PARTIONNED-CUBE (Ross et Srivastava, 1997) qui propose une technique pour fragmenter des données trop volumineuses pour être chargées en mémoire centrale, les traiter fragment après fragment et enfin fusionner les résultats obtenus. Cette approche est tellement efficace qu'elle est considérée comme une étape incontournable par tout algorithme de calcul de cubes (en particulier BUC).

Il est intéressant de souligner que jusqu'à présent, pour les différentes approches citées, le cube n'est vu que comme un résultat et non comme une structure et que l'espace de recherche, exploré lors du calcul, n'a pas été formellement défini. Il est vrai que ces approches plutôt pragmatiques ont surtout visé l'efficacité des calculs et ont, proposition après proposition, repousser les limites du temps d'exécution requis.
Les deux premières démarches, à avoir formaliser une représentation des cubes de données, ont été proposées concurremment et ont, tout d'abord, défini un espace de recherche [1] réellement adapté au problème de calcul du cube. Cet espace, appelé treillis cube (Lakshmanan *et al.*, 2002; Casali *et al.*, 2003a), organise les tuples représentant les tendances selon un ordre de généralisation. Ces approches offrent, toutes deux, une représentation réduite du cube en éliminant les redondances pouvant exister au sein des cubes de données. L'idée sous-jacente est de déterminer un ou des représentants pour chaque sous-ensemble de tuples redondants de manière à ne préserver que ce ou ces représentants au lieu de conserver la totalité du sous-ensemble. Il en résulte évidemment un gain d'espace de stockage. Les représentations introduites sont le cube fermé (Casali *et al.*, 2003b) d'une part, le cube quotient (Lakshmanan *et al.*, 2002) d'autre part. Le cube fermé est, à notre connaissance, la plus petite représentation sans perte d'information du cube de données. Le cube quotient offre une réduction moindre mais en revanche il préserve la capacité de naviguer entre les tuples de différents niveaux de granularité. À ces deux approches de réduction sémantique des cubes, nous avons ajouté une représentation réduite, l'approche PCube (Casali *et al.*, 2009), qui relève plutôt du niveau logique et que nous qualifions pour cela de réduction syntaxique. Elle met en œuvre une idée simple : il est inutile, pour chaque tuple du cube de données, de stocker la totalité des valeurs des attributs dimensions alors qu'il suffit pour les retrouver d'une référence à un tuple de la relation originelle partageant ces mêmes valeurs. Là encore, la notion de représentativité est primordiale pour réduire le volume d'informations nécessaires. Mais là, il relève du niveau logique.

Pour terminer cet état de l'art, nous faisons une brève incursion hors du domaine de l'Olap, en présentant le concept de Skyline(enveloppe de Pareto). Lorqu'une requête porte sur plusieurs critères et qu'elle cherche à tous les optimiser simultanément, elle ne va, dans la très grande majorité des cas, rendre aucun résultat. En effet, il est exceptionnel qu'un tuple (ou a fortiori un sous-ensemble de tuples) maximise et/ou minimise l'ensemble de tous les critères. Pour résoudre un tel problème, l'opérateur Skyline (Börzsönyi *et al.*, 2001) introduit une relation de dominance entre tuples (un tuple est non dominé s'il n'existe aucune autre meilleure solution) et ne retient comme résultat que les tuples qui ne sont dominés par aucun autre. Cette incursion, hors du domaine de l'Olap, est motivée par le fait que le concept de Skyline a été combiné à celui du cube de données pour donner naissance au concept de SkyCube (Pei *et al.*, 2006). À partir d'un ensemble d'attributs critères à optimiser, il s'agit de réunir l'ensemble des Skylines, selon toutes les combinaisons possibles de ces critères, de manière à pouvoir répondre efficacement à toute requête de l'utilisateur en recherchant les objets dominants selon diverses combinaisons de critères. De plus grâce à cette structure, il est possible d'observer le comportement des objets dominants à travers l'espace multidimensionnel et de mieux comprendre les facteurs de dominance.

Sans être un panorama exhaustif des recherches menées dans le domaine de l'Olap, l'état de l'art présenté dans ce mémoire retrace les travaux effectués sur les cubes de données selon trois

1. Nos propres propositions s'inscrivent également dans ce cadre.

axes, celui du calcul, celui de la représentation formelle et celui d'une nouveau type d'analyse. C'est dans ce cadre de travail général que se situent nos contributions dont une description résumée est donnée dans le paragraphe suivant.

1.2 Synthèse des contributions

Les approches Olap que nous avons présentées ont permis, en calculant ou représentant un cube, au décideur de faire de l'analyse de tendances. Avec le SkyCube, il est possible de procéder à l'analyse de préférences. Dans le même esprit, notre démarche vise l'analyse des renversements de tendances.

Comme le cube iceberg (Beyer et Ramakrishnan, 1999) qui combine cube et motif fréquent (Agrawal *et al.*, 1996) ou que le SkyCube associant enveloppe de Pareto et data cube, le travail que nous avons proposé fait une synthèse entre le concept de cube de données et celui de motif émergent (Dong et Li, 2005), introduit en fouille de données binaires. En effet nous introduisons les Cubes Émergents. Supposons que nous disposons de deux relations de schéma comparable et imaginons qu'un cube a été calculé à partir de chacune d'elles. Il peut être extrêmement intéressant de pouvoir comparer ces deux cubes afin de détecter les tendances non significatives dans un cube mais très pertinentes dans l'autre. Nous appelons de telles tendances des tendances émergentes et elles sont réunies au sein du Cube Émergent. De telles tendances ont, dans le cube de la première relation, une valeur de la mesure non intéressante pour le décideur (car inférieure à un seuil donné). En revanche, elles sont dotées, dans le cube de la deuxième relation, d'une valeur de la mesure supérieure à un seuil. Nous appelons cette double condition la contrainte d'émergence. Dans diverses applications, la connaissance des tendances émergentes présente un intérêt certain. Par exemple, si dans les deux relations sont collectées des données décrivant des « populations » différentes, calculer le Cube Émergent met en exergue les différences marquées entre les caractéristiques de ces deux populations ou entre leur comportement suivant la nature des données répertoriées. Par exemple, il sera possible de comparer l'état de santé d'une population méditerranéenne et celui d'une population nordiste puis éventuellement de détecter leurs différences alimentaires.
Si l'une des relations décrit un échantillon type et l'autre une « population » ciblée, il est possible de mettre en évidence les écarts ou déviations entre cette population et l'échantillon type. Par exemple, nous pourrons étudier les différences significatives entre les populations méditerranéenne et nordiste et celle pratiquant au quotidien un régime chypriote (l'échantillon type dont la durée de vie est exceptionnelle).
Nous pouvons aussi considérer que les deux relations stockent des données relatives à une même « population » mais collectées sur différentes périodes. Dans ce contexte, les tendances émergentes vont souligner les évolutions, au cours du temps, de cette population. Nouvelles habitudes alimentaires ou nouveaux problèmes de santé peuvent alors être décelés. Si des flots de données sont collectés, comme dans n'importe quelle application Web, l'administrateur du site, disposant du Cube Émergent, sera informé des engouements (ou au contraire des desintérêts) pour tel ou tel produit, telle ou telle page ou URL, et pourra au mieux gérer les stocks ou allouer les ressources nécessaires.
L'importance de la chronologie est fondamentale dans la gestion d'entrepôt de données. En effet, lors de son insertion dans un entrepôt, toute nouvelle donnée reçoit une estampille temporelle au cours d'une opération de mise à jour appelée rafraîchissement. Dans ce contexte, les deux relations permettant le calcul d'un Cube Émergent sont d'une part l'ensemble des données col-

lectées jusqu'à présent et d'autre part les données correspondant au rafraîchissement. Dès lors, le Cube Émergent capture des tendances non appuyées jusqu'à présent mais qui deviennent significatives (ou, à l'inverse des tendances importantes mais qui s'effrondent au point de devenir insignifiantes).

Calculer un Cube Émergent est un problème difficile car il s'agit de calculer les cubes de données de deux relations puis de les comparer. Comme souligné précédemment, le calcul de chacun des cubes est coûteux et il est vraisemblable que la comparaison des deux cubes a un coût non négligeable car la taille de chacun d'entre eux est réellement importante. Dès lors, il devient fondamental pour effectivement exploiter les nouvelles connaissances véhiculées par un Cube Émergent d'éviter de calculer chacun des deux cubes de données concernés et pouvoir les comparer à moindre coût.

En fouille de données, de nombreux travaux ont porté sur des représentations réduites et démontré leur intérêt. Citons par exemple les couvertures pour les motifs fréquents (Pasquier *et al.*, 1999; Zaki et Hsiao, 2005). Des approches de calcul ou de représentation de cubes ont, elles aussi, cherché à réduire la taille des cubes de données. Nous les avons évoqué au paragraphe précédent. Il était naturel d'explorer ces représentations réduites pour le Cube Émergent et il était tout aussi naturel de commencer par transposer, dans notre contexte de travail, les bordures L et U (pour Lower et Upper) classiquement utilisées en fouille de données (Raedt et Kramer, 2001). Elles correspondent respectivement aux tuples minimaux et aux tuples maximaux satisfaisant la contrainte d'émergence. Elles représentent les frontières de l'espace de recherche et constituent effectivement une représentation très réduite des Cubes Émergents. Dans le même esprit, nous introduisons un nouveau couple de bordures. Au lieu de considérer les tuples minimaux satisfaisant la contrainte d'émergence (i.e. L), nous prenons en compte les tuples maximaux ne la satisfaisant pas. Cette nouvelle frontière basse, appelée $U^\sharp$ (Nedjar *et al.*, 2008) se situe, dans le treillis cube, juste en dessous de la bordure L. Bien que très proche de L, la bordure $U^\sharp$ est plus réduite en pratique que L. De plus, elle nous permet de donner une caractérisation de la taille exacte des Cubes Émergents (Nedjar, 2009). Nous avons également établi le lien entre les bordures L et $U^\sharp$ (Nedjar *et al.*, 2007a). Il s'appuie sur le concept de minimal transversal cubique (Casali *et al.*, 2003b) dont nous étudions la complexité du calcul. Outre le résultat théorique et son intérêt si l'on dispose du couple de bordures classiques et que nos nouvelles bordures s'avèrent plus intéressantes, l'intérêt pratique est de pouvoir ré-utiliser des algorithmes existants pour obtenir la bordure L.
Pour pouvoir quantifier la réduction apportée, nous avons procédé à des évaluations expérimentales sur des jeux de données variées, réelles ou synthétiques, visant à mesurer et comparer la taille du Cube Émergent et celle de chacune des représentations par bordures. Les résultats sont ceux que nous attendions puisque chacun des couples de bordures apporte une réduction extrêment importante et que $U^\sharp$ est, de manière significative, plus restreint que L. Les bordures proposées permettent de répondre à des requêtes du type : "est-ce que cette tendance est émergente ?". De plus, à l'instar des motifs émergents, elles peuvent être des classifieurs efficaces pour répondre systématiquement à ce type de requête. Malheureusement, les bordures, qu'elles qu'elles soient, ne sont pas des représentations sans perte d'information : les valeurs de l'attribut mesure ne peuvent pas être retrouvées.

Aussi, nous nous sommes intéressés à des représentations sans perte, permettant d'obtenir toute valeur de mesure pour les tendances émergentes. Naturellement, nous nous sommes tournés vers les travaux s'intéressant à la modélisation des cubes de données permettant une réduction si-

gnificative de l'information à stocker. Parmi eux, deux approches ont retenu notre attention. La première, le cube fermé (Casali *et al.*, 2003b), est, à notre connaissance, la représentation sans perte d'information la plus réduite possible d'un cube de données. La seconde, le cube quotient (Lakshmanan *et al.*, 2002), offre la possibilité de naviguer à l'intérieur d'un cube.

En nous appuyant sur le concept de la fermeture cubique (Casali *et al.*, 2003b), nous proposons une famille de représentations. Chacune d'elles inclut l'ensemble des tuples fermés émergents. Le calcul de tels tuples s'appuie sur la fermeture cubique. À partir d'un ensemble de tuples véhiculant la même sémantique mais à différents niveaux de granularité, la fermeture cubique permet d'en désigner un représentant. Ce dernier contient toute l'information nécessaire pour redériver l'ensemble de tuples initial et peut donc être seul préservé. Néanmoins, l'ensemble des tuples fermés émergents ne constituent pas une représentation sans perte. Nous montrons qu'il faut lui adjoindre des informations complémentaires, heureusement contenues dans l'une des bordures basses précédemment décrites. Ainsi, nous définissons le L-Cube Fermé Émergent (Nedjar *et al.*, 2010) et le $U^\sharp$-Cube Fermé Émergent (Nedjar *et al.*, 2007b). De plus, nous constatons que $U^\sharp$ inclut des redondances qui peuvent être identifiées et éliminées. En écartant ces redondances de $U^\sharp$, nous obtenons une nouvelle bordure appelée $U^{\sharp\sharp}$ dont la taille est encore plus réduite que celle de $U^\sharp$. Nous proposons alors le $U^{\sharp\sharp}$-Cube Fermé Émergent (Nedjar *et al.*, 2009) qui est la plus petite représentation dans cette famille.

Avec ces trois représentations, il est possible de répondre à toute requête Olap qui pourrait être posée sur le Cube Émergent lui-même. Ainsi, en conservant le minimum d'informations et en s'épargnant le calcul des deux cubes, il est possible de savoir quelles tendances ont émergé entre les deux cubes et de les quantifier par la valeur de leur mesure.

Certaines applications Olap nécessitent, en plus de l'expression de requêtes, des capacités de navigation au sein des cubes pour observer les données à différents niveaux de granularité. Par exemple, si un important renversement de tendances apparaît dans un tuple très agrégé, l'utilisateur peut vouloir comprendre l'origine du phénomène et « forer » le Cube Émergent pour retrouver les tuples plus détaillés qui l'expliquent. Avec les représentations basées sur le Cube Fermé Émergent, ces navigations ne sont pas possibles. En revanche, le cube quotient offre de telles capacités. Adapter cette structure pour isoler les renversements de tendances ne suffit pas pour caractériser le Cube Quotient Émergent. Nous avons établi le lien entre cube quotient et fermeture cubique et revisité les définitions originelles du cube quotient. À partir de ce résultat, nous avons proposé une nouvelle structure, le Cube Quotient Émergent (Nedjar *et al.*, 2010). En le définissant nous poursuivons les mêmes objectifs de réduction de la redondance que pour le Cube Fermé Émergent. Cependant pour un ensemble de tuples redondants, au lieu d'en préserver un seul représentant comme le Cube Fermé, nous conservons les frontières de cet ensemble (tuples minimaux et tuples maximaux).

Nous avons également établi le lien existant entre le couple de bordures (L, U), le L-Cube Fermé Émergent et le Cube Quotient Émergent. Ce lien est celui d'inclusion (Nedjar *et al.*, 2010). Les bordures sont contenues dans le L-Cube Fermé Émergent, lui-même étant un sous-ensemble du Cube Quotient Émergent. Ces inclusions illustrent particulièrement bien la dépendance entre les fonctionnalités proposées et le volume d'information requis : plus riches sont les fonctionnalités, plus volumineuses sont les informations à préserver. Ainsi, les représentations proposées se révèlent adaptées à certains usages et, en fonction de ses besoins, l'utilisateur peut choisir la meilleure : les bordures pour effectuer des tâches de classification, les Cubes Fermés Émergents s'il s'agit de répondre aux requêtes ou, enfin, le Cube Quotient Émergent lorsque des capacités

de navigation dans les cubes sont nécessaires.

Toutes les propositions de représentations de notre démarche ont fait l'objet d'évaluations expérimentales qui nous ont permis de conforter nos résultats et de quantifier la réduction apportée par chaque représentation. À partir de jeux d'essai variés, incluant données réelles et synthétiques, nous avons mesuré la taille du Cube Émergent et celle de chacune des représentations. Les résultats obtenus sont convaincants. Comme nous l'avons déjà mentionné, les représentations par bordures apportent une réduction considérable du Cube Émergent. Les Cubes Fermé et Quotient Émergents montrent une réduction moindre mais cependant significative de la taille d'un Cube Émergent lorsque les jeux d'essai sont des données réelles et donc fortement corrélées. Comme nous l'attendions, les cas les plus défavorables à ces deux représentations sont les jeux de données synthétiques faiblement corrélées et donc faiblement redondantes.

Nous avons également proposé des méthode de calcul du Cube Émergent et ses représentations. Elles adoptent soit un point de vue bases de données, soit un point de vue algorithmique. Les premières, basées sur des requêtes SQL, ont surtout un objectif pédagogique mais peuvent être utilisées si les volumes de données ont une taille raisonnable. En revanche, les approches algorithmiques sont conçues pour être performantes. Pour cela, nous avons retenu l'algorithme BUC particulièrement efficace, simple et intégrable dans un SGBD et l'avons adapté à notre contexte de travail. Le résultat est une solution logicielle homogène capable de calculer le Cube Émergent et ses représentations. Ainsi, nous proposons E-IDEA (Emergent cube Integrable DatabasE Algorithme) dédié au calcul du Cube Émergent qui exploite la double contrainte d'émergence pour élaguer l'espace de recherche. Cependant BUC est conçu pour calculer le cube d'une seule relation. Pour pouvoir l'adapter à notre contexte où deux relations sont en jeu, nous avons mis en œuvre une structure de données astucieuse qui "fusionne" les deux relations (et qui permet aussi de simplifier les requêtes SQL). En traitant d'abord les tuples les plus agrégés, E-IDEA calcule des tuples qui vraisemblablement vérifient la contrainte anti-monotone C_2 sans satisfaire la contrainte monotone C_1. Ces tuples ne sont pas émergents. En générant des tuples moins agrégés, les tuples émergents vont être détectés jusqu'au moment où le niveau d'agrégation sera insuffisant pour que la contrainte C_2 reste valide.
Dans un contexte de fouille de données binaires, des algorithmes ont été proposés pour calculer les bordures L et U (Burdick *et al.*, 2005; Gouda et Zaki, 2005; Jr., 1998). Cependant, il n'existe pas d'algorithme permettant de calculer les bordures dans un contexte multidimensionnel. Pour calculer les bordures du Cube Émergent, nous définissons l'algorithme F-IDEA qui retourne L, $U^\sharp$ et/ou U en fonction des besoins de l'utilisateur. En écrivant sur disque que les tuples maximaux, F-IDEA retourne la bordure U en épargnant le temps important que BUC passe à écrire les résultats. Nous montrons ensuite les modifications à apporter à l'algorithme pour obtenir les autres bordures.
Pour calculer les représentations réduites basées sur le Cube Fermé Émergent nous proposons une approche intégrable au cœur d'un SGBD, l'algorithme $\mathbb{C}$-IDEA. Il utilise un schéma algorithmique similaire à F-IDEA et concilie à la fois simplicité et efficacité.
Des expérimentations ont été menées pour évaluer le temps d'exécution des approches proposées. Pour calculer le Cube Émergent, nous n'avons évidemment pas la possibilité de comparer E-IDEA à un autre algorithme. Nous avons alors simplement mesuré ses temps d'exécution pour différents jeux de données et nous les avons mis en regard des temps que nécessite la requête SQL la plus performante parmi celles proposées. Évidemment E-IDEA est très efficace mais il s'avère aussi que la requête retenue permet d'obtenir les résultats escomptés en un temps raisonnable. Pour évaluer et comparer les performances de F-IDEA, nous avons retenu l'algorithme

MAFIA (Burdick *et al.*, 2005) qui calcule, de manière très performante, les bordures classiques dans un contexte binaire. Nous l'avons adapté au contexte multidimensionnel. Là encore, une batterie de jeux d'essai a été utilisée. Dans tous les cas de figure, F-IDEA s'est révélé plus efficace que MAFIA.
Le volume des résultats obtenus avec un Cube Émergent dépend directement de la contrainte d'émergence. Si cette double contrainte n'est pas adaptée aux données collectées, le décideur risque d'être submergé par des résultats tellement volumineux qu'ils ne seront pas, en pratique, exploitables. En revanche, si une contrainte trop forte est formulée, il pourra ne filtrer que quelques renversements de tendances tout à fait exceptionnels. Pouvoir calibrer au plus près la double contrainte d'émergence est un réel enjeu pour une utilisation effective des résultats. Pour permettre à l'utilisateur d'affiner au mieux cette contrainte, nous proposons une approche permettant d'estimer la taille du futur Cube Émergent. Cependant pour qu'un véritable processus de calibrage s'instaure, il faut que cette taille soit rendue quasi instanément à l'utilisateur. Tout d'abord, nous en proposons une borne plutôt grossière mais dont l'obtention est immédiate. Ensuite nous introduisons une caractérisation de la taille exacte du Cube Émergent basée sur le concept d'idéal d'un ordre et le principe d'inclusion/exclusion. Néanmoins, calculer cette taille exacte à partir de notre caractérisation reste un problème difficile qui ne peut pas être résolu efficacement. Comme le facteur temps est particulièrement critique dans la problématique abordée, nous avons choisi de relâcher la contrainte sur l'exactitude de la taille et d'en donner une approximation. Pour ce faire, la stratégie adoptée met en œuvre les bordures proposées et adapte, à notre contexte de travail, l'algorithme quasi optimal HYPERLOGLOG (Flajolet *et al.*, 2007). Pour montrer la faisabilité de la démarche, nous procédons là aussi à des expérimentations. Elles ont un double objectif : d'une part comparer la taille exacte des Cubes Émergents avec leur taille approximative et d'autre part comparer les temps de calcul. Sur le premier point, les résultats sont extrêmement satisfaisants car, pour toutes les évaluations menées, l'erreur de l'approximation reste en dessous de 5%. Bien évidemment le temps de réponse pour obtenir la taille approximative est incomparable avec celui nécessaire pour calculer la taille réelle. Mais un résultat plus intéressant est de constater que la taille approximative peut être retournée à l'utilisateur de manière quasi instanée, ce qui permet au processus de calibrage d'être réellement opérant.
La démarche globale que nous développons dans cette thèse s'articule autour du concept de Cube Émergent. Néanmoins, différents résultats obtenus peuvent être exploités dans d'autres contextes. Par exemple, les nouvelles bordures introduites, qui par définition même sont plus restreintes que les bordures classiques, ont montré expérimentalement une réduction significative. Elles pourraient sans difficulté être adaptées aux contextes utilisant des bordures classiques comme l'extraction des motifs fréquents ou la classification. De la même façon, toute notre démarche peut s'appliquer directement à tout type de cube contraint. L'approche de calcul d'une taille approximative pour le Cube Émergent peut être utilisée pour tout type de cube et devenir un outil d'aide à l'administration d'un entrepôt.

1.3 Organisation du mémoire

Ce mémoire de thèse est organisé en deux parties. La première est dédiée à un état de l'art centré sur les méthodes de calcul de cubes, les approches de représentation réduites de cubes et l'analyse multidimensionnelle des préférences que permet le SKYCUBE. Dans le premier chapitre de cette partie, après un rappel des concepts de base, nous présentons des approches classiques de calcul de cube de données qu'il soit complet ou partiel. Une attention particulière est portée

à l'algorithme BUC pour son efficacité et son caractère novateur par rapport aux démarches précédentes, mais aussi parce nous l'exploitons ensuite pour notre approche algorithmique de calcul des représentations du Cube Émergent.
Le chapitre 3, intitulé « Analyse des bases de données multidimensionnelles », propose trois approches qui, parmi les premières, ont défini formellement trois représentations réduites des cubes de données. Les deux premières, le cube fermé et le cube quotient, ont introduit un espace de recherche réellement adapté au calcul du cube. Nous le présentons en détail car c'est dans ce contexte que s'inscrivent également nos contributions. Les cubes fermé et quotient sont également présentés car ils proposent, pour le premier, la représentation la plus réduite d'un cube sans perte des mesures et pour le second, la représentation la plus réduite d'un cube sans perte des mesures ni des capacités de navigation au sein des données plus ou moins agrégées. De plus, grâce à leurs qualités respectives, nous retenons ces représentations pour faire différentes propositions concernant les Cubes Émergents.
Le chapitre 4 décrit la problématique du SKYLINE, les concepts associés ainsi que différents algorithmes permettant son calcul. Nous présentons ensuite l'analyse multidimensionnelle des skylines à travers le concept de SKYCUBE. Nous en présentons une représentation réduite ainsi qu'un algorithme de calcul.

La deuxième partie de cette thèse est dédiée à nos contributions. Le chapitre 5 introduit l'ensemble des concepts nécessaires à la définition du Cube Émergent et à ses représentations par bordure. Il donne également les approches développées pour obtenir le Cube Émergent et ses bordures : requêtes SQL et les algorithmes E-IDEA et F-IDEA. Des experimentations conséquentes confirment nos résultats et permettent de les quantifier tant sur le plan des réductions de taille apportées que sur celui des temps de calcul.
Le chapitre 6 aborde l'approche d'estimation de la taille du Cube Émergent permettant de mettre en œuvre un véritable processus de calibrage des contraintes. Nous y décrivons en particulier notre adaptation de l'algorithme HYPERLOGLOG. Ce travail est conforté par les résultats d'évaluations expérimentales qui montrent la très bonne précision de la taille approximative et la rapidité de son obtention.
Dans le chapitre 7, nous introduisons les représentations sans perte de la valeur de la mesure du Cube Émergent. Nous détaillons le famille des représentations basées sur le Cube Fermé Émergent associé à l'une des bordures introduites (L, U et $U^\sharp$) et montrons comment réduire encore la bordure $U^\sharp$. Nous revisitons également le concept du cube quotient afin de pouvoir proposer une nouvelle représentation : le Cube Quotient Émergent. Là encore, nous avons mené des expérimentations conséquentes pour mesurer et comparer les tailles des différentes représentations et obtenu les résultats espérés.

Le dernier chapitre de ce document établit un bilan des contributions apportées puis trace différentes perspectives de recherche qui pourraient être ultérieurement menées.

Première partie

Autour de l'analyse des bases de données

2

Analyse Olap classique

Sommaire

2.1 Introduction

Les entrepôts de données permettent le stockage d'énormes volumes de données accumulées au fil du temps dans les bases opérationnelles. En effet, récemment l'évolution des technologies a conduit les entreprises à conserver leurs données et ainsi préserver la « mémoire » de leurs activités. Les entrepôts de données ont été conçus dans cet objectif. Contrairement aux bases opérationnelles, ils ont des caractéristiques notables. Tout d'abord, ils ne sont pas destinés à la gestion quotidienne du système d'information mais à offrir une véritable aide à la prise de décision. Leurs utilisateurs, des décideurs, sont donc peu nombreux et s'intéressent non pas au détail des données mais à des tendances générales, selon tel ou tel critère, non explicitement stockées. Les données des entrepôts sont également particulières. Régulièrement insérées dans l'entrepôt lors de rafraîchissements, ces données sont pérennes dans le sens où elles ne font l'objet d'aucune mise à jour. De plus, à leur insertion, ces données sont pourvues d'une estampille temporelle. On dit qu'elles sont historisées. À partir des gisements de données ainsi constitués, il

était naturel de chercher à les exploiter au mieux. Là encore, il ne s'agit pas de formuler des requêtes classiques, simples et fréquentes, sélectionnant généralement quelques dizaines de tuples, mais de procéder à des analyses nécessitant d'agréger les données afin de dégager des grandes tendances. De telles requêtes sont particulièrement coûteuses car elles demandent des balayages d'importants volumes de données et qu'elles sont par nature même complexes. Cependant, s'inscrivant dans un processus d'aide à la décision (d'où le sigle OLAP pour On Line Analytical Processing), la formulation de ces requêtes dépend à la fois des connaissances antérieures et des besoins des décideurs. Elle a donc lieu de manière *ad hoc* et idéalement devrait être interactive. Or le calcul sous-jacent est complexe et long, d'où l'idée de pré-calculer les résultats. Cependant, ne pouvant prévoir les souhaits de l'utilisateur, le pré-calcul doit concerner l'ensemble de tous les résultats possibles. Le concept de cube de données a été introduit pour répondre à cette problématique (Gray *et al.*, 1997). C'est le concept central pour l'analyse OLAP. Le cube de données est constitué de l'union des résultats des requêtes agrégatives GROUP BY sur toutes les combinaisons possibles des critères d'analyse (attributs catégories). Grâce au pré-calcul du cube, l'utilisateur peut avoir une réponse quasi instantanée à toutes les requêtes qui lui seront utiles.

La matérialisation complète du cube de données permet d'avoir des temps de réponse rapides pour l'utilisateur mais en contre partie nécessite un coût énorme autant en temps de calcul qu'en espace de stockage. Plusieurs approches ont tenté de trouver le meilleur compromis entre temps de réponse aux requêtes utilisateur et besoin en ressources pour le calcul et le stockage du cube de données (Gray *et al.*, 1997; Agarwal *et al.*, 1996; Ross et Srivastava, 1997; Beyer et Ramakrishnan, 1999; Harinarayan *et al.*, 1996; Shukla *et al.*, 1998; Kotsis et McGregor, 2000). Elles se séparent en deux familles distinctes. La première regroupe les méthodes qui ne matérialisent qu'une partie du cube sacrifiant ainsi le temps de réponse utilisateur au profit d'un coût de stockage moindre. Dans la seconde, les approches refusent de sacrifier le temps de réponse utilisateur tout en optimisant autant que possible le calcul et le stockage du cube de données complet.

Dans ce chapitre, nous introduisons les concepts associés au cube de données. Puis nous nous intéressons aux méthodes de calcul et matérialisation de ces cubes, en décrivant les approches les plus intéressantes matérialisant partiellement ou totalement le cube.

2.2 Cubes de données

Dans les entrepôts, les données sont dites multidimensionnelles car l'utilisateur voit et manipule ces données suivant différents critères, aussi nommés *attributs dimensions*. Durant le processus d'analyse, le décideur a besoin d'appréhender ses données suivant plusieurs combinaisons de ces dimensions. Pour ce faire, il va devoir effectuer des requêtes agrégeant les données par rapport à l'ensemble des critères choisis. Typiquement, ces requêtes mettent aussi en jeu des fonctions agrégatives (ou statistiques) appliquées sur des *attributs mesures* selon les diverses dimensions retenues. Ces différentes interrogations sont généralement calculées par le SGBD en utilisant l'opérateur GROUP BY de SQL.

Considérons par exemple la relation DOCUMENT (*cf.* Table 2.1) donnant les quantités de livres vendues par Type (T), Ville (V), Éditeur (E) et Langue (L). Lors d'une analyse faite par un décideur, il peut consulter la base de données pour connaitre la quantité totale de livres vendus, puis voyant que cette quantité est inhabituelle affiner en regardant soit la quantité de livres vendus par Ville et Éditeur ou bien par Éditeur et Langue. Comme on le constate sur cet exemple, les interrogations faites sur un entrepôt vont se décider au fur et à mesure de manière *ad hoc*. En outre, ces requêtes nécessitent le balayage de milliers voire de millions de tuples. Elles

Table 2.1 – Relation exemple Document

RowId	Type	Ville	Éditeur	Langue	Quantité
1	Nouvelles	Marseille	Collins	Français	400
2	Nouvelles	Marseille	Hachette	Anglais	100
3	Pédagogie	Paris	Hachette	Français	100
4	Pédagogie	Marseille	Collins	Anglais	300
5	Essai	Paris	Collins	Français	200

sont donc extrêmement coûteuses en temps de calcul alors que l'idéal pour les utilisateurs est que ce processus soit interactif.

Gray *et al.* (1997) à introduit le concept de *cube de données* et l'opérateur associé Cube By pour concilier ces deux besoins contradictoires. On peut les voir comme une généralisation aux données multidimensionnelles des agrégats et de l'opérateur Group by. L'idée sous-jacente est de pré-calculer et stocker tous les agrégats possibles, à différents niveaux de granularité. Ainsi, répondre à toute requête Olap se réduit à une simple sélection de résultats préalablement stockés.

Dans la suite de ce mémoire nous supposons que les relations respectent les hypothèses et les notations suivantes : soit r une relation de schéma $\mathcal{R}$. Les attributs de $\mathcal{R}$ sont divisés en deux ensembles :

1. $\mathcal{D}$ est l'ensemble des attributs dimensions, appelés aussi catégories. Il correspond aux critères d'analyse pour l'Olap. De plus, les attributs de $\mathcal{D}$ sont totalement ordonnés (un tel ordre est noté $\geq_{\mathcal{D}}$) et $\forall A \in \mathcal{D}$, r$(A)$ est la projection de r sur A.
2. $\mathcal{M}$ l'ensemble des attributs mesures, sur lesquels sont appliquées les fonctions agrégatives. Par la suite, pour simplifier les formulations, on ne considérera qu'une seule mesure.

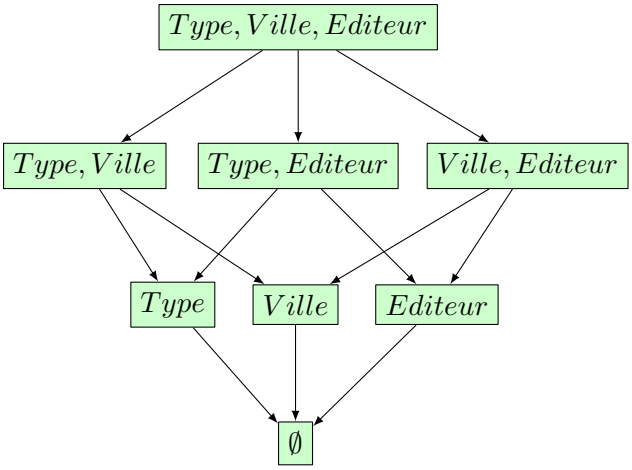
En plus des attributs de $\mathcal{R}$, nous considérons un attribut supplémentaire *RowId* qui est implicite et ne fait donc pas partie du schéma de la relation. Les valeurs de cet attribut servent d'identifiant unique à chaque tuple et sont attribuées au moment de leur insertion. On note t_i le tuple tel que $t(RowId) = i$.

Le résultat d'une requête Group-By, selon un ensemble $X \subseteq \mathcal{D}$ de dimensions, est appelé cuboïde [2], noté $cuboide_X(r)$. L'ensemble de tous les cuboïdes constitue le cube de données, noté $datacube(r)$. Les cuboïdes peuvent être ordonnés par rapport à leur niveau de détail par la relation d'ordre partiel $\preceq$. Par exemple le cuboïde selon les dimensions Type, Ville est plus détaillé que le cuboïde selon le Type ou celui selon la Ville. En d'autres termes, $cuboide_{Type}(r) \preceq cuboide_{Type,Ville}(r)$ et $cuboide_{Ville}(r) \preceq cuboide_{Type,Ville}(r)$. L'ensemble des cuboïdes muni de cet ordre forme un treillis.

Les cuboïdes de ce treillis sont regroupés par niveau en fonction de leur nombre de dimensions. Ces niveaux sont numérotés en partant du bas du treillis (cuboïde portant sur aucune dimension) et en remontant vers le sommet (cuboïde suivant tous les critères possibles appelé *cuboïde de base*). Soit deux cuboïdes $cuboide_{\mathcal{U}}(r)$ selon le sous-ensemble de dimensions $\mathcal{U}$ et $cuboide_{\mathcal{V}}(r)$ selon $\mathcal{V}$, si $\mathcal{U} \subset \mathcal{V}$ alors $cuboide_{\mathcal{U}}(r) \preceq cuboide_{\mathcal{V}}(r)$, on dit que les cuboïdes ont un lien de parenté, $cuboide_{\mathcal{V}}(r)$ est nommé *l'ancêtre* du $cuboide_{\mathcal{U}}(r)$ et $cuboide_{\mathcal{U}}(r)$ *le descendant* de $cuboide_{\mathcal{V}}(r)$.

2. La requête typique de calcul d'un cuboïde est : Select X, $f(\mathcal{M})$ From r Group By X.

FIGURE 2.1 – Ensemble des cuboïdes de la relation DOCUMENT suivant les dimensions Type, Ville, Éditeur



De plus, dans le cas où $|\mathcal{U}| = |\mathcal{V} - 1|$, $cuboide_{\mathcal{V}}(r)$ est appelé le cuboïde père de $cuboide_{\mathcal{U}}(r)$ et $cuboide_{\mathcal{U}}(r)$ le cuboïde fils de $cuboide_{\mathcal{V}}(r)$.

Les figures 2.1 et 2.2 donnent la représentation du treillis des cuboïdes pour la relation DOCUMENT suivant les dimensions Type, Ville, Éditeur.

Chaque cuboïde est a priori doté du schéma $X \cup \mathcal{M}$. Si le cube de données est calculé avec f comme fonction agrégative, nous notons $f_{val}(t,r)$ [3] la valeur de la fonction agrégative f associée au tuple t dans $datacube(r)$.

Dans notre exemple (*cf.* table 2.1), pour exprimer en SQL le cube de données nécessaire à une analyse des quantités de livres vendus par Type, Ville, Éditeur (en ignorant l'attribut Langue), la requête suivante peut être formulée :

```
SELECT Type, Ville, Editeur, SUM(Quantite)
FROM Document
CUBE BY Type, Ville, Editeur
```

Cette requête va calculer les $2^3 = 8$ groupe-by : 'Type, Ville, Éditeur', 'Type, Ville', 'Type, Éditeur', 'Ville,Éditeur', 'Type', 'Ville', 'Éditeur' et '$\emptyset$', où $\emptyset$ représente le groupe-by selon aucune dimension, i.e. qu'il agrège toute la relation en un seul tuple.

La manière naïve de calculer ce type de requête est de la réécrire comme une collection de 8 requêtes agrégatives et de les exécuter séparément. Cependant, chacune de ses sous-requêtes a un schéma qui lui est propre. Pour rendre tous ces cuboïdes uni-compatibles, les attributs ne faisant pas partie des attributs de partitionnement (*i.e.* $\forall\ A \in \mathcal{D} \backslash X$) prennent la valeur $'ALL'$. Cette valeur (Gray *et al.*, 1997) a une sémantique particulière : c'est une généralisation de toutes les valeurs du domaine d'un attribut dimension. Ainsi, les cuboïdes partagent tous le même schéma ($\mathcal{D} \cup \mathcal{M}$) et peuvent être regroupés dans une seule et même relation : le cube de données. Les éléments de ce cube sont appelés tuples (ou cellules) multidimensionnels. Chaque tuple multidimensionnel est constitué d'un ensemble de valeurs pour ses dimensions et d'une valeur numérique pour la mesure. La valeur de la mesure est calculée en agrégeant l'ensemble des tuples de la relation originelle partageant les mêmes valeurs des dimensions sélectionnées.

3. *cf.* Chapitre 3

Figure 2.2 – Matérialisation de l'ensemble des cuboïdes de la relation Document suivant les dimensions Type, Ville, Éditeur

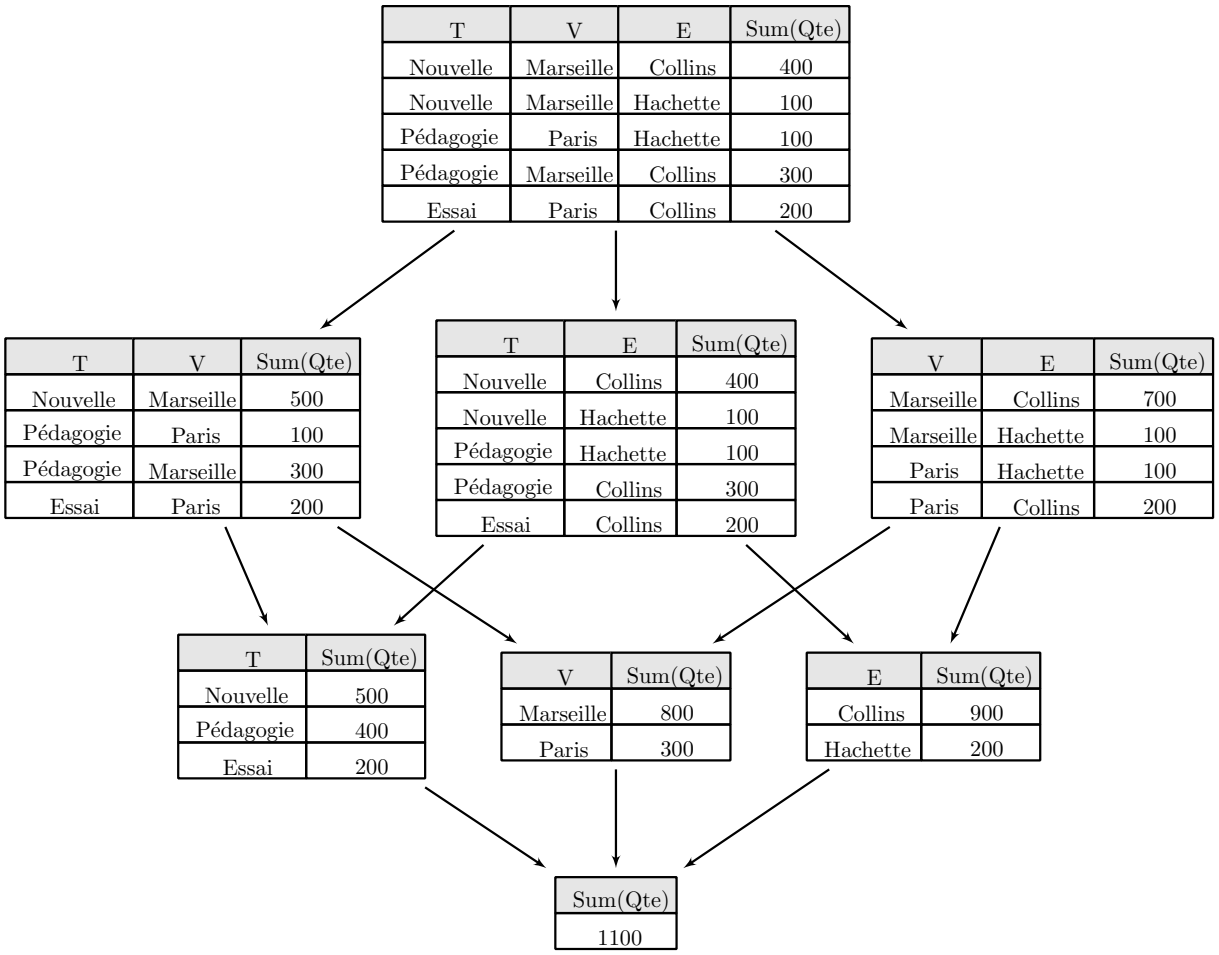


TABLE 2.2 – Représentation sous forme d'une relation du cube de données de DOCUMENT

Type	Ville	Éditeur	Quantité
Nouvelles	Marseille	Collins	400
Nouvelles	Marseille	Hachette	100
Pédagogie	Paris	Hachette	100
Pédagogie	Marseille	Collins	300
Essai	Paris	Collins	200
Nouvelles	Marseille	$'ALL'$	500
Pédagogie	Paris	$'ALL'$	100
Pédagogie	Marseille	$'ALL'$	300
Essai	Paris	$'ALL'$	200
Nouvelles	$'ALL'$	Collins	400
Nouvelles	$'ALL'$	Hachette	100
Pédagogie	$'ALL'$	Hachette	100
Pédagogie	$'ALL'$	Collins	300
Essai	$'ALL'$	Collins	200
$'ALL'$	Marseille	Collins	700
$'ALL'$	Marseille	Hachette	100
$'ALL'$	Paris	Hachette	100
$'ALL'$	Paris	Collins	200
Nouvelles	$'ALL'$	$'ALL'$	500
Pédagogie	$'ALL'$	$'ALL'$	400
Essai	$'ALL'$	$'ALL'$	200
$'ALL'$	Marseille	$'ALL'$	800
$'ALL'$	Paris	$'ALL'$	300
$'ALL'$	$'ALL'$	Collins	900
$'ALL'$	$'ALL'$	Hachette	200
$'ALL'$	$'ALL'$	$'ALL'$	1100

Par exemple le tuple multidimensionnel t = (Nouvelles, Marseille, $'ALL'$) est obtenu en agrégeant par Type et Ville, les tuples t_1 = (Nouvelles, Marseille, Collins, Français) et t_2 = (Nouvelles, Marseille, Hachette, Anglais) de la relation exemple (*cf.* table 2.1). La mesure de t est obtenue en appliquant la fonction mesure sur l'ensemble des tuples nécessaires à l'agrégation. Dans notre cas on a : $f_{val}(t,r) = t_1(\mathcal{M}) + t_2(\mathcal{M}) = 400 + 100 = 500$.

Avec la relation exemple (*cf.* table 2.1) la requête pour calculer naïvement le cube de données suivant les dimensions Type, Ville, Éditeur est la suivante :

```
SELECT Type, Ville, Editeur, SUM(Quantite)
FROM Document
GROUP BY Type, Ville, Editeur
UNION
SELECT Type, Ville, 'ALL', SUM(Quantite)
FROM Document
GROUP BY Type, Ville
```

```
UNION
SELECT Type, 'ALL', Editeur, SUM(Quantite)
FROM Document
GROUP BY Type, Editeur
UNION
SELECT 'ALL', Ville, Editeur, SUM(Quantite)
FROM Document
GROUP BY Ville, Editeur
UNION
SELECT Type, 'ALL', 'ALL', SUM(Quantite)
FROM Document
GROUP BY Type
UNION
SELECT 'ALL', Ville, 'ALL', SUM(Quantite)
FROM Document
GROUP BY Ville
UNION
SELECT 'ALL', 'ALL', Editeur, SUM(Quantite)
FROM Document
GROUP BY Editeur
UNION
SELECT 'ALL', 'ALL', 'ALL', SUM(Quantite)
FROM Document
```

Le résultat de cette requête est donné par la table 2.2. Par souci de clarté les différents cuboïdes sont séparés par une ligne horizontale.

Cette approche naïve montre en partie la difficulté du calcul d'un cube de données. En effet, si la relation d'entrée possède $|\mathcal{D}|$ dimensions, le nombre de cuboïdes est $2^{|\mathcal{D}|}$. Cela implique que le cube de données a une taille exponentielle par rapport au nombre de dimensions. Dans la plupart des entrepôts, la relation d'entrée est de l'ordre du Giga-octet avec un nombre de dimensions compris entre 3 et 20. Le développement de méthodes de calcul et stockage efficaces pour le cube de données est donc un problème critique. Ce problème a fait l'objet de nombreux travaux au cours des 15 dernières années (Morfonios *et al.*, 2007). Nous détaillons les plus importantes dans les paragraphes suivants.

Avant de s'intéresser à ces méthodes à proprement parlé, nous allons d'abord étudier les propriétés sur lesquelles se basent les algorithmes pour optimiser le calcul.

2.2.1 Catégorie de fonctions agrégatives

Dans le calcul du cubes de données, la majeure partie du temps est utilisée à agréger les données et à calculer les valeurs de la fonction mesure. Ces fonctions ont été abondamment étudiées (Han et Kamber, 2006). Les plus connues d'entre elles sont celles disponibles directement en SQL tel que : COUNT, SUM, MIN, MAX, AVG, VAR, STDDEV, ...Ces fonctions mesures peuvent être organisées en 3 catégories :

Fonction distributive : Une fonction agrégative f est distributive s'il existe une fonction g telle que $\forall I \subseteq r(RowId)$ et $\forall J$ partition de I on a :

$$f(\{t_i \mid i \in I\}) = g(\{f(\{t_j \mid j \in \mathcal{J}\}) \mid \mathcal{J} \in J\})$$

En d'autres termes, une fonction f est distributive si elle peut être calculée comme suit : Supposons que la relation soit partitionnée en n sous-ensembles. Nous appliquons d'abord la fonction f sur chacun des sous-ensemble. La valeur de f sur l'intégralité de la relation est dérivée (grâce à une fonction g donnée) à partir de la valeur de f précédemment calculée sur les éléments de la partition. La fonction distributive f est dite additive si la fonction g est la somme.

Par exemple, la fonction COUNT peut être calculée pour un cube de données en partitionnant le cube en un ensemble de « sous-cubes ». Le cube de données est obtenu en sommant les valeurs des comptages pour chacun des sous-cubes. Pour les même raisons les fonctions SUM, MIN, MAX sont aussi distributives. Cette famille de fonctions est intéressante car elle permet de distribuer (ou découper) le calcul du cube de données.

Fonction algébrique : Une fonction agrégative f est algébrique (ou calculable) s'il existe une fonction h à M arguments (avec M un entier naturel borné) et un ensemble $G = \{g_1, \ldots, g_M\}$ de M fonctions distributives, tel que : $\forall I \subseteq r(RowId)$ et $\forall J$ partition de I on a :

$$f(\{t_i \mid i \in I\}) = h(\{g_j(\{t_i \mid i \in I\}) \mid j = 1 \ldots M\})$$

Exprimé plus simplement, une fonction algébrique est une fonction qui peut être calculée à partir du résultat de M fonctions distributives. Généralement, on stocke uniquement les résultats des M fonctions distributives dans le cube et la mesure est redérivée au besoin. Cette approche permet de bénéficier de tous les avantages des fonctions distributives pour des fonctions qui ne le sont pas.

Par exemple, AVG (la moyenne) peut être calculée en divisant le résultat de la fonction SUM par le résultat de la fonction COUNT. Dans Chen et McNamee (1989) une approche de paramétrisation est proposée et les expressions de calcul des principales fonctions statistiques sont détaillées : la covariance s'exprime à partir du comptage et de la somme de produits, l'analyse de régression est calculable à partir du comptage, somme, somme de produits, somme de carrés...

Fonction holistique : Les fonctions holistiques sont celles qui ne sont ni distributives ni algébriques. Pour ces fonctions il n'existe pas de borne constante pour décrire l'agrégat avec des sous-agrégats. Dans ce cas, la mesure doit être recalculée à partir des données initiales sans pouvoir réutiliser des résultats intermédiaires. Les fonctions agrégatives holistiques typiques sont celles basées sur le rang comme la médiane, les k-meilleurs, les quartiles, ...

Parmi ces catégories de fonctions, celles qui sont les plus intéressantes pour les entrepôts sont les fonctions distributives. En effet, le fait de pouvoir calculer un cube de données de manière distribuée a plusieurs conséquences importantes :

- lors d'un rafraîchissement de l'entrepôt (insertion de nouveaux tuples), si l'on considère ce dernier et l'ensemble des tuples à insérer comme une partition d'une nouvelle relation, il suffit de calculer le cube de données correspondant aux nouveaux tuples et, à partir des résultats obtenus, d'effectuer les mises à jour sur le cube de données précédemment calculé. Nous évitons ainsi un recalcul complet du cube de données. Le rafraichissement est donc beaucoup moins coûteux ;
- pendant la phase de calcul du cube de données, des résultats intermédiaires peuvent être exploités pour optimiser le temps de calcul. En effet, lorsque l'on agrège une relation r suivant un ensemble de dimensions X, les tuples de r sont partitionnés en fonction de leurs valeurs sur X. Tous les tuples ayant même valeur pour X sont regroupés dans une même classe d'équivalence. Si maintenant nous souhaitons agréger r suivant $Y \subset X$, il est possible

TABLE 2.3 – Cube de données de t_6 = (Essai, Marseille, Collins, Français)

Type	Ville	Éditeur	Quantité
Essai	Marseille	Collins	200
Essai	Marseille	$'ALL'$	200
Essai	$'ALL'$	Collins	200
$'ALL'$	Marseille	Collins	200
Essai	$'ALL'$	$'ALL'$	200
$'ALL'$	Marseille	$'ALL'$	200
$'ALL'$	$'ALL'$	Collins	200
$'ALL'$	$'ALL'$	$'ALL'$	200

d'utiliser le résultat du partitionnement selon X car chaque classe d'équivalence selon Y provient d'un ensemble de classes d'équivalence selon X. Grâce à la nature distributive de ses fonctions, nous pouvons agréger un agrégat et nous avons la garantie d'obtenir le bon résultat.

Exemple 2.1 - Pour la relation DOCUMENT (*cf.* table 2.1) où la fonction mesure utilisée est la somme (notéeSUM) :

- si durant un rafraîchissement nous rajoutons le tuple t_6 = (Essai, Marseille, Collins, Français) avec $Sum_{val}(t_6,r) = 200$, alors pour mettre à jour le cube de données il nous suffit de calculer le cube de données de ce tuple et de mettre à jour la table 2.2. La table 2.3 donne ce cube de données. Les tuples grisés sont ceux à ajouter dans la table 2.2, pour les autres, il faut modifier la valeur de la mesure en cumulant les valeurs présentes dans chacune des deux tables ;
- le tuple multidimensionnel t = (Pédagogie, $'ALL'$, $'ALL'$) peut être obtenu en agrégeant par Type, les tuples multidimensionnels u = (Pédagogie, Marseille, $'ALL'$) et v = (Pédagogie, Paris, $'ALL'$) (*cf.* table 2.2). La fonction agrégative étant distributive, $Sum_{val}(t,r) = Sum_{val}(u,r) + Sum_{val}(v,r) = 300 + 100 = 400$.

La plupart des applications requérant le calcul d'un cube de données utilisent des fonctions distributives ou algébriques. Cependant, si la relation comporte de nombreux tuples, le calcul rapide du cube de données pour une fonction agrégative holistique peut poser des problèmes. C'est pourquoi, il existe des techniques d'approximation pour ces fonctions (Han et Kamber, 2006)

2.2.2 Variantes du cube de données

Le cube de données a été proposé comme un outil, permettant aux décideurs d'analyser leurs données selon plusieurs dimensions pour en extraire les grandes tendances apparaissant et en comprendre leurs origines. Selon le type d'analyse à mener, il peut être intéressant de se focaliser seulement sur une certaine partie du cube. Les données trop détaillées ou trop générales par exemple, peuvent n'apporter que peu d'informations utiles. C'est pour répondre à ce type de besoin que plusieurs variantes du cube de données sont apparues ces dernières années. Chacune d'entre elles permet à l'utilisateur de définir des contraintes que doivent respecter les tuples pour qu'ils soient considérés pertinents.

TABLE 2.4 – Représentation sous forme d'une relation du cube iceberg de DOCUMENT

Type	Ville	Éditeur	Quantité
Nouvelles	Marseille	Collins	400
Pédagogie	Marseille	Collins	300
Nouvelles	Marseille	*'ALL'*	500
Pédagogie	Marseille	*'ALL'*	300
Nouvelles	*'ALL'*	Collins	400
Pédagogie	*'ALL'*	Collins	300
'ALL'	Marseille	Collins	700
'ALL'	Paris	Collins	200
Nouvelles	*'ALL'*	*'ALL'*	500
Pédagogie	*'ALL'*	*'ALL'*	400
'ALL'	Marseille	*'ALL'*	800
'ALL'	Paris	*'ALL'*	300
'ALL'	*'ALL'*	Collins	900
'ALL'	*'ALL'*	*'ALL'*	1100

Cubes icebergs

En s'inspirant des motifs fréquents, Beyer et Ramakrishnan (1999) introduisent les cubes icebergs. Ces derniers, sont présentés comme des sous-ensembles de tuples du cube de données qui satisfont, pour les valeurs de la mesure, une contrainte de seuil. L'objectif sous-jacent est triple. En premier lieu, il s'agit d'exhiber les tendances suffisamment fortes pour être pertinentes pour le décideur. Il en découle deux intérêts techniques importants : ne pas calculer ni matérialiser la totalité du cube, d'où un gain notable à la fois en temps d'exécution et en espace disque. La requête SQL permettant de calculer un cube iceberg a la forme suivante :

```
SELECT D, f({M|*})
FROM r
CUBE BY D
HAVING f({M|*}) >= MinSeuil;
```

Exemple 2.2 - Avec la relation exemple DOCUMENT (*cf.* Table 2.1), le cube iceberg obtenu pour la contrainte « SUM(Quantité) $\geq$ 300 » est donné dans la table 2.4.

Cubes intervallaires

Le cube intervallaire, introduit par Casali *et al.* (2007), ne contient que les tuples du cube de données pour lesquels les valeurs de la mesure sont comprises dans un intervalle donné. La requête SQL permettant de calculer un tel cube est la suivante :

```
SELECT D, f({M|*})
FROM r
CUBE BY D
HAVING f({M|*}) BETWEEN MinSeuil AND MaxSeuil;
```

TABLE 2.5 – Représentation sous forme d'une relation du cube intervallaire de DOCUMENT

Type	Ville	Éditeur	Quantité
Nouvelles	Marseille	Collins	400
Pédagogie	Marseille	Collins	300
Nouvelles	Marseille	*'ALL'*	500
Pédagogie	Marseille	*'ALL'*	300
Nouvelles	*'ALL'*	Collins	400
Pédagogie	*'ALL'*	Collins	300
'ALL'	Marseille	Collins	700
'ALL'	Paris	Collins	200
Nouvelles	*'ALL'*	*'ALL'*	500
Pédagogie	*'ALL'*	*'ALL'*	400
'ALL'	Paris	*'ALL'*	300

Exemple 2.3 - Avec la relation exemple DOCUMENT (*cf.* Table 2.1), le cube intervallaire obtenu pour la contrainte « $300 \leq$ SUM(Quantité) ≤ 700 » est donné dans la table 2.5.

Il est important de remarquer que tout algorithme pouvant calculer le cube iceberg ou le cube intervallaire peut aussi être utilisé pour calculer le cube de données complet. Il suffit pour le cube iceberg de choisir comme contrainte « COUNT(*) ≥ 1 » ou pour le cube intervallaire la contrainte « $1 \leq$ COUNT(*) $\leq \infty$ ».

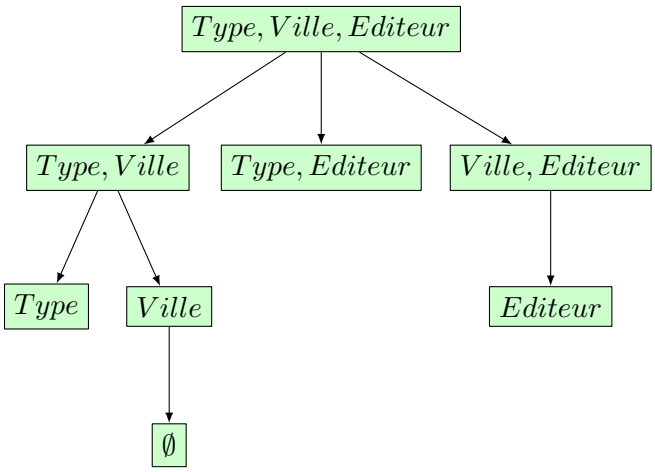
2.3 Méthodes de calcul du cube de données complet

2.3.1 L'algorithme 2^D

2^D est l'algorithme le plus simple pour calculer un cube de données. C'est la première méthode de calcul qui a été proposée par Gray *et al.* (1997). Cet algorithme calcule tous les cuboïdes à partir de la relation d'entrée et renvoie l'union de tous ces résultats partiels. Cette approche a une complexité qui est exponentielle par rapport au nombre de dimensions. Elle ne tire aucun bénéfice des résultats communs à chacun des agrégats. C'est pour cela qu'elle ne peut être utilisée en pratique. Toutefois, il est important de noter que cet algorithme a été proposé en même temps que le concept du cube de données. Il a l'avantage de bien expliciter ce qu'est le cube de données et permet de comprendre quels sont les problèmes liés à son calcul.

Pour surmonter la difficulté principale de 2^D, tous ses successeurs directs ont essayé d'identifier un ordre selon lequel les cuboïdes doivent être calculés. Cet ordonnancement des calculs a pour objectif de minimiser le coût de calcul de chaque sous-résultat en utilisant le mieux possible les précédents. Cet ordre devant être minimal, il forme nécessairement un arbre couvrant du treillis des cuboïdes. Cet arbre est appelé *l'arbre d'exécution* de l'algorithme. Chacune de ses arêtes indique quel est le cuboïde (la source) utilisé comme base pour le calcul d'un autre cuboïde (la destination). La principale différence entre tous les algorithmes basés sur cette idée est la méthode de construction de l'arbre d'exécution.

FIGURE 2.3 – Arbre d'exécution de l'algorithme GBLP pour la relation DOCUMENT



2.3.2 L'algorithme GBLP

L'algorithme GBLP (Gray *et al.*, 1997) a été proposé pour améliorer l'efficacité de 2^D. Cette méthode calcule chaque nœud du treillis des cuboïdes non pas en utilisant la relation d'origine mais en choisissant le plus réduit de ses antécédents. Une fois l'ordre d'exécution choisi, l'opération d'agrégation peut être implémentée soit avec des tris, soit avec du hachage, le choix étant laissé au système.

Lorsque l'on traverse le treillis des cuboïdes en descendant, la cardinalité des agrégats diminue car les tuples de chaque classe d'équivalence sont remplacés par un tuple multidimensionnel de moins en moins détaillé (plus en plus agrégé). Choisir l'antécédent ayant la cardinalité la plus réduite permet d'économiser du temps de calcul.

Exemple 2.4 - Pour la relation DOCUMENT (*cf.* table 2.1) la figure 2.3 donne l'arbre d'exécution de l'algorithme GBLP.

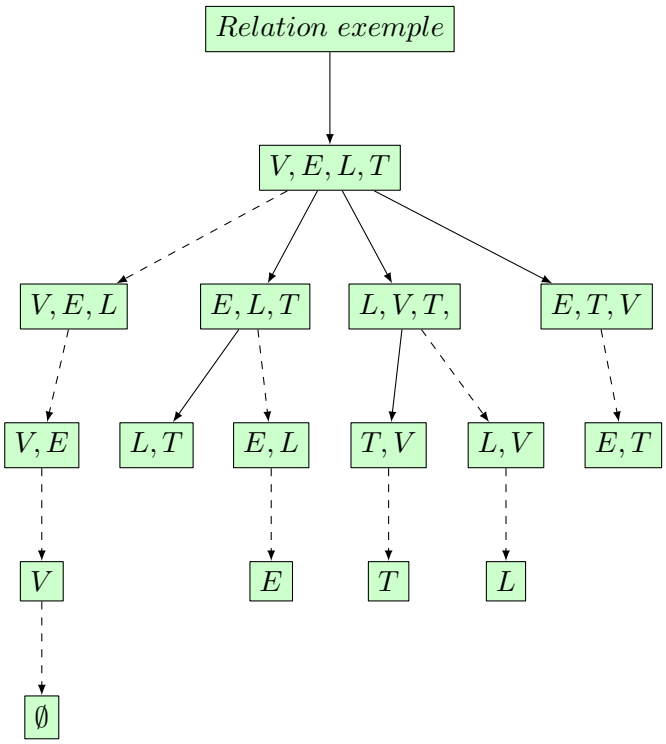
En réalité GBLP (ainsi que tous les autres algorithmes fonctionnant sur le même principe d'arbre d'exécution) ne calcule pas la taille de chaque cuboïde mais se satisfait d'une simple estimation. Ce type d'estimation a largement été étudié dans la littérature et donne de très bonnes approximations (Shukla *et al.*, 1996).

2.3.3 L'algorithme PipeSort

L'algorithme PIPESORT (Agarwal *et al.*, 1996) est la première approche qui a tenté d'améliorer l'algorithme GBLP, en essayant de minimiser le coût total de calcul du cube de données. Il utilise différentes estimations de coût pour choisir à partir de quel nœud N_y du treillis des cuboïdes il va calculer le nœud N_x (avec $x \subset y$). Pour commencer, PIPESORT estampille chacune des arêtes N_{xy} ($N_x \rightarrow N_y$) du treillis avec deux valeurs : S_{xy} l'estimation du coût de calcul du nœud N_y à partir de N_x si ce dernier n'est pas trié, et A_{xy} l'estimation du coût de calcul du nœud N_y à partir de N_x si ce dernier déjà est convenablement trié.

Une fois cette opération effectuée, PIPESORT parcourt le treillis, en partant de la racine et en montant niveau par niveau. À chaque étape, il détermine pour chaque nœud (cuboïde) l'ordre des dimensions dans lequel il sera trié. Si l'ordre choisi pour un nœud N_y est un préfixe de

FIGURE 2.4 – Arbre d'exécution de l'algorithme PIPESORT pour la relation DOCUMENT



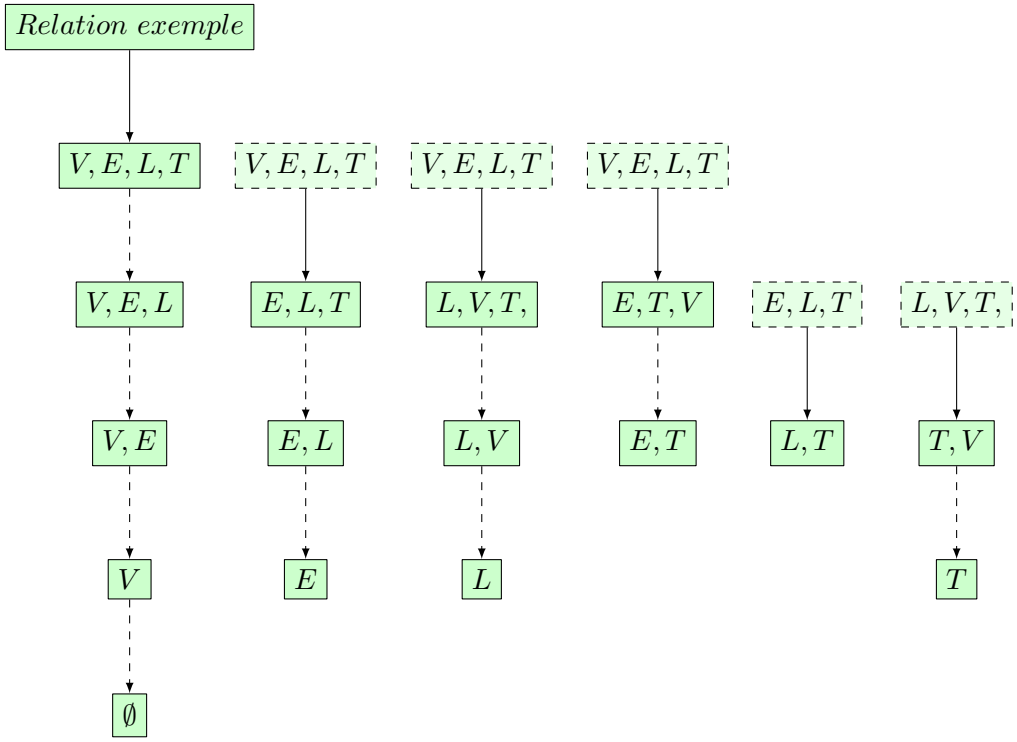
l'ordre de l'un de ses parents N_x, alors N_y peut être obtenu directement à partir de N_x sans avoir besoin de faire un tri supplémentaire. Dans ce cas, l'arête N_{xy} est marquée A. Sinon N_x doit être trié selon l'ordre de N_y, l'arête correspondante est marquée S. Clairement, pour un nœud N_x, comme un seul niveau est considéré à la fois, il y a au plus un seul nœud pouvant avoir un ordre préfixe de celui de N_x. Donc une seule arête est marquée A. Pour minimiser la somme des coûts des arêtes sélectionnées par l'algorithme à chaque niveau, PIPESORT se ramène à un problème de couplage de poids minimal dans un graphe biparti. Le treillis des cuboïdes est ainsi transformé en un arbre d'exécution efficace. Une fois calculé, PIPESORT ajoute à l'arbre d'exécution un nœud pour la relation d'entrée r et une arête marquée S qui connecte r à la racine de l'arbre.

Enfin, l'arbre est transformé en un ensemble de chemins, de sorte que chaque arête apparaisse dans un seul chemin et qu'à part la première arête d'un chemin toutes les suivantes soient marquées A. Ainsi, chaque chemin ne nécessite qu'un tri. Pendant la phase de calcul de l'algorithme, chaque chemin est calculé séparément. Une fois l'opération de tri réalisée pour la racine d'un chemin, les autres cuboïdes de ce chemin ne nécessitent plus qu'un simple passage sur les données pour être calculés.

Exemple 2.5 - Pour la relation DOCUMENT (*cf.* table 2.1) la figure 2.4 donne l'arbre d'exécution de l'algorithme PIPESORT. Les arêtes en pointillés sont celles marquées A dont le nœud destination peut être obtenu sans tri. Le nombre de tris à effectuer correspond au nombre d'arêtes marquées S. Pour cet exemple restreint, il faut réaliser 6 tris.

Le principal défaut de PIPESORT est que ses performances se dégradent très vite lorsque le nombre de dimensions augmente. En effet, l'algorithme réalise autant de tris qu'il y a de

FIGURE 2.5 – Ensemble des chemins de l'algorithme PIPESORT pour la relation DOCUMENT



chemins. Or le nombre de chemins nécessaire pour couvrir un treillis est égal à la longueur de la plus grande antichaîne. Dans le cas du treillis des cuboïdes elle est minorée par $\binom{\mathcal{D}}{\lfloor \mathcal{D}/2 \rfloor}$. Le nombre de tris à réaliser est donc exponentiel. De plus, lorsque la relation est peu dense, certains agrégats risquent de ne pas tenir en mémoire, les tris doivent être fais en mémoire secondaire ce qui pénalise fortement le temps de réponse de l'algorithme.

Exemple 2.6 - La figure 2.5 illustre les chemins utilisés par PIPESORT pour calculer le cube de données de la relation DOCUMENT (*cf.* table 2.1). Le nombre de tris à effectuer est égal à $\binom{4}{\lfloor 4/2 \rfloor} = 6$ ce qui est la borne théorique minimale.

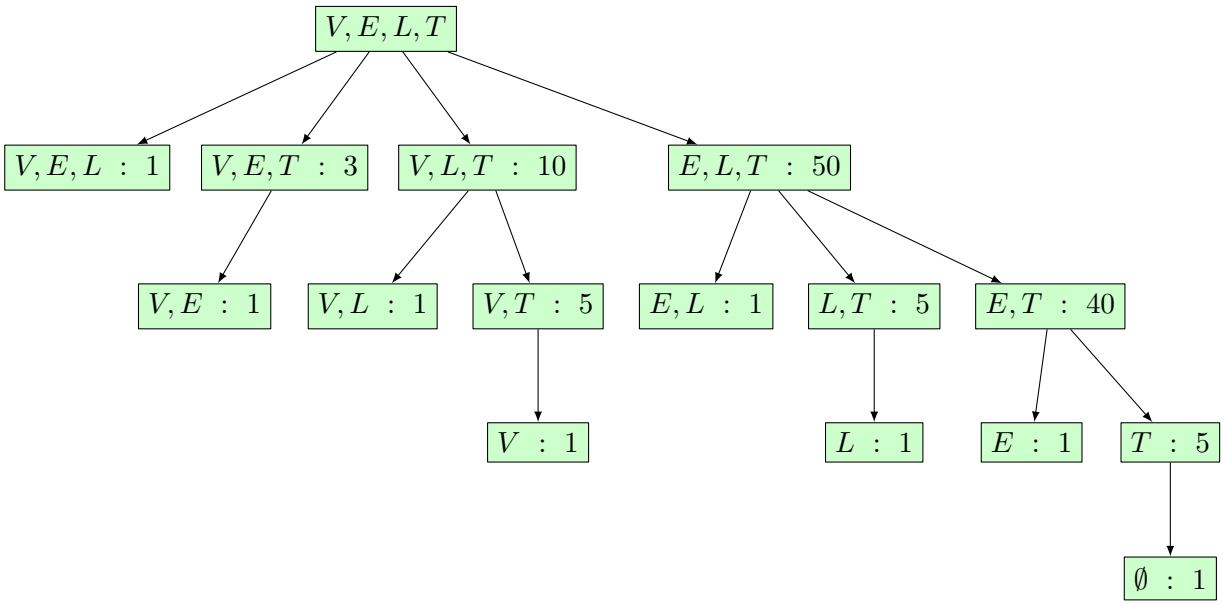
2.3.4 L'algorithme Overlap

L'algorithme OVERLAP, proposé par Agarwal *et al.*, s'efforce de minimiser autant que possible le nombre d'accès disques en réutilisant au maximum les calculs déjà effectués. Pour ce faire, il utilise la notion de correspondance d'ordre partiel qui permet de réduire le nombre de tris.

Pour commencer, l'algorithme calcule le cuboïde de base et le trie complètement selon un ordre choisi. En plus de la racine, l'ordre choisi est utilisé pour ordonner les dimensions des autres cuboïdes. Le choix de l'ordre n'affecte pas la justesse de l'algorithme mais peut jouer un rôle non négligeable sur son efficacité. Les auteurs de cet algorithme ont proposé plusieurs heuristiques pour trouver cet ordre.

La seconde étape de l'algorithme vise à élaguer le treillis des cuboïdes jusqu'à obtenir un arbre. La méthode utilisée est la suivante : Pour chaque nœud du treillis le parent avec lequel il partage le plus grand préfixe (par rapport à l'ordre initialement choisi) est sélectionné pour

FIGURE 2.6 – Arbre d'exécution de l'algorithme OVERLAP pour la relation DOCUMENT



devenir son père dans l'arbre d'exécution. Une fois cette étape réalisée, OVERLAP estime la mémoire nécessaire pour calculer chaque cuboïde à partir du père qui lui a été affecté (en supposant que le père soit déja trié et que ses données aient une distribution uniforme).

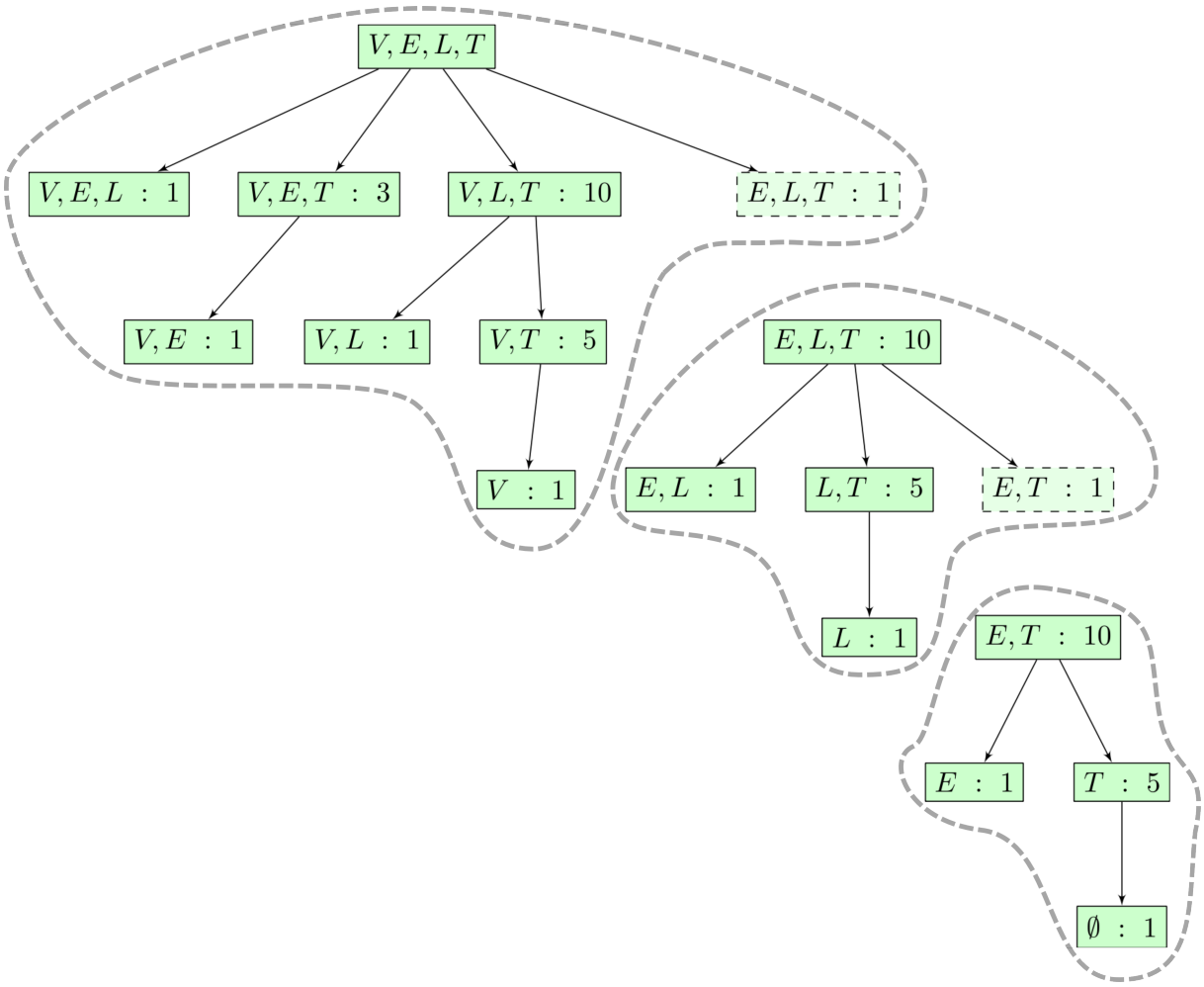
Exemple 2.7 - La figure 2.6 montre l'arbre créé par OVERLAP pour la relation DOCUMENT (*cf.* table 2.1). L'ordre initial choisi est $VELT$. La valeur dans chaque cuboïde correspond à l'estimation de la mémoire nécessaire pour le calculer.

Après la création de l'arbre, OVERLAP fonctionne de manière itérative. À chaque étape, il sélectionne un ensemble de cuboïdes non traités pouvant rentrer en mémoire centrale. Pour une partie des cuboïdes, la mémoire dont on estime avoir besoin pour le calcul, peut être allouée. De tels cuboïdes sont dit dans l'état « Partition ». Les autres, ne tenant pas en mémoire centrale, sont dits dans l'état « SortRun » et une page mémoire leur est allouée.

La sélection des nœuds de l'arbre d'exécution est basée sur l'heuristique suivante : les nœuds sont considérés en partant du haut et en descendant dans l'arbre. Le parcours s'effectue en largeur en choisissant en priorité les cuboïdes les plus petits (selon l'estimation statistique) et ayant le plus grand nombre d'attributs. Pendant le calcul du cube à proprement parler, chaque ensemble de nœuds est évalué séparément. Les nœuds dans l'état « Partition » sont calculés directement, puisque leur besoin mémoire est satisfait. Les tuples n'ont qu'à être totalement triés en mémoire en profitant des tris déjà effectués. Les tuples des cuboïdes dans l'état « SortRun » sont simplement écrits sur disque, les différentes parties sont par la suite fusionnées en les agrégeant au besoin. Les tuples résultants sont réinjectés pour la suite des calculs. Les nœuds « SortRun » économisent de nombreuses et coûteuses passes sur les données du cuboïde père en réduisant autant que possible le besoin en mémoire.

Exemple 2.8 - La figure 2.7 montre les trois ensembles de tuples qui sont choisis par l'algorithme OVERLAP pour calculer le cube de données de la relation DOCUMENT (*cf.* table 2.1). Nous supposons que nous avons à notre disposition 25 pages mémoires. Les nœuds « SortRun » sont représentés en pointillés.

FIGURE 2.7 – Ensemble de nœuds choisis par l'algorithme OVERLAP pour la relation DOCUMENT



Les algorithmes présentés jusqu'à présent suivent tous des stratégies similaires en utilisant des tris pour calculer les différents agrégats. Ce qui différencie ces algorithmes, c'est principalement leur manière plus ou moins élaborée de transformer le treillis des cuboïdes en un arbre d'exécution efficacement calculable. Malgré tout, ces approches restent inapplicables dans deux cas :

- les relations ayant un grand nombre de dimensions pour lesquelles le treillis des cuboïdes explose. Le temps de calcul de l'arbre couvrant lui aussi devient exorbitant. Les stratégies essayant d'élaguer le treillis dans son intégralité ne peuvent donc pas s'appliquer.
- les relations d'entrée qualifiées de « sparse » (traduit par *peu dense*). Une relation est dite « sparse » si sa cardinalité représente juste une petite fraction de celle du produit cartésien des domaines des attributs qui la composent. Avec de telles relations, les regroupements sont moins fréquents et les cuboïdes tendent à avoir une taille importante. Il devient ainsi fort possible de ne pas pouvoir faire tenir un cuboïde complet en mémoire centrale. Donc, en plus d'augmenter considérablement le coût de l'écriture du résultat sur disque, les relations « sparse » font aussi exploser les temps de calcul à cause des allers/retours entre mémoire centrale et disque.

2.3.5 L'algorithme Partitioned-Cube

L'algorithme PARTITIONED-CUBE, proposé par Ross et Srivastava, vise à corriger les défauts des approches antérieures avec les jeux de données peu denses. Ce travail est particulièrement important car les relations réelles ont souvent des attributs avec de larges domaines et sont donc généralement « sparse ». La méthode présentée repose sur deux algorithmes distincts. Le premier, PARTITIONED-CUBE, a pour objectif de découper (partitionner) récursivement la relation d'entrée jusqu'à obtenir une sous-relation dont le cube de données peut être directement calculé en mémoire. En réunissant les différents sous-cubes, l'algorithme dérive l'intégralité du cube de données souhaité. Le second algorithme, nommé MEMORY-CUBE, est dédié au calcul de cube de données. Il est tout à fait classique mais optimisé pour fonctionner exclusivement en mémoire centrale. C'est grâce à cet algorithme que PARTITIONED-CUBE calcule tous les sous-cubes. Les deux algorithmes étant indépendants, tout algorithme peut bénéficier des qualités de PARTITIONED-CUBE. Cette approche est tellement efficace qu'à l'heure actuelle, les méthodes de calcul de cubes font quasi-systématiquement l'hypothèse implicite de disposer de cette méthode de partitionnement et supposent donc que les relations rentrent toujours en mémoire centrale.

PARTITIONED-CUBE est décrit dans l'algorithme 1. C'est un algorithme récursif utilisant une stratégie « diviser pour régner » déjà employée avec succès pour répondre efficacement à des opérations complexes telles que la jointure ou le tri : découper la relation d'entrée jusqu'à obtenir des fragments tenant en mémoire, calculer l'opération souhaitée sur chacun d'eux, puis réunir les résultats pour obtenir la solution désirée. L'algorithme transforme le calcul d'un cube à k dimensions avec une relation d'entrée de taille T en $n+1$ calculs de sous-cubes. Les n premiers sous-cubes ont k dimensions et sont basés sur approximativement T/n tuples. Le dernier sous-cube se base sur moins de T tuples et comporte $k-1$ dimensions.

À chaque étape l'algorithme procède comme suit : il choisit une dimension $d \in \mathcal{D}$, il partitionne la relation d'entrée r en n fragments en fonction des valeurs des attributs pour cette dimension. Sur chacun de ces fragments, plus réduit que la relation r, il calcule les sous-cubes de données. Puis il réunit les résultats au sein d'un premier sous-cube noté $SousCube_1$. Il est constitué de tous les cuboïdes contenant la dimension d. Après cette opération, l'algorithme calcule un deuxième sous-cube, noté $SousCube_2$, qui porte sur les dimensions $\mathcal{D} \setminus d$. Ce cube est donc constitué de tous les cuboïdes ne contenant pas d. Pour le déterminer, on prend comme entrée, non pas la relation r, mais le cuboïde le moins agrégé de $SousCube_1$ déjà calculé. Ce

cuboïde étant un agrégation de r, il a une cardinalité inférieure ou égale à celle de r.

Les deux sous-cubes contiennent chacun la moitié des nœuds du cube. Les sous-cubes sont tous de taille inférieure à celle de r et ont moins de dimensions. L'algorithme a donc divisé le problème en deux sous-problèmes plus facilement résolvables. On applique récursivement cette procédure jusqu'à obtenir des jeux de données traitables. Cette méthode, réduisant à chaque étape la difficulté, est certaine de se terminer en un temps fini.

Algorithme 1 Algorithme PARTITIONED-CUBE

Entrée :
La relation r
L'ensemble de dimensions $\mathcal{D}$
Sortie :
Le cube de données de r divisé en deux parties, F correspondant au cuboïde le moins agrégé et D l'ensemble des autres cuboïdes.

1: **si** r tient en mémoire centrale **alors**
2: **retourner** MEMORY-CUBE(r, $\mathcal{D}$)
3: **fin si**
4: choisir une dimension $d \in \mathcal{D}$
5: partitionner r, selon les valeurs de l'attribut d, en n sous-relations : $r_1, \ldots, r_n$
6: **pour** $i = 1, \ldots, n$ **faire**
7: (F_i, D_i) =PARTITIONED-CUBE(r_i, $\mathcal{D}$) //Calcul de $SousCube_1$
8: **fin pour**
9: $F := \bigcup_{i=1}^{n} F_i$
10: (F', D') =PARTITIONED-CUBE(F, $\mathcal{D} \setminus d$) //Calcul de $SousCube_2$
11: $D := (\bigcup_{i=1}^{n} D_i) \cup F' \cup D'$
12: **retourner** (F, D)

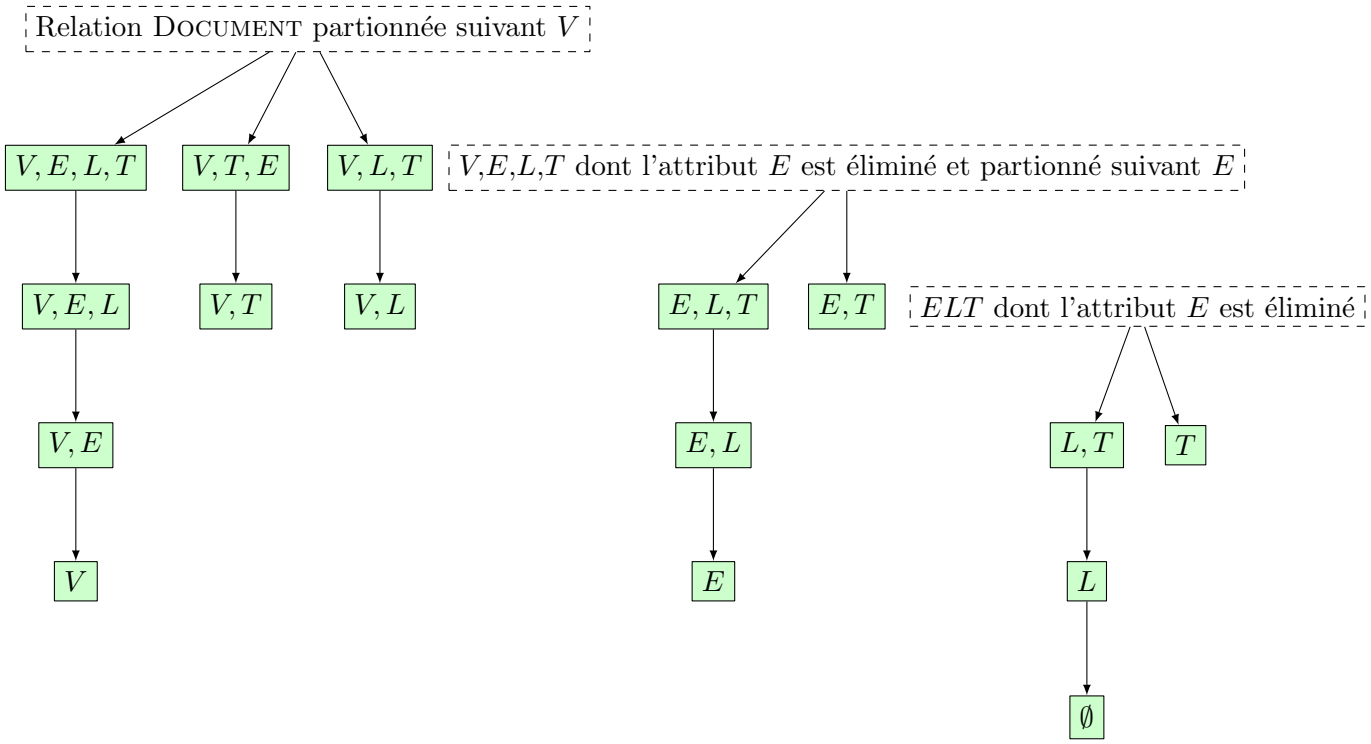
Exemple 2.9 - La figure 2.8 montre le partitionnement choisi par l'algorithme PARTITIONED-CUBE pour calculer le cube de données de la relation DOCUMENT (*cf.* table 2.1). Les nœuds en pointillés correspondent aux différents appels de la fonction PARTITIONED-CUBE.

Les principaux avantages de l'algorithme PARTITIONED-CUBE sont sa souplesse et son faible coût en E/S :

- en effet, l'algorithme est particulièrement souple car le choix des dimensions et la méthode de partitionnement sont libres. L'algorithme peut donc toujours s'adapter quelle que soit la nature de la relation. Par exemple, si la dimension de partitionnement contient des données fortement biaisées, il est possible de découper une partition en plusieurs fragments et inversement, regrouper en une seule autant de partitions que la mémoire le permet ;
- de plus, si un attribut supplémentaire est ajouté, on ne fait dans le pire des cas qu'un appel supplémentaire à l'algorithme en réutilisant les résultats calculés. Les E/S sont donc largement diminuées. Lorsque l'algorithme est utilisé avec MEMORY-CUBE, le coût des E/S croît proportionnellement au nombre de dimensions. Son comportement est ainsi bien meilleur que les algorithmes OVERLAP et PIPESORT qui ont respectivement un coût quadratique et exponentiel.

Dans la suite du travail PARTITIONED-CUBE sera utilisé quasi-systématiquement pour garantir aux algorithmes que les relations d'entrée rentrent toujours en mémoire centrale.

FIGURE 2.8 – Graphe d'exécution de l'algorithme PARTITIONED-CUBE pour la relation DOCUMENT

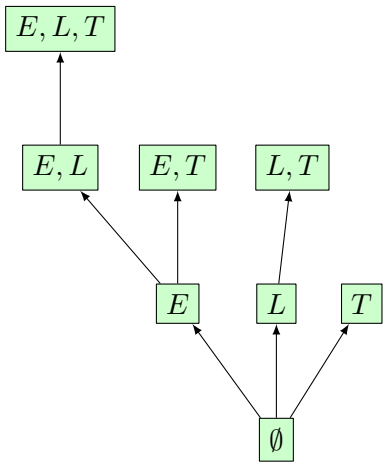


2.3.6 L'algorithme Buc

L'algorithme BUC (pour *Bottom Up Cube*), proposé par Beyer et Ramakrishnan (1999), est le premier algorithme de calcul efficace pour le cube iceberg et le cube de données. La plus grande innovation introduite par cet algorithme provient de son sens de parcours du treillis des cuboïdes de bas en haut. En effet, tous ses prédécesseurs ont voulu optimiser le temps de réponse en évitant de faire des opérations de tris inutiles et en partageant autant que possible les calculs d'agrégats entre les différents cuboïdes. Or BUC, originellement conçu pour le cube iceberg, choisit de parcourir le treillis des cuboïdes en partant du bas (du plus agrégé vers le moins agrégé) pour pouvoir appliquer une stratégie d'élagage, inspirée de la fouille de données binaire, à la APRIORI (Agrawal *et al.*, 1993). Ce choix lui permet d'économiser énormément de temps en ne calculant que les agrégats utiles. En plus d'être particulièrement efficace pour le cube iceberg et le cube de données, l'approche adoptée par BUC a le mérite de ne pas s'appuyer sur des structures de données complexes supplémentaires. Il est ainsi directement intégrable au sein d'un SGBD. Morfonios et Ioannidis (2008) ont même montré qu'en plus de calculer le cube de données, on pouvait aussi l'utiliser avec un grand bénéfice tout au long du cycle de vie d'un data cube (de la construction à maintenance incrémentale en passant par le calcul des requêtes et l'indexation). C'est grâce à toutes ces qualités, que les algorithmes basés sur BUC restent aujourd'hui encore parmi les meilleurs concurrents pour le calcul du cubes de données et de certaines de ses variantes.

BUC, comme son nom l'indique, parcourt son espace de recherche de bas en haut. Il part du cuboïde selon $\emptyset$ et remonte à travers le treillis des cuboïdes en utilisant un ordre lexicographique. Pour simplifier la présentation, nous supposerons que l'ordre entre attributs ($\geq_{\mathcal{D}}$) est l'ordre alphabétique. Ainsi, l'ordre lexicographique de parcours des nœuds fait par l'algorithme

FIGURE 2.9 – Arbre de parcours des cuboïdes par l'algorithme BUC pour la relation DOCUMENT



est identique à l'ordre du dictionnaire. La relation d'ordre $\geq_{\mathcal{D}}$ n'a aucune influence sur la justesse de l'algorithme mais peut avoir des conséquences importantes pour l'efficacité. En pratique, plusieurs heuristiques pour trouver le meilleur ordre ont été proposées par Beyer et Ramakrishnan (1999) et expérimentées par Nasr et Badr (2003).

BUC est un algorithme récursif basé sur le partitionnement horizontal. Assez grossièrement, on peut imaginer BUC comme une version de PARTITIONED-CUBE qui ne fait jamais appel à MEMORY-CUBE. Ainsi, il bénéficie à la fois de la bonne efficacité des E/S de PARTITIONED-CUBE et de l'élagage à la APRIORI. Il est décrit dans l'algorithme 2. Au premier appel, la cellule courante passée en paramètre est $('ALL', \ldots, 'ALL')$. La première agrégation fait donc un passage sur r pour calculer la valeur de la mesure (*cf.* algorithme 3). Une fois cette étape réalisée, on vérifie si r contient qu'un seul tuple. Dans ce cas, le data cube devient évident à calculer, l'algorithme s'arrête et ÉCRIRESUIVANTS est appelé (*cf.* algorithme 4). Ce dernier remplace, attribut après attribut, chaque valeur $'ALL'$ de la cellule courante par celle de l'unique tuple de la relation. Cette étape est particulièrement importante dans le cas des données peu denses où il y a potentiellement de nombreuse partitions très réduites. L'algorithme, dans ce cas, économise de nombreux appels récursifs et donc beaucoup de temps. Si la relation d'entrée n'est pas réduite à un seul tuple, la cellule courante est écrite. Expliquons maintenant le cœur de l'algorithme. Pour chacune des dimensions d_j passées en paramètre, la relation courante est partitionnée en fonction de ses valeurs sur l'attribut d_j. Deux tuples t et t' seront dans la même partition si et seulement si $t[d_j] = t'[d_j]$. Cette étape nécessite de faire un tri, c'est donc la plus coûteuse. Pour obtenir une efficacité optimale, les auteurs ont choisi d'utiliser un tri par comptage dont la complexité en temps est linéaire. PARTITIONNER est décrit dans l'algorithme 5. Une fois cette opération effectuée, BUC est appelé récursivement pour chacune des partitions vérifiant la contrainte iceberg. Lorsque toutes les partitions ont été traitées pour la dimension d_j, tous les cuboïdes contenant d_j l'ont été également. La valeur $'ALL'$ est alors mise dans la dimension d_j de la cellule courante.

Exemple 2.10 - La figure 2.9 montre le parcours du treillis des cuboïdes employé par l'algorithme BUC pour calculer le cube de données de la relation DOCUMENT (*cf.* table 2.1). Lors du premier appel, BUC calcule la valeur de la fonction agrégative en parcourant l'intégralité de la relation DOCUMENT obtenant ainsi le tuple multidimensionnel ('ALL', 'ALL', 'ALL') : 1100,

Algorithme 2 Algorithme BUC

Entrée :
Une relation r
L'ensemble des dimensions $\mathcal{D} = \{d_1, \ldots, d_n\}$ restant à traiter.
La cellule courante $CelluleCour$
Sortie :
Le cube de données de r.

1: $DimCour := d_1$;
2: AGRÉGER($r, CelluleCour,\ f$) ; //Calcule la valeur de la fonction mesure et l'écrit dans la cellule courante
3: **si** $|r| = 1$ **alors**
4: ÉCRIRESUIVANTS($r[0], \mathcal{D}, CelluleCour$) ; //La relation courante ne contient qu'un seul tuple, on peut écrire directement les résultats sans agréger
5: **retourner**
6: **fin si**
7: ÉCRIRE($CelluleCour$)
8: **pour tout** $d_j \in \mathcal{D}$ **faire**
9: C := $|r(d_j)|$;
10: PARTITIONNER($r,\ d_j$) ; //r est partitionnée suivant ses valeurs pour l'attribut d_j, en C fragments : $r_1, \ldots, r_C$
11: **pour** $i = 1, \ldots, C$ **faire**
12: **si** $|r_i| \geq MinSeuil$ **alors**
13: $CelluleCour.d_j := r_i[0].d_j$; //On affecte à la cellule courante la valeur de la dimension d_j pour r_i
14: BUC($r_i, \{d_{j+1}, \ldots, d_n\}, CelluleCour$) ;
15: **fin si**
16: **fin pour**
17: $CelluleCour.d_j := ALL$; //La dimension d_j est entièrement traitée
18: **fin pour**

Algorithme 3 Algorithme AGRÉGER

Entrée :
Une relation r
La cellule courante $CelluleCour$
Une fonction agrégative additive f

1: **pour tout** $t \in r$ **faire**
2: $CelluleCour.M = CelluleCour.M + f(t)$;
3: **fin pour**

Algorithme 4 Algorithme ÉCRIRESUIVANTS($t, \mathcal{D}, CelluleCour$)

Entrée :
Un tuple t
L'ensemble des dimensions restant à traiter $\mathcal{D} = \{d_1, \ldots, d_n\}$.
La cellule courante $CelluleCour$

1: **pour tout** $d_j \in \mathcal{D}$ **faire**
2: $CelluleCour.d_j := t.d_j$;
3: ÉCRIRE($CelluleCour$) ;
4: **fin pour**

directement écrit sur le disque.

Id	Éditeur	Langue	Type	Qté
1	Collins	Français	Nouvelles	400
2	Hachette	Anglais	Nouvelles	100
3	Hachette	Français	Pédagogie	100
4	Collins	Anglais	Pédagogie	300
5	Collins	Français	Essai	200

Aggréger → Écriture sur disque : ('ALL','ALL','ALL') : 1100

Une fois cette étape réalisée, BUC partitionne la relation DOCUMENT suivant les valeurs de l'attribut *Éditeur* (le premier dans l'ordre alphabétique). Cette dimension prenant pour valeurs simplement *Collin* et *Hachette*, la relation est alors découpée en deux fragments $P_1 = \{t_1, t_4, t_5\}$ et $P_2 = \{t_2, t_3\}$.

Id	Editeur	Langue	Type	Qté
1	Collins	Français	Nouvelles	400
2	Hachette	Anglais	Nouvelles	100
3	Hachette	Français	Pédagogie	100
4	Collins	Anglais	Pédagogie	300
5	Collins	Français	Essai	200

P_1 →

Id	Editeur	Langue	Type	Qté
1	Collins	Français	Nouvelles	400
4	Collins	Anglais	Pédagogie	300
5	Collins	Français	Essai	200

P_2 →

Id	Editeur	Langue	Type	Qté
2	Hachette	Anglais	Nouvelles	100
3	Hachette	Français	Pédagogie	100

L'algorithme traite ensuite les fragments un par un. Pour P_1, la contrainte iceberg est controlée et si elle est satisfaite, la cellule courante est mise à jour avec la valeur des tuples de P_1 sur la dimension *Éditeur* ($CelluleCour$ = (Collins, 'ALL', 'ALL')). Puis BUC est appelé récursivement pour P_1.

Id	Éditeur	Langue	Type	Qté
1	Collins	Français	Nouvelles	400
4	Collins	Anglais	Pédagogie	300
5	Collins	Français	Essai	200

Aggréger → Écriture sur disque : (Collins,'ALL','ALL') : 900

P_1 est partitionné sur la dimension suivante, *Langue*, en deux nouveaux fragments $P_3 = \{t_4\}$ et $P_4 = \{t_1, t_5\}$.

Algorithme 5 Algorithme PARTITIONNER

Entrée :

Une relation r

La dimension selon laquelle r doit être partitionnée $dimCour$

Sortie :

La relation r partitionnée suivant les valeurs de l'attribut $dimCour$ en C fragments : $r_1, \ldots, r_C$

1: **soit** C : un entier contenant le nombre de valeurs distinctes dans r pour l'attribut $dimCour$
2: **soit** $hist$: un tableau de $|Dom(dimCour)|$ entiers intialisés à 0
3: **soit** $nbTuples$: un entier initialisé avec le nombre de tuples contenus dans r
4: **pour** $i = 1, \ldots, nbTuples$ **faire**
5: $hist[t_i.dimCour] := hist[t_i.dimCour]+1$; //On compte le nombre d'occurences de chaque valeur de l'attribut $dimCour$
6: **fin pour**
7: //On exploite l'histogramme
8: **soit** $indTab$: un tableau d'entiers initialisés à 0 ;
9: **soit** $premier$: un entier initialisé à 0 ;
10: **soit** $dernier$: un entier initialisé à 0 ;
11: **pour** $i = 1, \ldots, |Dom(dimCour)|$ **faire**
12: **si** $hist[i] \neq 0$ **alors**
13: $C := C + 1$;
14: **fin si**
15: $dernier := hist[i]$;
16: $indTab[i] := premier$;
17: $premier := premier + dernier$;
18: **fin pour**
19: **soit** $indTab2$: une copie locale de $indTab$;
20: **pour** $i = 1, \ldots, nbTuples$ **faire**
21: **soit** v : la valeur du tuple t_i pour la dimension $dimCour$;
22: $rTmp[indTab2[v]] := t_i$;
23: $indTab2[v] := indTab2[v] + 1$;
24: **fin pour**
25: **soit** j : un entier initialisé à 1 ;
26: **pour** $i = 1, \ldots, |Dom(dimCour)|$ **faire**
27: **si** $hist[i] \neq 0$ **alors**
28: $r_j := rTmp[indTab[i]..indTab[i] + hist[i]]$;
29: $j := j + 1$;
30: **fin si**
31: **fin pour**
32: **retourner** $r_1, \ldots, r_C$;

Id	Editeur	Langue	Type	Qté
1	Collins	Français	Nouvelles	400
4	Collins	Anglais	Pédagogie	300
5	Collins	Français	Essai	200

P_3

Id	Editeur	Langue	Type	Qté
4	Collins	Anglais	Pédagogie	300

P_4

Id	Editeur	Langue	Type	Qté
1	Collins	Français	Nouvelles	400
5	Collins	Français	Essai	200

$P3$ est maintenant traité, la cellule courante devient $CelluleCour$ = (Collins, Anglais, 'ALL'). Le fragment ne contenant plus qu'un seul tuple il est inutile de calculer la valeur de la fonction aggrégative. L'algorithme écrit directement sur disque tous les déscendants de (Collins, Anglais, 'ALL').

Id	Éditeur	Langue	Type	Qté
4	Collins	Anglais	Pédagogie	300

EcrireSuivant →

Écriture sur disque :

(Collins,Anglais,'ALL') : 300
(Collins,Anglais,Pédagogie) : 300

Le même processus est exécuté pour le fragment P_4.

Id	Éditeur	Langue	Type	Qté
1	Collins	Français	Nouvelles	400
5	Collins	Français	Essai	200

Aggréger →

Écriture sur disque :

(Collins, Français,'ALL') : 600

Id	Editeur	Langue	Type	Qté
1	Collins	Français	Nouvelles	400
5	Collins	Français	Essai	200

P_5

Id	Editeur	Langue	Type	Qté
5	Collins	Français	Essai	200

P_6

Id	Editeur	Langue	Type	Qté
1	Collins	Français	Nouvelles	400

Id	Éditeur	Langue	Type	Qté
5	Collins	Français	Essai	200

EcrireSuivant →

Écriture sur disque :

(Collins, Français, Essai) : 200

Id	Éditeur	Langue	Type	Qté
1	Collins	Français	Nouvelles	400

EcrireSuivant →

Écriture sur disque :

(Collins, Français, Nouvelles) : 400

Chacun des fragments P_5 et P_6 n'est constitué que d'un seul tuple, les résultats sont écrits sans faire de calcul. L'algorithme remonte jusqu'à P_1 qu'il repartitionne suivant l'attribut *Type*. Il obtient ainsi trois nouveaux fragments P_7, P_8 et P_9. Chacun d'entre eux est constitué d'un unique tuple, l'algorithme écrit les tuples multidimensionnels associés sans calcul.

Id	Editeur	Langue	Type	Qté
1	Collins	Français	Nouvelles	400
4	Collins	Anglais	Pédagogie	300
5	Collins	Français	Essai	200

P_7

Id	Editeur	Langue	Type	Qté
5	Collins	Français	Essai	200

P_8

Id	Editeur	Langue	Type	Qté
1	Collins	Français	Nouvelles	400

P_9

Id	Editeur	Langue	Type	Qté
4	Collins	Anglais	Pédagogie	300

FIGURE 2.10 – Première partie de l'arbre de parcours des partitions par l'algorithme BUC pour la relation DOCUMENT

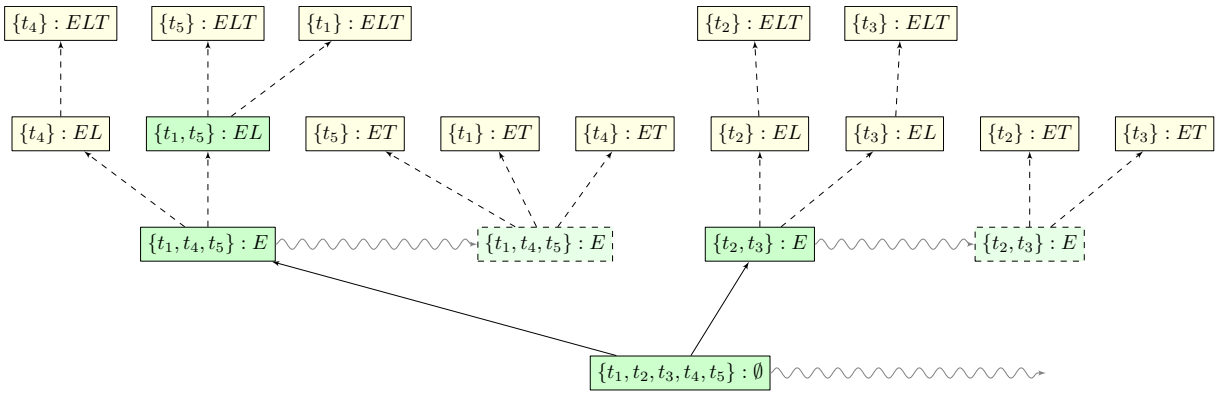
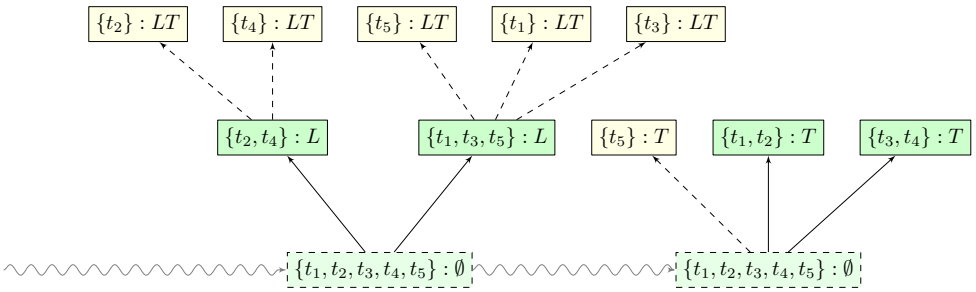


FIGURE 2.11 – Seconde partie de l'arbre de parcours des partitions par l'algorithme BUC pour la relation DOCUMENT



Id	Éditeur	Langue	Type	Qté
5	Collins	Français	Essai	200

EcrireSuivant → Écriture sur disque : (Collins,'ALL', Essai) : 200

Id	Éditeur	Langue	Type	Qté
1	Collins	Français	Nouvelles	400

EcrireSuivant → Écriture sur disque : (Collins,'ALL', Nouvelles) : 400

Id	Éditeur	Langue	Type	Qté
4	Collins	Anglais	Pédagogie	300

EcrireSuivant → Écriture sur disque : (Collins,'ALL',Pédagogie) : 300

Une fois tous ces fragments traités, l'algorithme remonte et traite P_2 récursivement. Les cuboïdes contenant la dimension *Éditeur* ont été générés. Cette dimension est donc ignorée par BUC à partir de maintenant. L'algorithme repart de la relation d'origine en démarrant le partitionnement avec la dimension *Langue* et effectue le processus précédemment décrit. BUC se termine lorsque tous les attributs de la relation DOCUMENT ont été traités de la sorte.

Les figures 2.10 et 2.11 donnent les détails de tous les fragments calculés par BUC durant le parcours de son espace de recherche. À l'intérieur de chaque nœud figurent les tuples de ce fragment ainsi que les attributs de partitionnement. Les arêtes en pointillés correspondent aux nœuds obtenus sans calcul d'agrégation grâce à la fonction ECRIRESUIVANT. Les nœuds en pointillés, quant à eux, correspondent au moment où BUC fait un retour sur trace et réutilise la même entrée pour continuer son calcul.

2.4 Méthodes de calcul du cube de données partiel

Dans le paragraphe précédent, nous avons présenté les approches de calcul du cube de données complet. Avec de telles méthodes, la réponse à chaque requête des utilisateurs peut se faire en un temps extrêmement court. Le principal problème posé par cette approche est que la taille du cube de données peut être plusieurs ordres de magnitudes au dessus de celle de la relation d'entrée. Le stockage de tous les agrégats peut donc devenir critique dans certains cas. Pour trouver un compromis entre le temps de réponse et le coût de stockage, les approches de matérialisation partielle (aussi appelées vues matérialisées) choisissent un sous-ensemble minimal de cuboïdes permettant de conserver un temps de réponse rapide aux requêtes.

2.4.1 L'algorithme Glouton

L'algorithme Glouton (Harinarayan *et al.*, 1996) est la première approche permettant de choisir un sous-ensemble du cube adéquat à la fois pour le calcul et pour le stockage. Trouver un tel sous-ensemble minimal du treillis des cuboïdes est un problème NP-Difficile. Les algorithmes utilisent généralement des heuristiques pour trouver une solution acceptable.

Comme son nom l'indique l'algorithme, présenté par Harinarayan *et al.* (1996), est basé sur une stratégie gloutonne pour éviter une recherche exhaustive de l'ensemble des solutions. L'algorithme suppose disposer à l'avance des coûts de stockage de chaque cuboïde. Dans la pratique, nous ne disposons jamais d'une telle information mais on peut en déterminer rapidement une approximation satisfaisante en utilisant par exemple, les méthodes proposées par Shukla *et al.* (1996). Nous notons $C(v)$ le coût associé au nœud v. L'ensemble S des cuboïdes choisis par l'algorithme doit toujours contenir le cuboïde de base (nœud le moins agrégé du datacube), car il n'existe aucun nœud du treillis permettant de répondre aux requêtes portant sur ce cuboïde. En plus du nœud racine, le nombre de cuboïdes contenus dans S est limité à k. Pour chaque nœud v non encore dans S, on définit une fonction bénéfice qui retourne une quantité correspondant à l'amélioration apportée au temps de réponse utilisateur par l'ajout du nœud v à l'ensemble S. Elle est notée $B(v,S)$ et calculée comme suit :

1. Pour chaque $w \preceq v$ la quantité B_w est définie de la manière suivante. Soit $u \in S$ le nœud de moindre coût tel que $w \preceq u$. $B_w = \max(C(v) - C(u),0)$
2. $B(v,S) = \sum_{w \preceq v} B_w$

L'algorithme Glouton est itératif, à chaque étape il ajoute à S le nœud le plus bénéfique. Il se termine après k itérations. Sa description est donnée dans l'algorithme 6

Algorithme 6 Algorithme Glouton$(\mathcal{D}, k)$

Entrée :
L'ensemble des dimensions $\mathcal{D} = \{d_1, \ldots, d_n\}$ du cube de données.
Le nombre de cuboïdes à sélectionner k
Sortie : L'ensemble S des cuboïdes à matérialiser

soit $S := cuboide_{\mathcal{D}}$ //S est initialisé avec le cuboïde de base
pour $i = 1 \ldots k$ **faire**
  Choisir le cuboïde $v \notin S$ maximisant $B(v,S)$
  $S := S \cup v$;
fin pour
retourner S

La complexité de l'algorithme GLOUTON est $O(k \times n^2)$ avec n le nombre de nœuds du treillis et k le nombre de cuboïdes à matérialiser. Harinarayan *et al.* ont prouvé que le bénéfice total de l'algorithme est au moins 0.63 fois le bénéfice total de l'algorithme optimal. Cette garantie de performance n'implique pas une solution quasi optimale mais permet au moins d'être certain de n'avoir jamais une solution trop irréaliste.

2.4.2 L'algorithme PBS

La complexité de l'algorithme GLOUTON est quadratique par rapport au nombre de cuboïdes n. Ce nombre étant égal à $2^{|\mathcal{D}|}$, GLOUTON est inapplicable dès que le nombre de dimensions de la relation d'entrée devient trop important. PBS (Shukla *et al.*, 1998) a été proposé pour contourner ce problème en parcourant uniquement la partie du treillis des cuboïdes qui lui est utile. Contrairement à l'algorithme précédent qui avait comme contrainte le nombre de cuboïdes à matérialiser, PBS (Shukla *et al.*, 1998) a pour limite la quantité mémoire à sa disposition pour stocker les cuboïdes choisis (hormis le cuboïde de base). Cette limitation est bien plus naturelle car l'administrateur d'un entrepôt est davantage préoccupé par l'espace disque requis que par le nombre de cuboïdes résultats. En effet, dans la pratique le fait d'avoir 10 ou 1000 vues matérialisées n'a que peu d'importance tant que la capacité de stockage est suffisante.

PBS commence par initialiser l'ensemble S avec la racine du treillis puis, il parcourt itérativement l'ensemble des cuboïdes. À chaque étape, il sélectionne le nœud le plus petit, vérifie qu'il entre en mémoire et l'ajoute à S. L'algorithme se termine lorsque l'espace restant n'est plus suffisant pour le moindre cuboïde. La description de PBS est donnée dans l'algorithme 7.

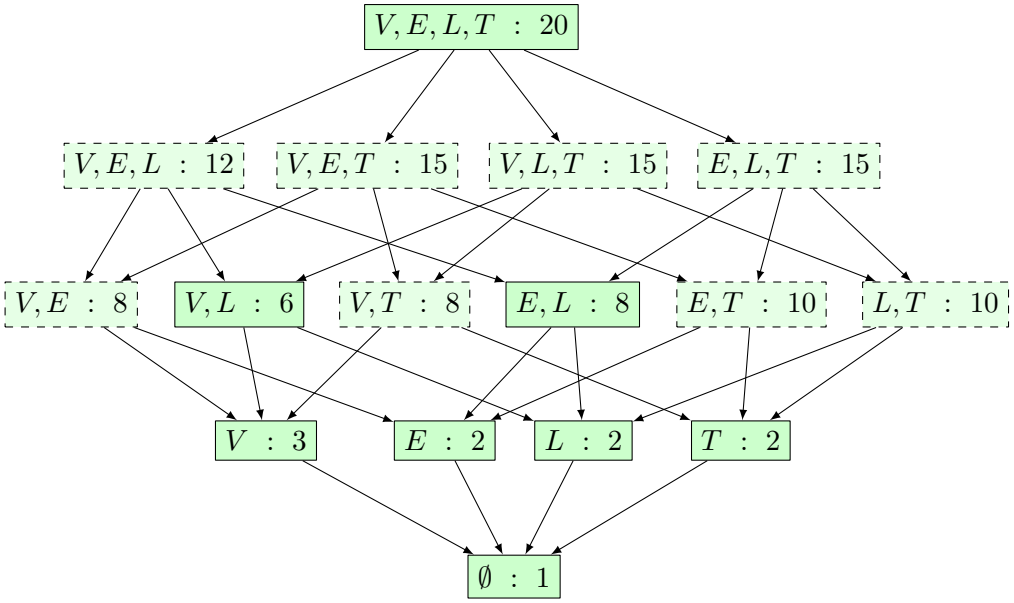
Algorithme 7 Algorithme PBS($\mathcal{D}, espace$)

Entrée :
L'ensemble des dimensions $\mathcal{D} = \{d_1, \ldots, d_n\}$ du cube de données.
La capacité mémoire à disposition *espace*.
Sortie : L'ensemble S des cuboïdes à matérialiser

soit $S := cuboide_{\emptyset}$ //S est initialisé avec le cuboïde selon $\emptyset$
tant que $espace > 0$ **faire**
  Choisir le cuboïde v non sélectionné ayant la plus petite taille.
  si $espace - taille(v) \geq 0$ **alors**
    $espace := espace - taille(v)$;
    $S := S \cup v$;
  sinon
    $espace := 0$; //Espace insuffisant pour le plus petit cuboïde
  fin si
fin tant que
retourner $S \cup cuboide_{\mathcal{D}}$

L'algorithme choisissant les cuboïdes par ordre de taille, le parcours se fait des cuboïdes les plus agrégés vers les plus détaillés. Comme BUC, il adopte donc une stratégie dite *bottom-up*. En procédant ainsi, l'algorithme crée une sorte de bordure qui sépare les nœuds sélectionnés de ceux qui ne le sont pas. La complexité en temps de PBS est $O(n \times \log n)$, elle est liée au coût du tri effectué sur la taille des cuboïdes pour pouvoir les choisir du plus petit au plus grand. La stratégie employée est très simple, elle est particulièrement efficace vis à vis du temps de réponse. En contrepartie, cette rapidité est obtenue en sacrifiant la garantie de qualité du résultat dans le cas général.

FIGURE 2.12 – Treillis des cuboïdes sélectionnés par l'algorithme PBS pour la relation DOCUMENT



Exemple 2.11 - La figure 2.12 montre les cuboïdes choisis par l'algorithme PBS pour la relation DOCUMENT (*cf.* table 2.1) avec 25 pages de mémoire à disposition. Les nœuds en pointillés correspondent aux cuboïdes non sélectionnés.

2.4.3 L'algorithme Key

L'algorithme KEY (Kotsis et McGregor, 2000) adopte une approche innovante pour choisir le sous-ensemble du treillis des cuboïdes à matérialiser. Au lieu de se focaliser sur une optimisation du coût de réponse aux requêtes de l'utilisateur, cet algorithme choisit de ne pas matérialiser les cuboïdes qu'il considère comme redondants. Pour les auteurs, un cuboïde est redondant s'il peut être obtenu à partir d'un autre cuboïde par une simple projection. Par exemple, en reprenant notre relation DOCUMENT (*cf.* table 2.1), la figure 2.2 montre que le cuboïde $cuboide_{\{Type,Editeur\}}$ est redondant car il peut être obtenu par projection de $cuboide_{\{Type,Ville,Editeur\}}$. Les requêtes portant sur les cuboïdes redondants sont peu coûteuses à calculer. Le coût de stockage est ainsi diminué sans augmenter le temps de réponse.

Pour identifier la redondance, Kotsis et McGregor définissent le concept de *g-équivalence*. Un tuple multidimensionnel $t \in cuboide_X$ est *g-équivalent* à $t' \in cuboide_Y$ si et seulement si, $Y \subset X$ et que $t.\mathcal{M} = t'.\mathcal{M}$. En d'autres termes, deux tuples sont *g-équivalents* s'ils peuvent être obtenus l'un à partir de l'autre sans agrégation. Ils véhiculent tous deux exactement la même information. De plus, un cuboïde $cuboide_X$ est dit *g-équivalent* à $cuboide_Y$ si et seulement si tous les tuples de $cuboide_X$ sont *g-équivalents* à ceux de $cuboide_Y$ et $|cuboide_X| = |cuboide_Y|$. De tels nœuds sont redondants. Ils peuvent être obtenus l'un à partir de l'autre par simple projection. Il suffit donc d'en conserver un.

L'algorithme KEY tire son nom du fait qu'il va rechercher les clefs candidates pour trouver les cuboïdes redondants. Il parcourt le treillis des cuboïdes de bas en haut et niveau par niveau. Une fois une clef candidate X identifiée, comme tout sur-ensemble de cette clef est aussi une clef, tous

les cuboïdes parents de $cuboide_X$ possèdent le même nombre de tuples. Ainsi, tous ces nœuds sont *g-équivalents*, donc redondants, et n'auront pas à être matérialisés. L'algorithme élague toute une partie du treillis chaque fois qu'il trouve une clef candidate. C'est pour cela que le parcours de bas en haut se prête le mieux à cet algorithme. KEY est décrit dans l'algorithme 8. Pour éliminer un plus grand nombre de nœuds, les auteurs proposent d'appeler l'algorithme récursivement pour chacun des nœuds restants.

Algorithme 8 Algorithme KEY$(\mathcal{D}, r)$

Entrée :
L'ensemble des dimensions $\mathcal{D} = \{d_1, \ldots, d_n\}$ du cube de données.
La relation d'entrée r.
Sortie : L'ensemble K des clefs candidates.

soit $K := \emptyset$;
soit $i := 0$;
tant que $i < 2^{|\mathcal{D}|}$ **faire**
  soit *cuboideCour* le i-ème cuboide du cube de données de r.
  soit *DimensionsCour* le sous-ensemble de $\mathcal{D}$ sur lequel porte *cuboideCour*.
  si $|cuboideCour| < |r.size|$ **alors**
    si $\exists k \in K$ tel que $k \subset DimensionsCour$ **alors**
      $i := i + 1$; //C'est un cuboïde redondant
    sinon si il existe un duplicat dans $r[DimensionsCour]$ **alors**
      $i := i + 1$; //Ce cuboïde s'agrège donc *DimensionsCour* n'est pas une clef
    sinon
      $K := K \cup cuboideCour$;
    fin si
  fin si
  $i := i + 1$;
fin tant que
retourner K

Exemple 2.12 - Avec la relation DOCUMENT (*cf.* table 2.1) et $\mathcal{D} = \{Type, Ville, Editeur\}$ l'algorithme KEY identifie $\{Type, Editeur\}$ comme la seule clef candidate.

Cette approche, qui en éliminant la redondance du cube de données optimise les calculs et le stockage, a vraiment été novatrice par rapport à ce qui existait en OLAP. Le principal défaut de cette méthode est qu'elle recherche les redondances à un niveau de granularité élevé (au niveau des cuboïdes). Une telle redondance n'arrive que peu fréquemment et a de fortes chances de disparaître lors des rafraichissements. En effet, un seul tuple peut suffire à rendre non redondants plusieurs cuboïdes alors que globalement la redondance existe toujours.

2.5 Conclusion

Le cube de données est le concept majeur pour la gestion des entrepôts de données. Dans ce chapitre, nous avons proposé un survol de travaux centrés sur ce concept. Plus précisément, nous avons rappelé ce qu'est une cube de données et deux des variantes qui ont été introduites. Ces dernières intègrent une ou deux contraintes portant sur les tuples du cube et ainsi en produisent des versions partielles. Puis nous nous sommes focalisés sur les approches permettant de calculer

des cubes complets et partiels. De très nombreux travaux ont abordé cette problématique si bien qu'il était délicat d'en brosser un tableau exhaustif. Un des principaux critères de choix qui nous a fait retenir les approches présentées est leur propriété d'intégration au sein des SGBD. Cette qualité reste rare car de nombreuses approches ont recours à des structures de données trop spécifiques et non adaptables dans un contexte relationnel. Citons les tries, FP-trees et leurs extensions (Han *et al.*, 2000; Pei *et al.*, 2000; Han *et al.*, 2001, 2005; Xin *et al.*, 2006; Cheng *et al.*, 2006; Wang *et al.*, 2007; Xin *et al.*, 2007). Les algorithmes que nous avons décrits représentent une sorte d'historique en matière de calcul de cubes. Le point commun de la première génération d'approches est de se concentrer sur la construction du meilleur arbre d'exécution possible à partir du treillis des cuboïdes. L'idée sous-tendant ces approches est de ré-exploiter des résultats intermédiaires afin d'optimiser les temps de calcul. Évidemment, ceci implique un parcours des cuboïdes les plus détaillés vers ceux qui sont de plus en plus agrégés. Avec la seconde génération, l'idée de construire un arbre d'exécution a été abandonnée. Ainsi en se libérant de cette contrainte, le sens de parcours du treillis a été inversé, donnant naissance à des algorithmes bien plus efficaces que leurs prédécesseurs. Nous avons présenté PARTITIONED-CUBE car il est incontournable pour toute méthode souhaitant s'affranchir des problèmes d'accès à la mémoire secondaire. Les raisons qui nous ont fait retenir BUC sont ses très bonnes performances, sa simplicité, sa polyvalence et bien sur, son intégrabilité.

La leçon à retenir de cet état de l'art est que le treillis des cuboïdes ne peut pas constituer un espace de recherche satisfaisant pour calculer efficacement les cubes de données. En effet, en examinant l'algorithme BUC et plus précisément son parcours, pourtant présenté comme celui du treillis des cuboïdes, il apparaît que le niveau de granularité requis est celui des tuples et non des cuboïdes. C'est de cette particularité que BUC tire sa très grande efficacité. Il est assez étonnant de constater qu'à ce stade, le cube de données n'est vu que comme le résultat d'un calcul (soit celui d'une requête soit celui plus performant d'un algorithme) et n'est pas précisément défini.

Nous explorerons dans le chapitre suivant les approches qui ont proposé des structures pour formaliser solidement le cube de données.

3

Analyse des bases de données multidimensionnelles

Sommaire

3.1 Introduction

Il existe bien sûr des points communs et des convergences entre les travaux sur les cubes de données et les approches de fouille de données classiques. Témoin, l'algorithme BUC (Beyer et Ramakrishnan, 1999) qui tire profit des propriétés exploitées par APRIORI (Agrawal *et al.*, 1996) pour optimiser le calcul de l'iceberg cube. Bien que développés dans un contexte de données binaires, les algorithmes de fouille peuvent être adaptés pour le calcul de cubes. Mais ce type d'extension a des effets largement indésirables. Typiquement, les approches binaires classiques considèrent un ensemble d'items qui représentent les valeurs d'un seul attribut (*e.g.* l'identifiant des produits pour les bases de transactions commerciales). Dès lors, transposer des solutions performantes lorsqu'un seul attribut est considéré au cas où quelques dizaines d'attributs doivent être manipulés a deux conséquences importantes :

1. si l'on considère le treillis des parties comme l'espace de recherche dans un contexte multidimensionnel, l'ensemble des items à considérer est l'ensemble de toutes les valeurs de

tous les attributs dimensions. Il y a alors explosion de l'espace de recherche. Dans le cadre d'applications OLAP, le problème s'aggrave car les attributs dimensions peuvent avoir des domaines de définition extrêmement larges (Ross et Srivastava, 1997; Beyer et Ramakrishnan, 1999) ;

2. À partir de l'ensemble des valeurs de tous les attributs, le treillis des parties contient toutes les combinaisons possibles de ces valeurs. Ces dernières, par construction même, vont pouvoir associer deux valeurs d'un même attribut. De telles combinaisons sont sémantiquement invalides d'un point de vue multidimensionnel. En effet, le point de départ du calcul de cube étant une relation classique et normalisée (1-NF), toute combinaison incluant deux valeurs d'un même attribut est un « *motif impossible* ». Pourtant, il est considéré par les approches transposant des solutions binaire à l'environnement multidimensionnel (Srikant et Agrawal, 1996; Liu *et al.*, 1998; Li *et al.*, 2001). Il en résulte une triple conséquence : (*i*) le calcul des mesures, qui nécessite un parcours des données, est effectué pour ces motifs impossibles alors qu'il est inutile, (*ii*) si la contrainte antimonotone de fréquence est mise en œuvre, de tels motifs sont éliminés du résultat mais dégradent la complexité et (*iii*) pour d'autres types de contraintes monotones ou antimonotones, ces motifs impossibles deviennent bel et bien des solutions du problème.

Pour éviter les problèmes présentés, certaines méthodes dédiées au contexte OLAP ont été proposées. Ce chapitre leur est consacré.

Deux approches ont concurremment tenu compte de la spécificité des données multidimensionnelles à savoir qu'elles sont structurées. En effet, dans le contexte OLAP, il ne s'agit pas de manipuler des ensembles d'items mais des tuples dont chaque valeur a une sémantique particulière en fonction de l'attribut concerné.

Ces deux approches, le cube fermé (Casali *et al.*, 2003b) et le cube quotient (Lakshmanan *et al.*, 2002), ont parallèlement caractérisé un espace de recherche similaire, le treillis cube. Tous ses tuples sont des solutions potentielles sémantiquement valides. Cet espace est muni d'une relation d'ordre entre tuples, la généralisation (ou de façon duale la spécialisation), et de plusieurs opérateurs de parcours.

Outre le fait que ce treillis soit commun à deux des approches qui nous intéressent, cet espace de recherche propose un cadre de travail formellement et solidement défini dans les contributions présentées ultérieurement. Pour ces raisons, nous le présentons séparément des approches qui l'ont originellement défini.

La première des approches présentées, le cube fermé (Casali *et al.*, 2003b), en s'inspirant des motifs fermés fréquents (Pasquier *et al.*, 1999), vise à éliminer les redondances intrinsèquement présentes au sein d'un cube. Le résultat est une structure solidement fondée et sans perte d'information. À partir de cette structure, les réponses aux requêtes sur les cubes peuvent être dérivées. De plus elle a le mérite d'être, à notre connaissance, l'une des représentations sans perte parmi les plus réduites d'un cube de données. Cette réduction se fait au prix d'une impossibilité de naviguer entre les différents niveaux de granularité des données du cube. La deuxième approche présentée, le cube quotient, a l'objectif similaire d'éliminer la redondance tout en conservant les capacités de parcours à travers le cube. Le lien entre ces deux approches est également décrit.

À l'inverse des cubes quotient et fermé qui s'intéressent à réduire la représentation des cubes de données en écartant les redondances « sémantiques », la troisième approche présentée, le Cube partition (Laporte *et al.*, 2002), tire profit des redondances « syntaxiques » pour proposer une représentation compacte du cube de données. Elle s'appuie sur le concept de partition (Spyratos, 1987) et met en œuvre l'opérateur de produit de partitions (Spyratos, 1987; Novelli et Maabout, 2003).

3.2 Cadre formel

3.2.1 Le treillis cube

Contrairement aux motifs recherchés en fouille de bases de données binaires, les motifs multidimensionnels sont dotés d'une structure et il importe de les caractériser en exhibant cette structure. D'autre part, les liens qui existent entre de tels motifs véhiculent une sémantique importante dans un contexte de travail multidimensionnel.

La solution proposée par Casali *et al.* (2003a) a consisté à introduire la notion d'espaces multidimensionnels dont les éléments représentent les tuples multidimensionnels possibles. Pour formaliser leurs liens, ils s'appuient sur une relation d'ordre et définissent deux opérateurs fondamentaux de construction. Dès lors, les espaces de recherche pour les problèmes de fouille de bases multidimensionnelles peuvent être caractérisés, ce qui est fait en définissant le concept de treillis cube. Dans ce contexte multidimensionnel, le treillis cube est un nouveau langage de description de concepts qui peut être utilisé non seulement en fouille de bases de données multidimensionnelles mais aussi en apprentissage automatique (Mitchell, 1997).

Espaces multidimensionnels

Comme dans le chapitre précédent, nous considérons une relation r dont le schéma $\mathcal{R}$ comporte deux types d'attributs : $\mathcal{D}$ l'ensemble des attributs dimensions et $\mathcal{M}$ l'ensemble des attributs mesures (pour l'OLAP). L'espace multidimensionnel de la relation r regroupe toutes les combinaisons sémantiquement valides des ensembles de valeurs existant dans r pour les attributs dimensions, enrichis de la valeur ALL. Cette dernière, introduite dans (Gray *et al.*, 1997) pour la définition de l'opérateur Cube-By, est une généralisation de toutes les valeurs possibles de tout attribut. Ainsi $\forall A \in \mathcal{D}, \forall a \in Dim(A),\ \{a\} \subset \text{ALL}$ (*cf.* paragraphe 2.2 p. 13).

Définition 3.1 (Espace multidimensionnel) **-** L'espace multidimensionnel de r est noté et défini comme suit : $Space(r) = \{\times_{A \in \mathcal{D}}(Dim(A) \cup \{ALL\})\} \cup \{(\emptyset, \ldots, \emptyset)\}$ où $\times$ symbolise le produit cartésien, et $(\emptyset, \ldots, \emptyset)$ la combinaison de valeurs vides. Toute combinaison de $Space(r)$ est un tuple et représente un motif multidimensionnel.

Exemple 3.1 - Considérons à nouveau la relation Document utilisée dans le chapitre précédent (*cf.* table 3.1). Son espace multidimensionnel est illustré par le tableau 3.2.

Les tuples d'identifiants 1, ..., 12 et 37 sont des tuples possibles pour r (car toutes leurs valeurs sont réelles), même si le tuple d'identifiant 37 n'est pas présent explicitement dans la relation r. Dans ce tuple, le symbole $\emptyset$ signifie « valeur vide ». Les autres tuples (identifiants 13, ..., 36) ne peuvent pas être des tuples de r car ils comportent au moins une valeur ALL. Ils véhiculent donc une information à un niveau de détail plus agrégé que les précédents.

Ordre de généralisation

Les tuples véhiculent une information à différents niveaux de granularité et, dans l'espace multidimensionnel, ils sont, par nature même, redondants. En effet, tout tuple de la relation initiale participe généralement à la construction de plusieurs tuples (premier niveau de synthèse de données) ; ces derniers peuvent à nouveau être synthétisés et ainsi de suite. Au niveau le plus général, la synthèse consiste en un unique tuple dont toutes les valeurs sont ALL et qui résume la relation initiale r de la manière la plus compacte qui soit, mais aussi la plus grossière.

TABLE 3.1 – Relation exemple DOCUMENT

RowId	Type	Ville	Éditeur	Langue	Quantité
1	Nouvelles	Marseille	Collins	Français	400
2	Nouvelles	Marseille	Hachette	Anglais	100
3	Pédagogie	Paris	Hachette	Français	100
4	Pédagogie	Marseille	Collins	Anglais	300
5	Essai	Paris	Collins	Français	200

L'espace multidimensionnel de r est structuré par la relation de généralisation/spécialisation entre tuples. Cet ordre est originellement introduit par T. Mitchell (Mitchell, 1982, 1997) dans le cadre de l'apprentissage de concepts.

Définition 3.2 (Généralisation/Spécialisation) **-** Soit u, v deux tuples de l'espace multidimensionnel de r, nous définissons la relation d'ordre $\preceq_g$ comme suit :

$$u \preceq_g v \Leftrightarrow \begin{cases} \forall A \in \mathcal{D} \text{ tel que } u[A] \neq \text{ALL}, u[A] = v[A] \\ \text{où } v = (\emptyset, \ldots, \emptyset) \end{cases}$$

Si $u \preceq_g v$, nous disons que u est plus général que v dans $Space(r)$. La relation de généralisation $\preceq_g$ est un ordre dual à la relation de spécialisation ($\succeq_s$). La relation de couverture de $\preceq_g$ est notée $\prec_g$ et définie comme ceci : $\forall t,t' \in Space(r), t \prec_g t' \Leftrightarrow t \preceq_g t'$ et $\forall t'' \in Space(r), t \preceq_g t'' \preceq_g t' \Rightarrow t = t''$ ou $t' = t''$.

Exemple 3.2 - Dans l'espace multidimensionnel de notre relation exemple (*cf.* table 3.2), en considérant que t_i est le tuple d'identifiant i, nous avons : $t_{13} \preceq_g t_1$ et $t_{29} \preceq_g t_1$. Donc t_{13} et t_{29} sont plus généraux que t_1 et t_1 plus spécifique que t_{13} et t_{29}. De plus, nous avons $t_{13} \succ_g t_1$ et $t_{29} \not\succ_g t_1$.

Appliqués sur un ensemble de tuples, les opérateurs *min* et *max* rendent respectivement les tuples les plus généraux et les plus spécifiques de l'ensemble considéré.

Définition 3.3 (Opérateurs min/max selon $\preceq_g$) **-** Soit un ensemble de tuples $T \subseteq Space(r)$:

$$min_{\preceq_g}(T) = \{t \in T \mid \nexists u \in T : u \preceq_g t\}$$
$$max_{\preceq_g}(T) = \{t \in T \mid \nexists u \in T : t \preceq_g u\}$$

En utilisant l'ordre dual de la généralisation (*i.e.* la spécialisation), nous avons : $min_{\succeq_s}(T) = max_{\preceq_g}(T)$ et $max_{\succeq_s}(T) = min_{\preceq_g}(T)$.

Exemple 3.3 - Dans notre espace multidimensionnel exemple (*cf.* table 3.2), soit un ensemble de tuples $T = \{t_1, t_{13}, t_{23}\}$. Nous avons $min(T) = \{t_{13}, t_{23}\}$ et $max(T) = \{t_1\}$.

Opérateurs de base

Nous présentons les deux opérateurs de construction de tuples : Somme (notée $+$) et Produit (noté $\bullet$). Le Semi-Produit (noté $\odot$) est une variante contrainte de l'opérateur produit.

La somme de deux tuples rend le tuple le plus spécifique généralisant les deux opérandes. Elle est définie comme suit.

TABLE 3.2 – Espace multidimensionnel de la relation DOCUMENT

RowId	Type	Ville	Éditeur
1	Nouvelles	Marseille	Collins
2	Nouvelles	Marseille	Hachette
3	Nouvelles	Paris	Collins
4	Nouvelles	Paris	Hachette
5	Pédagogie	Marseille	Hachette
6	Pédagogie	Marseille	Collins
7	Pédagogie	Paris	Hachette
8	Pédagogie	Paris	Collins
9	Essai	Marseille	Hachette
10	Essai	Marseille	Collins
11	Essai	Paris	Hachette
12	Essai	Paris	Collins
13	Nouvelles	Marseille	*ALL*
14	Nouvelles	Paris	*ALL*
15	Pédagogie	Paris	*ALL*
16	Pédagogie	Marseille	*ALL*
17	Essai	Paris	*ALL*
18	Essai	Marseille	*ALL*
19	Nouvelles	*ALL*	Collins
20	Nouvelles	*ALL*	Hachette
21	Pédagogie	*ALL*	Collins
22	Pédagogie	*ALL*	Hachette
23	Essai	*ALL*	Collins
24	Essai	*ALL*	Hachette
25	*ALL*	Marseille	Collins
26	*ALL*	Marseille	Hachette
27	*ALL*	Paris	Collins
28	*ALL*	Paris	Hachette
29	Nouvelles	*ALL*	*ALL*
30	Pédagogie	*ALL*	*ALL*
31	Essai	*ALL*	*ALL*
32	*ALL*	Marseille	*ALL*
33	*ALL*	Paris	*ALL*
34	*ALL*	*ALL*	Collins
35	*ALL*	*ALL*	Hachette
36	*ALL*	*ALL*	*ALL*
37	$\emptyset$	$\emptyset$	$\emptyset$

Définition 3.4 (Opérateur Somme) **-** Soit u et v deux tuples de $Space(r)$, $t = u + v \Leftrightarrow \forall A \in \mathcal{D}$,

$$t[A] = \begin{cases} u[A] \text{ si } u[A] = v[A] \\ \text{ALL sinon.} \end{cases}$$

t est nommé la somme des tuples u et v.

Exemple 3.4 - Dans l'espace multidimensionnel exemple (*cf.* table 3.2), nous avons $t_1 + t_2 = t_{13}$, ce qui signifie que t_{13} est construit à partir des tuples t_1 et t_2. D'autre part $(t_{13}+t_{14})+(t_{19}+t_{20}) = t_{13} + t_{14} = t_{19} + t_{20} = t_{29}$.

Le produit de deux tuples retourne le tuple le plus général spécialisant les deux opérandes. Si pour ces deux tuples, il existe un attribut A prenant des valeurs distinctes et réelles (*i.e.* existant dans la relation initiale), alors seul le tuple $(\emptyset, \ldots, \emptyset)$ les spécialise (hormis ce tuple, les ensembles permettant de les construire sont disjoints).

Définition 3.5 (Opérateur Produit) **-** Soit u et v deux tuples de $Space(r)$, $t = u \bullet v \Leftrightarrow$

$$\begin{cases} t = (\emptyset, \ldots, \emptyset) \text{ si } \exists A \in \mathcal{D} \text{ tel que } u[A] \neq v[A] \neq \text{ALL}, \\ \text{sinon } \forall A \in \mathcal{D} \begin{cases} t[A] = u[A] \text{ si } v[A] = \text{ALL} \\ t[A] = v[A] \text{ si } u[A] = \text{ALL}. \end{cases} \end{cases}$$

t est dit le produit des tuples u et v.

Exemple 3.5 - Dans l'espace multidimensionnel pris en exemple (*cf.* table 3.2), nous avons $t_{13} \bullet t_{25} = t_1$ et $t_1 \bullet t_2 = (\emptyset,\emptyset,\emptyset)$, ce qui signifie que t_{13} et t_{25} généralisent t_1 et t_1 participe à la construction de t_{13} et t_{25} (directement ou indirectement). Les tuples t_1 et t_2 n'ont d'autre point commun que le tuple vide.

L'opérateur semi-produit a été proposé pour disposer d'une base rigoureuse lors de l'étape de génération des candidats dans un algorithme par niveau. Avant de présenter cet opérateur, la définition de la fonction *Attribute* est rappelée. Pour un tuple t, elle rend l'ensemble des attributs dont la valeur est différente de ALL.

Définition 3.6 (Fonction *Attribute*) **-** Soit t un tuple de $Space(r)$,

$$Attribute(t) = \{A \in \mathcal{D} \mid t[A] \neq ALL\}$$

Exemple 3.6 - Dans $Space(\text{DOCUMENT})$ (*cf.* table 3.2), nous avons $Attribute(t_1) = \{Nouvelle, Marseille, Collins\}$ et $Attribute(t_{13}) = \{Nouvelle, Marseille\}$.

L'opérateur semi-produit est un produit contraint utilisé pour la génération de candidats dans une approche utilisant des algorithmes par niveau (Agrawal *et al.*, 1996; Mannila et Toivonen, 1997). À tout niveau i, il permet de ne générer que des candidats de niveau $i + 1$, s'ils existent (sinon le semi-produit retourne le tuple $(\emptyset, \ldots, \emptyset)$). De plus, chaque tuple de niveau $i + 1$ n'est généré qu'une seule fois. Plus précisément, pour tout couple de tuples t, u de l'espace, les ensembles d'attributs (ordonnés selon $<_{\mathcal{D}}$) pour lesquels t et u admettent des valeurs réelles (différentes de ALL) sont considérés. Si ces ensembles ne diffèrent que par leur dernier élément alors le semi-produit $t \odot u$ génère un nouveau tuple plus agrégé que t et u. La condition de précédence selon $<_{\mathcal{D}}$ entre les derniers éléments des deux ensembles d'attributs est inspirée d'Apriori-Gen.

Définition 3.7 (Opérateur Semi-produit) **-** Soit (u,v) un couple de tuples de $Space(r)$ et deux ensembles $X = Attribute(u)$ et $Y = Attribute(v)$.

$$t = u \odot v \Leftrightarrow t = \begin{cases} u \bullet v \text{ si } X \backslash max_{<_{\mathcal{D}}}(X) = Y \backslash max_{<_{\mathcal{D}}}(Y) \\ \quad \text{et } max_{<_{\mathcal{D}}}(X) <_{\mathcal{D}} max_{<_{\mathcal{D}}}(Y) \\ (\emptyset, \ldots, \emptyset) \text{ sinon.} \end{cases}$$

où $<_{\mathcal{D}}$ est un ordre total sur $\mathcal{D}$.

Exemple 3.7 - Dans $Space(\text{DOCUMENT})$, nous avons $t_{13} \odot t_{19} = t_1$.

Définition 3.8 (Opérateur Différence) **-** La différence de deux tuples u et v de $Space(r)$ est définie comme suit :

$$t = u \backslash v \Leftrightarrow \forall A \in \mathcal{D}, t[A] = \begin{cases} u[A] \text{ si } u[A] \neq v[A] \\ \text{ALL sinon.} \end{cases}$$

Exemple 3.8 - Dans $Space(\text{DOCUMENT})$, nous avons $t_1 \backslash t_{19} = t_{32}$.

Caractérisation du treillis cube

En dotant l'espace multidimensionnel $Space(r)$ de la relation de généralisation entre tuples et en utilisant les opérateurs produit et somme, Casali *et al.* proposent une structure algébrique appelée treillis cube qui fixe un cadre théorique et général pour la fouille de bases multidimensionnelles. Les différents lemmes proposés donnent les propriétés fondamentales du treillis cube, qui sont reprises dans le théorème 3.5.

Lemme 3.1 - L'ensemble ordonné $CL(r) = \langle Space(r), \preceq_g \rangle$ est un treillis complet appelé treillis cube pour lequel :

1. $\forall\ T \subseteq CL(r), \bigwedge T = +_{t \in T}\ t$ où $\bigwedge$ symbolise l'infimum.
2. $\forall\ T \subseteq CL(r), \bigvee T = \bullet_{t \in T}\ t$ où $\bigvee$ symbolise le supremum.

Rappels : les co-atomes (respectivement les atomes) sont les tuples maximaux, c'est à dire les tuples les plus spécifiques (respectivement les tuples minimaux), du treillis privé de son majorant (respectivement minorant) universel. Les co-atomes (respectivement les atomes) du treillis cube d'une relation r sont notés $\mathcal{CA}t(CL(r))$ (respectivement $\mathcal{A}t(CL(r))$).

Lemme 3.2 - Le treillis $CL(r) = \langle Space(r), \preceq_g \rangle$ est un treillis co-atomique et atomique.

Dans le treillis des parties d'un ensemble $\mathcal{I}$, chaque élément peut être généré avec soit (i) les co-atomes (motifs de cardinalité $|\mathcal{I}| - 1$) et l'opérateur infimum ($\cap$), soit (ii) les atomes (motifs de cardinalité 1) et l'opérateur supremum ($\cup$). De manière similaire, mais dans notre contexte, chaque tuple du treillis cube peut être généré avec (i) les co-atomes du treillis cube et l'opérateur infimum ($+$), ou (ii) les atomes du treillis cube et l'opérateur supremum ($\bullet$). En conséquence, les atomes et les co-atomes du treillis cube sont deux familles génératrices de celui-ci.

La proposition ci-dessous permet à la fois de définir un plongement d'ordre du treillis cube vers le treillis des parties des attributs binaires ainsi qu'une fonction rang. Cette dernière est utilisée pour donner une autre caractérisation des atomes/co-atomes du treillis cube et pour montrer que le treillis cube est un treillis non distributif dans la majeure partie des cas et gradué (*cf.* lemme 3.4). Pour éviter toute ambiguïté, chaque valeur est préfixée par le nom de l'attribut concerné.

Proposition 3.3 - Soit $\mathcal{L}(r)$ le treillis des parties des attributs binaires de la relation binaire, *i.e.* le treillis $\langle \mathcal{P}(\bigcup_{A \in \mathcal{D}} A.a, \forall a \in Dim(A)), \subseteq \rangle$. Alors il existe un plongement d'ordre Φ : $CL(r) \rightarrow \mathcal{L}(r)$

$$t \mapsto \begin{cases} \bigcup_{A \in \mathcal{D}} A.a, \forall a \in Dim(A) \text{ si } t = (\emptyset, \ldots, \emptyset) \\ \{A.t[A] \mid \forall A \in Attribute(t)\} \text{ sinon} \end{cases}$$

Le rang d'un tuple t, noté $rank(t)$, est la longueur du plus petit chemin (nombre d'arcs minimal) dans le treillis cube le reliant au tuple (ALL, ..., ALL). Nous avons donc : $rank(t) = |\Phi(t)|$ si $t \neq (\emptyset, \ldots, \emptyset), |\mathcal{D}| + 1$ sinon.

Cette proposition permet une autre caractérisation des atomes et des co-atomes du treillis cube. En effet, les atomes du treillis cube sont les tuples tels que $|\Phi(t)| = 1$ et les co-atomes du treillis cube sont les tuples tels que $|\Phi(t)| = |\mathcal{D} - 1|$.

Le lemme suivant montre que le treillis cube est un treillis gradué. En conséquence, nous pouvons appliquer des algorithmes par niveau sur cet espace de recherche.

Lemme 3.4 - Le treillis cube $CL(r)$ est gradué. Si $|\mathcal{D}| \preceq 2$ alors $CL(r)$ n'est pas distributif.

Théorème 3.5 - Soit r une relation d'attributs catégories sur $\mathcal{D} \cup \mathcal{M}$. L'ensemble ordonné $CL(r) = \langle Space(r), \preceq_g \rangle$ est un treillis complet, atomique, co-atomique et gradué appelé treillis cube dans lequel :

1. $\forall\ T \subseteq CL(r), \bigwedge T = +_{t \in T}\ t$
2. $\forall\ T \subseteq CL(r), \bigvee T = \bullet_{t \in T}\ t$.

Exemple 3.9 - La figure 3.1 représente le treillis cube de notre relation exemple (*cf.* table 3.1). Dans ce diagramme, les arêtes représentent les liens de généralisation ou spécialisation entre tuples.

1. En faisant abstraction de $(\emptyset, \emptyset, \emptyset)$, les co-atomes sont des tuples véhiculant l'information au niveau le plus détaillé, *i.e.* celui des valeurs réelles des dimensions. En d'autres termes, les co-atomes sont les tuples potentiels d'une relation. Ainsi, nous avons :

$$\mathcal{CA}t(CL(\textsc{Document})) = \left\{ \begin{array}{l} \text{(Nouvelles , Marseille , Collins)}, \\ \text{(Nouvelles , Marseille , Hachette)}, \\ \text{(Nouvelles , Paris , Collins)}, \\ \text{(Nouvelles , Paris , Hachette)}, \\ \text{(Pédagogie , Marseille , Hachette)}, \\ \text{(Pédagogie , Marseille , Collins)}, \\ \text{(Pédagogie , Paris , Hachette)}, \\ \text{(Pédagogie , Paris , Collins)}, \\ \text{(Essai , Marseille , Hachette)}, \\ \text{(Essai , Marseille , Collins)}, \\ \text{(Essai , Paris , Hachette)}, \\ \text{(Essai , Paris , Collins)} \end{array} \right\}$$

2. Les atomes du treillis offrent l'information la plus synthétique possible, à l'exception du tuple (ALL, ALL, ALL). Étant donné que nous ne considérons ici que trois attributs dimensions, tous les atomes comportent 2 fois la valeur synthétique ALL. Plus précisément,

nous avons :

$$\mathcal{A}t(CL(r)) = \left\{ \begin{array}{l} \text{(Nouvelles , 'ALL', 'ALL'),} \\ \text{(Pédagogie , 'ALL', 'ALL'),} \\ \text{(Essai , 'ALL', 'ALL'),} \\ \text{('ALL', Paris , 'ALL'),} \\ \text{('ALL', Marseille , 'ALL'),} \\ \text{('ALL', 'ALL', Hachette),} \\ \text{('ALL', 'ALL', Collins),} \end{array} \right\}$$

La proposition suivante donne une analyse du nombre d'éléments contenus dans un niveau du treillis cube ainsi que le nombre d'éléments total. Cette proposition est particuliérement importante car elle permet de caractériser la complexité des algorithmes utilisant le treillis cube comme espace de recherche.

Proposition 3.6 - La hauteur (nombre de niveaux) du treillis cube est $|\mathcal{D}|+1$. Le nombre d'éléments pour un niveau i ($i \in 1..|\mathcal{D}|$) est :

$$\sum_{\substack{X \subseteq \mathcal{D} \\ |X|=i}} \left(\prod_{A \in X} |Dim(A)| \right) \leq \binom{|\mathcal{D}|}{i} \max_{A \in \mathcal{D}} (|Dim(A)|)^i.$$

Le nombre total d'éléments dans le treillis cube est :

$$\sum_{i=1..|\mathcal{D}|} \left(\sum_{\substack{X \subseteq \mathcal{D} \\ |X|=i}} \left(\prod_{A \in X} |Dim(A)| \right) \right) + 2 = \left(\prod_{A \in \mathcal{D}} (|Dim(A)| + 1) \right) + 1$$

3.2.2 Le treillis cube convexe

Dans ce paragraphe, la structure du treillis cube en présence de conjonctions de contraintes monotones et/ou antimonotones est présentée. Munies d'une telle structure, les représentations classiques condensées (avec bordures) du treillis cube contraint sont décrites avec un double objectif : définir d'une manière compacte l'espace de solutions et décider si un tuple t appartient, ou pas, à cet espace. Nous prenons en compte les contraintes monotones et/ou antimonotones les plus couramment utilisées en fouille de base de données (Ng *et al.*, 1998; Jr. *et al.*, 2000; Grahne *et al.*, 2000; Lopes *et al.*, 2002). Celles-ci peuvent porter sur :

- des mesures d'intérêts comme la fréquence de motifs, la confiance, la corrélation Han et Kamber (2006) : dans ce cas, seuls les attributs dimensions de $\mathcal{R}$ sont nécessaires ;
- des agrégats selon des attributs mesures $\mathcal{M}$ calculés en utilisant des fonctions statistiques additives (Count, Sum, Min, Max).

Nous rappelons les définitions des contraintes monotones et antimonotones selon l'ordre de généralisation $\preceq_g$.

Définition 3.9 (Contraintes monotones/antimonotones) **-**

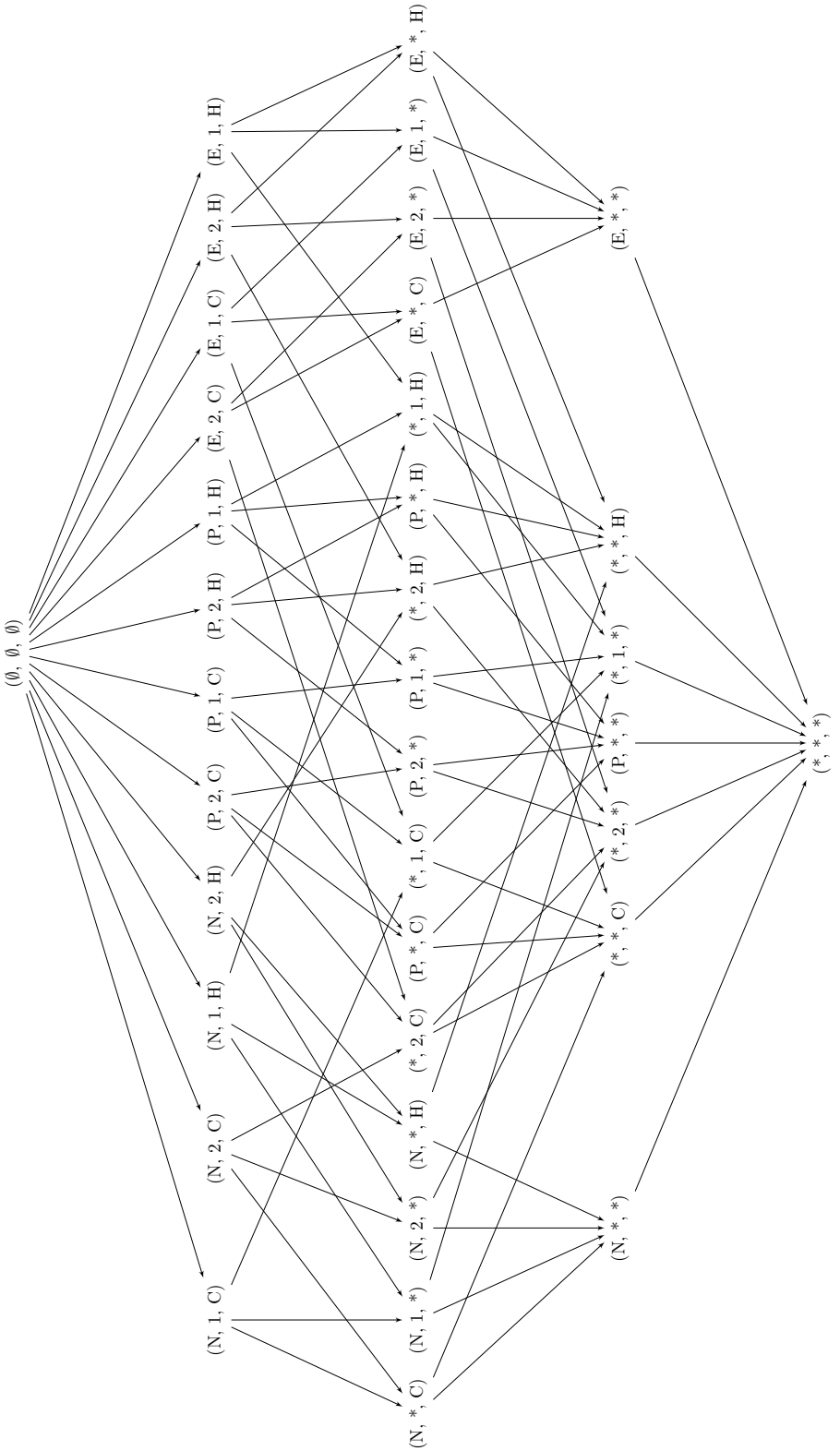
1. Une contrainte *Cont* est dite monotone si et seulement si :

$$\forall\ t,u \in CL(r) : [t \preceq_g u \text{ et } Cont(t)] \Rightarrow Cont(u)$$

2. Une contrainte *Cont* est dite antimonotone si et seulement si :

$$\forall\ t,u \in CL(r) : [t \preceq_g u \text{ et } Cont(u)] \Rightarrow Cont(t)$$

FIGURE 3.1 – Diagramme de Hasse du treillis cube de la relation DOCUMENT



Exemple 3.10 - Dans l'espace multidimensionnel de la relation DOCUMENT (*cf.* table 3.1), une requête porte sur tous les tuples dont la somme des valeurs pour l'attribut mesure *Quantité* est supérieure ou égale à 300. La contrainte « SUM(*Quantité*) ≥ 300 » est une contrainte antimonotone. Si la quantité de livres vendus par Type, Ville et Editeur est supérieure à 300, alors elle l'est *a fortiori* pour un niveau plus agrégé de granularité *e.g.* par Type et par Ville (tous les éditeurs confondus). Inversement, si la requête porte sur tous les tuples dont la somme des valeurs pour l'attribut *Quantité* est inférieure ou égale à 700, la contrainte exprimée « SUM(*Quantité*) ≤ 700 » est monotone.

Structure du treillis cube convexe

Le treillis cube, en présence de contraintes monotones et/ou antimonotones, n'est plus nécessairement un treillis. Il est rappelé dans ce paragraphe qu'un tel ensemble ordonné est un espace convexe et donc représentable par des bordures.

Définition 3.10 (Espace Convexe) **-** Soit $(\mathcal{P}, \leq)$ un ensemble partiellement ordonné, $\mathcal{C} \subseteq \mathcal{P}$ est un espace convexe (Vel, 1993) *si et seulement si* :

$$\forall x,y,z \in \mathcal{P} \text{ tels que } x \leq y \leq z \text{ et } x,z \in \mathcal{C} \Rightarrow y \in \mathcal{C}.$$

Donc $\mathcal{C}$ est borné par deux ensembles : (i) un majorant (ou *« Upper set »*), noté U, défini par $U = \max_{\leq}(\mathcal{C})$, (ii) un minorant (ou *« Lower set »*) noté L et défini par $L = \min_{\leq}(\mathcal{C})$.

Notations : *cmc* (respectivement *camc*) est utilisé pour noter une conjonction de contraintes monotones (respectivement antimonotones) et *chc* une conjonction de contraintes hybrides (monotones et antimonotones). En reprenant les symboles U et L classiques et suivant les cas considérés, les bornes introduites sont indicées par le type de contrainte considéré. Par exemple U_{camc} symbolise l'ensemble des tuples les plus spécifiques vérifiant une conjonction de contraintes antimonotones.

Remarques (cas extrêmes) :

- Nous supposons par la suite que le tuple $(\text{ALL}, \ldots, \text{ALL})$ vérifie toujours la conjonction de contraintes antimonotones et que le tuple $(\emptyset, \ldots, \emptyset)$ vérifie toujours la conjonction de contraintes monotones. Avec ces hypothèses, l'espace des solutions contient toujours au moins un élément.
- De plus, nous supposons que le tuple $(\text{ALL}, \ldots, \text{ALL})$ ne vérifie jamais la conjonction de contraintes monotones et que le tuple $(\emptyset, \ldots, \emptyset)$ ne vérifie jamais la conjonction de contraintes antimonotones, car sinon l'espace de solutions est $Space(r)$.

Théorème 3.7 - Tout treillis cube avec contraintes monotones et/ou antimonotones est un espace convexe. Son majorant U_{const} et son minorant L_{const} sont :

$$\text{si } const = cmc, \begin{cases} L_{cmc} = \min_{\preceq_g}(\{t \in CL(r) \mid cmc(t)\}) \\ U_{cmc} = (\emptyset, \ldots, \emptyset) \end{cases} \tag{3.1}$$

$$\text{si } const = camc, \begin{cases} L_{camc} = (\text{ALL}, \ldots, \text{ALL}) \\ U_{camc} = \max_{\preceq_g}(\{t \in CL(r) \mid camc(t)\}) \end{cases} \tag{3.2}$$

$$\text{si } const = chc, \begin{cases} L_{chc} = \min_{\preceq_g}(\{t \in CL(r) \mid chc(t)\}) \\ U_{chc} = \max_{\preceq_g}(\{t \in CL(r) \mid chc(t)\}) \end{cases} \tag{3.3}$$

TABLE 3.3 – Bornes du treillis cube contraint pour « 300 ≤ SUM(*Quantité*) ≤ 700 »

U_{camc}	(Nouvelles , Marseille , Collins) (Pédagogie , Marseille , Collins) ('ALL' , Paris , Collins)
U_{chc}	(Nouvelles , Marseille , Collins) (Pédagogie , Marseille , Collins) ('ALL' , Paris , Collins)
L_{cmc}	('ALL' , Marseille , Collins) (Nouvelles , 'ALL' , 'ALL') (Pédagogie , 'ALL' , 'ALL') ('ALL' , 'ALL' , Hachette) ('ALL' , Paris , 'ALL')
L_{chc}	('ALL' , Marseille , Collins) (Nouvelles , 'ALL' , 'ALL') (Pédagogie , 'ALL' , 'ALL') ('ALL' , Paris , 'ALL')

Le majorant U_{const} représente les tuples les plus spécifiques satisfaisant la conjonction de contraintes et le minorant L_{const} les tuples les plus généraux la satisfaisant aussi. Donc U_{const} et L_{const} permettent d'obtenir des représentations condensées du treillis cube en présence d'une conjonction de contraintes monotones et/ou antimonotones.

Le corollaire suivant permet la caractérisation des bordures du treillis cube contraint par une conjonction de contraintes hybrides $chc = camc \wedge cmc$ en ne connaissant que (i) soit la bordure maximale pour la contrainte antimonotone U_{camc} et la contrainte monotone cmc, (ii) soit la bordure minimale pour la contrainte monotone L_{cmc} et la contrainte antimonotone $camc$.

Corollaire 3.8 -

1. Étant données U_{camc} et cmc, une représentation condensée de l'ensemble des tuples satisfaisant la contrainte hybride $chc = cmc \wedge camc$ est :

$$\begin{cases} L_{chc} = min_{\preceq_g}(\{t \in CL(r) \mid \exists t' \in U_{camc} : t \preceq_g t' \text{ et } cmc(t)\}) \\ U_{chc} = \{t \in U_{camc} \mid \exists t' \in L_{chc} : t' \preceq_g t\} \end{cases}$$

2. Étant données L_{cmc} et $camc$, une représentation condensée de $CL(r)_{chc}$ est :

$$\begin{cases} U_{chc} = max_{\preceq_g}(\{t \in CL(r) \mid \exists t' \in L_{cmc} : t' \preceq_g t \text{ et } camc(t)\}) \\ L_{chc} = \{t \in L_{camc} \mid \exists t' \in U_{chc} : t \preceq_g t'\}. \end{cases}$$

Exemple 3.11 - Le tableau 3.3 donne les bornes U_{camc}, U_{chc} L_{cmc} et L_{chc} du treillis cube convexe de la relation exemple en considérant la contrainte hybride « 300 ≤ SUM(*Quantité*) ≤ 700 ».

De même que la caractérisation de l'ensemble des motifs fréquents comme étant un espace convexe nous permet de savoir, en ne connaissant seulement la bordure positive, si un motif est fréquent ou pas, la représentation par bordure du treillis cube contraint de la relation DOCUMENT permet de répondre facilement à certaines requêtes telles que :

1. Est-ce que le nombre d'achats à Marseille chez l'éditeur Collins est superieur à 300 ?
2. Est-ce que le nombre de livres pédagogiques vendus à Marseille est compris entre 300 et 700 ?
3. Est-ce que le nombre de romans vendus à Marseille par les éditions Hachette est compris entre 300 et 700 ?

La réponse à la première question est oui, car le tuple (ALL, Marseille, Collins) donnant les achats effectués dans la ville de Marseille pour l'éditeur Collins tout type confondu appartient à la bordure L_{cmc}. Il en est de même pour la seconde requête car le tuple (Pédagogie, Marseille, 'ALL') spécialise le tuple (Pédagogie, 'ALL', 'ALL') appartenant à L_{chc} et généralise le tuple (Pédagogie, Marseille, Collins) appartenant à la bordure U_{chc}. En revanche, la réponse à la troisième question est non car le tuple ('ALL', Paris, Hachette), c'est à dire tous les livres édités par Hachette achetés à Paris, ne spécialise aucun tuple de la bordure L_{chc}.

3.2.3 Transversaux cubiques

En considérant le treillis des parties des valeurs $\mathcal{L}(r)$ comme espace de recherche, un motif binaire X est un transversal de $\Phi(r)$ si et seulement s'il a au moins un 1-motif commun avec toutes les transactions de la base binaire. Par exemple, en utilisant la relation DOCUMENT et le plongement d'ordre Φ, le motif {Ville.Marseille, Langue.Français} est un transversal de $\Phi(r)$. De plus, ce motif est un minimal transversal car aucun de ses sous-ensembles n'est un transversal de $\Phi(r)$. Malheureusement, en utilisant un tel treillis, des motifs multidimensionnels sémantiquement invalides sont calculés et peuvent appartenir à l'espace de solutions. Par exemple, le motif {Ville.Marseille,Ville.Paris} est un transversal de $\Phi(r)$, mais n'est pas un motif multidimensionnel sémantiquement valide parce que chaque 1-motif le constituant appartient au même attribut.

Le concept de transversaux cubiques (Casali *et al.*, 2003b) est un cas particulier des transversaux d'un hypergraphe (Berge, 1989; Eiter et Gottlob, 1995; Gunopulos *et al.*, 1997). La contrainte « t est un transversal cubique » étant une contrainte monotone selon $\preceq_g$, nous présentons dans un premier temps, l'algorithme par niveau CTR pour l'extraction des minimaux transversaux cubiques. Cependant, si la relation ne tient pas en mémoire centrale, l'algorithme par niveau avec partitions et fusions $MCTR$ permet le calcul des minimaux transversaux cubiques. Le point fort de cet algorithme est de ne nécessiter qu'un seul balayage de la base de données.

Définition 3.11 (Transversal cubique) **-** Soit T un ensemble de tuples ($T \subseteq CL(r)$) et soit $t \in T$ un tuple, t est un transversal cubique de T sur $CL(r)$ si et seulement si t est un transversal cubique et $\forall t' \in T, t'$ est un transversal cubique et $t' \preceq_g t \Rightarrow t = t'$. Les minimaux transversaux cubiques de T sont notés $cTr(T)$ et définis comme suit :

$$cTr(T) = \min_{\preceq_g}(\{t \in T \mid \forall t' \in T, t + t' \neq (\text{ALL}, \ldots, \text{ALL})\})$$

Soit $\mathbb{A}$ une anti-chaîne de $CL(r)$ (tous les tuples de $\mathbb{A}$ sont incomparables selon $\preceq_g$), l'ensemble des minimaux transversaux cubiques de T peut être contraint en utilisant $\mathbb{A}$. La nouvelle définition associée est la suivante :

$$cTr(T,\mathbb{A}) = \{t \in cTr(r) \mid \exists u \in \mathbb{A} : t \preceq_g u\}$$

Définition 3.12 (Transversal cubique du complément) **-** Soit T un ensemble de tuples ($T \subseteq CL(r)$) et soit $t \in T$ un tuple, t est un transversal cubique de $\overline{T}$ (complément de T) sur $CL(r)$ si

et seulement si $\forall t' \in T, t \not\preceq_g t'$. Les minimaux transversaux cubiques du complément de T sont notés $ccTr(T)$ et définis comme suit :

$$ccTr(T) = \min_{\preceq_g}(\{t \in T \mid \forall t' \in T, t \not\preceq_g t'\})$$

Soit $\mathbb{A}$ une anti-chaîne de $CL(r)$ l'ensemble des minimaux transversaux cubiques du complément de T contraint par $\mathbb{A}$ est défini comme suit :

$$ccTr(T,\mathbb{A}) = \{t \in ccTr(T) \mid \exists u \in \mathbb{A} : t \preceq_g u\}$$

Proposition 3.9 - Soit $t \in CL(r)$, les contraintes *« t est un transversal cubique de T »* et *« t est un transversal cubique du complément de T »* sont des contraintes monotones selon l'ordre de généralisation.

Puisque les deux contraintes sont monotones (selon $\preceq_g$), les ensembles $cTr(T)$ et $ccTr(T)$ sont deux bordures représentant l'ensemble des transversaux cubiques de T et de son complément.

L'algorithme par niveau CTR dont le pseudo-code est donné ci-dessous, permet de calculer les minimaux transversaux d'une relation ou de son complément.

Algorithme 9 Algorithme CTR

Entrée : Relation d'attributs catégories r sur $\mathcal{D}$ [et une anti-chaîne $\mathbb{A}$]
Sortie : $cTr(r[,\mathbb{A}])$
  $L_1 := \{t \in \mathcal{A}t(CL(r))\}$
  $T := \{t \in L_1 \mid t$ est un transversal cubique [et $\exists w \in \mathbb{A} : v \preceq_g w]\}$
  $i := 1$
  $L_1 := L_1 \backslash T$
  $cTr := cTr \cup T$
  tant que $L_i \neq \emptyset$ **faire**
    $C_{i+1} := \{v = t \odot t' \mid t,t' \in L_i, v \neq (\emptyset, \ldots, \emptyset), [\exists w \in \mathbb{A} : v \preceq_g w]$ et $\nexists u \in cTr : u \preceq_g v\}$
    pour tout $t \in r$ **faire**
      pour tout $l \in L_{i+1}$ non marqué **faire**
        si $l + t = (\text{ALL}, \ldots, \text{ALL})$ **alors**
          marquer l $//l \preceq_g t$ si recherche de $ccTr(r)$
        fin si
      fin pour
    fin pour
    $T := \{l \in L_{i+1} \mid l$ n'est pas marqué $\}$
    $L_{i+1} := C_{i+1} \backslash T$
    $cTr := cTr \cup T$
    $i := i + 1$
  fin tant que
  retourner cTr

Exemple 3.12 - Soit DOCUMENT_{Fr} le résultat de la sélection sur la relation DOCUMENT avec la condition Langue = 'Français'. Cette relation est la suivante :

RowId	Type	Ville	Éditeur	Langue	Quantité
1	Nouvelles	Marseille	Collins	Français	400
3	Pédagogie	Paris	Hachette	Français	100
5	Essai	Paris	Collins	Français	200

Pour illustrer le fonctionnement de l'algorithme CTR, calculons les minimaux transversaux du complément de la relation DOCUMENT_{Fr} sur le treillis cube de la relation DOCUMENT.

Niveau 1 : Puisque nous cherchons les minimaux transversaux du complément de DOCUMENT_{Fr} sur le treillis cube de la relation DOCUMENT, l'ensemble des candidats de niveau 1, C_1, est initialisé avec les atomes du treillis cube de la relation DOCUMENT. Après un balayage de la base, seul le tuple ('ALL', 'ALL', 'ALL', Anglais) ne satisfait pas la contrainte monotone, il est donc marqué. L'ensemble cTr est donc uniquement constitué de ce tuple.

C_1
(Nouvelles , 'ALL' , 'ALL' , 'ALL')
(Pédagogie , 'ALL' , 'ALL' , 'ALL')
(Essai , 'ALL' , 'ALL' , 'ALL')
('ALL' , Paris , 'ALL' , 'ALL')
('ALL' , Marseille , 'ALL' , 'ALL')
('ALL' , 'ALL' , Hachette , 'ALL')
('ALL' , 'ALL' , Collins , 'ALL')
('ALL' , 'ALL' , 'ALL' , Français)
('ALL' , 'ALL' , 'ALL' , Anglais)

$\xrightarrow{Balayage}$

C_1
(Nouvelles , 'ALL' , 'ALL' , 'ALL')
(Pédagogie , 'ALL' , 'ALL' , 'ALL')
(Essai , 'ALL' , 'ALL' , 'ALL')
('ALL' , Paris , 'ALL' , 'ALL')
('ALL' , Marseille , 'ALL' , 'ALL')
('ALL' , 'ALL' , Hachette , 'ALL')
('ALL' , 'ALL' , Collins , 'ALL')
('ALL' , 'ALL' , 'ALL' , Français)
('ALL' , 'ALL' , 'ALL' , Anglais)

$\xrightarrow{Élagage}$

L_1
(Nouvelles , 'ALL' , 'ALL' , 'ALL')
(Pédagogie , 'ALL' , 'ALL' , 'ALL')
(Essai , 'ALL' , 'ALL' , 'ALL')
('ALL' , Paris , 'ALL' , 'ALL')
('ALL' , Marseille , 'ALL' , 'ALL')
('ALL' , 'ALL' , Hachette , 'ALL')
('ALL' , 'ALL' , Collins , 'ALL')
('ALL' , 'ALL' , 'ALL' , Français)

$ccTr$
('ALL' , 'ALL' , 'ALL' , Anglais)

– Niveau 2 : Les tuples du niveau 2 sont obtenus en appliquant l'opérateur semi-produit à partir des tuples constituant L_1. Parmi les tuples candidats générés, les minimaux transversaux du complément de la relation DOCUMENT_{Fr} sont marqués. Les autres tuples constituent donc l'ensemble L_2.

C_2
(Nouvelles , Paris , 'ALL' , 'ALL')
(Nouvelles , Marseille , 'ALL' , 'ALL')
(Nouvelles , 'ALL' , Hachette , 'ALL')
(Nouvelles , 'ALL' , Collins , 'ALL')
(Nouvelles , 'ALL' , 'ALL' , Français)
(Pédagogie , Paris , 'ALL' , 'ALL')
(Pédagogie , Marseille , 'ALL' , 'ALL')
(Pédagogie , 'ALL' , Hachette , 'ALL')
(Pédagogie , 'ALL' , Collins , 'ALL')
(Pédagogie , 'ALL' , 'ALL' , Français)
(Essai , Paris , 'ALL' , 'ALL')
(Essai , Marseille , 'ALL' , 'ALL')
(Essai , 'ALL' , Hachette , 'ALL')
(Essai , 'ALL' , Collins , 'ALL')
(Essai , 'ALL' , 'ALL' , Français)
('ALL' , Paris , Hachette , 'ALL')
('ALL' , Marseille , Hachette , 'ALL')
('ALL' , Paris , Collins , 'ALL')
('ALL' , Marseille , Collins , 'ALL')
('ALL' , Paris , 'ALL' , Français)
('ALL' , Marseille , 'ALL' , Français)
('ALL' , 'ALL' , Hachette , Français)
('ALL' , 'ALL' , Collins , Français)

$\xrightarrow{Balayage}$

C_2
(Nouvelles , Paris , 'ALL' , 'ALL')
(Nouvelles , Marseille , 'ALL' , 'ALL')
(Nouvelles , 'ALL' , Hachette , 'ALL')
(Nouvelles , 'ALL' , Collins , 'ALL')
(Nouvelles , 'ALL' , 'ALL' , Français)
(Pédagogie , Paris , 'ALL' , 'ALL')
(Pédagogie , Marseille , 'ALL' , 'ALL')
(Pédagogie , 'ALL' , Hachette , 'ALL')
(Pédagogie , 'ALL' , Collins , 'ALL')
(Pédagogie , 'ALL' , 'ALL' , Français)
(Essai , Paris , 'ALL' , 'ALL')
(Essai , Marseille , 'ALL' , 'ALL')
(Essai , 'ALL' , Hachette , 'ALL')
(Essai , 'ALL' , Collins , 'ALL')
(Essai , 'ALL' , 'ALL' , Français)
('ALL' , Paris , Hachette , 'ALL')
('ALL' , Marseille , Hachette , 'ALL')
('ALL' , Paris , Collins , 'ALL')
('ALL' , Marseille , Collins , 'ALL')
('ALL' , Paris , 'ALL' , Français)
('ALL' , Marseille , 'ALL' , Français)
('ALL' , 'ALL' , Hachette , Français)
('ALL' , 'ALL' , Collins , Français)

$\xrightarrow{Élagage}$

L_2
(Nouvelles , Marseille , 'ALL' , 'ALL')
(Nouvelles , 'ALL' , Collins , 'ALL')
(Nouvelles , 'ALL' , 'ALL' , Français)
(Pédagogie , Paris , 'ALL' , 'ALL')
(Pédagogie , 'ALL' , Hachette , 'ALL')
(Pédagogie , 'ALL' , 'ALL' , Français)
(Essai , Paris , 'ALL' , 'ALL')
(Essai , 'ALL' , Collins , 'ALL')
(Essai , 'ALL' , 'ALL' , Français)
('ALL' , Paris , Hachette , 'ALL')
('ALL' , Paris , Collins , 'ALL')
('ALL' , Marseille , Collins , 'ALL')
('ALL' , Paris , 'ALL' , Français)
('ALL' , Marseille , 'ALL' , Français)
('ALL' , 'ALL' , Hachette , Français)
('ALL' , 'ALL' , Collins , Français)

$ccTr$
('ALL' , 'ALL' , 'ALL' , Anglais)
(Nouvelles , Paris , 'ALL' , 'ALL')
(Nouvelles , 'ALL' , Hachette , 'ALL')
(Pédagogie , Marseille , 'ALL' , 'ALL')
(Pédagogie , 'ALL' , Collins , 'ALL')
(Essai , Marseille , 'ALL' , 'ALL')
(Essai , 'ALL' , Hachette , 'ALL')
('ALL' , Marseille , Hachette , 'ALL')

– Niveau 3 : Lors de la phase de génération des candidats, dix tuples sont générés. L'ensemble $ccTr$ n'est pas mis à jour.

C_3
(Nouvelles , Marseille , Collins , 'ALL')
(Nouvelles , Marseille , 'ALL' , Français)
(Pédagogie , Paris , 'ALL' , Français)
(Pédagogie , 'ALL' , Hachette , Français)
(Essai , Paris , 'ALL' , Français)
(Essai , 'ALL' , Collins , Français)
('ALL' , Paris , Hachette , Français)
('ALL' , Paris , Collins , Français)
('ALL' , Marseille , Collins , Français)

$\xrightarrow{Balayage}$

C_3
(Nouvelles , Marseille , Collins , 'ALL')
(Nouvelles , Marseille , 'ALL' , Français)
(Pédagogie , Paris , 'ALL' , Français)
(Pédagogie , 'ALL' , Hachette , Français)
(Essai , Paris , 'ALL' , Français)
(Essai , 'ALL' , Collins , Français)
('ALL' , Paris , Hachette , Français)
('ALL' , Paris , Collins , Français)
('ALL' , Marseille , Collins , Français)

$\xrightarrow{Élagage}$

L_3
(Nouvelles , Marseille , Collins , 'ALL')
(Nouvelles , Marseille , 'ALL' , Français)
(Pédagogie , Paris , 'ALL' , Français)
(Pédagogie , 'ALL' , Hachette , Français)
(Essai , Paris , 'ALL' , Français)
(Essai , 'ALL' , Collins , Français)
('ALL' , Paris , Hachette , Français)
('ALL' , Paris , Collins , Français)
('ALL' , Marseille , Collins , Français)

$ccTr$
('ALL' , 'ALL' , 'ALL' , Anglais)
(Nouvelles , Paris , 'ALL' , 'ALL')
(Nouvelles , 'ALL' , Hachette , 'ALL')
(Pédagogie , Marseille , 'ALL' , 'ALL')
(Pédagogie , 'ALL' , Collins , 'ALL')
(Essai , Marseille , 'ALL' , 'ALL')
(Essai , 'ALL' , Hachette , 'ALL')
('ALL' , Marseille , Hachette , 'ALL')

– Niveau 4 : L'algorithme se termine à ce niveau. Aucun tuple n'est ajouté à $ccTr$

C_3
(Nouvelles , Marseille , Collins , Français)
(Pédagogie , Paris , Hachette , Français)
(Essai , Paris , Collins , Français)

$\xrightarrow{Balayage}$

C_3
(Nouvelles , Marseille , Collins , Français)
(Pédagogie , Paris , Hachette , Français)
(Essai , Paris , Collins , Français)

$\xrightarrow{Élagage}$

L_3
(Nouvelles , Marseille , Collins , Français)
(Pédagogie , Paris , Hachette , Français)
(Essai , Paris , Collins , Français)

$ccTr$
('ALL' , 'ALL' , 'ALL' , Anglais)
(Nouvelles , Paris , 'ALL' , 'ALL')
(Nouvelles , 'ALL' , Hachette , 'ALL')
(Pédagogie , Marseille , 'ALL' , 'ALL')
(Pédagogie , 'ALL' , Collins , 'ALL')
(Essai , Marseille , 'ALL' , 'ALL')
(Essai , 'ALL' , Hachette , 'ALL')
('ALL' , Marseille , Hachette , 'ALL')

Donc, l'ensemble des minimaux transversaux du complément de la relation DOCUMENT_{Fr}

est :

$$ccTr(\text{DOCUMENT}_{Fr}) = \left\{ \begin{array}{l} (\text{'ALL'} , \text{'ALL'} , \text{'ALL'} , \text{Anglais}), \\ (\text{Nouvelles} , \text{Paris} , \text{'ALL'} , \text{'ALL'}), \\ (\text{Nouvelles} , \text{'ALL'} , \text{Hachette} , \text{'ALL'}), \\ (\text{Pédagogie} , \text{Marseille} , \text{'ALL'} , \text{'ALL'}), \\ (\text{Pédagogie} , \text{'ALL'} , \text{Collins} , \text{'ALL'}), \\ (\text{Essai} , \text{Marseille} , \text{'ALL'} , \text{'ALL'}), \\ (\text{Essai} , \text{'ALL'} , \text{Hachette} , \text{'ALL'}), \\ (\text{'ALL'} , \text{Marseille} , \text{Hachette} , \text{'ALL'}), \end{array} \right\}$$

Transversaux cubiques vs transversaux d'un hypergraphe

Lorsque l'espace de recherche est le treillis des parties, l'ensemble $\mathcal{H} = \{\Phi(t) \mid t \in r\}$ est un hypergraphe simple sur $E = \bigcup_{A \in \mathcal{D}} A.a, \forall a \in Dim(A)$, plus précisément un hypergraphe $|\mathcal{D}|$-uniforme. Les transversaux minimaux de $\mathcal{H}$ sont $Tr(\mathcal{H}) = \min_{\subseteq}(\{X \subseteq E \mid \forall Y \in \mathcal{H}, X \cap Y \neq \emptyset\})$ et de $\overline{\mathcal{H}}$ sont $\min_{\subseteq}(\{X \subseteq E \mid \forall Y \in \mathcal{H}, X \nsubseteq Y\})$. D'après le plongement d'ordre Φ, il est clair que $\{\Phi(t) \mid t \in cTr(r)\} \subset Tr(\mathcal{H})$ et le problème de l'extraction des transversaux cubiques a une forte corrélation avec celui de la recherche des transversaux d'un hypergraphe.

Exemple 3.13 - En considérant l'espace multidimensionnel de notre relation exemple, le tuple (ALL,Marseille,ALL) appartient à $cTr(r)$ et Φ((ALL,Marseille,ALL)) appartient à $Tr(\mathcal{H})$. Cependant, le motif binaire {Ville.Marseille, Langue.Français} $\in Tr(\mathcal{H})$, mais ce motif ne peut être un tuple sémantiquement valide car il regroupe deux valeurs d'un même attribut.

La complexité de l'algorithme CTR est du même ordre que celle de l'algorithme TR (Berge, 1989; Eiter et Gottlob, 1995; Gunopulos *et al.*, 1997), à savoir $\mathcal{O}(2^k|E||cTr(r)|)$, où $E = \bigcup_{A \in \mathcal{D}} A.a, \forall\ a \in Dim(A)$ et $k = |\mathcal{D}|$. Cependant, la cardinalité de l'ensemble $cTr(r)$ est de loin inférieure à celle de l'ensemble $Tr(\phi(r))$ car en utilisant le cadre du treillis cube, les combinaisons sémantiquement invalides ne sont pas considérées.

Partitions et fusions pour le calcul des minimaux transversaux cubiques

Lorsque la relation d'attributs catégories est stockée sur disque, l'algorithme par niveau CTR fait k balayages de la base de données. Une caractérisation basée sur les partitions et les fusions est présentée. À partir cette caractérisation l'algorithme $MCTR$ est défini. Cet algorithme ne requiert qu'un seul balayage de la base de données pour la découverte des minimaux transversaux cubiques.

Proposition 3.10 - Soit $r = r_1 \cup r_2$ une relation d'attributs catégories, $cTr(r_1)$ et $cTr(r_2)$ les minimaux transversaux cubiques de r_1 et r_2. Si $cTr(r_1)$ est vide, nous considérons que $cTr(r_1) = \{(\emptyset, \ldots, \emptyset)\}$ (idem pour $cTr(r_2)$). $cTr(r)$ peut être caractérisé comme suit :

$$cTr(r) = \min_{\preceq_g}(\{t \bullet t' \neq (\emptyset, \ldots, \emptyset) \mid t \in cTr(r_1) \text{ et } t' \in cTr(r_2)\})$$

Le résultat reste vrai pour l'obtention de $ccTr(r)$.

La complexité de l'algorithme MCTR est du même ordre que celle de l'algorithme CTR, à savoir $\mathcal{O}(2^k|E||cTr(r)|)$, où $E = \bigcup_{A \in \mathcal{D}} A.a, \forall\ a \in Dim(A)$ et $k = |\mathcal{D}|$.

Algorithme 10 Algorithme $MCTR$

Entrée : Relation d'attributs catégories $r = r_1 \cup r_2 \cup ... \cup r_p$ sur $\mathcal{D}$ [et une anti-chaîne $\mathbb{A}$]
Sortie : $cTr(r)\backslash\backslash$ ou $cTr(\overline{r})$
  $cTr := (\emptyset, \ldots, \emptyset)$
  pour $i = 1$ **à** p **faire**
    $T := CTR(r_i[,\mathbb{A}])$ //ou $T := CTR(\overline{r_i}[,\mathbb{A}])$
    si $T \neq \{\emptyset\}$ **alors**
      $cTr := min_{\preceq_g}(\{v = t \bullet t' \mid v \neq (\emptyset, \ldots, \emptyset), t \in cTr \text{ et } t' \in T\ \})$
    fin si
  fin pour
  retourner cTr

Exemple 3.14 - DOCUMENT est partitionné en deux relations DOCUMENT_{Fr} (l'ensemble des livres de langue française) et DOCUMENT_{En} (l'ensemble des livres de langue anglaise), suivant les valeurs de l'attribut *Langue*. En utilisant l'algorithme cTr sur le complément de la relation DOCUMENT_{En} nous obtenons :

$$ccTr(\text{DOCUMENT}_{En}) = \left\{ \begin{array}{l} \text{(Essai , 'ALL' , 'ALL' , 'ALL'),} \\ \text{('ALL' , Paris , 'ALL' , 'ALL'),} \\ \text{('ALL' , 'ALL' , 'ALL' , Français),} \\ \text{(Nouvelles , 'ALL' , Collins , 'ALL'),} \\ \text{(Pédagogie , 'ALL' , Hachette , 'ALL')} \end{array} \right\}$$

D'après l'exemple 3.12,

$$ccTr(\text{DOCUMENT}_{Fr}) = \left\{ \begin{array}{l} \text{('ALL' , 'ALL' , 'ALL' , Anglais),} \\ \text{(Nouvelles , Paris , 'ALL' , 'ALL'),} \\ \text{(Nouvelles , 'ALL' , Hachette , 'ALL'),} \\ \text{(Pédagogie , Marseille , 'ALL' , 'ALL'),} \\ \text{(Pédagogie , 'ALL' , Collins , 'ALL'),} \\ \text{(Essai , Marseille , 'ALL' , 'ALL'),} \\ \text{(Essai , 'ALL' , Hachette , 'ALL'),} \\ \text{('ALL' , Marseille , Hachette , 'ALL')} \end{array} \right\}$$

Donc, l'ensemble des minimaux transversaux cubiques du complément de la relation DOCUMENT est :

$$ccTr(\textsc{Document}) = \min_{\preceq_g} \left(\left\{ \begin{array}{l} \text{(Essai , 'ALL' , 'ALL' , Anglais),} \\ \text{(Essai , Marseille , 'ALL' , 'ALL'),} \\ \text{(Essai , 'ALL' , Hachette , 'ALL'),} \\ \text{(Essai , Marseille , Hachette , 'ALL'),} \\ \text{('ALL' , Paris , 'ALL' , Anglais),} \\ \text{(Nouvelles , Paris , 'ALL' , 'ALL'),} \\ \text{(Nouvelles , Paris , Hachette , 'ALL'),} \\ \text{(Nouvelles , Paris , Hachette , 'ALL'),} \\ \text{(Pédagogie , Paris , Collins , 'ALL'),} \\ \text{(Essai , Paris , Hachette , 'ALL'),} \\ \text{(Nouvelles , Paris , 'ALL' , Français),} \\ \text{(Nouvelles , 'ALL' , Hachette , Français),} \\ \text{(Pédagogie , Marseille , 'ALL' , Français),} \\ \text{(Pédagogie , 'ALL' , Collins , Français),} \\ \text{(Essai , Marseille , 'ALL' , Français),} \\ \text{(Essai , 'ALL' , Hachette , Français),} \\ \text{('ALL' , Marseille , Hachette , Français),} \\ \text{(Nouvelles , 'ALL' , Collins , Anglais),} \\ \text{(Nouvelles , Paris , Collins , 'ALL'),} \\ \text{(Pédagogie , 'ALL' , Hachette , Anglais),} \\ \text{(Pédagogie , Marseille , Hachette , 'ALL')} \end{array} \right\} \right)$$

$$= \left\{ \begin{array}{l} \text{(Nouvelles , Paris , 'ALL' , 'ALL'),} \\ \text{(Nouvelles , 'ALL' , Collins , Anglais),} \\ \text{(Nouvelles , 'ALL' , Hachette , Français),} \\ \text{(Pédagogie , 'ALL' , Hachette , Anglais),} \\ \text{(Pédagogie , Marseille , Hachette , 'ALL'),} \\ \text{(Pédagogie , Paris , Collins , 'ALL'),} \\ \text{(Pédagogie , Marseille , 'ALL' , Français),} \\ \text{(Pédagogie , 'ALL' , Collins , Français),} \\ \text{(Essai , 'ALL' , 'ALL' , Anglais),} \\ \text{(Essai , Marseille , 'ALL' , 'ALL'),} \\ \text{(Essai , 'ALL' , Hachette , 'ALL'),} \\ \text{('ALL' , Paris , 'ALL' , Anglais),} \\ \text{('ALL' , Marseille , Hachette , Français)} \end{array} \right\}$$

3.2.4 Comparaisons entre le treillis cube et le treillis des parties

Pour conclure ce paragraphe, les deux espaces de recherche utilisés pour la fouille de bases de données multidimensionnelles sont comparés. À notre connaissance, le treillis cube est la premiére approche visant à établir un fondement algébrique pour la fouille de bases de données multidimensionnelles. Précédemment, des chercheurs ont tenté d'étendre au contexte multidimensionnel, des solutions éprouvées dans un cadre de bases de données binaires, en utilisant le treillis des parties comme espace de recherche notamment pour les règles d'associations quantitatives (Srikant et Agrawal, 1996) et la classification (Ganascia, 1993; Liu *et al.*, 1998; Dong et Li, 2005; Li *et al.*, 2001). Dans (Mannila et Toivonen, 1997), une approche par langage (langage de motifs) est introduite comme un modèle théorique général pour la fouille de données.

Malheureusement, cette tentative de formalisation ne constitue pas une panacée car la propriété fondamentale de graduation, qui est nécessaire pour les algorithmes par niveau et la structure convexe, n'est pas toujours vérifiée. C'est pourquoi il apparaît qu'un cadre de travail spécifique, treillis des parties pour les bases de données binaires et treillis cube pour les bases de données multidimensionnelles, est indispensable.

Sur un plan théorique, nous présentons dans ce paragraphe une analyse comparative des démarches d'extension du treillis des parties au contexte multidimensionnel et le treillis cube. Cette comparaison est menée en étudiant les espaces de recherche à explorer, les espaces de solutions, ainsi que le comportement des algorithmes par niveau.

En considérant le treillis de parties $\mathcal{L}(r)$ et le treillis cube $CL(r)$ comme des espaces de recherche pour l'extraction de motifs multidimensionnels contraints, la comparaison porte sur quatre points distincts : la taille des treillis, leurs caractéristiques, la justesse des solutions obtenues pour les conjonctions de contraintes et la complexité des algorithmes par niveau. Le tableau 3.4 récapitule les différences entre ces deux treillis.

– **Taille des treillis :** dans un premier temps, la taille des deux treillis ainsi que la taille maximale des niveaux associés sont examinées. Cette dernière mesure permet d'affiner la complexité des algorithmes de fouille de données dans le cadre multidimensionnel.

  $|\mathcal{L}(r)| = 2^{\sum_{A\in\mathcal{D}}|Dim(A)|}$, alors que $|CL(r)| = \Pi_{A\in\mathcal{D}}(|Dim(A)|+1)+1$ (proposition 3.6). Une borne supérieure pour la cardinalité du treillis cube est $\mathcal{O}((max_{A\in\mathcal{D}}(|Dim(A)|+1))^{|\mathcal{D}|})$. Considérons par exemple une relation avec 5 attributs ayant chacun 10 valeurs, nous avons : $|\mathcal{L}(r)| = 2^{50} = 1125899906842624$, alors que $|CL(r)| = 11^5 + 1 = 161052$.
  Soit $n = \sum_{A\in\mathcal{D}} |Dim(A)|$. La taille du niveau le plus large dans $\mathcal{L}(r)$ est bornée par $\binom{n}{n/2}$, qui est asymptotique à $\frac{2^n}{\sqrt{n}}\sqrt{\frac{2}{\pi}}$ (Beeri *et al.*, 1984), alors que la taille maximale d'un niveau du treillis cube est bornée par $\binom{|\mathcal{D}|}{|\mathcal{D}|/2} \max_{A\in\mathcal{D}}(|Dim(A)|)^{|\mathcal{D}|}$ qui est asymptotique à $\frac{2^{|\mathcal{D}|}}{\sqrt{|\mathcal{D}|}}\sqrt{\frac{2}{\pi}} * \max_{A\in\mathcal{D}}(|Dim(A)|)^{|\mathcal{D}|}$.

  En conclusion, la taille du niveau le plus large dans $\mathcal{L}(r)$ est donc exponentielle par rapport au nombre de valeurs des attributs de la relation (*i.e.* $\sum_{A\in\mathcal{D}} |Dim(A)|$) alors que celle de $CL(r)$ est exponentielle dans le nombre d'attributs (*i.e.* $|\mathcal{D}|$). Notons qu'en pratique $|\mathcal{D}|$ est généralement une constante.

– **Caractéristiques des treillis :** comme la relation d'ordre sur les deux treillis n'est pas la même, nous pouvons déduire deux conséquences :
  – les opérateurs infimum et supremum ne sont pas les mêmes dans les deux treillis : dans le treillis des parties $\bigwedge = \cap$ et $\bigvee = \cup$, alors que dans le treillis cube $\bigwedge = +$ et $\bigvee = \bullet$;
  – le treillis des parties est un treillis distributif, alors que le treillis cube est un treillis atomique, co-atomique et gradué (*cf.* théorème 3.5).

  D'un point de vue plus conceptuel, nous pouvons dire que le treillis des parties est le treillis d'un ensemble d'atomes (valeurs d'attributs) alors que le treillis cube est le treillis d'un ensemble de molécules (tuples).

– **Justesse des solutions :** nous avons précédement expliqué que, pour les problèmes multidimensionnels, le treillis des parties $\mathcal{L}(r)$ englobe des solutions sémantiquement erronées alors que le treillis cube $CL(r)$ est exactement l'espace de recherche valide. Plus

TABLE 3.4 – Différences entre le treillis cube de r et le treillis des parties correspondant

	Treillis cube $CL(r)$	Treillis des parties $\mathcal{L}(r)$
Relation d'ordre	Généralisation ($\preceq_g$)	Inclusion ($\subseteq$)
Opérateur $\bigwedge$	Somme ($+$)	Intersection ($\cap$)
Opérateur $\bigvee$	Produit ($\bullet$)	Union ($\cup$)
Taille du niveau le plus large	$\mathcal{O}(\frac{2^{\lvert\mathcal{D}\rvert}}{\sqrt{\lvert\mathcal{D}\rvert}}\sqrt{\frac{2}{\pi}})$	$\mathcal{O}(\frac{2^n}{\sqrt{n}}\sqrt{\frac{2}{\pi}})$

précisément, le plongement d'ordre Φ (*cf.* proposition 3.3) montre que pour tout tuple du treillis cube, il existe un élément équivalent dans le treillis des parties et que sa sémantique est correcte alors que la réciproque est fausse car Φ n'est pas bijective. $\forall t \in CL(r)$, $\nexists a_i,a_j \in \Phi(t)$ et $a_i,a_j \in Dim(A_k)$ d'après la définition 3.1. Or, $\forall A_k \in \mathcal{D}, \forall a_i,a_j \in Dim(A_k)$ avec $i \neq j$, $(a_i,a_j) \in \mathcal{L}(r)$ et pourtant nous savons que ces combinaisons sont invalides. Si la contrainte antimonotone de fréquence « $Freq(t) \leq minfreq$ » est utilisée pour l'extraction, un algorithme de type APRIORI élague ces couples invalides au deuxième niveau. Par contre, pour les autres contraintes (monotones et/ou antimonotones) ces couples peuvent faire partie du résultat.

– **Complexité des algorithmes par niveau :** le fait de générer des motifs erronés a évidemment des conséquences pour les algorithmes sous-jacents. Elles sont mises en évidence en comparant la taille des bordures pertinentes pour les contraintes monotones du type « $Freq() \leq minfreq$ » et antimonotones du type « $Freq() \geq minfreq$ ». Considérons les solutions les plus générales vérifiant *cmc* pour $\mathcal{L}(r)$ et $CL(r)$. Nous avons :

$$|L_{cmc}(\mathcal{L}(r)_{cmc})| \geq |L_{cmc}(CL(r)_{cmc})| + \sum_{A \in \mathcal{D}} \frac{|Dim(A)|^2 - |Dim(A)|}{2}$$

Dans le cadre des contraintes antimonotones, la bordure négative pour $\mathcal{L}(r)$ comporte, elle aussi, des motifs multidimensionnels erronés (les couples de valeurs d'un même attribut), donc sa taille est plus grande que L^-_{camc} [4] pour $CL(r)$. En fait, le nombre d'éléments supplémentaires dans la bordure L^-_{camc} pour $\mathcal{L}(r)$ est exactement le nombre maximal donné précédemment (les mêmes couples sont considérés). Cette augmentation de la bordure négative a un impact d'autant plus néfaste sur la complexité des algorithmes par niveau que les attributs ont de grands ensembles de valeurs. C'est la raison de l'inefficacité, dans un contexte multidimensionnel, des algorithmes par niveau, explorant $\mathcal{L}(r)$, qui, au deuxième niveau, construisent toutes les combinaisons possibles, calculent leurs supports et par conséquent, pour les niveaux suivants, augmentent le temps d'élagage en utilisant de trop grandes bordures.

Les auteurs du treillis cube (Casali *et al.*, 2003a) ont montré la nécessité d'un fondement pour la fouille de bases multidimensionnelles qui ne soit pas celui de la fouille de bases binaires, en développant des points indiscutables d'argumentation :

1. un espace de recherche dont les éléments ne sont pas structurés est incomparablement plus volumineux qu'un espace dans lequel ils le sont. Entre la taille des deux espaces étudiés, le facteur de plus d'un milliard, donné en exemple à travers une « petite » relation de

4. Ensemble des tuples minimaux ne vérifiant pas la contrainte monotone

cinq attributs, est suffisamment explicite. Compte tenu des volumes mis en jeu dans les entrepôts et en fouille de données, cette raison doit à elle seule convaincre.

2. un espace de recherche dont les éléments ne sont pas structurés englobe un ensemble de combinaisons sémantiquement « impossibles » mais pour lesquelles des calculs doivent être effectués. Le nombre de ces combinaisons est exponentiel dans la cardinalité des dimensions (dont le domaine est réputé très large (Ross et Srivastava, 1997; Beyer et Ramakrishnan, 1999; Han et Kamber, 2006)). Le coût du calcul évoqué est donc très important. De plus,
   (*i*) dans le meilleur des cas, ces combinaisons sont élaguées de l'espace des solutions mais ajoutées à la bordure négative. Les résultats sont donc corrects, mais avec dégradation des performances ;
   (*ii*) dans le pire des cas, ces combinaisons font partie de l'espace de solution du problème.

3.3 Approches de réduction sémantique

En fouille de données aussi bien que pour la gestion d'entrepôts, un problème majeur concerne la taille des jeux de données manipulés et surtout des résultats générés. Il est donc naturel que de nombreuses approches se soient préoccupées de cette question (Lakshmanan *et al.*, 2002; Casali *et al.*, 2003b; Morfonios et Ioannidis, 2006; Xin *et al.*, 2006). C'est pour cela que nous nous sommes intéréssés à deux approches de réduction sémantique, le cube fermé et le cube quotient, car elles sont particuliérement représentatives de ce type de méthodes.

3.3.1 Cubes fermés

Lors du calcul de cube de données, l'explosion combinatoire des résultats est un phénoméne bien connu (Han et Kamber, 2006). L'approche par cube fermé, présentée ici, vise à représenter un cube de données sans perte d'information et avec une réduction notable de l'espace de stockage nécessaire. L'idée sous-jacente est d'éliminer les redondances éventuelles en ne conservant qu'un représentant pour un ensemble de tuples issus des mêmes données de la relation d'origine. Cette représentation s'appuie sur le concept de connexion cubique. Dans le même esprit que la connexion de Galois (Ganter et Wille, 1999), la connexion cubique est un couple de fonctions (λ,σ), tel que λ est une application définie du treillis cube de r vers le treillis des parties de $Tid(r)$ (ensemble des identifiants des tuples de r) et σ est la fonction duale à λ. Finalement, un opérateur de fermeture de $CL(r)$ sur r est défini en utilisant la connexion cubique.

Définition 3.13 (Connexion cubique) **-** Soit $Rowid : r \to \mathbb{N}^*$ une fonction qui associe à chaque tuple un unique entier positif et $Tid(r) = \{Rowid(t) \mid t \in r\}$. Soit λ et σ deux fonctions définies comme suit :

- $\lambda : CL(r) \to \langle \mathcal{P}(Tid(r)), \subseteq \rangle$
  $t \mapsto \cup\{Rowid(t') \in Tid(r) \mid t \preceq_g t' \text{ et } t' \in r\}$
- $\sigma : \langle \mathcal{P}(Tid(r)), \subseteq \rangle \to CL(r)$
  $P \mapsto +\{t \in r \mid Rowid(t) \in P\}$

La fonction λ associe à tout tuple du treillis cube l'ensemble des identifiants des tuples qui le généralisent. La fonction σ quant à elle s'applique sur un ensemble d'identifiants de tuples et retourne la somme (*cf.* definition 3.4) de tous les tuples dotés de ces identifiants. En d'autres termes, elle renvoie le tuple le plus spécifique généralisant tous les tuples pourvus de ces identifiants.

Proposition 3.11 - La connexion cubique $rc = (\lambda,\sigma)$ est une connexion de Galois entre le treillis cube de r et le treillis des parties de $Tid(r)$.

La connexion cubique étant un cas particulier de la connexion de Galois, la composition des deux fonctions, $\sigma \circ \lambda$, est un opérateur de fermeture dont une définition équivalente est donnée ci-dessous.

Définition 3.14 (Opérateur de fermeture cubique) **-** L'opérateur de fermeture cubique associe à tout tuple t du treillis cube un unique tuple appelé la fermeture de t. Il est obtenu en considérant l'ensemble des tuples plus spécifiques que t et en déterminant le tuple le général de cet ensemble grâce à l'opérateur de somme. Cet opérateur est noté $\mathbb{C}$ et défini comme suit :

$$\begin{array}{rcl} \mathbb{C} & : & CL(r) \rightarrow CL(r) \\ t & \mapsto & \left\{ \begin{array}{l} +\, t' \mid t \preceq_g t' \text{ et } t' \in r \\ (\emptyset, \ldots, \emptyset) \text{ sinon.} \end{array} \right. \end{array}$$

Exemple 3.15 - En considérant l'espace multidimensionnel de la relation DOCUMENT, nous avons :

$$\begin{array}{rl} \mathbb{C}((\text{Nouvelle,ALL,ALL})) & = (\text{Nouvelles, Marseille, Collins }) + (\text{Nouvelles, Marseille, Hachette }) \\ & = (\text{Nouvelles, Marseille, ALL }) \\ \mathbb{C}((\text{Essai, ALL, ALL})) & = (\text{Essai, Paris, Collins}) \end{array}$$

Proposition 3.12 - $\mathbb{C}$ est un opérateur de fermeture de $CL(r)$ sur r.

L'opérateur $\mathbb{C}$ étant un opérateur de fermeture, il bénéficie des trois propriétés données dans le corollaire suivant.

Corollaire 3.13 - $\mathbb{C}$ satisfait les propriétés suivantes :

1. $t \preceq_g t' \Rightarrow \mathbb{C}(t,r) \preceq_g \mathbb{C}(t',r)$ (isotonie)
2. $t \preceq_g \mathbb{C}(t,r)$ (extensivité)
3. $\mathbb{C}(t,r) = \mathbb{C}(\mathbb{C}(t,r),r)$ (idempotence).

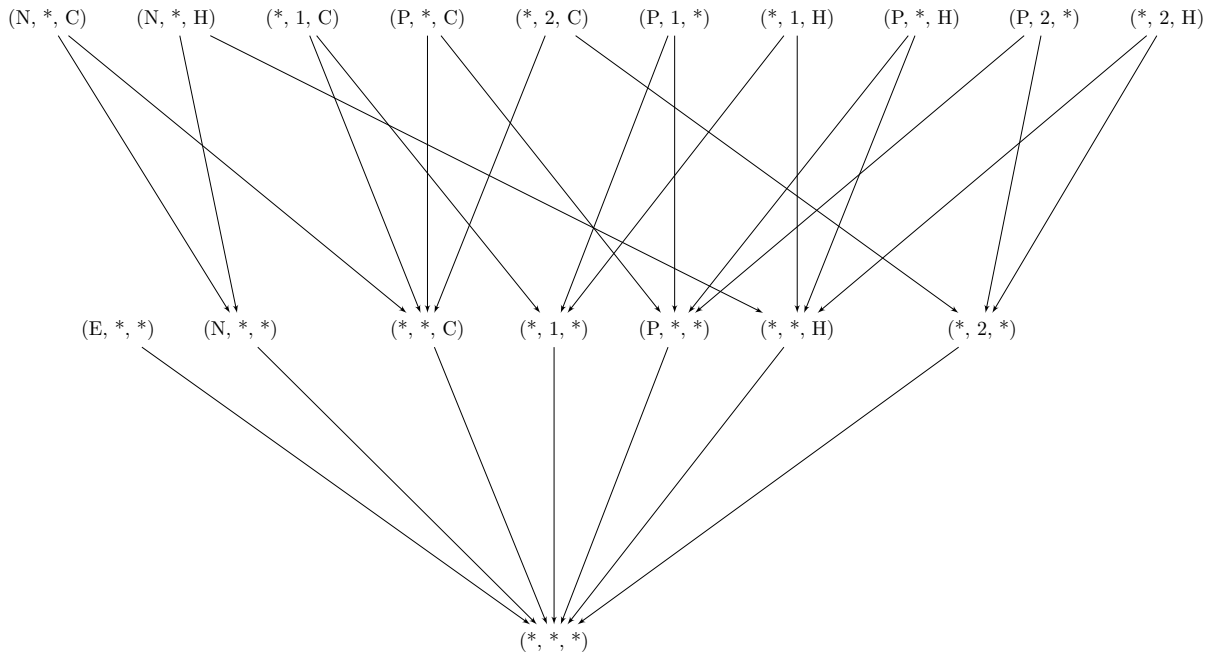
Le système de fermeture contient l'ensemble des fermetures de tous les tuples du treillis cube. Chacun d'eux est nommé tuple fermé cubique. La propriété d'extensivité garantit que tout fermé cubique est plus général que les tuples qui l'ont généré. De plus, il est égal à sa propre fermeture de par la propriété d'idempotence. L'isotonie assure le respect de l'ordre des fermés par rapport à l'ordre des tuples qui les ont générés.

Définition 3.15 (Système de fermeture cubique) **-** Soit $\mathbb{C}(r) = \{\ t \in CL(r) \mid \mathbb{C}(t,r) = t\}$. $\mathbb{C}(r)$ est un système de fermeture sur r et l'opérateur de fermeture associé est $\mathbb{C}$. Tout tuple appartenant à $\mathbb{C}(r)$ est un tuple fermé ou un fermé cubique.

Exemple 3.16 - En considérant l'espace multidimensionnel de la relation exemple, nous avons :

$$\mathbb{C}(r) = \left\{ \begin{array}{ll} (\text{Nouvelles , Marseille , Collins }), & (\emptyset, \ldots, \emptyset), \\ (\text{Nouvelles , Marseille , Hachette }), & (\text{Nouvelles , Marseille , 'ALL' }), \\ (\text{Pédagogie , Marseille , Collins }), & (\text{Pédagogie , 'ALL' , 'ALL' }), \\ (\text{Pédagogie , Paris , Hachette }), & (\text{'ALL' , Marseille , 'ALL' }), \\ (\text{Essai , Paris , Collins }), & (\text{'ALL' , Paris , 'ALL' }), \\ (\text{'ALL' , Marseille , Collins }), & (\text{'ALL' , 'ALL' , Hachette }), \\ (\text{'ALL' , 'ALL' , Collins }) & \end{array} \right\}$$

FIGURE 3.2 – Représentation de l'ensemble des clefs de la relation DOCUMENT



Pour chaque tuple fermé cubique t, tous les tuples de $CL(r)$ dont la fermeture est t sont appelés les générateurs de t. Les tuples générateurs minimaux de t selon $\preceq_g$ sont les clés cubiques de ce tuple.

Définition 3.16 (Clef cubique) **-** Soit t un tuple fermé. $Key(t) = \min_{\preceq_g}(\{t' \in CL(r) \mid t' \preceq_g t \text{ et } \mathbb{C}(t',r) = t\})$ est l'ensemble de générateurs minimaux de t. Chaque tuple de $Key(t)$ est une clef cubique.

Exemple 3.17 - En considérant l'espace multidimensionnel de la relation DOCUMENT, nous avons : Key((Nouvelle,ALL,ALL)) = (Nouvelle,ALL,ALL) et Key((Essai, Paris, Collins)) = (Essai,ALL,ALL). La figure 3.2 représente l'ensemble de toutes les clefs cette relation.

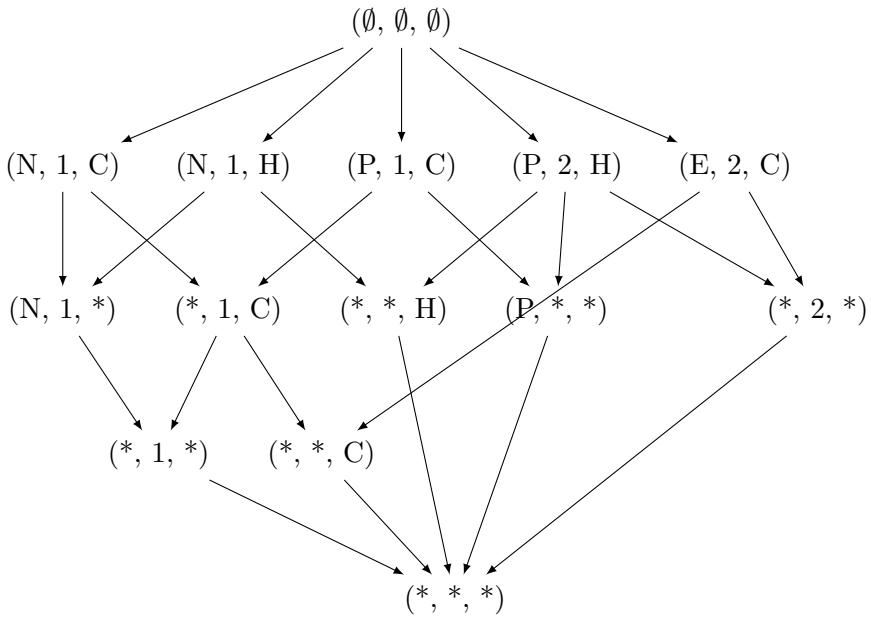
Le lemme suivant caractérise le nombre maximal de clefs cubiques que peut avoir un tuple fermé. Cependant, contrairement au modèle binaire, le nombre de clefs n'est plus exponentiel en fonction du nombre de valeurs, mais est exponentiel en fonction du nombre d'attributs dimensions.

Lemme 3.14 - Soit t un tuple fermé, le nombre maximal de clés cubiques pour t est $\binom{|\Phi(t)|}{|\Phi(t)|/2}$.

En appliquant le théorème de G. Birkhoff (Birkhoff, 1970) sur le treillis cube avec l'opérateur $\mathbb{C}$ comme opérateur de fermeture, il est possible de construire un treillis regroupant tous les fermés cubiques, l'ordre de généralisation étant la relation d'ordre de ce treillis. Cependant, contrairement aux autres treillis des fermés (Ganter et Wille, 1999) ce treillis n'est pas quelconque : il est co-atomique.

Théorème 3.15 - L'ensemble partiellement ordonné $CCL(r) = \langle \mathbb{C}(r), \preceq_g \rangle$ est un treillis complet et co-atomique appelé treillis cube fermé . De plus, nous avons :

FIGURE 3.3 – Diagramme de Hasse du treillis cube fermé de la relation DOCUMENT



1. $\forall\ T \subseteq CCL(r),\ \bigwedge T = +_{t \in T}\ t$
2. $\forall\ T \subseteq CCL(r),\ \bigvee T = \mathbb{C}(\bullet_{t \in T}\ t, r)$

Exemple 3.18 - La figure 3.3 représente le treillis cube fermé de notre relation DOCUMENT.

Tous les tuples ayant la même fermeture généralisent les mêmes tuples de la relation originelle. Comme ils généralisent les mêmes tuples originaux, ils partagent la même valeur agrégée de la mesure (propriété du GROUP-BY). Pour tout tuple du treillis cube, il suffit de calculer sa fermeture pour retrouver sa mesure. Ainsi le cube fermé est une couverture du cube de données.

3.3.2 Cubes quotients

Un cube quotient (Lakshmanan *et al.*, 2002) est un résumé du cube de données pour certaines fonctions agrégatives comme COUNT,SUM, MIN, MAX, AVG et TOP-k. De plus, le cube quotient préserve la sémantique des opérateurs ROLL-UP/DRILL-DOWN sur le cube. Pour définir le cube quotient, les auteurs utilisent le concept de classe d'équivalence (*i.e.* ensemble de tuples satisfaisant une relation d'équivalence).

Définition 3.17 (Classes d'équivalence convexes) **-** Soit $\mathcal{C} \subseteq CL(r)$ une classe d'équivalence. Nous disons que $\mathcal{C}$ est convexe si et seulement si :

$$\forall t \in CL(r) \text{ si } \exists t', t'' \in \mathcal{C} \text{ tels que } t' \preceq_g t \preceq_g t'' \text{ alors } t \in \mathcal{C}$$

Une partition $\mathcal{P}$ de $CL(r)$ qui comprend uniquement des classes d'équivalence convexes est appelée partition convexe.

La propriété de convexité rend possible la représentation de chaque classe d'équivalence à travers ses tuples minimaux et et ses tuples maximaux. Les tuples intermédiaires ne sont, dès lors, plus utiles et la représentation sous-jacente est réduite. Pour être sûr que la partition est convexe, la relation d'équivalence suivante est utilisée.

Définition 3.18 (Relation d'équivalence quotient) **-** Soit f_{val} une fonction mesure. Nous définissons la relation d'équivalence $\equiv_f$ comme la fermeture réflexive transitive de la relation suivante τ : soit t, t' deux tuples, $t \ \tau \ t'$ est vraie si et seulement si (i) $f_{val}(t,r) = f_{val}(t',r)$ et (ii) t est soit un parent soit un enfant de t'.
La relation d'équivalence $\equiv_f$ est dite *relation d'équivalence quotient* si et seulement si elle satisfait la proprieté de congruence faible : $\forall t,t',u,u' \in CL(r)$, si $t \equiv_f t', u \equiv_f u', t \preceq_g u$ et $u' \preceq_g t'$, alors $t \equiv_f u$.

Nous notons $[t]_{\equiv_f}$ la classe d'équivalence de t ($[t]_{\equiv_f} = \{t' \in CL(r)$ tel que $t \equiv_f t'\}$). Alors le cube quotient est défini comme l'ensemble des classes d'équivalence, chacune d'entre elles étant pourvue de la valeur de la mesure.

Définition 3.19 (Cube quotient) **-** Soit $CL(r)$ le treillis cube de la relation r et $\equiv_f$ une relation d'équivalence quotient. Le Cube Quotient de r, noté $QuotientCube(r, \equiv_f)$, est défini comme suit :

$$QuotientCube(r, \equiv_f) = \{([t]_{\equiv_f}, f_{val}(t,r)) \text{ tel que } t \in CL(r)\}.$$

Le cube quotient de r est une partition convexe de $CL(r)$.

Pour deux classes d'équivalence $\mathcal{C},\mathcal{C}' \in QuotientCube(r, \equiv_f)$, $\mathcal{C} \preceq_{QC} \mathcal{C}'$ quand $\exists t \in \mathcal{C}$ et $\exists t' \in \mathcal{C}'$ tels que $t \preceq_g t'$.

Exemple 3.19 - En considérant l'espace multidimensionnel de la relation exemple, la table 3.5 et la figure 3.4 donnent le cube quotient de DOCUMENT pour la fonction agrégative SUM. Chaque classe d'équivalence est représentée par ses tuples minimaux et ses tuples maximaux.

La construction d'un cube quotient dépend de la relation d'équivalence quotient choisie. Par conséquent, pour deux relations d'équivalence quotients, leur cubes quotients associés peuvent différer.

3.4 Approche de réduction syntaxique : le cube partition

Comme les deux approches précédentes, le cube partition (Casali *et al.*, 2009; Laporte *et al.*, 2002) propose une représentation des cubes plus concise que les cubes eux-mêmes. Cependant, elle n'exploite pas les redondances incluses dans les cubes de données comme le font le cube fermé et le cube quotient mais relève plutôt d'une optimisation au niveau logique. L'approche part d'un constat simple : pour chaque tuple t du cube de données, il n'est pas nécessaire de stocker les valeurs de tous les attributs dimensions, mais il suffit simplement de connaître d'une part les dimensions pourvues de valeurs différentes de ALL et d'autre part l'identifiant d'un tuple dans la relation originelle ayant les mêmes valeurs que t pour les dimensions considérées. À l'aide de ces deux informations, toutes les valeurs des dimensions peuvent être retrouvées. L'approche s'appuie sur le modèle partitionnel (Spyratos, 1987) pour définir une nouvelle structure réduite pour les cubes de données. Elle en propose une représentation relationnelle simple, puis introduit un algorithme efficace permettant son calcule.

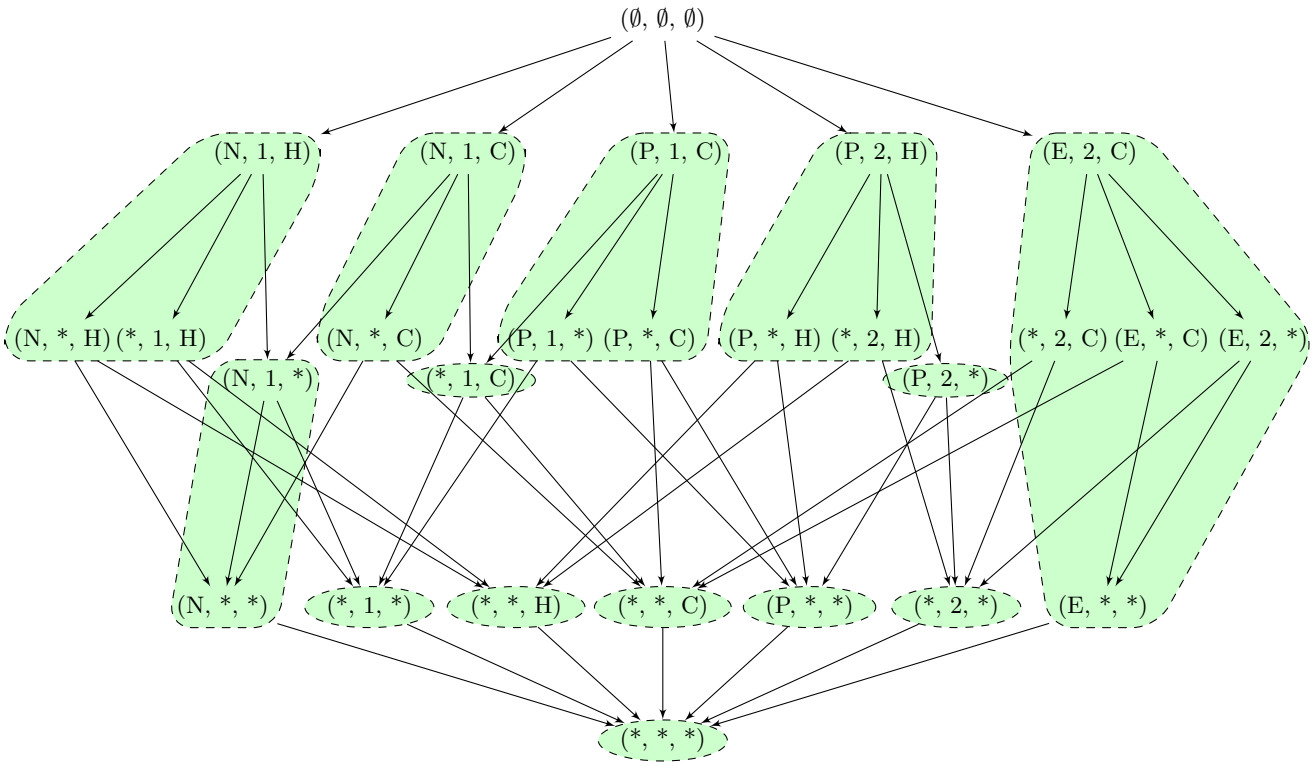
3.4.1 Concepts de base

La représentation proposée pour le cube de données est basée sur un ensemble de concepts simples et s'appuie essentiellement sur la notion de partition. La première définition revisite le

TABLE 3.5 – *QuotientCube*(DOCUMENT, $\equiv_{\text{SUM}}$)

Tuples Minimaux	Tuples Maximaux	SUM_{val}
('ALL' , 'ALL' , 'ALL')	('ALL' , 'ALL' , 'ALL')	1100
(Pédagogie , 'ALL' , 'ALL')	(Pédagogie , 'ALL' , 'ALL')	400
('ALL' , Marseille , 'ALL')	('ALL' , Marseille , 'ALL')	800
('ALL' , Paris , 'ALL')	('ALL' , Paris , 'ALL')	300
('ALL' , 'ALL' , Collins)	('ALL' , 'ALL' , Collins)	900
('ALL' , 'ALL' , Hachette)	('ALL' , 'ALL' , Hachette)	200
(Nouvelles , 'ALL' , 'ALL')	(Nouvelles , Marseille , 'ALL')	500
(Nouvelles , 'ALL' , Collins)	(Nouvelles , Marseille , Collins)	400
(Nouvelles , 'ALL' , Hachette) ('ALL' , Marseille , Hachette)	(Nouvelles, Marseille, Hachette)	100
(Pédagogie , Paris , 'ALL') (Pédagogie , 'ALL' , Hachette) ('ALL' , Paris , Hachette)	(Pédagogie , Paris , Hachette)	100
(Pédagogie , Marseille , 'ALL') (Pédagogie , 'ALL' , Collins)	(Pédagogie, Marseille, Collins)	300
(Essai , 'ALL' , 'ALL') ('ALL' , Paris , Collins)	(Essai, Paris, Collins)	200
('ALL' , Marseille , Collins)	('ALL' , Marseille , Collins)	700

FIGURE 3.4 – *QuotientCube*(DOCUMENT, $\equiv_{\text{SUM}}$)



concept de classe d'équivalence en utilisant celui d'ensemble en accord (Mannila et Toivonen, 1996; Lopes *et al.*, 2002).

Définition 3.20 (Classe-DM) **-** Soit r une relation d'une base de données et X un ensemble d'attributs dimensions. Une classe Dimension-Mesure (classe-DM) d'un tuple t selon X, notée $[t]_X$, est définie comme l'ensemble des couples (identifiant(u), mesure(u)) de tous les tuples u en accord avec t selon l'ensemble des attributs de X (*i.e.* l'ensemble des tuples u partageant avec t les mêmes valeurs pour X). Nous avons donc : $[t]_X = \{(u[RowId],u[\mathcal{M}])$ tels que $u[X] = t[X], \forall u \in r\}$.

Les classes-DM sont construites afin d'être agrégées lors du calcul de cube en appliquant une fonction sur l'attribut mesure. En fait, il n'est pas utile de conserver l'ensemble des couples d'éléments de chaque classe-DM. Pour capturer toute l'information véhiculée par une classe-DM, il suffit de connaître le résultat de la fonction agrégative appliquée sur tous les tuples de la classe ainsi que les valeurs des attributs dimensions concernés. Ces dernières sont communes à tous les tuples de la classe-DM qui sont, par définition, en accord sur les dimensions. L'idée pour retrouver les valeurs de ces dimensions est d'y faire référence via l'identifiant d'un des tuples qui les partagent. Ce tuple est en quelque sorte le représentant de la classe-DM. Ainsi, chaque classe-DM est représentée par un couple de valeurs numériques : la première est un des identifiants des tuples rassemblés dans la classe considérée et la seconde est la valeur calculée de la mesure pour toute la classe.

Exemple 3.20 - Avec notre relation exemple (*cf.* table 3.1), la classe-DM associée au premier tuple selon l'attribut $Ville$ regroupe tous les couples (identifiant, mesure) des tuples satisfaisant, comme t_1, la contrainte ($Ville$='Marseille') :

$$[t_1]_{Ville} = \{(1,400), (2,100), (4,300)\}$$

. Dans un contexte multidimensionnel, il est possible d'associer, à chaque classe-DM, un tuple multidimensionnel. Ainsi, la classe $[t_1]_{Ville}$ correspond au tuple multidimensionnel t = (Marseille, ALL, ALL, ALL) : 800.

Toutes les classes-DM pour un ensemble d'attributs dimensions X sont réunies en un seul ensemble appelé partition dimension-mesure ou partition-DM.

Définition 3.21 (Partition-DM) **-** Soit r une relation et X un ensemble de dimensions, la partition-DM de r selon X est définie par : $\Pi_X(r) = \{[t]_X, \forall\ t \in r\}$.

Exemple 3.21 - Dans les exemples, pour une meilleure lisibilité, les classes-DM sont délimitées par les symboles '<' et '>' dans l'écriture des partitions-DM. La partition selon l'attribut $Ville$ est : $\Pi_{Ville}(r) = \{$ <(1,400), (2,100), (4,300)>, <(3,100), (5, 200)> $\}$, celle selon l'attribut $Type$ est : $\Pi_{Type}(r) = \{$<(1,400),(2,100) >, <(3,100),(4,300) >, <(5,200) >$\}$.

Considérons deux partitions-DM calculées selon les ensembles de dimensions X et Y. Leur produit retourne la partition-DM selon $X \cup Y$. Un tel produit est effectué en réalisant l'intersection, deux à deux, des classes-DM des deux partitions-DM et en conservant seulement les classes non vides (dont la cardinalité est supérieure ou égale à 1).

Définition 3.22 (Produit de Partitions-DM) **-** Soit r une relation, X et Y deux ensembles de dimensions, $\Pi_X(r)$ et $\Pi_Y(r)$ leur partition-DM respective. Le produit des partitions-DM $\Pi_X(r)$

et $\Pi_Y(r)$, noté $\Pi_X(r) \bullet_p \Pi_Y(r)$, rend la Partition selon $X \cup Y$ et est obtenu comme suit :

$$\begin{aligned}\Pi_X(r) \bullet_p \Pi_Y(r) &= \Pi_{X \cup Y}(r) \\ &= \{[t]_{X \cup Y} = [t]_X \cap [t]_Y : [t]_{X \cup Y} \neq \emptyset,\ [t]_X \in \Pi_X(r) \text{ et } [t]_Y \in \Pi_Y(r)\}\end{aligned}$$

Exemple 3.22 - Dans notre relation example, les partitions-DM selon les attributs Ville (V) et Type (T) sont les suivantes :

$$\begin{aligned}\Pi_V(r) =&\{< (1{,}400), (2{,}100), (4{,}300) >, < (3{,}100), (5, 200) >\} \text{ et} \\ \Pi_T(r) =&\{< (1{,}400), (2{,}100) >, < (3{,}100), (4{,}300) >, < (5{,}200) >\}\end{aligned}$$

Nous avons donc :

$$\begin{aligned}\Pi_{TV}(r) &= \Pi_T(r) \bullet_p \Pi_V(r) \\ &= \{< (1{,}400), (2{,}100) >, < (4{,}300) > , < (3{,}100) >, < (5{,}200) >\}\end{aligned}$$

Le cuboïde selon Ville et Type est donc composé de 4 tuples (un par classe d'équivalence DM) :

(Nouvelles , Marseille , 'ALL' , 'ALL') : 500
(Pédagogie , Marseille , 'ALL' , 'ALL') : 100
(Pédagogie , Paris , 'ALL' , 'ALL') : 300
(Essai , Paris , 'ALL' , 'ALL') : 200

Tous les couples représentant les classes-DM sont groupés ensemble dans un cuboïde partition.

Définition 3.23 (Cuboïde Partition) **-** Soit $\Pi_X(r)$ la partition-DM de r selon X et f une fonction agrégative. Pour chaque classe-DM, $[t]_X.\mathcal{M}$ est la valeur de l'attribut mesure. Le cuboïde partition selon l'ensemble de dimensions X, noté $Cuboid_X(r)$, est défini comme suit :

$$Cuboid_X(r) = \{(t[RowId], f([t]_X.\mathcal{M})), \forall\ [t]_X \in \Pi_X(r)\}$$

Exemple 3.23 - Le cuboïde partition selon les dimensions Villle (V) et Type (T) est le suivant :

$$Cuboid_{VT}(r) = \{(1{,}500), (3{,}100), (4{,}300), (5{,}200)\}$$

La représentation du cube de données peut être définie comme l'ensemble de tous les cuboïdes partitions selon toutes les combinaisons de dimensions.

Définition 3.24 (cube partition) **-** Soit r une relation. Le cube partition associé à r est défini par : Partition_Cube$(r) = \{Cuboid_X(r), \forall\ X \in \mathcal{P}(\mathcal{D})\}$, où $\mathcal{P}$ est le treillis des parties.

Exemple 3.24 - Le cube partition de notre relation exemple pour la fonction agrégative Sum est donné dans la table 3.6. Il contient $2^3 = 8$ cuboïdes partitions, chacun correspondant à une combinaison de dimensions (utilisée comme index pour identifier les cuboïdes).

3.4.2 L'algorithme Pcube

La solution algorithmique proposée avec la structure de cube partition est décrite ci-après.

Tout d'abord, une définition simple de l'ordre lectique (ou ordre co-lexicographique) (Ganter et Wille, 1999) est donnée. Ensuite l'algorithme récursif permettant l'énumération, selon l'ordre lectique, des sous-ensembles de $\mathcal{P}(\mathcal{D})$ est décrit.

TABLE 3.6 – Cube partition de la relation DOCUMENT

$Cuboid_{\emptyset} =$	$\{(1,1100)\}$
$Cuboid_{T} =$	$\{(1,500),(3,400),(5,200)\}$
$Cuboid_{V} =$	$\{(1,800),(3,300)\}$
$Cuboid_{E} =$	$\{(1,900),(2,200)\}$
$Cuboid_{TV} =$	$\{(1,500),(3,100),(4,300),(5,200)\}$
$Cuboid_{TE} =$	$\{(1,400),(2,100),(3,100),(4,300),(5,200)\}$
$Cuboid_{VE} =$	$\{(1,700),(2,100),(3,100),(5,200)\}$
$Cuboid_{TVE} =$	$\{(1,400),(2,100),(3,100),(4,300),(5,200)\}$

Définition 3.25 (Ordre Lectique) **-** Soit $(\mathcal{D}, <_{\mathcal{D}})$ un ensemble fini totalement ordonné. Nous supposons, par simplicité, que $\mathcal{D}$ peut être défini de la manière suivante : $\mathcal{D} = \{A_1, A_2, \dots A_n\}$. $\mathcal{D}$ est pourvu de l'opérateur défini ci-dessous.

$$\begin{aligned} Max \ : \ & \mathcal{P}(\mathcal{D}) \rightarrow \mathcal{D} \\ & X \ \mapsto \text{le dernier élément de} X \text{ selon } <_{\mathcal{D}} \end{aligned}$$

L'ordre lectique, noté $<_l$, est défini comme suit :

$$\forall\ X, Y\ \in\ \mathcal{P}(\mathcal{D}),\ X\ <_l\ Y \Leftrightarrow Max(X \setminus Y) < Max(Y \setminus X)$$

Cet ordre est un ordre linéaire strict sur l'ensemble des parties d'un ensemble.

Exemple 3.25 - Considérons l'ensemble suivant totalement ordonné $\mathcal{D} = \{E, L, T, V\}$. L'énumération des combinaisons de $\mathcal{P}(\mathcal{D})$, selon l'ordre lectique, donne le résultat suivant : $\emptyset <_l E <_l L <_l EL <_l T <_l ET <_l LT <_l ELT <_l V <_l EV <_l LV <_l ELV <_l TV <_l ETV <_l LTV <_l ELTV$.

Proposition 3.16 - (Ganter et Wille, 1999) $\forall\ X, Y\ \in\ \mathcal{P}(\mathcal{D}),\ X \subset Y \Rightarrow X\ <_l\ Y$.

LS (Lectic Subsets) donne le schéma algorithmique général utilisé par PCUBE qui construit le cube partition.

Schéma algorithmique récursif pour énumérer les sous-ensembles dans l'ordre lectique

L'algorithme LS admet comme paramètres deux sous-ensembles d'attributs dimensions X et Y. Il est basé sur une double récursion. Les appels récursifs forment un arbre binaire équilibré dans lequel chaque branche d'exécution renvoie un sous-ensemble de dimensions. La stratégie générale pour énumérer les combinaisons d'attributs dimensions consiste à considérer tout d'abord les sous-ensembles de dimensions ne contenant qu'une certaine dimension, puis tous ceux qui incluent cette dimension. Plus précisément, l'attribut maximal selon l'ordre $<_{\mathcal{D}}$ est écarté de Y et ajouté à X dans la variable Z. L'algorithme est récursivement appliqué avec (i) X et un nouveau sous-ensemble Y (duquel l'attribut maximal est éliminé), puis (ii) Z et Y. Pour le premier appel de LS, les deux paramètres fournis sont $X = \emptyset$ et $Y = \mathcal{D}$.

La preuve que l'algorithme LS est correct est basée sur la proposition 3.16 et la propriété distributive du treillis des attributs dimensions :

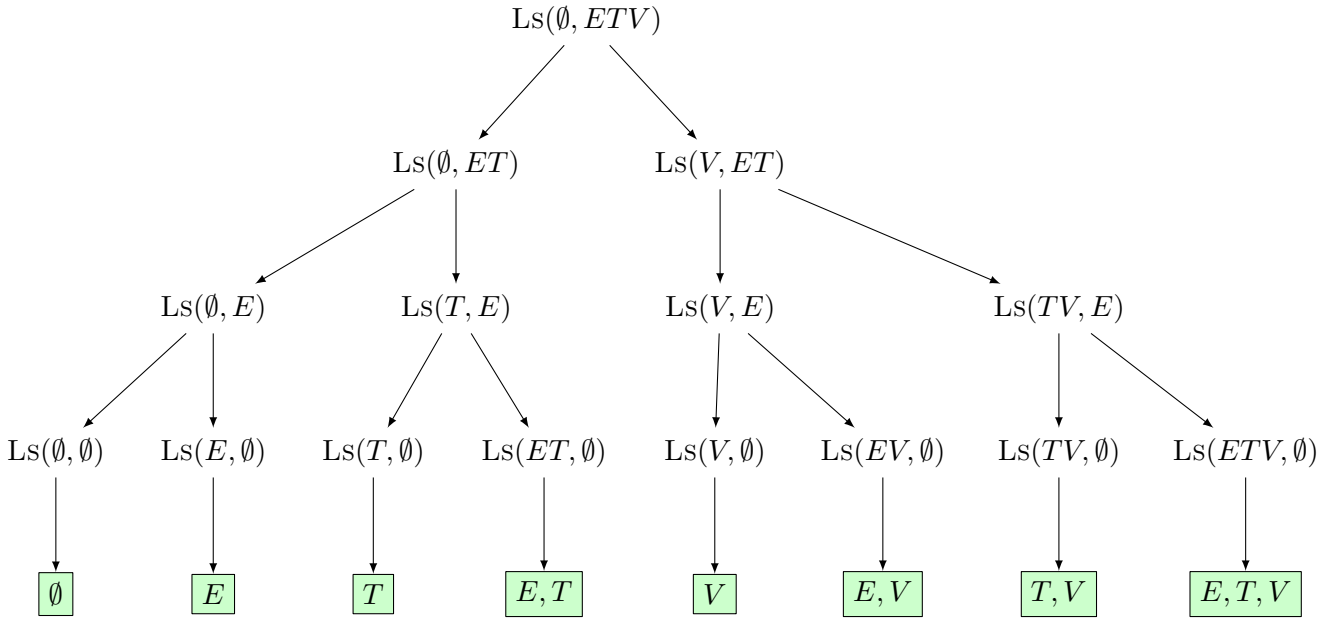
$$\forall\ A \in \mathcal{D}, \forall\ X \subseteq \mathcal{D}, \mathcal{P}(X \cup A) \cap \mathcal{P}(\mathcal{D} \setminus (X \cup A)) = \emptyset.$$

Algorithme 11 Algorithm Ls

Entrée : X et Y deux ensembles de dimensions (au départ $X = \emptyset$ et $Y = \mathcal{D}$)
Sortie : $\mathcal{P}(Y)$

```
si Y = ∅ alors
    Return X
sinon
    A := Max(Y)
    Y := Y\{A}
    LS(X,Y)
    Z := X ∪ {A}
    LS(Z,Y)
fin si
```

FIGURE 3.5 – Arbre des appels récursifs de Ls pour l'ensemble $\{E, T, V\}$



Ainsi, chaque sous-ensemble d'attributs dimensions est énuméré exactement une fois.

Exemple 3.26 - Considérons notre relation exemple. L'arbre binaire des appels récursifs lors du fonctionnement de l'algorithme est illustré par la figure 3.5. Les feuilles de l'arbre correspondent aux sorties qui sont, de gauche à droite, ordonnées dans l'ordre lectique. Dans chaque sous-arbre de gauche, les sous-ensembles ne comprenant pas l'attribut maximal dans la racine de ce sous-arbre sont considérés. Dans les sous-arbres de droite cet attribut maximal est préservé.

L'algorithme, appelé PCUBE, calcule le cube partition. Une étape de pré-traitement est nécessaire afin de construire les partitions-DM selon un seul attribut dimension à partir de la relation d'entrée. Réaliser cette étape initiale permet aussi de calculer le cuboïde selon l'ensemble vide ($Cuboid_{\emptyset}$) et son résultat est retourné. Si les partitions suivant un unique attribut ($\cup_{A \in \mathcal{D}} \Pi_A(r)$) ne tiennent pas en mémoire centrale, alors la stratégie de fragmentation proposée dans (Ross et Srivastava, 1997) et utilisée dans (Beyer et Ramakrishnan, 1999) est appliquée (*cf.*

paragraphe 2.3.5 p. 28). Son idée principale est de diviser, selon un attribut, la relation d'entrée en fragments jusqu'à ce que les partitions-DM associées puissent être chargées. PCUBE adopte le schéma algorithmique général de LS mais il est conçu pour calculer tous les agrégats désirés et donc il restitue la représentation condensée de tous les cuboïdes possibles. PCUBE s'applique sur les partitions-DM et met en œuvre le produit de partitions-DM. Comme LS, ses paramètres d'entrée sont les sous-ensembles de X et Y. La partition-DM associée à Z est calculée en appliquant l'opérateur produit sur les deux partitions en mémoire : $\Pi_X(r)$ and $\Pi_A(r)$. Le deuxième appel récursif est alors effectué. Le pseudo-code de PCUBE est donné dans l'algorithme 12.

Algorithme 12 Algorithm PCUBE

Entrée : Ensemble des partitions-DM $\{\Pi_A,\ A \in \mathcal{D}\}$, X et Y deux ensembles d'attributs dimensions
Sortie : Partition Cube
1: **si** $Y = \emptyset$ **alors**
2: Write_Cuboid(X)
3: **sinon**
4: $A := \max_{<_{\mathcal{D}}}(Y)$
5: $Y := Y \backslash \{A\}$
6: PCUBE(r,X,Y)
7: $Z := X \cup \{A\}$
8: $\Pi_Z(r) := \Pi_X(r) \bullet_p \Pi_A(r)$
9: PCUBE(r,Z,Y)
10: **fin si**

3.4.3 Représentation relationnelle du cube partition

Quand les applications OLAP sont gérées par un système relationel, le cube partition peut être stocké dans une relation, appelée cube, dans laquelle chaque tuple décrit une classe-DM d'un cuboïde selon X. Plus précisément, pour chaque classe-DM, sont connus l'identifiant de son élément représentatif, la valeur de la mesure et la combinaison de dimensions ($DimId$). Comme dans d'autres approches calculant les cubes, les valeurs réelles des dimensions sont codées sous forme d'entiers (Ross et Srivastava, 1997; Beyer et Ramakrishnan, 1999; Han *et al.*, 2001). Le schéma suivant appelé cube partition relationnel est utilisé :

$$r(\underline{RowId},\mathcal{D},\mathcal{M})$$
$$Dimension(\underline{DimId},\mathcal{D})$$
$$Cube(\underline{RowId,DimId}, f(\mathcal{M}))$$

La relation *Dimension* est conçue pour stocker toutes les combinaisons de dimensions. Ses valeurs sont binaires et pour chaque attribut A, A prend la valeur 0 s'il n'appartient pas à la combinaison considérée, 1 sinon. De plus, à partir de la relation originelle, les valeurs réelles des attributs dimensions peuvent être retrouvées pour les éléments représentatifs des classes-DM.

Exemple 3.27 - Pour notre exemple, les deux derniéres relations du cube partition relationnel sont respectivement données dans les tables 3.7 et 3.8.

TABLE 3.7 – Relation *Dimension*

DimId	*T*	*V*	*E*
1	0	0	0
2	1	0	0
3	0	1	0
4	0	0	1
5	1	1	0
6	1	0	1
7	0	1	1
8	1	1	1

TABLE 3.8 – Relation *Cube*

RowId	*DimId*	SUM(Q)
1	1	1100
1	2	500
3	2	400
5	2	200
1	3	800
3	3	300
1	4	900
2	4	200
1	5	600
3	5	100
4	5	300
5	5	200
1	6	400
2	6	100
3	6	100
4	6	300
5	6	200
1	7	700
2	7	100
3	7	100
5	7	200
1	8	400
2	8	100
3	8	100
4	8	300
5	8	200

3.4.4 Évaluation analytique

Lors du calcul de data cubes, $2^{|Dim|}$ combinaisons de dimensions doivent être examinées, à chacune d'entre elles correspond un cuboïde. Pour chaque cuboïde, le nombre de tuples générés dépend de la cardinalité de la combinaison considérée (Shoshani, 1997), qui est notée $|X|$. Comme mentionné précédemment, toutes les approches ne traitent pas les données originelles mais des données codées (pour des raisons évidentes d'optimisation). Effectivement, chaque valeur d'un attribut dimension A_i est remplacée par un entier dans l'intervalle $[0...|A_{i-1}|]$ lors d'une étape de pré-traitement (Shoshani, 1997; Beyer et Ramakrishnan, 1999). Avec cette hypothèse, l'espace de stockage requis pour conserver un cuboïde selon X est :

$$4(|Dim| + 1)|X|$$

L'espace total pour stocker un cube complet est borné par :

$$2^{|Dim|}4(|Dim| + 1)Max(|X|), \forall X \in \mathcal{P}(Dim)$$

En revanche, Pcube génère des représentations concises des cubes de données. Pour chaque cuboïde selon la combinaison de dimensions X, $|X|$ tuples doivent être calculés mais chacun ne nécessite que le stockage de trois valeurs (un identifiant, la valeur agrégée associée et la combinaison de dimensions), chaque valeur requérant 4 octets. Donc les besoins de stockage d'un cuboïde sont :

$$12|X|$$

Comparé à la représentation classique (utilisée par Buc par exemple), ce dernier résultat est vraiment significatif car sitôt que le nombre de dimensions est supérieur à 2, la représentation par cube partition est plus compacte. Bien sûr cet avantage est de plus en plus important au fur et à mesure que le nombre de dimensions croît car le besoin de stockage de Pcube pour tout cuboïde est indépendant du nombre de dimensions. Cette dernière remarque explique les résultats reportés dans la figure 3.9. Ils donnent, en fonction du nombre de dimensions considérées, le pourcentage d'espace occupé par la représentation concise par rapport au stockage classique d'un cube de données. Quand le nombre de dimensions est égal à 10, la représentation par cube partition ne nécessite que 27,2% de l'espace de la représentation classique et seulement 14,2% pour 20 dimensions.

3.5 Conclusion

Aborder la problématique du calcul et du stockage des cubes de données est un challenge car le calcul requiert un temps d'exécution coûteux et un large espace de mémoire centrale. De plus, le volume énorme des résultats générés nécessite un très large espace de stockage sur disque. Dans ce chapitre, nous avons décrit trois approches de représentation des cubes de données qui ont en commun de réduire le volume des données sans aucune perte d'information. Les deux premières, le cube fermé et le cube quotient, sont définies dans le cadre du treillis cube, un espace de recherche parfaitement adapté aux données multidimensionnelles. Nous avons longuement décrit cette structure et ses propriétés car nos contributions sur le Cube Émergent s'inscrivent également dans ce cadre de travail. Le cube fermé est la représentation la plus concise d'un cube de données. Il exploite les redondances généralement incluses dans un cube de données. En les éliminant, il permet de réduire la taille nécessaire sans perdre de données utiles. Lorsque les données sont très corrélées, la représentation par cube fermé montre tout son intérêt car la

TABLE 3.9 – pourcentage de l'espace de stockage pour PCUBE vs. représentation classique

$\|Dim\|$	1	2	3	4	5	6
% de l'espace du cube partition vs. stockage classique	150%	100%	75%	60%	50%	42.8%
$\|Dim\|$	7	9	10	12	20	25
% de l'espace du cube partition vs. stockage classique	37.5%	33.3%	27.2%	23%	14.2%	11.5%

réduction apportée est réellement significative. Ces fortes corrélations apparaissent dans les jeux de données réelles et s'accroissent lorsque les données des dimensions sont très éparpillées sur des domaines très vastes. De telles caractéristiques sont typiques des entrepôts de données (Ross et Srivastava, 1997; Beyer et Ramakrishnan, 1999). Le cube quotient adopte une stratégie similaire quant à la redondance. La réduction apportée à la taille des cubes est un peu moins importante que pour le cube fermé mais le cube quotient a un avantage dont ne dispose pas le cube fermé : il préserve les possibiltés de naviguer entre les différents niveaux de granularité des données. La troisième approche présentée, le cube partition, est une méthode alternative qui permet aussi une réduction de l'espace de stockage, néanmoins moins importante que ne le font les deux approches précédentes. Elle ne stocke les valeurs des dimensions que dans la relation originelle et pas dans le data cube, ce qui permet de limiter la taille de ce dernier. Ainsi, dans les cas les plus défavorables aux deux précédentes représentations, quand les données sont très faiblement corrélées, le cube partition reste toujours une représentation réduite pour peu que le nombre de dimensions soit supérieur à deux, ce qui est un scénario parfaitement plausible dans la gestion des entrepôts de données.

4

Analyse multicritère dans les bases de données

Sommaire

4.1 Introduction

Dans un contexte décisionnel, certaines requêtes ne renvoient aucun résultat. Dans ces requêtes, l'utilisateur recherche les tuples pour lesquels les valeurs de certains critères sont optimales. C'est le caractère « multicritère » de ces interrogations qui les rend généralement infructueuses. En effet, tel tuple peut être optimal pour un critère mais pas pour un autre, il est alors éliminé du résultat alors qu'il aurait pu être pertinent pour l'utilisateur. Par exemple, si l'on considère une base de données immobilières, la recherche du logement « idéal » peut combiner des conditions sur le prix, le plus bas possible, la surface, la plus grande possible, et l'éloignement du lieu de travail, le plus réduit possible. Évidemment il est vraisemblable que ce logement idéal n'existe pas, d'où l'absence de réponse à ce type de requête. Pourtant certains logements pourraient s'avérer pertinents pour l'utilisateur parce que, situés dans une zone proche, mais non voisine, ils réunissent les critères de surface maximale et de prix minimal.

Afin d'apporter une réponse adéquate au type de requêtes décrites, l'opérateur Skyline a été introduit. Il considère l'ensemble des critères de choix d'une recherche comme autant de préférences et extrait les tuples globalement optimaux pour cet ensemble de préférences. Ainsi plutôt que de rechercher une hypothétique solution idéale, il extrait les candidats les plus proches possibles des souhaits de l'utilisateur. Son principe général s'appuie sur la notion de dominance. Un objet ou un tuple est dit dominé par un autre si, pour tous les critères intéressant le décideur,

il est moins optimal que cet autre. Un tel tuple est éliminé du résultat, non pas parce qu'il est non pertinent pour un des critères mais parce qu'il est non optimal selon la combinaison de tous les critères. En d'autres termes, il existe au moins une meilleure solution pour l'utilisateur qui, elle, sera retenue.

De la même manière que le cube de données permet de comprendre les liens existant entre plusieurs niveaux d'agrégation, une généralisation multidimensionnelle du SKYLINE a été proposée à travers le SKYCUBE (Pei *et al.*, 2006). Cette structure réunit tous les SKYLINES possibles suivant une combinaison de critères. Il est alors possible de rechercher efficacement des objets dominants selon différentes combinaisons de critères. De plus, grâce à cette structure, il devient possible d'observer le comportement des objets dominants à travers l'espace multidimensionnel et ainsi d'analyser et de comprendre les différents facteurs de dominance. Ce concept étant inspiré du cube de données, il souffre des mêmes inconvénients de coût de calcul et d'explosion de l'espace de stockage. Ainsi il est naturel, comme pour le data cube, d'essayer d'en proposer des représentations réduites et les algorithmes associés.

Dans ce chapitre, nous présentons d'abord l'opérateur SKYLINE ainsi que la problématique à laquelle il répond. Ensuite nous nous intéressons aux algorithmes de calcul du SKYLINE. Nous avons distingué deux familles d'algorithmes selon qu'ils utilisent ou non des techniques d'indexation. Dans un deuxième temps, nous présentons l'analyse multidimensionnelle des SKYLINES à travers le concept de SKYCUBE. Nous en décrivons une représentation réduite ainsi qu'un algorithme de calcul.

4.2 L'opérateur Skyline

Avant définir de manière formelle l'opérateur SKYLINE, il est important de bien situer le contexte dans lequel la problématique se pose. En effet, l'opérateur ne s'applique pas à n'importe quelle relation. Pour qu'il puisse effectuer les comparaisons nécessaires, il faut que les différents domaines sur lesquels sont définis les attributs, critères de choix de l'utilisateur, soient totalement ordonnés. Par simplicité, dans la suite du document, les attributs seront systématiquement codés sous forme de valeurs numériques.

Les tuples de nos relations pouvant être utilisées par l'opérateur SKYLINE sont de la forme $t = (a_1,a_2,\ldots,a_k,c_1,c_2,\ldots,c_l)$ où les a_i sont les attributs inutiles pour le SKYLINE alors que les c_i sont les critères sur lesquels l'utilisateur se fonde pour porter son choix.

Exemple 4.1 - La relation illustrée par la table 4.1 est typique pour l'utilisation du SKYLINE. Elle répertorie différents logements. Les attributs classiques sont ici PROPRIÉTAIRE et VILLE et les critères de choix pour trouver le « meilleur logement » sont :

- le PRIX de vente en euros,
- l'ÉLOIGNEMENT par rapport au lieu de travail en kilomètres,
- la CONSOMMATION Énergétique en kilowattheures par an et par mètre carré,
- le nombre de VOISINS.

Pour des raisons de clarté, la table est divisée en trois parties. La première contient le *RowId*. C'est un attribut implicite ne faisant pas partie du schéma de la relation. Sa valeur sert d'identifiant unique à chaque tuple. Nous notons t_i le tuple ayant i pour *RowId*. La deuxième contient toutes les informations qui ne n'entrent pas en compte dans les critères de préférence de l'utilisateur. Il n'y a pas de restriction particulière sur leur domaine. Enfin, la troisième partie regroupe les attributs préférences qui vont être pris en considération pour le calcul du SKYLINE.

De plus, il est important de préciser que nous souhaitons que le calcul du SKYLINE reste le plus

TABLE 4.1 – La relation LOGEMENTS

RowId	Propriétaire	...	Ville	Prix	Éloignement	Consommation	Voisins
1	Dupont		Marseille	220000	15	275	5
2	Dupond		Aix-en-Provence	100000	15	85	1
3	Martin		Marseille	220000	7	180	1
4	Sanchez		Aubagne	340000	7	85	3
5	LIF		Aix-en-Provence	100000	7	180	1

général possible. Nous ne prenons pas en compte certains cas particuliers qui sont exploitables pour calculer efficacement le SKYLINE. Il s'agit par exemple de calculs sur des attributs critères dont les domaines ont tous de faibles cardinalités (Morse *et al.*, 2007) ou bien de SKYLINE portant sur une ou deux dimensions, cas traités dans (Börzsönyi *et al.*, 2001). Nous ne considérerons pas non plus que l'entrée, les tuples de la relation qui nous intéressent, est toujours assez petite pour entrer en mémoire centrale comme dans (Preparata et Shamos, 1985; Kung *et al.*, 1975).

Le travail présenté dans ce paragraphe porte sur l'opérateur SKYLINE, les conditions dans lesquelles il peut être appliqué et les algorithmes de mise en œuvre. Une typologie de ces algorithmes est dressée et des critères d'évaluation examinés. Pour ces critères, nous avons retenu :

- la progressivité qui vise à rendre accessibles les résultats le plus rapidement possible afin de conforter l'utilisateur dans ses choix.
- le caractère universel de l'algorithme qui ne doit pas tirer profit de telle ou telle spécificité.
- les limitations techniques, en particulier le passage à l'échelle.

Nous avons identifié deux familles d'algorithmes, avec ou sans méthode d'indexation, implémentant l'opérateur SKYLINE. Nous avons étudié les algorithmes les plus pertinents par rapport aux critères d'évaluation retenus et, le cas échéant, les structures de données sous-jacentes.

4.2.1 Définition du problème

Nous avons décrit les conditions d'application de l'opérateur SKYLINE, nous en donnons à présent une définition formelle et la solution de calcul qui en découle via une requête SQL.

Définition 4.1 (Relation de dominance) **-** Soit $C = \{c_1, c_2, \ldots, c_d\}$ l'ensemble des critères sur lesquels va porter l'opérateur SKYLINE. Sans perte de généralité, nous considérons qu'ils doivent tous être minimisés. Soit deux tuples t et u à comparer, la relation de dominance suivant l'ensemble de critères C est définie comme suit :

$$t \succeq_C u \;\Leftrightarrow\; t.c_1 \leq u.c_1 \; et \; t.c_2 \leq u.c_2 \; et \; \ldots \; et \; t.c_d \leq u.dc_d$$

Lorsque $t \succeq_C u \; et \; \exists \, t.c_i \neq u.c_i$ on parle d'une relation de dominance stricte, notée $t \succ_C u$.

Exemple 4.2 - Avec notre relation exemple (*cf.* Table 4.1), en supposant que l'utilisateur souhaite trouver les meilleurs logements suivant les critères suivant $C = \{Eloignement, Prix\}$, la table 4.2 donne la projection de cette relation selon l'ensemble d'attributs C.

Le tuple t_1 est dominé par le tuple t_2 ($t_2 \succ_C t_1$) car ils ont le même éloignement mais que $t_2.Prix < t_1.Prix$. Pour l'utilisateur le tuple t_2 répond mieux à ses souhaits que le tuple t_1, il n'est donc pas intéréssant (vis à vis des préférences formulées) de conserver le tuple t_1. Pour les tuples t_2 et t_3, on a $t_2 \succeq_C t_3$ et $t_3 \succeq_C t_2$, ils sont donc tous les deux incomparables par rapport à la relation de dominance $\succeq_C$.

TABLE 4.2 – Projection de la relation LOGEMENTS selon les critères $C = \{Eloignement, Prix\}$

RowId	Prix	Éloignement
1	220000	15
2	100000	15
3	220000	7
4	340000	7
5	100000	7

Lorsqu'un tuple t domine un tuple u ($i.e$ $t \succeq_C u$), cela signifie que t est équivalent ou « meilleur » que le tuple u pour tous les critères choisis. Comme nous considérons que les critères sont minimisés, les valeurs de t pour tous les critères sont inférieures ou égales à celles de u. Ainsi dans le cadre d'une recherche multicritère les tuples dominés par d'autres (au moins un) ne sont pas pertinents et sont éliminés du résultat par l'opérateur SKYLINE que nous définissons ci-après.

Définition 4.2 (L'opérateur SKYLINE) **-** À partir d'une relation r, l'opérateur SKYLINE renvoie l'ensemble des tuples qui ne sont dominés par aucun autre, suivant l'ensemble de critères C :

$$SKY_C(r) = \{t \in r \mid \nexists\, u \in r \setminus t,\ u \succ_C t\}$$

Exemple 4.3 - Avec notre relation exemple (*cf.* Table 4.1) et les critères suivants $C = \{Eloignement, Prix\}$, $SKY_C(\text{LOGEMENT}) = \{t_5\}$ car le tuple t_5 domine tous les autres. Il est donc le meilleur logement possible pour l'utilisateur.

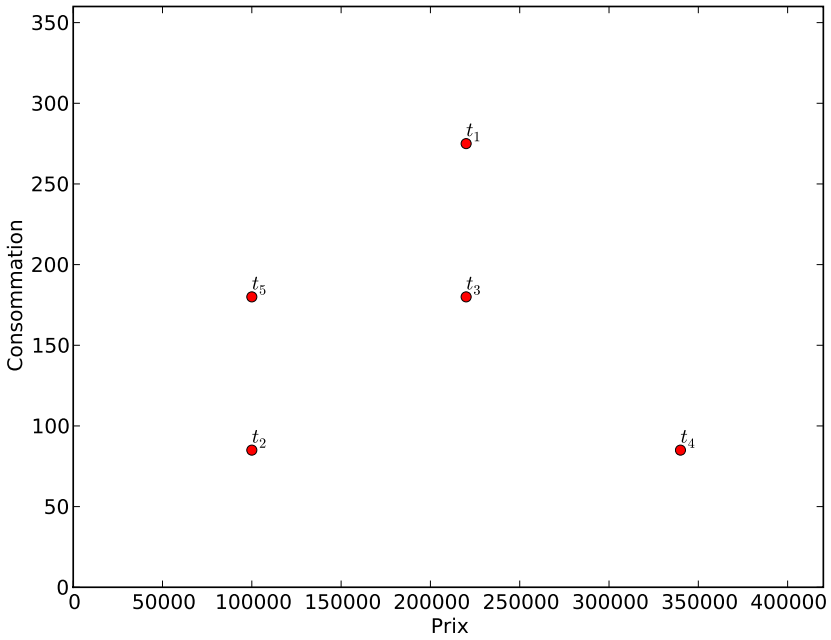
Si les tuples sont considérés comme des points dans un espace d-dimensionnel, leurs coordonnées sont alors les valeurs respectives des attributs critères du SKYLINE. Le problème se ramène alors à celui du *Maximal Vector Computation* (Preparata et Shamos, 1985; Kung *et al.*, 1975).

Exemple 4.4 - La figure 4.1 représente la projection de la relation exemple (*cf.* Table 4.1) sur les critères $C = \{Consommation, Prix\}$.

En traduisant directement la définition précédente en SQL, nous pouvons calculer simplement le résultat de l'opérateur SKYLINE. Cette requête doit réaliser les opérations suivantes :

1. comparer chaque tuple de r avec tous les autres. Une auto-jointure est donc nécessaire avec une imbrication d'un bloc corrélé ;
2. chaque tuple examiné t doit être écarté du résultat s'il existe un autre tuple u tel que $u \succ t$ selon les critères définis par l'utilisateur.

FIGURE 4.1 – Représentation géométrique de la projection de la relation LOGEMENT



La requête est de la forme suivante (remarquons que la dernière ligne permet d'obtenir une dominance stricte) :

```
SELECT *
FROM   R t
WHERE NOT EXISTS (SELECT *
                  FROM   R u
                  WHERE u.c1 <= t.c1
                  AND    ...
                  AND   u.cd <= t.cd
                  AND   (u.c1 < t.c1 OR ... OR u.cd < t.cd));
```

Nous disposons donc d'un moyen simple de calculer le SKYLINE. Il faut maintenant évaluer son efficacité en examinant la complexité de la requête. Chaque tuple de la relation est comparé avec tous les autres. En considérant que les comparaisons s'effectuent en temps constant, la complexité de la requête est en $O(n^2)$ (avec n le nombre de tuples de la relation). Une telle complexité est généralement acceptable lorsqu'on travaille en mémoire centrale. Dans le contexte des bases de données, le nombre de tuples est généralement très important et en pratique, la relation complète ne tient pas dans la mémoire centrale. Cela a une conséquence immédiate très importante : le temps d'exécution se retrouve considérablement décuplé car les temps d'accès disques sont incomparablement plus longs que les accès mémoire. Cette requête, viable en mémoire centrale, devient tout simplement inexploitable en pratique comme l'illustre l'exemple suivant.

Exemple 4.5 -

- **Scénario**
  - La taille d'une page disque est de $8KB$.
  - La taille de la base de données est de $1GB$, soit 131072 pages disques.
  - Chaque E/S disque prend $1ms$.
- **Complexité linéaire** $O(n)$
  Le temps de lecture total est de $131s \approx 2min$.
  $\Rightarrow$ pas très performant.
- **Complexité quadratique** $O(n^2)$
  Le temps de lecture total est de $17179869s \approx 200jours$.
  $\Rightarrow$ totalement inexploitable.

Ces chiffres montrent la nécessité d'utiliser des algorithmes spécifiquement conçus pour des données à traiter en mémoire secondaire, ils ont besoin de tenir compte des temps d'accès disques et par conséquent doivent limiter le plus possible le nombre d'E/S coûteuses.

4.2.2 Les algorithmes de calcul

Dans ce paragraphe, nous étudions des algorithmes de calcul de SKYLINE, mais au préalable, nous nous intéressons aux critères que nous avons utilisés pour évaluer et comparer ces algorithmes.

Les critères d'évaluation

Avant de présenter différents algorithmes, il nous faut pouvoir les comparer sur des critères identiques afin de mettre en évidence leurs forces et leurs faiblesses. Kossmann *et al.* (2002) en proposent une liste sur laquelle nous nous appuyons sans la suivre strictement car certains points sont contradictoires et d'autres peu pertinents dans le cadre fixé. Voici la liste des critères choisis :

- **Universalité**
  L'algorithme doit utiliser des structures standard facilement intégrables au cœur d'un SGBD.
- **Limitations**
  Ce critère a trait à l'efficacité mais surtout au passage à l'échelle des algorithmes. Il s'agit de donner les limites, par exemple au nombre de critères du SKYLINE et/ou à la taille de l'entrée, au delà desquelles l'algorithme n'est plus utilisable.
- **Progressivité**
  Les volumes des bases de données sont souvent énormes. Un simple parcours de la base peut prendre un temps non négligeable. C'est pourquoi nous souhaitons que nos algorithmes calculent et affichent les résultats trouvés le plus rapidement possible, au fur et à mesure qu'ils sont découverts. Idéalement, les premiers résultats devraient être envoyés quasiment instantanément.

Les algorithmes qui vont être détaillés maintenant peuvent se classer en deux catégories. Les premiers n'utilisent pas de structure d'indexation particulière et n'ont donc pas besoin d'un *pré-traitement* pour fonctionner. Les seconds regroupent tous ceux qui sont basés sur des index. Ils les exploitent pour améliorer l'efficacité du calcul et modifier leur comportement, vis-à-vis par exemple de leur progressivité. La contrepartie est toutefois qu'il faut disposer de l'index nécessaire, ou bien le cas échéant de devoir le calculer, avant de pouvoir d'utiliser ces algorithmes.

Les algorithmes sans index

L'algorithme Block-Nested-Loop « *BNL* » L'idée directrice de cet algorithme (Börzsönyi *et al.*, 2001), extension de l'algorithme *Nested-Loops* (Grust *et al.*, 1997; Berchtold *et al.*, 1997), est de limiter le nombre d'E/S effectuées en chargeant en mémoire un ensemble de tuples candidats au Skyline appelé « fenêtre ». Les accès disque sont faits par bloc, les comparaisons ont lieu directement en mémoire centrale et sont donc bien plus rapides. Cependant, cette fenêtre a une taille limitée. De plus, comme la taille du Skyline est du même ordre de grandeur que la relation d'entrée, il est vraisemblable que la fenêtre ne puisse pas accueillir tous les candidats. C'est pour cela que l'algorithme gère, en plus de la fenêtre, un fichier temporaire stocké en mémoire secondaire dans lequel sont écrits tous les candidats non considérés faute de place. Ils seront traités lors d'une prochaine itération.

Examinons les différents cas de figure qui se présentent lors de la comparaison d'un tuple t avec ceux de la fenêtre W.

1. t **est dominé par un tuple de la fenêtre**
   t est alors directement écarté et ne sera plus pris en compte pour le reste du calcul : il est dominé et ne fera donc jamais partie du Skyline. On élimine t dès la première occurrence de ce cas et il n'est pas nécessaire de poursuivre la comparaison avec les autres tuples de la fenêtre.
2. t **domine un ou plusieurs tuples de la fenêtre**
   Les tuples de la fenêtre dominés par t sont écartés et ne seront plus pris en compte pour les itérations futures. t est inséré dans la fenêtre sans problème, puisqu'il y a au moins une place libre.
3. t **est incomparable avec l'ensemble des tuples de la fenêtre**
   C'est le cas le plus problématique : t doit être ajouté à la fenêtre mais il n'a éliminé aucun tuple sur son passage et il se peut que celle-ci soit pleine. Si ce n'est pas le cas, t est ajouté normalement à la fenêtre. Sinon, t est mis de côté dans le fichier temporaire et sera examiné à nouveau au cours de la prochaine itération de l'algorithme.

Exemple 4.6 - Nous allons illustrer avec notre relation (*cf.* Table 4.1) exemple les trois cas de figure présentés ci-dessus. Nous considérons que les préférences utilisateur sont $C = \{Eloignement, Prix, Consommation\}$. Soit W la fenêtre des candidats. Elle a une capacité de deux éléments. Dans chacun des cas, on montre l'étape principale de l'algorithme *BNL* pour un tuple t.

1. t **est dominé par un tuple de la fenêtre**

$$t = (1, 220000, 15, 275)$$

RowId	Prix	Élgnt	Conso
2	100000	15	85
3	220000	7	180

$\Longrightarrow$

RowId	Prix	Élgnt	Conso
2	100000	15	85
3	220000	7	180

   Le tuple t est comparé à $(2, 100000, 15, 85)$, qui le domine : ce tuple ne fait donc pas partie du Skyline. Il est éliminé. La fenêtre reste inchangée.

2. t **domine un ou plusieurs tuples de la fenêtre**

$$t = (5, 100000, 7, 180)$$

RowId	Prix	Élgnt	Conso
2	100000	15	85
3	220000	7	180

$\Longrightarrow$

RowId	Prix	Élgnt	Conso
2	100000	15	85
5	100000	7	180

Le tuple t est comparé à $(2, 100000, 15, 85)$. Il n'est pas dominé. On continue avec le tuple suivant $(3, 220000, 7, 180)$, qui lui est dominé. Il est supprimé de la fenêtre et on insère t à sa place.

3. **t est incomparable avec l'ensemble des tuples de la fenêtre**

$$t = (4, 340000, 7, 85)$$

RowId	Prix	Élgnt	Conso
2	100000	15	85
5	100000	7	180

$\Longrightarrow$

RowId	Prix	Élgnt	Conso
2	100000	15	85
5	100000	7	180

$t = (4, 340000, 7, 85)$

Le tuple p est incomparable avec tous les tuples de la fenêtre. Il reste un candidat potentiel mais il ne peut pas y être inséré. On l'ajoute dans le fichier temporaire.

À la fin de chaque itération, on peut extraire les tuples candidats qui ont été comparés à tous les autres, y compris ceux du fichier temporaire : ils ne sont dominés par aucun autre et font donc partie du SKYLINE. Il n'est pas nécessaire de les garder dans la fenêtre plus longtemps. Celle-ci ne contient donc que les tuples réellement candidats, c'est à dire dont on ne peut affirmer qu'il font partie du SKYLINE. Pour savoir facilement quels tuples ont déjà été comparés entre eux, un système d'estampillage est utilisé : chaque tuple étiqueté t a été comparé avec tous les tuples d'étiquette inférieure à t. Ceci permet de garantir que l'algorithme se termine toujours et que deux tuples ne seront jamais comparés ensemble plusieurs fois.

Cet algorithme fonctionne particulièrement bien si le SKYLINE est petit et qu'il n'y a pas beaucoup de candidats à la fois : le fichier temporaire n'est alors pas utilisé puisqu'il y a toujours de la place dans la fenêtre. L'algorithme se termine ainsi après une seule itération, avec sa meilleure complexité $O(n)$, avec n le nombre de tuples de l'entrée à examiner. Inversement, dans le pire des cas, sa complexité est quadratique $O(n^2)$. Il reste cependant toujours bien plus efficace que la requête SQL puisqu'il limite considérablement les E/S, la fenêtre résidant en mémoire centrale.

Les critères d'évaluation pour cet algorithme sont donnés dans le tableau suivant :

Critère	**Satisfaisant ?**	**Commentaires**
Progressivité	NON	Doit d'abord finir une itération complète
Universalité	OUI	N'a besoin d'aucun tri ou index préalable
Préférences	NON	Compare strictement selon la définition de la dominance
Limitations	OUI	Fonctionne bien si la mémoire disponible est grande

Deux variantes de cet algorithme ont été proposées par Chomicki *et al.* (2005), pour améliorer ses performances. Elles utilisent les techniques suivante :

– **Réorganisation dynamique de la fenêtre**
La majorité du temps de l'algorithme est passée à comparer un tuple t avec ceux de la fenêtre. Il est intéressant d'essayer de réduire ce temps moyen avec une heuristique très

simple : quand un tuple t domine un autre, il est placé en haut de la fenêtre. C'est donc lui qui sera comparé en premier lors des prochains tests de dominance. Si la distribution est corrélée, cette heuristique est particulièrement efficace puisqu'il y aura des tuples « tueurs » qui vont dès la première comparaison éliminer un très grand nombre de tuples. De plus, cette heuristique à un coût quasi nul, car l'opération s'effectue en temps constant si la fenêtre est une liste.

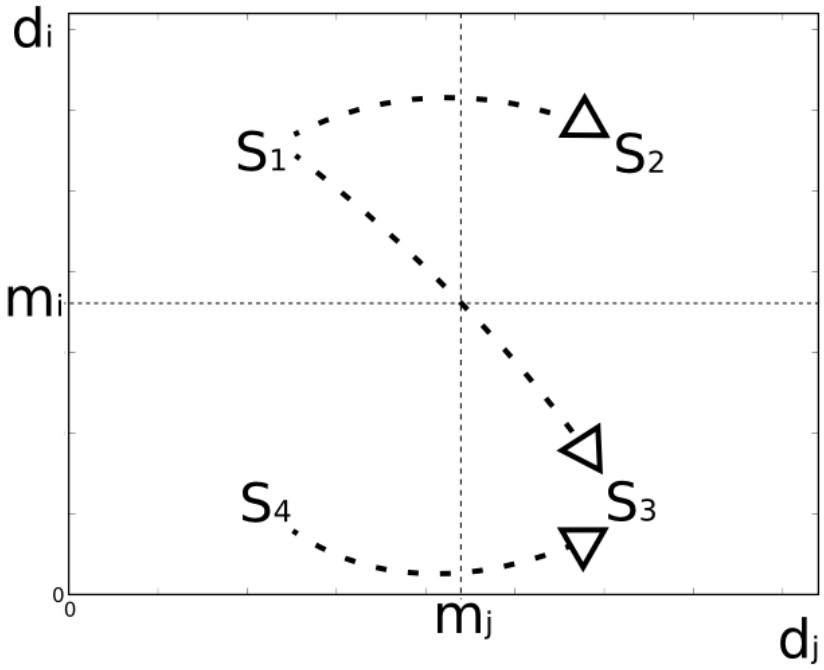
– **Pré-tri de la relation**
L'idée ici est de trier l'entrée selon une fonction de score (monotone) correspondant aux attentes de l'utilisateur. Les tuples candidats sont insérés dans la liste selon cet ordre car il est plus probable qu'un tuple avec un « bon » score domine un grand nombre d'éléments. Grâce à cette heuristique, on élimine donc plus vite les tuples dominés. De plus, *Sort First* Skyline (Chomicki *et al.*, 2005) se comporte de manière plus progressive que son algorithme de base puisque le tri permet de s'assurer qu'un tuple t dominant un autre tuple t' sera examiné avant et on peut donc directement ajouter les points insérés dans la liste en tant qu'éléments du Skyline final.

L'algorithme Divide-and-Conquer « *DC* » Pour résoudre le problème du « *Maximal Vector Computation* » un algorithme efficace a été proposé par Preparata et Shamos (1985); Kung *et al.* (1975). Comme nous l'avons souligné, ce problème est équivalent au calcul du Skyline mais est défini dans un contexte extrêmement différent de celui des bases de données. Börzsönyi *et al.* (2001) en ont proposé une adaptation qui prend en compte les contraintes associées à ce nouveau contexte. Dans un premier temps nous présentons l'algorithme originel puis nous examinons les améliorations qui lui ont été apportées.

Pour comprendre les idées sous-tendant cet algorithme, il faut se représenter le problème de l'opérateur Skyline de manière géométrique. Les tuples à examiner sont alors considérés comme des points, dont les coordonnées sont les valeurs numériques des attributs-critères du Skyline. Le principe général de cette méthode est de travailler de manière récursive en divisant l'espace en des sous-espaces plus petits jusqu'à rendre le calcul du Skyline évident. Puis, il faut reconstruire le résultat final en réunissant tous les Skylines partiels. Voici les trois étapes principales de l'algorithme.

1. **Calculer la médiane**
La médiane m_i est calculée pour l'entrée par rapport à la dimension d_i. Elle est utilisée pour diviser l'entrée en deux partitions, P_1 dans laquelle se trouvent tous les tuples t pour lesquels $t.d_i$ est meilleur que m_i et P_2 contient les tuples restants.
2. **Calculer les Skylines**
C'est la partie récursive de l'algorithme : les Skylines SKY_1 et SKY_2 des partitions P_1 et P_2 sont calculés. L'espace est partitionné de manière récursive jusqu'à ce que chaque partition ne contienne plus que très peu de tuples. Calculer le Skyline devient alors simple.
3. **Fusionner les Skylines**
Le Skyline résultant de la fusion de SKY_1 et SKY_2 est calculé. Remarquons qu'aucun des tuples de SKY_1 ne peut être dominé par un tuple de SKY_2 pour la dimension d_i, puisque la médiane m_i a été utilisée pour diviser l'espace. De plus, une autre propriété géométrique du Skyline peut être exploitée en divisant SKY_1 et SKY_2 à l'aide d'une médiane m_j pour une autre dimension $d_j \neq d_i$. Il en résulte alors quatre partitions $SKY_{1,1}$, $SKY_{1,2}$, $SKY_{2,1}$, $SKY_{2,2}$. L'avantage est que nous n'avons pas à essayer de fusionner $SKY_{1,2}$ et $SKY_{2,1}$ car leurs tuples sont forcément incomparables. Il faut maintenant fusionner les paires restantes en procédant de manière récursive.

FIGURE 4.2 – Principe de fusion des SKYLINES partiels dans l'algorithme *DC*



La figure 4.2 illustre la manière de découper et parcourir l'espace utilisé par l'algorithme *DC*. Les flèches indiquent comment l'algorithme effectue les fusions entre les résultats partiels. Examinons l'algorithme selon les critères que nous avons définis.

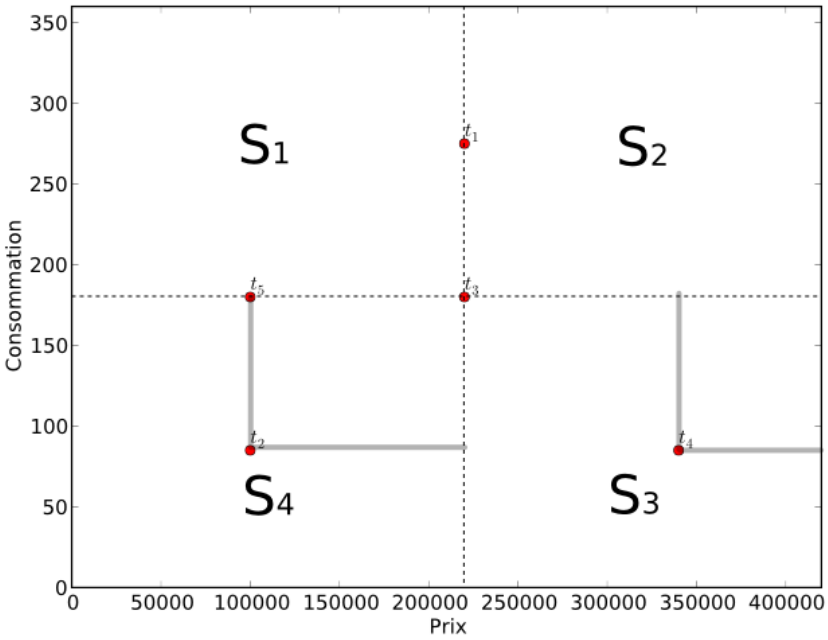
Critère	Satisfaisant ?	Commentaires
Progressivité	NON	Doit d'abord partitionner l'espace récursivement
Universalité	OUI	N'a besoin d'aucun index ou tri
Préférences	NON	Aucune possibilité d'intégration
Limitations	OUI	Fonctionne bien si la mémoire disponible est grande

La complexité de cet algorithme (Preparata et Shamos, 1985; Kung *et al.*, 1975) est de l'ordre de $\Theta(n \cdot (\log n)^{d-2}) + O(n \cdot \log n)$ (avec d le nombre de dimensions sur lesquelles porte le SKYLINE et n le nombre de tuples en entrée). Ce résultat est meilleur que la complexité en $O(n^2)$ de la requête SQL et de *BNL*. Cet algorithme se comporte donc en général bien mieux que *BNL* lorsque que le résultat est de grande taille et qu'il ne tient pas en mémoire.

Exemple 4.7 - La figure 4.3 illustre le fonctionnement de l'algorithme *DC*. Une fois que tous les SKYLINES partiels sont calculés, l'algorithme effectue leur fusion. Les SKYLINES S_1 et S_2 ne contiennent aucun tuple, nous n'avons pas besoin de les considérer pour la fusion. Nous avons donc $SKY_{\{Prix,Consommation\}} = fusion(S_3,S_4) = \{t_2\}$

Une adaptation de l'algorithme dédiée au domaine des bases de donnée a été proposée par Börzsönyi *et al.* (2001). Cette méthode à l'origine purement géométrique est transformée en

FIGURE 4.3 – Illustration de l'algorithme *DC* sur notre relation exemple



une méthode dédiée aux bases de données en prenant en compte les contraintes déjà évoquées (*cf.* paragraphe 4.2). Pour ce faire, les auteurs apportent les modifications suivantes :

- **Partitionnement en α partitions**
  Dans l'algorithme *DC* originel, l'entrée est lue, partitionnée, écrite sur disque, lue à nouveau et ainsi de suite... Ceci est extrêmement coûteux en E/S et détériore grandement les performances de cet algorithme. Au lieu de diviser l'espace seulement en deux, il est divisé en une multitude d'espaces plus petits, selon des α-quantiles, de manière à assurer qu'ils rentrent en mémoire.
- **Fusion en mémoire centrale**
  Avec le partitionnement précédent, l'étape (3) de fusion devient plus délicate : il faut partitionner à nouveau en de nombreuses sous-partitions telles que leur fusion puisse être contenue en mémoire (c'est à dire que chaque sous-partition doit occuper au maximum la moitié de la mémoire restante).

Cet algorithme devient particulièrement intéressant par rapport à *BNL* lorsque le nombre de critères de recherche est élevé.

Les algorithmes avec index

Plusieurs types d'indexation ont été exploités pour mettre au point des algorithmes de calcul du SKYLINE. Nous allons brièvement présenter les deux grandes familles d'index :

- **Les index « classiques »**
  Il s'agit des index couramment utilisés dans les bases de données, tels que les B-Trees ou bien les index Bitmap. Deux algorithmes se basent sur ces index : le premier nommé *Index* et le second *Bitmap*, présentés dans Papadias *et al.* (2003). Ils ne sont pas étudiés dans ce

paragraphe car ils présentent de nombreux désavantages. Le plus important d'entre eux est que l'index pré-calculé est adapté pour une demande particulière de SKYLINE. Ainsi, si l'utilisateur souhaite formuler plusieurs requêtes SKYLINE, il sera nécessaire de calculer un index spécifique pour chacune d'elles.

– **Les index « spatiaux »**
Comme nous l'avons vu avec l'algorithme *Divide and Conquer*, appréhender le problème du calcul du SKYLINE dans sous son aspect géométrique permet d'exploiter certaines propriétés pour optimiser le temps de réponse des algorithmes. Développés dans le contexte des bases de données géographiques, les index spatiaux, permettent d'exploiter pleinement certaines de ces propriétés.

Nous allons dans ce paragraphe étudier les index spatiaux les plus utilisés pour le calcul du SKYLINE : les R-Trees (Guttman, 1984).

Les R-Trees Il s'agit d'une structures de données conçue pour indexer spatialement des objets (dans notre cas les tuples de la relation). Cette structure est particulièrement performante lorsque le nombre de dimensions n'est pas trop important, c'est typiquement le cas pour les requêtes SKYLINE. En effet si le nombre de critères est trop important, le nombre de résultats de l'opérateur devenant très grand, l'interprétation de ces résultats devient plus délicate. Si l'utilisateur souhaite malgré cela prendre en compte un grand nombre de critère, d'autres structures d'indexation seront plus adaptées, comme par exemple les X-Trees (Berchtold *et al.*, 1996), les SH-Trees (Dang, 2006), ...

Les R-Trees (pour Rectangle-Trees) se basent sur les B-Trees en adaptant leur principe au contexte des bases de données géographiques. Leur nom vient de la manière dont les objets sont représentés à l'intérieur de la structure de données. Au lieu de conserver la forme (possiblement complexe) d'un objet, on utilise son MBR (Minimum Bounding Rectangle). C'est le plus petit hyper-rectangle dont les côtés sont parallèles aux axes du repère, contenant l'objet. Toutes les opérations de recherche se basent ainsi uniquement sur les MBR ce qui uniformise, simplifie et accélère les calculs.

Exemple 4.8 - La figure 4.4 illustre ce concept pour la projection sur les attributs Prix et Consommation de notre relation exemple (*cf.* Table 4.1).

Un R-Tree d'ordre (m,M) possède les caractéristiques suivantes :

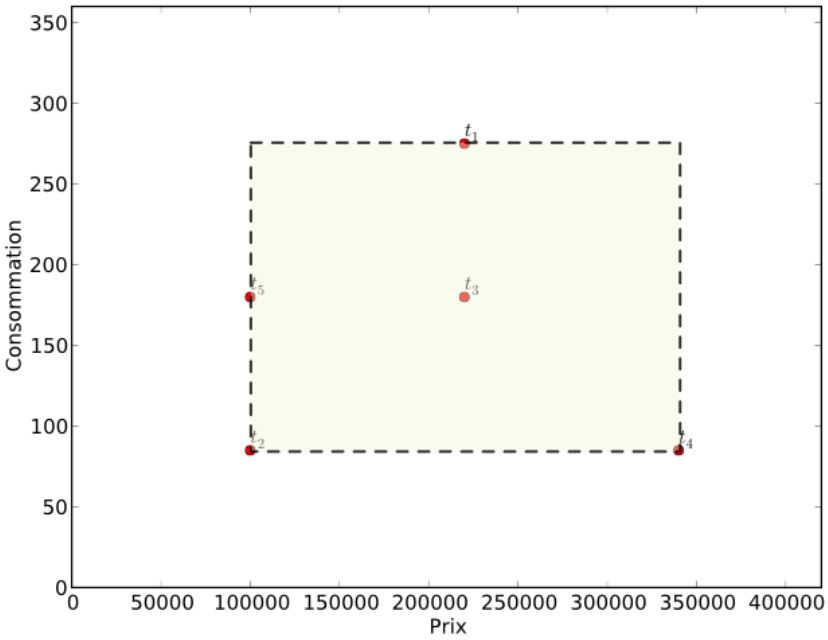
– il dispose de deux types de nœuds. Les nœuds internes contiennent des entrées de la forme *(MBR, idNœud)*. ils associent à un nœud-fils la région qu'il couvre, c'est à dire la plus petite zone qui contient tous les MBR de ses entrées. Les entrées des nœuds feuilles sont quant à elles de type *(MBR, rowID)*. Elles associent à un tuple de la relation la zone géographique qu'il représente.

– La racine de l'arbre contient au moins deux entrée, sauf si c'est une feuille (dans ce cas, elle peut contenir 0 ou 1 entrée).

– Les deux paramètres m et M sont respectivement le nombre minimum et maximum d'entrées contenues par nœud interne. On a $2 \leq m \leq \frac{M}{2}$ et généralement on choisit M tel que $M = \frac{taille(page)}{taille(entree)}$. Ce paramétrage permet de limiter au maximum les accès à la mémoire secondaire. En effet, en une lecture d'une page disque, on a accès à l'ensemble des informations contenues dans un nœud.

– Les R-Trees sont des arbres équilibrés, toutes les feuilles sont donc à la même profondeur p. Par construction cette profondeur respecte l'inégalité suivante : $\lceil \log_M(N-1) \rfloor \leq p \leq$

FIGURE 4.4 – MBR de la projection de la relation LOGEMENT



$\lceil \log_m(N-1) \rfloor$.

Exemple 4.9 - Pour pouvoir illustrer les R-Tree convenablement, nos exemples utilisent une nouvelle relation LOGEMENT' donnée dans la table 4.3. Cette relation a la même structure que la relation LOGEMENT mais elle contient plus de tuples. Un R-Tree ainsi que la représentation dans le plan de tous ses MBR sont donnés dans la figure 4.5. Les MBRs des entrées intermédiaires e_i sont notés N_i.

Intéressons nous maintenant au fonctionnement des R-Trees en décrivant les principaux algorithmes de manipulation associés à cette structure, c'est à dire ceux de recherche, d'insertion et de suppression.

1. **L'algorithme de recherche**
   L'énoncé de la requête est le suivant : soit un hyper-rectangle de recherche H et la racine de l'arbre A. On veut connaître tous les éléments stockés dont le MBR intersecte H. L'algorithme est classique pour les arbres de recherche : il est récursif et part de la racine pour aller jusqu'aux feuilles. Le parcours n'est effectué que losqu'il est pertinent : le MBR de l'entrée intersecte la zone cherchée. L'algorithme se termine toujours car les MBRs se réduisent au fur et à mesure de la descente dans l'arbre. CHERCHER est décrit dans l'algorithme 15.

   Exemple 4.10 - La figure 4.6 illustre le fonctionnement de l'algorithme CHERCHER pour le tuple t_3. Les entrées grisées sont celles examinées par l'algorithme.

   La recherche peut, dans le pire des cas, se faire en temps linéaire car rien n'interdit dans

TABLE 4.3 – La relation LOGEMENTS'

RowId	Prix	Éloignement	Consommation	Voisins
1	220000	15	275	5
2	100000	15	85	1
3	220000	7	180	1
4	340000	7	85	3
5	100000	7	180	1
6	80000	12	200	6
7	150000	11	120	2
8	60000	10	275	4
9	330000	3	260	1
10	350000	2	120	4
11	380000	4	70	2
12	70000	17	300	1
13	430000	3	40	5
14	330000	12	200	3

Algorithme 13 Algorithme CHERCHER

Entrée : L'arbre A, la zone H
Sortie : Q contenant les points intersectant H

```
1: si A n'est pas une feuille alors
2:    pour tout entrée E de A faire
3:       si E.mbr ∩ H alors
4:          CHERCHER(E.nœud)
5:       fin si
6:    fin pour
7: sinon
8:    pour tout entrée E de A faire
9:       si E.mbr ∩ H alors
10:         Ajouter E.tuple à Q
11:      fin si
12:   fin pour
13: fin si
```

FIGURE 4.5 – Un exemple de R-Tree

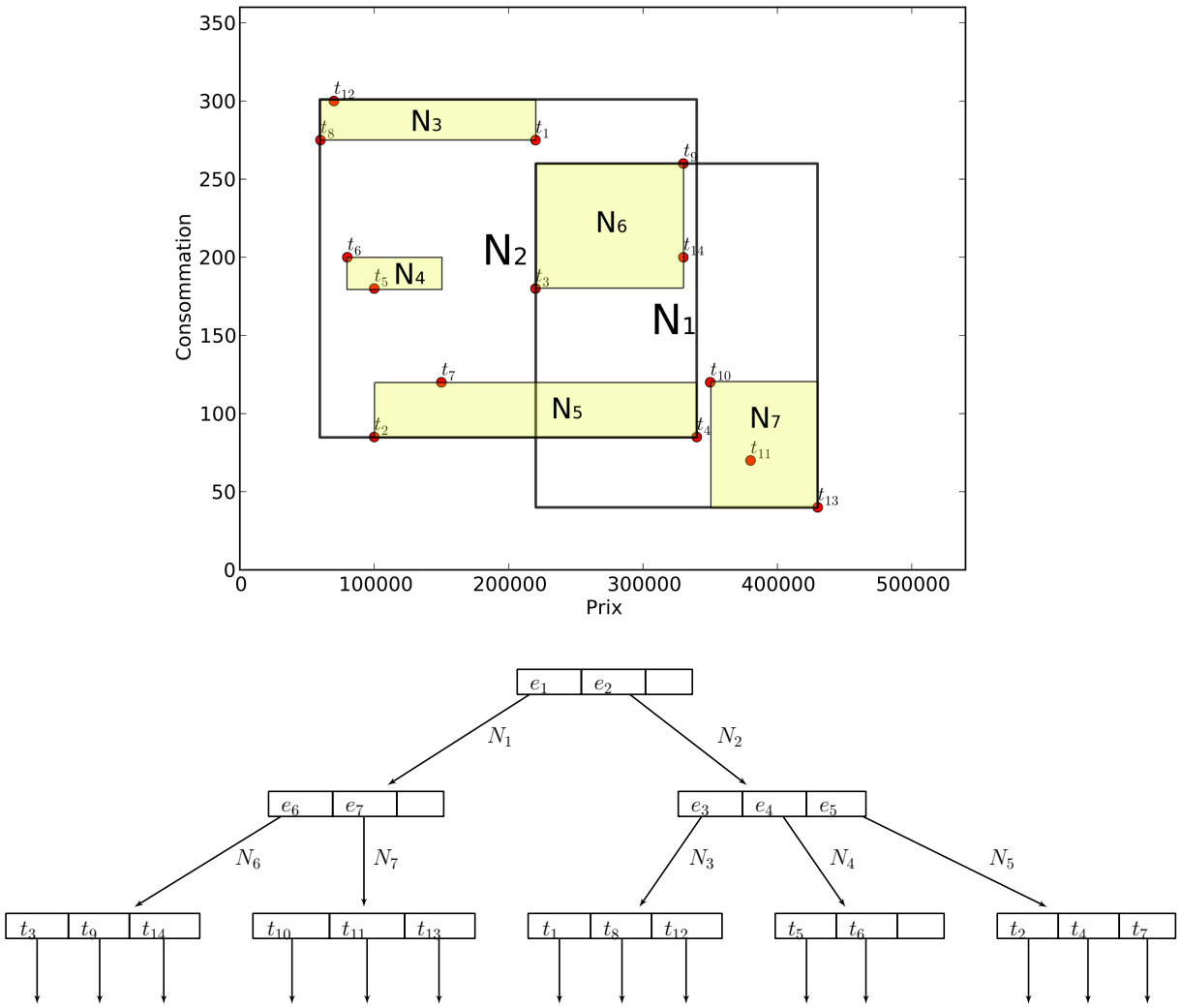
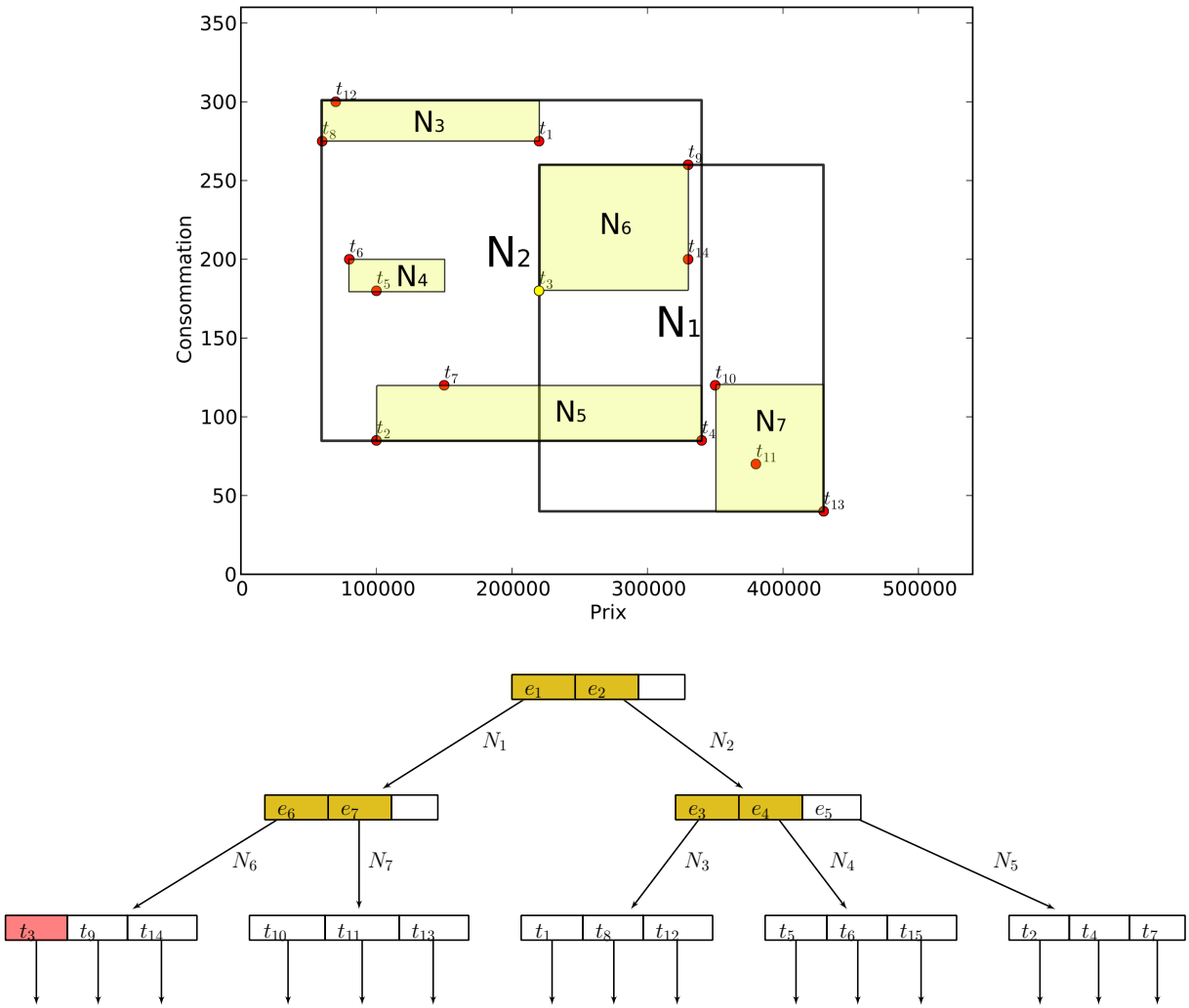


FIGURE 4.6 – Recherche du tuple t_3



les R-Trees d'avoir des MBRs qui se chevauchent. Si H n'intersecte pas plusieurs entrées, alors la complexité devient de l'ordre de $O(\log_m(N))$.

2. **L'algorithme d'insertion**
considérons l'insertion d'une entrée E dans l'arbre A. La procédure est similaire à l'insertion dans les B-Trees. Pour des raisons de lisibilité, l'algorithme est découpé en sous-parties ChoisirFeuille, DiviserNoeud et AjusterArbre.

Algorithme 14 Algorithme Inserer

Entrée : L'entrée E, l'arbre A
1: F = ChoisirFeuille(A, E) //Trouver où insérer
2: **si** F est pleine **alors**
3: F, FF = DiviserNoeud(F, E) //On obtient F et FF remplis correctement
4: **sinon**
5: Insérer E dans F
6: **fin si**
7: AjusterArbre(F, FF) //Les changements sont propagés vers le haut
8: **si** la racine R s'est divisée **alors**
9: Soit R_2 la nouvelle racine dont les fils sont les deux nœuds résultants //Faire grandir l'arbre
10: **fin si**

Pour la partie ChoisirFeuille, l'objectif est de trouver le meilleur emplacement pour accueillir E. On cherche à minimiser l'élargissement des MBRs des nœuds parents de la feuille. En effet, plus les MBRs sont grands plus ils ont des chances d'être sollicités lors de recherches et il s'agit d'éviter si possible de descendre dans l'arbre inutilement. L'algorithme ChoisirFeuille est décrit dans l'algorithme 15.

Algorithme 15 Algorithme ChoisirFeuille

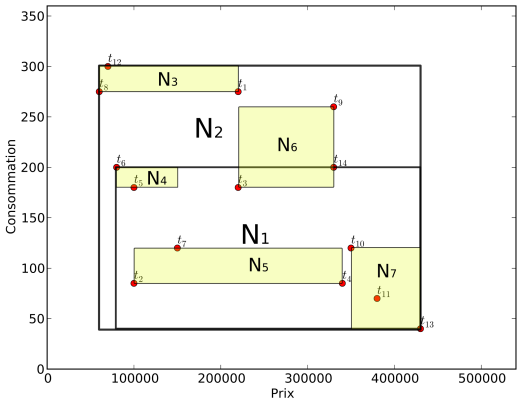
Entrée : Le nœud courant N, l'entrée E
Sortie : La feuille F où insérer
1: **boucle**
2: **si** N est une feuille **alors**
3: **retourner** N
4: **fin si**
5: Soit l'entrée S de N telle que l'élargissement de son MBR est minimal
6: $N = S.noeud$
7: **fin boucle**

La partie la plus importante de l'algorithme d'insertion concerne DiviserNœud. En effet, il s'agit de diviser l'espace couvert par le nœud de la meilleure façon possible. Ce point est très important car un arbre « mal entretenu » est moins efficace lors de futures recherches, comme illustré ci-dessous.

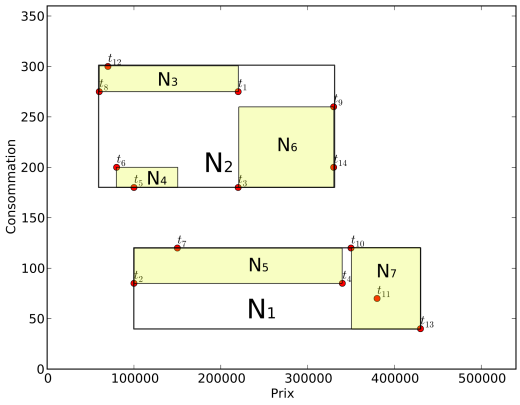
Exemple 4.11 - Avec notre relation exemple (*cf.* Table 4.3), la figure 4.7 met en évidence l'importance du découpage de l'espace. En effet, dans le premier cas (4.7a) la recherche du tuple t_3 nécessite le parcours de la totalité de l'arbre car t_3 est contenu à la fois dans les rectangles N_1 et N_2. Dans le second cas (4.7b), t_3 n'est inclus que dans le rectangle N_2, la recherche est bien ciblée donc plus efficace.

FIGURE 4.7 – Importance de bien diviser le MBR

(a) Mauvais découpage



(b) Bon découpage



Pour obtenir un découpage correct, l'une des heuristiques est de minimiser l'aire totale des MBRs produits tout en limitant le chevauchement de ces derniers. L'une des solutions serait de tester chacune des paires de MBRs possibles mais il y en a approximativement 2^{M-1} et cette approche est beaucoup trop coûteuse. Il vaut mieux se contenter d'un résultat approximatif mais calculable efficacement. L'idée est simple : on choisit les deux éléments à ne surtout pas mettre dans le même groupe et on les sépare. Ensuite, successivement, on cherche l'entrée qui va agrandir le plus possible la zone couverte par l'un des deux groupes et on l'ajoute dans l'autre. Le détail de l'algorithme DIVISERNOEUD est donné par l'algorithme 16.

Algorithme 16 Algorithme DIVISERNŒUD

Entrée : Le nœud N à diviser, l'entrée E à insérer
Sortie : Les nœuds N et NN

Soit NN le nouveau nœud résultant de la division.
Soit E_1 et E_2 le pire couple d'entrées à mettre dans un même groupe.
On place E_1 dans N et E_2 dans NN. //On a choisi la première entrée des groupes
boucle
  si tout a été assigné **alors**
    retourner N, NN //Test de terminaison
  fin si
  si un groupe va avoir moins de m entrées **alors**
    On lui assigne le reste pour atteindre ce nombre. //Impératif
    retourner N, NN
  fin si
  On ajoute la prochaine entrée prioritaire S au groupe qu'elle pénalisera le moins.
fin boucle

Maintenant que nous savons diviser les nœuds et chercher la feuille dans laquelle insérer, il nous faut expliquer comment propager l'information d'ajout dans tout l'arbre pour qu'il reste valide. Le principe est le même que celui employé pour les B-Trees : on part de la feuille

où la modification a eu lieu et on remonte jusqu'à la racine. Le détail de AjusterArbre est présenté dans l'algorithme 17.

Algorithme 17 Algorithme AjusterArbre

Entrée : Le nœud N où on a voulu insérer, l'éventuel nœud créé NN

boucle
  si N est la racine de l'arbre **alors**
    quitter
  fin si
  Soit P le parent de N.
  On met à jour le MBR de l'entrée de P correspondante.
  si NN existe **alors**
    si P n'est pas plein **alors**
      On ajoute une nouvelle entrée E dans P pointant vers NN
    sinon
      $P, PP =$ DiviserNoeud(P, E)
      $NN = PP$
    fin si
  fin si
  $N = P$
fin boucle

Exemple 4.12 - La figure 4.8 montre l'état de l'arbre après l'insertion du tuple $t_{15} = (15, 150000, 15, 180, 5)$. Les rectangles examinés durant l'insertion sont grisés.

3. **L'algorithme de suppression**
   Ici, on souhaite supprimer une entrée E de l'arbre.

Algorithme 18 Algorithme AjusterArbre

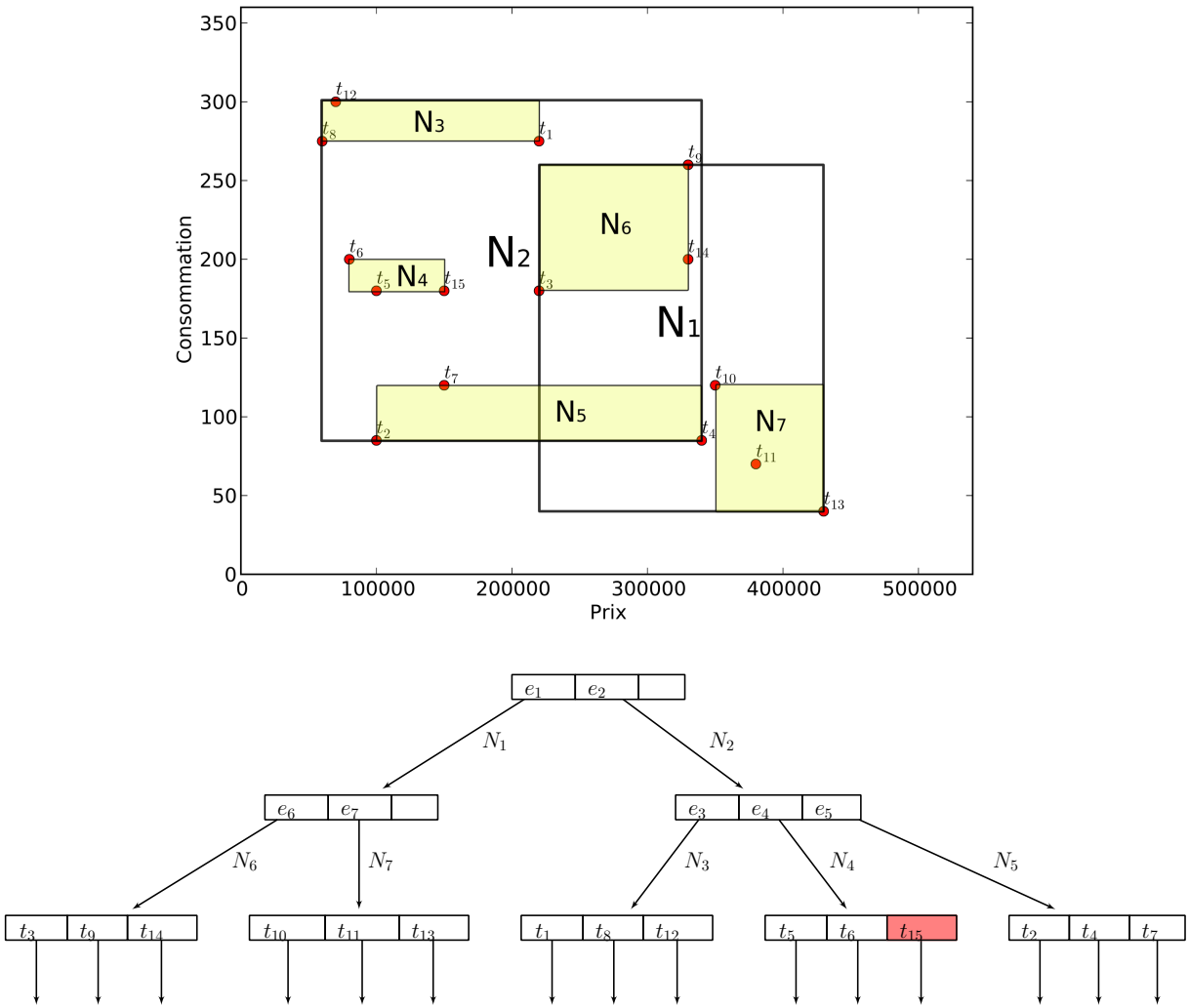
Entrée : L'arbre A à modifier, l'entrée E à supprimer

$F =$ TrouverFeuille(A, E) //On cherche l'emplacement de la suppression
si F n'existe pas **alors**
  quitter
fin si
On supprime E de F
CondenseArbre(F) //On propage les modifications
si la racine n'a plus qu'un seul fils **alors**
  Faire de ce fils la nouvelle racine //Raccourcir l'arbre
fin si

L'algorithme de suppression est similaire à celui des B-Trees. Il est rappelé dans l'algorithme 18. La seule différence vient de la partie CondenserArbre. Ici, on n'essaye pas de rattacher à ses « voisins » un nœud dont le nombre d'entrées est inférieur à m. Les auteurs de (Guttman, 1984) ont préféré supprimer le nœud trop vide et réinsérer les entrées perdues afin de raffiner la structure spatiale de l'index et d'éviter de détériorer l'arbre. CondenserArbre est décrit dans l'algorithme 19.

Notons qu'il existe d'autres variantes telles que les R+-Trees (Sellis *et al.*, 1987) ou les R*-

FIGURE 4.8 – Insertion du tuple t_{15} dans l'arbre



Algorithme 19 Algorithme CondenserArbre

Entrée : Le nœud N qui a été modifié
1: Soit Q l'ensemble des nœuds éliminés
2: **boucle**
3: **si** N est la racine **alors**
4: **break**
5: **fin si**
6: Soit P le parent de N et E_N l'entré pointant vers N dans P
7: **si** N a moins que m entrées **alors**
8: Supprimer E_N de P //N n'est plus valide
9: Ajouter N à Q
10: **sinon**
11: Ajuster le MBR de E_N //N est toujours pointé par E_N
12: **fin si**
13: $N = P$
14: **fin boucle**
15: **pour tout** Nœud N de Q **faire**
16: On réinsère N avec Insérer //Les nœuds internes doivent être insérés au bon niveau
17: **fin pour**

Trees (Beckmann *et al.*, 1990) qui tentent d'améliorer les recherches dans l'arbre.

L'algorithme Nearest Neighbor *« NN »* L'algorithme *« NN »* a été la première méthode développée pour résoudre le problème Skyline en exploitant les index spatiaux. Ce dernier utilise une constatation géométrique très simple (on considère sans perte de généralité que les points intéressants sont ceux qui sont proches de l'origine du repère et qu'on utilise une distance de Manhattan) : La zone spatiale que domine un tuple $i = (c_1, c_2, \ldots, c_d)$ est la zone $[c_1, \infty)\,[c_2, \infty) \ldots [c_d, \infty)$. Nous la nommerons par la suite la région de dominance d'un tuple.

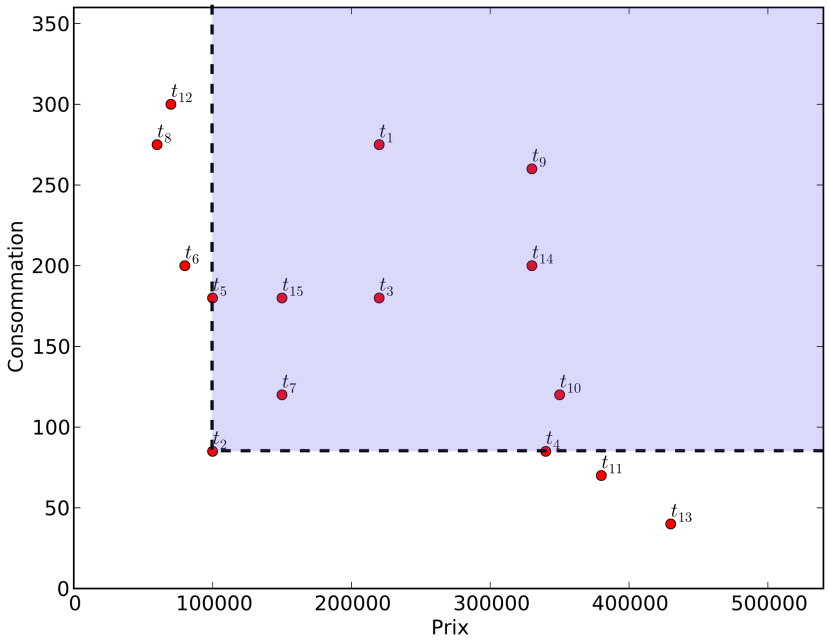
Exemple 4.13 - La figure 4.9 illustre le concept de région de dominance en considérant le tuple t_2 de notre exemple. Tous les tuples contenus dans la zone grisée sont dominés par t_2. Cette caractéristique est importante car en examinant uniquement le tuple t_2, il est possible d'éliminer tous les tuples situés dans sa zone de dominance (*ie.* $\{t_1, t_3, t_4, t_5, t_7, t_9, t_{10}, t_{14}, t_{15}\}$).

Lors de la découverte d'un nouveau point, l'algorithme utilise la susdite propriété, pour éliminer des régions de l'espace à ne plus explorer. Ainsi, il suffit de chercher uniquement dans les régions susceptibles de contenir d'autres points candidats.

Prenons le cas d'un Skyline à deux dimensions. Si notre point p a pour coordonnées (p_x, p_y),les candidats restants doivent être recherchés seulement dans la zone $[0, p_x)\,[0, \infty)$ et $[0, \infty)\,[0, p_y)$. La zone $[p_x, \infty)\,[p_y, \infty)$ est écartée. Les index de type spatiaux permettent de connaître très efficacement quels points appartiennent à ces zones de l'espace. Il est donc simple d'éliminer des tuples de la relation contenus dans les régions dominées.

Comme son nom l'indique, l'algorithme *NN* est basé sur des requêtes de plus proche voisinage. Elles sont implémentées à l'aide d'index spatiaux tels que les R-Trees présenté précédemment. l'algorithme commence en cherchant le point le plus proche de l'origine du repère. Ce dernier fait forcément partie du Skyline puisqu'aucun autre point n'a pu le dominer. À chaque étape,

FIGURE 4.9 – Région de dominance du tuple t_2



il met à jour une liste de zones qu'il devra explorer nommée *« to-do liste »*. Cette liste est maintenue en mémoire principale tout au long de l'algorithme. Après la découverte du premier point, il ajoute les zones non dominées dans la liste. Il va maintenant explorer les régions de la to-do liste, tant que celle-ci n'est pas vide. Aussi, remarque-t-on qu'une zone explorée vide ne partitionnera pas l'espace à son tour et en conséquence, n'agrandira pas la to-do liste. Ce n'est que la découverte d'un nouveau point qui partitionne l'espace.

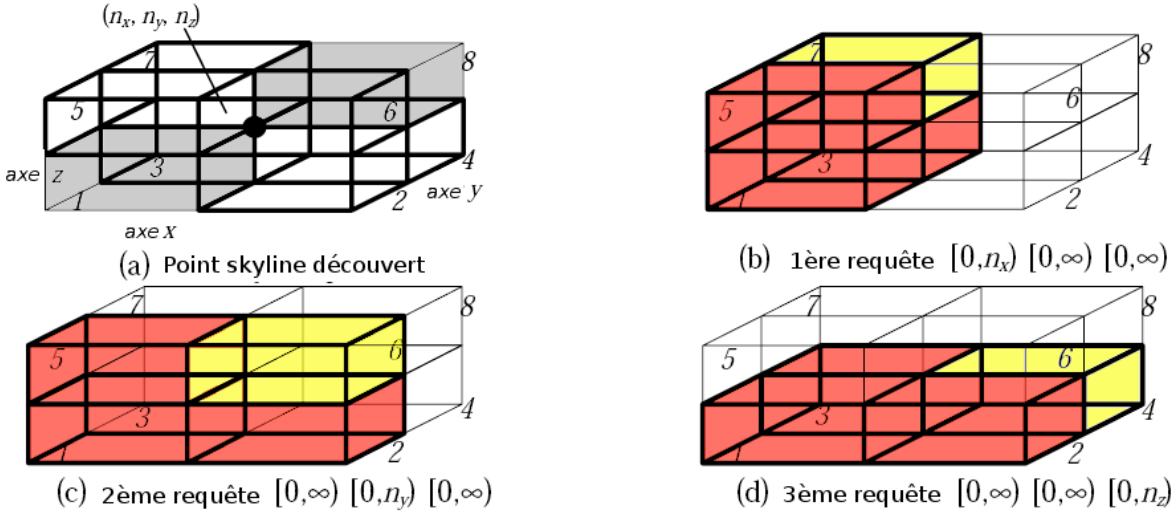
En général, pour un espace d-dimensionnel, chaque nouveau point SKYLINE découvert p est à l'origine de d appels récursifs de l'algorithme, une nouvelle zone de recherche étant ajoutée pour chaque coordonnée de p. Plusieurs problèmes se posent. Tout d'abord, pour $d \geq 3$, certaines zones de l'espace peuvent être recherchées de multiples fois. L'exemple 4.10 donne une illustration de ce cas.

Exemple 4.14 - Après la découverte du premier point SKYLINE, trois nouvelles requêtes *NN* sont exécutées. Les zones à explorer sont indiquées en jaune et les parties qui sont explorées de multiples fois sont colorées en rouge.

Cela signifie qu'un point SKYLINE peut être découvert à plusieurs reprises. Il est important de prendre en compte ce problème pour ne pas fausser les résultats et faire des calculs inutiles. Kossmann *et al.* (2002) propose plusieurs méthodes d'élimination des duplicats :

– **Laisser-faire**
  L'idée est d'utiliser une table de hachage stockée en mémoire principale pour savoir si un point p découvert a déjà été ajouté auparavant. Si ce n'est pas le cas, p est ajouté. Cette technique a le mérite d'être assez directe et implique un faible coût de calcul. Toutefois, elle entraîne aussi beaucoup d'E/S car de larges parties de l'espace risquent d'être examinées

FIGURE 4.10 – Chevauchements des zones de l'espace durant le déroulement de l'algorithme *NN*



à de nombreuses reprises.

– **Propager**
Lorsqu'un point p est découvert, toutes les zones sont supprimées de la to-do liste, partitionnées à nouveau selon p puis réinsérées. Avec cette méthode, les points ne sont pas redécouverts plusieurs fois, mais la to-do liste doit être entièrement scannée à chaque fois. Cette dernière opération entraine un surcoût important.

– **Fusionner**
Cette fois-ci, on cherche à fusionner les partitions de l'espace contenues dans la to-do liste entre elles, afin de réduire le nombre de requêtes à effectuer. En effet, on va pouvoir supprimer des zones, contenues dans d'autres. Le problème ici est qu'il est coûteux de trouver de bons candidats pour les fusions et le temps de calcul augmente fortement.

D'après les auteurs (Kossmann *et al.*, 2002), les méthodes *laisser-faire* et *fusionner* sont inacceptables en pratique. *Propager* est un peu plus efficace mais une méthode hybride entre *laisser-faire* et *propager* donne les meilleurs résultats.

Nous donnons à présent les critères d'évaluation pour l'algorithme :

Critère	Satisfaisant ?	Commentaires
Progressivité	OUI	Retourne les premiers points SKYLINE très rapidement
Universalité	OUI	Utilise les R-Trees
Limitations	NON	Ne se termine pas dans la majorité des cas pour $d \geq 4$

Pour conclure, *NN* est un bon algorithme pour les SKYLINES de faible dimension. Toutefois il a de très sérieux défauts qui l'empêchent d'être exploitable dès que le nombre de dimension dépasse 4.

L'algorithme Branch-and-Bound Skyline *« BBS »* Le dernier algorithme présenté, *BBS*, est développé dans le même esprit que *NN* puisque lui aussi est aussi basé sur des recherches de plus proches voisins. En effet, celui-ci part de la racine du R-Tree et l'explore en descendant

seulement dans les branches qui lui sont utiles. C'est pour cela que l'on dit qu'il est optimal en nombre d'E/S (Papadias *et al.*, 2003). Sa recherche des plus proches voisins ressemble à l'algorithme *best-first nearest neighbor* (Hjaltason et Samet, 1999).

Pour parvenir à ses fins, l'algorithme utilise un tas qui va contenir les entrées de l'arbre pris en considération, triées selon la distance de leur MBR par rapport à l'origine du repère. À chaque fois le premier nœud est examiné et ses entrées non dominées par des points SKYLINE déjà trouvés sont ajoutées au tas. Cela donne l'algorithme 20.

Algorithme 20 Algorithme BBS

Entrée : Le R-Tree R
Sortie : Les points SKYLINE S

```
1: Soit T un tas //trié selon la distance des MBRs par rapport à l'origine
2: Insérer les entrées de la racine de R dans T
3: tant que T non vide faire
4:    On enlève du tas la première entrée, e.
5:    si e dominée par un point de S alors
6:       Éliminer e
7:    sinon
8:       si e est une entrée intermédiaire alors
9:          pour tout e_i entrée de e faire
10:            si e_i n'est pas dominée par un point de S alors
11:               Ajouter e_i dans T //e_i pointe vers un nœud
12:            fin si
13:         fin pour
14:      sinon
15:         pour tout e_i entrée de e faire
16:            si e_i n'est pas dominée par un point de S alors
17:               Ajouter e_i.point dans S //e_i pointe vers un tuple
18:            fin si
19:         fin pour
20:      fin si
21:   fin si
22: fin tant que
```

On remarque que la dominance de chaque entrée est testée deux fois : avant d'être ajoutée dans le tas et avant d'être examinée. Ces deux tests sont nécessaires car il est possible qu'une entrée examinée soit dominée par un point SKYLINE découvert après son insertion dans le tas. (Papadias *et al.*, 2003) prouve que l'algorithme *BBS* est correct et optimal vis à vis des E/S.

Exemple 4.15 - Le déroulement de l'algorithme *BBS* pour notre relation exemple est décrit dans la table 4.4 qui indique, pour chaque étape de l'algorithme, l'entrée examinée, le contenu du tas et le SKYLINE courant. À chaque étape, l'algorithme retire du tas le nœud le plus proche de l'origine et insère tous ses fils dans le tas. Pour chacune de ces opérations, il vérifie la dominance. Les entrées dominées sont suprimées (elles sont barrées dans la table). Quand l'entrée courante est un tuple non dominé, elle est ajoutée au SKYLINE. L'algorithme s'arrête lorsque le tas est

TABLE 4.4 – Détail du déroulement l'algorithme *BBS*

Action	**Contenus du tas**	**Skyline**
Examiner R	$<e_2>$, $<e_1>$	$\varnothing$
Examiner e_2	$<e_5>$, $<e_1>$, $<e_4>$, $<e_3>$	$\varnothing$
Examiner e_5	$<t_2>$, $<t_7>$, $<e_1>$, $<e_4>$, $<e_3>$, $<t_4>$	$\varnothing$
Examiner t_2	~~$<t_7>$~~, $<e_1>$, $<e_4>$, $<e_3>$, ~~$<t_4>$~~	$\{t_2\}$
Examiner e_1	$<e_4>$, $<e_3>$, ~~$<e_6>$~~, $<e_7>$	$\{t_2\}$
Examiner e_4	~~$<t_5>$~~, $<t_6>$, $<e_3>$, $<e_7>$	$\{t_2\}$
Examiner t_6	$<e_3>$, $<e_7>$	$\{t_2, t_6\}$
Examiner e_3	$<t_8>$, $<t_{12}>$, ~~$<t_1>$~~, $<e_7>$	$\{t_2, t_6\}$
Examiner t_8	~~$<t_{12}>$~~, $<e_7>$	$\{t_2, t_6, t_8\}$
Examiner e_7	$<t_{11}>$, ~~$<t_{10}>$~~, $<t_{13}>$	$\{t_2, t_6, t_8\}$
Examiner t_{11}	$<t_{13}>$	$\{t_2, t_6, t_8, t_{11}\}$
Examiner t_{13}	$\varnothing$	$\{t_2, t_6, t_8, t_{11}, t_{13}\}$

vide.

La figure 4.11 représente le SKYLINE obtenu à la fin de l'algorithme. La zone grisée correspond à l'union des zones de dominance des tuples de ce SKYLINE ; la frontière de cette zone est matérialisée par la ligne noire brisée qui met en évidence le concept même de SKYLINE.

Examinons *BBS* selon les critères que nous avons définis.

Critère	**Satisfaisant ?**	**Commentaires**
Progressivité	OUI	Retourne les premiers points SKYLINE très rapidement
Universalité	OUI	Utilise les R-Trees
Limitations	OUI	peu efficace pour $d \geq 10$

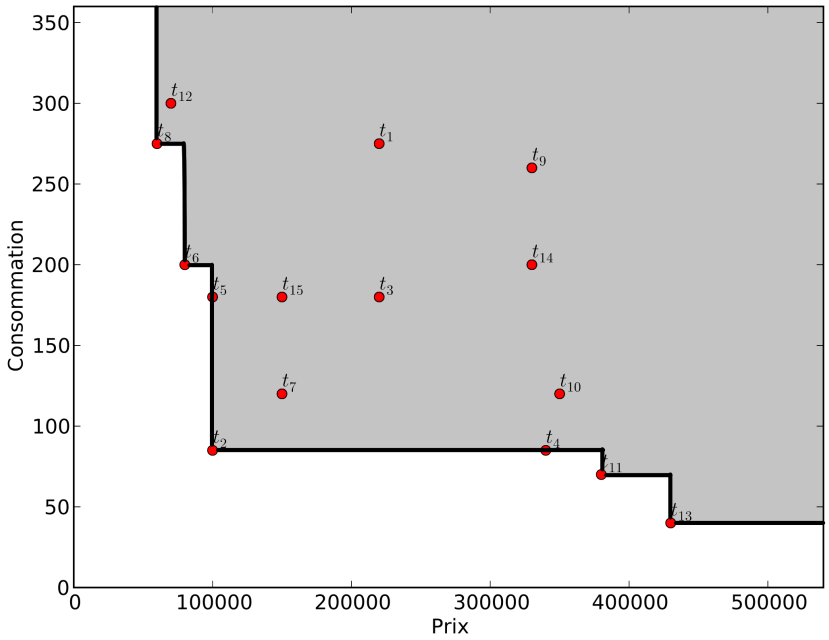
On peut améliorer les performances de l'algorithme en utilisant un deuxième R-Tree, pour les points du SKYLINE, afin d'augmenter la rapidité des tests de dominance. Il suffit alors d'exécuter une requête qui n'a pas besoin d'examiner autre chose que les entrées de la racine. Le gain de performance est très important.

L'algorithme *BBS* est la méthode de calcul la plus efficace que nous ayons étudiée pour le calcul du SKYLINE. Elle sera donc appropriée s'il s'agit de calculer de multiples SKYLINES.

4.3 Analyse multicritère Skyline dans l'espace multidimensionnel : l'approche SkyCube

L'opérateur SKYLINE est reconnu pour être un outil fondamental pour l'analyse multicritère des bases de données. Comme nous l'avons vu dans la section précédente, de nombreux travaux récents se sont penchés sur ce problème et proposent des méthodes de calcul efficaces. Grâce à ces algorithmes nous pouvons calculer un SKYLINE suivant un ensemble de critères définis par l'utilisateur, mais lorsqu'il s'agit d'en calculer plusieurs sur les mêmes données, aucun d'eux ne sait exploiter à son avantage les liens qui peuvent exister entre les différents SKYLINES. De plus la connaissance de ces liens peut être très intéressante non seulement pour économiser du

FIGURE 4.11 – Résultat final de l'algorithme *BBS*



temps de calcul mais aussi pour un décideur qui souhaite comprendre les raisons pour lesquelles un tuple donné devient dominant. C'est pour cela qu'une structure nommée SKYCUBE à été introduite (Pei *et al.*, 2006). Ainsi de la même manière que le cube de données permet d'analyser toutes les tendances présentes dans un jeu de données à plusieurs niveaux d'agrégation, le SKYCUBE rend possible quant à lui l'analyse de toutes les relations entre objets dominants dans tous les sous-espaces (tous les sous-ensembles de critères). En d'autre termes, on peut dire que le SKYCUBE est au SKYLINE ce que le cube de données est au GROUP-BY : une généralisation multidimensionnelle. Le calcul d'un SKYLINE étant généralement au moins aussi coûteux que le calcul d'un GROUP-BY, cette nouvelle structure a les mêmes inconvénients que le cube de données. Ainsi, dès lors que l'on souhaite répondre rapidement à toutes les requêtes posées sur un SKYCUBE, il faut là aussi opter pour une pré-matérialisation de ce cube.

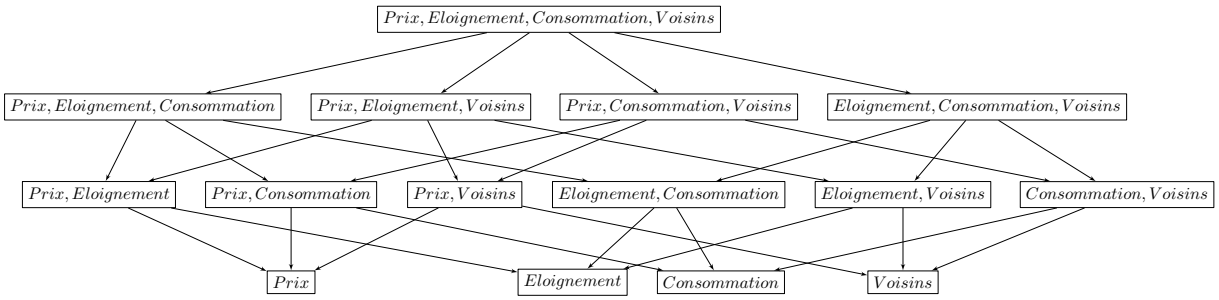
Ce paragraphe est dédié au SKYCUBE. Dans un premier temps nous décrivons le formalisme dans lequel est défini le SKYCUBE. Ensuite nous étudions les problèmes associés à l'analyse multidimensionnelle des SKYLINES et soulignons les nouvelles connaissances mises en évidence par le SKYCUBE. Enfin un algorithme de calcul efficace est présenté.

4.3.1 Concepts de base

Dans cette section, le concept de SKYLINE est étendu à l'espace multidimensionnel et nous présentons le concept de SKYCUBE.

Définition 4.3 (Sous-espace SKYLINE) **-** Un sous-ensemble de dimensions $\mathcal{B} \subseteq \mathcal{D}$ ($B \neq \emptyset$) forme un sous-espace $|\mathcal{B}|$-dimensionnel de $\mathcal{D}$. Pour un tuple p dans l'espace D, la projection de p dans le sous-espace $\mathcal{B}$, dénoté par $p[\mathcal{B}]$, est un $|\mathcal{B}|$-tuple $(p.Di_1, ..., p.Di_{|\mathcal{B}|})$, avec $p.Di_1, ..., p.Di_{|\mathcal{B}|} \in \mathcal{B}$

FIGURE 4.12 – Treillis des cuboïdes de la relation LOGEMENT



et $i_1 < ... < i_{|\mathcal{B}|}$. La projection d'un tuple p ($p \in r$) dans un sous-espace $\mathcal{B} \subseteq \mathcal{D}$ appartient au sous-espace SKYLINE selon $\mathcal{B}$, si aucun tuple $q[\mathcal{B}]$ (avec $q \in r$) ne domine $p[\mathcal{B}]$ dans $\mathcal{B}$. p est appelé un objet d'un sous-espace SKYLINE selon $\mathcal{B}$. Nous notons $SKY_{\mathcal{B}}(r)$ le sous-espace SKYLINE selon $\mathcal{B}$ pour la relation r.

Définition 4.4 (SKYCUBE) **-** Un SKYCUBE est l'ensemble de tous les SKYLINES dans tous les sous-espaces non vides possibles :

$$SKYCUBE(r, \mathcal{D}) = \{(\mathcal{B}, SKY_{\mathcal{B}}(r)) \mid \mathcal{B} \subseteq \mathcal{D}, \mathcal{B} \neq \emptyset\}$$

$SKY_{\mathcal{B}}(r)$ est appelé le cuboïde SKYLINE du sous-espace $\mathcal{B}$.

La structure du SKYCUBE peut être représentée par un treillis semblable à celui utilisé pour le cube de données (*cf.* figure 2.1). Les cuboïdes du SKYCUBE sont regroupés par niveau en fonction de leur nombre de dimensions. Ces niveaux sont numérotés en partant du bas du treillis (cuboïdes portant sur une seule dimension) et en remontant vers le sommet (cuboïde suivant tous les critères possibles). Soit deux cuboïdes $SKY_{\mathcal{U}}(r)$ selon le sous-ensemble de dimensions $\mathcal{U}$ et $SKY_{\mathcal{V}}(r)$ selon $\mathcal{V}$, si $\mathcal{U} \subset \mathcal{V}$ on dit alors que les cuboïdes ont un lien de parenté, $SKY_{\mathcal{V}}(r)$ est nommé *l'ancêtre* de $SKY_{\mathcal{U}}(r)$ et $SKY_{\mathcal{U}}(r)$ *le descendant* de $SKY_{\mathcal{V}}(r)$. De plus, dans le cas où $|\mathcal{U}| = |\mathcal{V} - 1|$, $SKY_{\mathcal{V}}(r)$ est appelé le cuboïde père de $SKY_{\mathcal{U}}(r)$ et $SKY_{\mathcal{U}}(r)$ le cuboïde fils de $SKY_{\mathcal{V}}(r)$.

Exemple 4.16 (SKYCUBE) **-** Dans ce paragraphe nous utilisons la même relation LOGEMENT (*cf.* Table 4.1) que précédement pour illustrer les concepts présentés. La table 4.5 donne le SKYCUBE associé à cette relation. La figure 4.12, quant à elle, représente le treillis des différents cuboïdes de ce SKYCUBE.

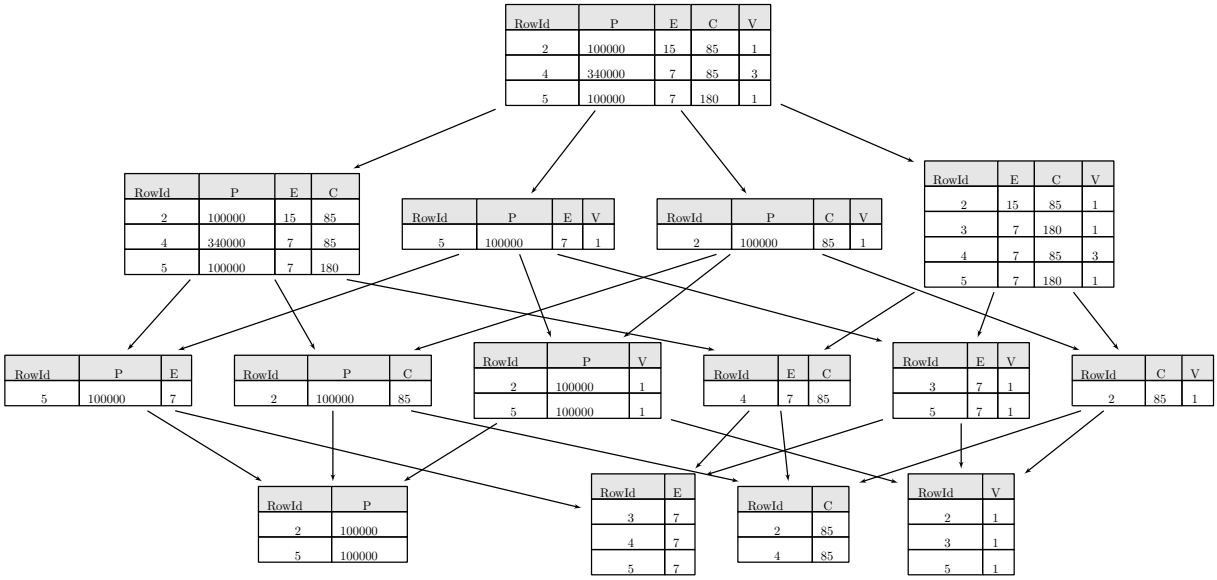
Une requête SKYLINE multidimensionnelle retourne le sous-ensemble de tuples de la relation originelle formant le SKYLINE dans un sous-espace donné. Clairement, une fois le SKYCUBE calculé, il est possible de répondre à toute requête efficacement.

Comme dans le cas du cube de données, le SKYCUBE peut contenir des informations superflues. Par exemple, dans la Figure 4.13, nous pouvons observer que les SKYLINES dans des sous-espaces (Prix, Éloignement, Consommation, Voisins) et (Prix, Éloignement, Consommation) sont constitués des mêmes tuples. Un autre exemple de cette redondance, les SKYLINES selon (Prix, Éloignement), (Prix, Voisins) et (Prix) sont tous les trois identiques. Pourquoi des tuples peuvent-ils apparaître ensemble dans les SKYLINES de plusieurs sous-espaces ? Pouvons-nous éliminer la redondance des représentations des SKYLINES ? Ce sont ces questions qui ont motivé l'exploration de la sémantique véhiculée par le SKYCUBE.

TABLE 4.5 – SKYCUBE de la relation LOGEMENT

Sous-espace	SKYLINE
Prix, Éloignement, Consommation, Voisins	t_2 =(100000, 15, 85, 1) t_4 =(340000, 7, 85, 3) t_5 =(100000, 7, 180, 1)
Prix, Éloignement, Consommation	t_2 =(100000, 15, 85) t_4 =(340000, 7 , 85) t_5 =(100000, 7 , 180)
Prix, Éloignement, Voisins	t_5 =(100000, 7 , 1)
Prix, Consommation, Voisins	t_2 =(100000, 85, 1)
Éloignement, Consommation, Voisins	t_2 =(15, 85, 1) t_3 =(7, 180, 1) t_4 =(7, 85, 3) t_5 =(7, 180, 1)
Prix, Éloignement	t_5 =(100000, 7)
Prix, Consommation	t_2 =(100000, 85)
Prix, Voisins	t_2 =(100000, 1) t_5 =(100000, 1)
Éloignement, Consommation	t_4 =(7, 85)
Éloignement, Voisins	t_3 =(7, 1) t_5 =(7, 1)
Consommation, Voisins	t_2 =(85, 1)
Prix	t_2 =(100000) t_5 =(100000)
Éloignement	t_3 =(7) t_4 =(7) t_5 =(7)
Consommation	t_2 =(85) t_4 =(85)
Voisins	t_2 =(1) t_3 =(1) t_5 =(1)

FIGURE 4.13 – Représentation sous forme de treillis du SKYCUBE de la relation LOGEMENT



4.3.2 Problèmes associés à l'analyse multidimensionnelle des Skylines

En général, si un tuple t est dans les SKYLINES des sous-espaces $\mathcal{B}_1$ et $\mathcal{B}_2$ tels que $\mathcal{B}_1 \subset \mathcal{B}_2$, pouvons nous affirmer que t appartiendra aussi au SKYLINE de n'importe quel sous-espace $\mathcal{B}$ situé entre $\mathcal{B}_1$ et $\mathcal{B}_2$ ($\mathcal{B}_1 \subset \mathcal{B} \subset \mathcal{B}_2$) ? Une telle propriété serait fort attrayante puisqu'elle pourrait considérablement simplifier la détermination des SKYLINES multidimensionnels. Malheureusement comme nous allons le voir avec l'exemple suivant, dans le cas général la situation est loin d'être aussi simple.

Exemple 4.17 - Avec la relation LOGEMENT, comme le montre la figure 4.13, en considérant les SKYLINES selon (Prix, Éloignement,Voisin) et (Éloignement,Voisin) on a : (Prix, Éloignement,Voisin) $\supseteq$ (Éloignement,Voisin) et SKY_{PEV}(LOGEMENT)) $\subseteq$ SKY_{EV}(LOGEMENT)). En revanche, si l'on regarde les SKYLINES suivants (Prix, Éloignement, Consommation, Voisins) et (Prix, Éloignement, Consommation) on a (Prix, Éloignement, Consommation, Voisins) $\supseteq$ (Prix, Éloignement, Consommation) et SKY_{PECV}(LOGEMENT)) $\supseteq$ SKY_{PEC}(LOGEMENT)).

Les tuples t_2 et t_5 partagent les mêmes valeurs sur les attributs Prix et Voisins. Il est important de remarquer que lorsqu'ils sont projetés sur {Prix, Voisins}, ces deux tuples deviennent indistinguables. Ainsi, si l'un des deux est un SKYLINE dans l'un des sous-espaces de (Prix, Voisins), alors l'autre tuple appartiendra lui aussi à ce SKYLINE.

Les observations mises en évidence par cet exemple peuvent être récapitulées comme suit.

- De manière générale, l'appartenance à un SKYLINE n'est pas monotone, c'est-à-dire qu'un tuple t appartenant à un cuboïde $SKY_{\mathcal{U}}(r)$) n'est pas automatiquement contenu dans les ancêtres de ce cuboïde. Le tuple t peut être dominé dans un des cuboïdes parents de $SKY_{\mathcal{U}}(r)$) par l'un des tuples ayant même projection que lui pour le sous-ensemble de dimensions $\mathcal{U}$.
- Les tuples partageant les mêmes valeurs pour les critères dans plusieurs sous-espaces peuvent être regroupés. L'appartenance aux SKYLINES multidimensionnels est alors partagée par tous les tuples d'un même groupe. Il est donc crucial de capturer de tels groupes

de tuples pour améliorer le calcul du SKYCUBE en supprimant la redondance.

Définition 4.5 (*c-groupe*) **-** Soit $G \subseteq r$ un ensemble de tuples et $\mathcal{B} \subseteq \mathcal{D}$ un sous-ensemble de dimensions. (G,$\mathcal{B}$) est un groupe coïncident (aussi nommé *c-groupe*), si pour chaque dimension de $\mathcal{B}$, les tuples de G partagent les mêmes valeurs. La projection du groupe sur $\mathcal{B}$, notée $G_{\mathcal{B}}$, est $u[\mathcal{B}]$ avec $u \in G$.

Un *c-groupe* est maximal s'il n'existe aucun autre tuple $t \in r \backslash G$ partageant les mêmes valeurs que les tuples de G sur les dimensions de $\mathcal{B}$, et les tuples de G ne partagent pas d'autres valeurs identiques pour d'autres attributs que ceux de $\mathcal{B}$. $\mathcal{B}$ est appelé le sous-espace signature de G.

Exemple 4.18 - En considérant la relation LOGEMENT, le sous-ensemble de tuples $G = \{t_2, t_5\}$ $= \{(100000, 15, 85, 1), (100000, 7, 180, 1)\}$ et $\mathcal{B} = \{$Prix, Voisins$\}$ forment un *c-groupe*. De plus, il est maximal car aucun autre tuple de la relation ne partage les mêmes valeurs sur $\mathcal{B}$ et t_2 et t_5 n'ont pas d'autres attributs égaux. La projection de ce groupe est $G_{\mathcal{B}} = (100000, 1)$.

Étant donné un ensemble de tuples G, $\mathcal{I}(G)$ est défini comme l'ensemble de dimensions maximal pour lequel tous les tuples de G partagent les mêmes valeurs.

$$\mathcal{I}(G) = \{D_i \mid D_i \in \mathcal{D}, \forall t,u \in G \ : \ t.D_i = u.D_i\}$$

De plus, pour un sous-espace $\mathcal{B}$ et un ensemble de tuples G, $\mathcal{O}(G, \mathcal{B})$ est défini comme l'ensemble maximal des tuples de r qui ont les mêmes valeurs sur des dimensions de $\mathcal{B}$ que les tuples de G.

$$\mathcal{O}(G, \mathcal{B}) = \{t \mid t \in r, \forall D_i \in \mathcal{B} \text{ et } \forall u \in G \ : \ t.D_i = u.D_i\}$$

Exemple 4.19 - Dans notre relation exemple on a :

$$\mathcal{I}(\{t_2, t_5\}) = \{\text{Prix}, \text{Voisins}\}$$
$$\mathcal{O}(\{t_2, t_5\}, \{\text{Voisins}\}) = \{t_2, t_3, t_5\}$$

Les deux opérateurs précédents peuvent être utilisés pour dériver les c-groupes maximaux pour n'importe quel ensemble de tuples ou pour un ensemble de tuples et un sous-ensemble de dimensions. De façon générale, étant donné un ensemble de tuples G, nous pouvons obtenir un c-groupe maximal en deux étapes. D'abord, nous pouvons trouver le sous-espace maximal $\mathcal{I}(G)$ où les tuples de G partagent la même projection. Puis, nous pouvons insérer dans G tous les autres tuples de la relation partageant la même projection que $G_{\mathcal{I}(G)}$. D'autre part, si nous voulons dériver un groupe dans un sous-espace $\mathcal{B}$, nous pouvons d'abord insérer dans G tous les autres tuples partageant la projection $G_{\mathcal{B}}$ (en d'autre termes l'ensemble de tuples devient $\mathcal{O}(G, \mathcal{B})$) et ensuite trouver le sous-espace maximal où les objets dans $\mathcal{O}(G, \mathcal{B})$ partagent la même projection.

Le lemme suivant formalise ces dérivations.

Lemme 4.1 (*c-groupe*) **-** Étant donné un ensemble de tuples G, ($\mathcal{O}(G, \mathcal{I}(G)), \mathcal{I}(G)$) est un c-groupe maximal. Étant donné c-groupe (G,$\mathcal{B}$), ($\mathcal{O}(G, \mathcal{B})$, $\mathcal{I}(\mathcal{O}(G, \mathcal{B}))$) est un c-groupe maximal.

Nous sommes particulièrement intéressés par les c-groupes maximaux dont les projections sont dans les SKYLINES de plusieurs cuboïdes. Intuitivement, nous voulons capturer les sous-ensembles de valeurs dans leurs projections qui sont décisives pour leur appartenance à des SKYLINES.

Définition 4.6 (Groupe SKYLINE et sous-espace décisif) **-** Un c-groupe maximal $(G,\mathcal{B})$ est appelé *groupe* SKYLINE si $G_{\mathcal{B}}$ appartient au SKYLINE multidimensionnel selon $\mathcal{B}$.

Étant donné un groupe SKYLINE $(G,\mathcal{B})$, un sous-espace $\mathcal{C} \subseteq \mathcal{B}$ est dit décisif si (1) $G_{\mathcal{C}}$ appartient au cuboïde selon $\mathcal{C}$; (2) $(\mathcal{O}(G, \mathcal{C}) = G$; et (3) il n'existe aucun sous-ensemble de dimensions $\mathcal{C}' \subseteq \mathcal{C}$ qui satisfasse les deux conditions précédentes.

La signature d'un groupe SKYLINE $(G,\mathcal{B})$ est définie comme suit :

$$Sig(G,\mathcal{B}) = \langle G_{\mathcal{B}}, \mathcal{C}_1, \cdots, \mathcal{C}_k \rangle$$

avec $\mathcal{C}_1, \cdots, \mathcal{C}_k$ tous les sous-espaces décisifs du groupe SKYLINE.

Exemple 4.20 - D'après la table 4.5, le *c-groupe* maximal $(\{t_2, t_5\}$,{Prix, Voisins}) est un groupe SKYLINE. Le sous-espace $\mathcal{C}_1$ = {Voisins} ne constitue pas un sous-espace décisif car $\mathcal{O}(\{t_2, t_5\}, \{\text{Voisins}\}) \neq G$ (*cf.* exemple 4.19). Le sous-espace $\mathcal{C}_2$ = {Prix}, quant à lui, est un sous-espace décisif. La signature de notre groupe SKYLINE exemple est donc :

$$Sig(\{t_2, t_5\}, \{\text{Prix, Voisins}\}) = \langle (100000, 1), \{\text{Prix}\} \rangle$$

Théorème 4.2 (Sous-espace décisif) **-** Étant donné un groupe SKYLINE $(G,\mathcal{B})$, si $\mathcal{C}$ est un sous-espace décisif, alors pour tous les sous-ensembles de dimensions $\mathcal{C}'$ tels que $\mathcal{C} \subseteq \mathcal{C}' \subseteq \mathcal{B}$, $G_{\mathcal{C}'}$ appartient au SKYLINE multidimensionnel.

Ce théorème garantit qu'en conservant uniquement les *c-groupes* maximaux et les sous-espaces décisifs associés, il possible de retrouver tous les tuples SKYLINE. La représentation sous-jacente est donc sans perte d'information et plus réduite que le SKYCUBE.

Soulignons les analogies entre cette représentation et les structures basées sur la reduction sémantique proposées pour le cube de données (*cf.* paragraphe 3.3 p.64). Le concept de signature d'un groupe avec son *c-groupe* maximal et ses sous-espaces décisifs s'apparente à la notion de classe d'équivalence du cube quotient avec son élément maximal et ses tuples minimaux. Bien que définie differemment, la notion de c-groupe est très proche de celle de tuple fermé et les sous-espaces décisifs corespondent aux tuples clefs associés à un fermé. D'autre part il existe aussi un lien entre cette représentation et le cube partition (*cf.* paragraphe 3.4 p.68). En effet la notion même de *c-groupe* est voisine des classes d'équivalence Dimension-Mesure. De plus les deux approches ont recours aux identifiants de tuples plutôt qu'aux tuples eux mêmes pour minimiser la redondance syntaxique. Enfin, certains algorithmes exploitent la même technique de réduction que les classes-DM, en ne conservant qu'un élément représentatif par *c-groupe* (Pei *et al.*, 2007).

4.3.3 Algorithme de calcul

Dans cette section, nous présentons l'algorithme Bottom-Up-Skycube (BUS). Il calcule les résultats de tous les sous-espaces SKYLINES non-vides possibles aussi bien que tous les c-groupes et leurs signatures. Cet algorithme adopte une approche basée sur Block-Nested-Loop (*cf.* paragraphe 4.2.2) pour calculer le SKYLINE de chaque sous-espace. BUS tire avantage pour optimiser son temps de calcul de deux stratégies : mutualisation des résultats et mutualisation des tris. Pour diminuer le nombre de comparaisons de tuples inutiles (tests de dominance), une amélioration heuristique suplémentaire à base de filtre est présentée.

Liens entre cuboïdes

Avant de présenter l'algorithme BUS, les liens entre objets SKYLINES présents dans des cuboïdes parents sont discutés. La condition de valeurs distinctes est présentée. Quand cette condition est satisfaite, le calcul s'en trouve grandement simplifié.

La condition de valeurs distinctes est satisfaite pour deux tuples s'ils n'ont aucune valeur commune pour l'ensemble des critères.

Définition 4.7 (Condition de valeurs distinctes) **-** Soit $\mathcal{T} \in r$ un ensemble de tuples, $\forall t, t' \in \mathcal{T}$. La condition est vraie si et seulement si $\forall d \in \mathcal{D}$, $t.d \neq t'.d$

Cette condition impose qu'aucune paire de tuples ne partage la même projection dans aucun sous-espace. Grâce à cette propriété, s'il existe un lien d'inclusion entre deux sous-espaces alors il existe un lien similaire entre les SKYLINES sous-jacents. Ainsi les objets SKYLINES appartenant aux fils d'un cuboïde donné sont aussi des objets SKYLINES pour ce cuboïde. En d'autres termes plus le sous-espace considéré comporte de dimensions plus la taille du cuboïde associé s'accroît. Cette caractéristique de croissance, vérifiée également pour le cube de données, permet aux algorithmes, à l'instar de BUC, de réutiliser les résultats intermédiaires pour optimiser le calcul.

Dans le cas général (condition de valeurs distinctes non vérifiée), la caractéristique précédente n'existe pas. Cependant, il existe toujours un lien entre objets SKYLINE de sous-espaces parents qui est décrit par le théorème suivant.

Théorème 4.3 - Etant donné un ensemble de tuples $\mathcal{T} \subset r$ et $\mathcal{U}$, $\mathcal{V}$ deux sous-espaces tels que $\mathcal{U} \subset \mathcal{V}$. Dans le sous-espace $\mathcal{V}$, tout tuple $t \in SKY_{\mathcal{U}}(\mathcal{T})$ peut être dominé soit par un tuple $t' \in SKY_{\mathcal{U}}(\mathcal{T})$ avec $t[\mathcal{U}] = t'[\mathcal{U}]$, soit par un tuple déjà présent dans $SKY_{\mathcal{V}}(\mathcal{T})$.

L'intuition sous-tendant ce théorème est la suivante : considérons deux tuples présents dans le SKYLINE du sous-espace $\mathcal{U}$ et ayant même projection ($t[\mathcal{U}] = t'[\mathcal{U}]$). En les considérant dans un espace père $\mathcal{V} = \mathcal{U} \cup d$, ces tuples resteront soit égaux soit diffèreront sur d. Dans ce cas l'un dominera forcément l'autre. De plus, un espace $\mathcal{V}$ a généralement plusieurs sous-espaces fils et les tuples SKYLINES sont nécessairement issus de ces sous-espaces.

L'algorithme Bottom Up Skyline

L'algorithme BUS est dédié au calcul du SKYCUBE. Cet algorithme va être présenté en deux parties. Dans la première, nous étudions l'algorithme à proprement parler en se restreignant uniquement aux relations satisfaisant la condition de valeurs distinctes (*cf.* définition 4.7). Puis à la fin de ce paragraphe, nous décrivons les modifications à apporter à l'algorithme pour traiter le cas général.

L'idée de base de BUS est de calculer chaque cuboïde du SKYCUBE niveau par niveau et de bas en haut. Ces différents SKYLINES sont calculés avec un algorithme similaire à *Sort First* SKYLINE (Chomicki *et al.*, 2005). Au lieu de les calculer directement, BUS va exploiter les liens existant entre eux pour mutualiser certaines opérations et ainsi économiser du temps de calcul. Cette mutualisation des calculs repose sur deux stratégies, *mutualisation des résultats* et *mutualisation des tris*, présentées ci-dessous.

- **Mutualisation des résultats :** par hypothèse notre relation d'entrée satisfait la condition de valeurs distinctes ; nous pouvons donc affirmer qu'un cuboïde père contient l'union de ses fils. Lorsque l'on calcule un cuboïde, on sait donc qu'un tuple présent dans un des fils sera forcément un tuple SKYLINE. Cette propriété a le double avantage de réduire l'entrée pour le calcul d'un cuboïde et de diminuer le nombre de tests de dominance effectués. En

effet, comme l'algorithme parcourt le treillis des cuboïdes de bas en haut et niveau par niveau, lors du calcul d'un cuboïde, au lieu de considérer toute la relation, l'entrée utilisée est l'union de ses cuboïdes fils précédemment calculés.

– **Mutualisation des tris :** pour éviter de trier l'entrée avant de calculer chacun des $2^{|\mathcal{D}|}$ cuboïdes , Yuan *et al.* proposent de changer le critère de tri de l'algorithme *Sort First* Skyline (Chomicki *et al.*, 2005). Effectivement pour un cuboïde selon $\mathcal{V}$, au lieu de trier l'entrée suivant tous les critères $d_i \in \mathcal{V}$, il suffit qu'elle soit triée suivant au moins un critère d_i. En utilisant une telle méthode, le nombre de tris à effectuer sur la relation d'entrée est réduit à $|\mathcal{D}|$ (exactement une fois par dimension). En plus de réduire considérablement le temps consacré aux tris, cette stratégie préserve les avantages de *SFS* : conserver un nombre de tuples candidats minimal en mémoire et découvrir les tuples Skylines plus rapidement.

Algorithme 21 Algorithme Bus

Entrée : S : un ensemble de tuples $|\mathcal{D}|$-dimensionnel
Sortie : tous les cuboïdes suivant $\mathcal{V} \subseteq \mathcal{D}$, $SKY_{\mathcal{V}}(S)$

```
1: Trier S suivant toutes les dimensions d_i ∈ D (dans l'ordre croissant) pour former |D| listes
   triées l_{a_i} (1 ≤ i ≤ |D|).
2: pour chaque niveau L_j du treillis pris de bas en haut faire
3:   pour chaque sous-espace V ∈ L_j faire
4:     SKY := l'union des cuboïdes fils du sous-espace courant V
5:     Choisir une liste triée l_{a_i} tel que a_i ∈ V
6:     pour chaque tuple t ∈ l_{a_i} faire
7:       si t ∈ SKY alors
8:         insérer t dans SKY_V(S)
9:       sinon
10:        Évaluer(t, SKY_V(S))
11:      fin si
12:    fin pour
13:  fin pour
14: fin pour
```

À partir des deux stratégies de mutualisation susdites, l'algorithme Bus calcule le SkyCube niveau par niveau du bas vers le haut. Il est détaillé dans l'Algorithme 21. Pour calculer chaque cuboïde selon $\mathcal{V}$, Bus examine chaque tuple t de manière ordonnée. Ils sont comparés avec les tous les tuples Skyline déjà obtenus. Si t appartient à un cuboïde fils, il est alors directement inséré dans le Skyline. Dans le cas contraire, avec la fonction Évaluer (*cf.* Algorithme 22), ce tuple est comparé avec les tuples présents dans le Skyline pour déterminer s'il fait lui aussi partie du Skyline. C'est cette fonction qui fait les tests de dominance.

Filtre heuristique

Dans sa version la plus simple la fonction Évaluer se contente de faire uniquement des tests de dominance. Ces tests ont une complexité linéaire par rapport au nombre de dimensions, donc plus ce nombre augmente plus le calcul est coûteux. De plus, comme cette fonction est appelée très fréquemment, il est important de diminuer autant que possible son temps de calcul. Pour ce faire, à la place du test de dominance relativement coûteux, un filtrage simple est réalisé comme étape préalable au test de dominance. De la sorte, si un tuple ne passe pas le filtrage il est inutile

Algorithme 22 Algorithme ÉVALUER

Entrée :
t : un tuple $|\mathcal{D}|$-dimensionnel à évaluer,
$SKY_{\mathcal{V}}(S)$: le cuboïde selon $\mathcal{V}$ en cours de calcul.
Sortie :
Le tuple t est inséré dans $SKY_{\mathcal{V}}(S)$ si c'est un SKYLINE.

```
1: Trier S suivant toutes les dimensions d_i ∈ D (dans l'ordre croissant) pour former |D| listes
   triées l_{a_i} (1 ≤ i ≤ |D|).
2: pour chaque tuple u ∈ SKY_V(S) faire
3:   si t ≻_V u alors
4:     insérer t dans SKY_V(S)
5:     retourner
6:   sinon si u ≻_V t alors
7:     retourner   // t est ignoré car ce n'est pas un tuple SKYLINE
8:   fin si
9: fin pour
10: Insérer t dans SKY_V(S)
```

de faire le test de dominance. La fonction de filtrage est une fonction monotone croissante qui prend en paramètre un tuple et qui retourne une valeur réelle permettant d'éliminer facilement une partie des tuples non dominants. Cette fonction est définie comme suit :

$$f_{\mathcal{U}}(t) = \sum_{\forall a_i \in \mathcal{U}} t.a_i$$

L'Algorithme 23 présente la modification apportée à ÉVALUER pour prendre en compte le filtrage. L'algorithme maintient une liste des candidats dans laquelle les tuples sont ordonnés par ordre croissant par rapport à la valeur de la fonction de filtrage. Si la valeur de filtrage de t est plus petite que celle de u, lorsque l'on compare un tuple t avec un point u du SKYLINE on est certain que t est un nouveau tuple SKYLINE ; dans le cas contraire un test de dominance classique est effectué.

Bottom Up Skyline dans le cas général

Si l'ensemble de données ne satisfait pas à la condition de valeurs distinctes, BOTTOM UP SKYLINE est étendu comme suit. Comme nous l'avons souligné au paragraphe précédent avec le Théorème 4.3, une seule partie des tuples SKYLINES du cuboïde selon $\mathcal{U}$ appartient, en général, à son cuboïde père suivant $\mathcal{V}$. Selon ce théorème, quand on souhaite examiner un tuple $t \in SKY_{\mathcal{U}}(S)$ on a besoin uniquement de le comparer avec un tuple u de $SKY_{\mathcal{U}}(S)$ ayant même projection que t sur $\mathcal{U}$ (*ie.* $u[\mathcal{U}] = u[\mathcal{U}]$). S'il n'existe aucun tuple u de la sorte, t est un tuple SKYLINE dans le sous-espace $\mathcal{V}$. Dans le cas contraire, t et u sont comparés sur l'ensemble de dimension $\mathcal{V} - \mathcal{U}$ pour déterminer celui des deux qui appartient à $SKY_{\mathcal{V}}(S)$.

Il est important de noter que le test précédent a un coût insignifiant parce que les tuples SKYLINES sont toujours maintenus dans l'ordre croissant par rapport à leur valeur pour la fonction de filtrage. Dans la mesure où deux tuples SKYLINES avec des valeurs différentes pour cette fonction ne sont pas égaux, seuls sont comparés entre eux, les tuples ayant la même valeur.

Quand la condition de valeur distincte n'est pas satisfaite, l'algorithme précédent ne peut plus garantir que seuls les tuples SKYLINES sont insérés dans la liste des candidats. Pour éliminer

Algorithme 23 Algorithme ÉVALUER avec filtre heuristique

Entrée :
t : un tuple $|\mathcal{D}|$-dimensionnel à évaluer,
$SKY_{\mathcal{V}}(S)$: le cuboïde selon $\mathcal{V}$ en cours de calcul.
Sortie :
Le tuple t est inséré dans $SKY_{\mathcal{V}}(S)$ si c'est un SKYLINE.

```
1: Trier S suivant toutes les dimensions d_i ∈ D (dans l'ordre croissant) pour former |D| listes
   triées l_{a_i} (1 ≤ i ≤ |D|).
2: pour chaque tuple u ∈ SKY_V(S) faire
3:    si f_V(t) < f_V(u) alors
4:       insérer t dans SKY_V(S)
5:       retourner
6:    sinon si u ≻_V t alors
7:       retourner   // t est ignoré car ce n'est pas un tuple Skyline
8:    fin si
9: fin pour
10: Insérer t dans SKY_V(S)
```

un tel « faux-SKYLINE », l'algorithme lit le groupe des tuples du buffer qui ont les mêmes valeurs sur les dimensions triées. Évidement, un tel tuple n'est pas dominé par l'un de ceux apparaissant après lui dans la liste triée, car il a une valeur inférieure au second sur les dimensions considérées. Les tuples de ce type sont donc considérés comme des candidats. Étant donné que la taille d'un tel groupe est réduite, calculer ses tuples SKYLINES est très peu coûteux. Par la suite BUS compare ces candidats avec les SKYLINES déjà calculés. Il est clair qu'une telle extension de BUS posséde elle aussi la propriété de n'insérer que les tuples SKYLINES dans la liste des candidats.

4.4 Conclusion

Dans la première partie de ce mémoire, nous avons proposé un tour d'horizon de l'analyse des bases de données. Pour le conclure, nous nous sommes penchés sur l'analyse multicritère à travers l'opérateur SKYLINE. Ce dernier permet à l'utilisateur d'extraire les solutions satisfaisant au mieux un ensemble de critères spécifiés. Après une description du contexte d'utilisation de cet opérateur, nous avons étudié sa définition, ainsi qu'une solution « bases de données » sous forme d'une requête SQL. Malheureusement celle-ci est trop peu performante pour être utilisée, aussi nous avons étudié les meilleurs représentants des deux grandes familles d'algorithmes remédiant à ce problème d'efficacité. Pour pouvoir opérer des choix entre ces différents algorithmes, nous avons pris en considération plusieurs critères d'évaluation, notamment la progressivité, incontournables pour notre contexte de travail.

Outre la réponse à des requêtes bien particulières, la problématique de l'opérateur SKYLINE nous a intéressés car ce concept a été couplé à celui de data cube. Ainsi une nouvelle problématique OLAP a t-elle fait son apparition. Comme pour le cube de données, il s'agit de combiner, au sein d'une même structure, tous les SKYLINES possibles selon un ensemble de critères. Ce concept de SKYCUBE partage avec le cube de données les mêmes avantages mais aussi les mêmes défauts dont celui d'un énorme volume. Pour remédier à cet inconvénient, une représentation visant à réduire les redondances a été présentée ainsi que l'algorithme associé. Ainsi que nous l'avons déjà remarqué, cette représentation a de nombreuses similitudes avec les structures réduites

proposées pour le data cube et basées sur la fermeture. En fait, il n'est pas étonnant que des approches visant le même objectif de réduction des représentations développent des méthodes voisines. Cette étude nous a conforté dans l'intérêt porté aux concepts liés à la fermeture qui, rappelons le, ont donné lieu dans le contexte de la fouille de données binaire à une représentation parmi les plus compactes pour les motifs fréquents et, dans le contexte OLAP classique, à l'une des couvertures les plus réduites du cube de données : le cube fermé.

Deuxième partie

Analyse des renversements de tendances dans les bases de données multidimensionnelles

5

Le Cube Émergent

Sommaire

5.1 Introduction et Motivations

SUPPOSONS que nous disposions d'un cube de données coûteusement calculé à partir de l'ensemble des données accumulées jusqu'à présent dans un entrepôt. Imaginons qu'une opération de rafraîchissement doive être effectuée. Une connaissance particulièrement intéressante peut être exhibée de la comparaison de ces deux ensembles de données : Quelles nouveautés le rafraîchissement a-t-il apportées ? Quelles tendances inconnues jusqu'alors apparaissent ? Ou au contraire, quelles sont les tendances avérées qui disparaissent ? Une connaissance similaire peut être exhibée chaque fois que deux data cubes sémantiquement semblables doivent être comparés. Par exemple, si deux ensembles de données sont collectés dans deux zones géographiques différentes pour plusieurs échantillons de populations ou plusieurs classes d'individus, il est possible de mettre en évidence les modifications de comportement entre deux populations, les contrastes entre leurs caractéristiques ou les écarts par rapport à un échantillon type. Afin de capturer les renversements de tendances dans les bases de données OLAP, nous proposons le concept de Cube

Émergent (Nedjar *et al.*, 2007a). Ce concept résulte du couplage de deux structures intéressantes, le data cube (Gray *et al.*, 1997) et les motifs émergents (Dong et Li, 1999), dans le même esprit que le Skycube (Pei *et al.*, 2006) qui combine le concept de data cube et la dominance de Pareto (Skyline (Börzsönyi *et al.*, 2001)). À partir des cubes de deux relations d'une base de données pourvues de dimensions (ou d'attributs catégories), le Cube Émergent rassemble tous les tuples satisfaisant une *double contrainte d'émergence* : la valeur de leur mesure est faible dans une relation (contrainte C_1) et significative dans l'autre (contrainte C_2). Nous introduisons le concept du Cube Émergent et proposons une représentation condensée à travers les bordures classiques $[L; U]$ (pour les bordures *L*ower et *U*pper) (Dong et Li, 1999; Raedt et Kramer, 2001; Bonchi et Lucchese, 2004). De plus, nous introduisons un nouveau couple de bordures appelé $]U^{\sharp}; U]$. Bien sûr cette représentation a les mêmes avantages que tout autre couple de bordures. Tout d'abord, elle offre une représentation condensée du Cube Émergent et permet de répondre efficacement à des requêtes telles que : « Est-ce que cette tendance croît significativement ? ». Elle fournit aussi un classifieur pour décider si un renversement de tendance est émergent ou pas dans le même esprit que dans (Dong et Li, 2005). Enfin, elle attire l'attention de l'utilisateur sur les tuples les plus intéressants à partir desquels une exploration fructueuse dans le cube peut débuter. En fait, cette nouvelle représentation accroît les différents avantages indiqués par rapport aux bordures classiques car elle apporte une réduction très significative de la taille de la représentation tout en véhiculant la même information. Nous établissons également le lien formel entre les deux représentations par bordures en nous appuyant sur le concept de transversal cubique (Casali *et al.*, 2003b). Ainsi il est possible, pour un utilisateur disposant des bordures classiques de dériver notre nouvelle représentation plus compacte. De plus, des algorithmes existants et réputés efficaces peuvent être mis à profit non seulement pour calculer la seconde représentation mais aussi pour en dériver la première. Après la caractérisation du Cube Émergent et de ses représentations, nous nous intéressons aux différentes approches permettant de calculer ces dernières. Dans un premier temps, nous étudions les requêtes Sql qui retournent le Cube Émergent partiellement ou complètement. Elles apportent une solution au problème sans aucun investissement de développement et peuvent être mises en œuvre facilement à condition toutefois que la taille des données initiales soit raisonnable et que l'on ne souhaite pas calculer les bordures. Lorsque cette double condition n'est pas remplie, les approches algorithmiques s'imposent. Pour éviter de développer un *n*ième algorithme, nous avons choisi de nous appuyer sur Buc (Beyer et Ramakrishnan, 1999) pour deux raisons essentielles : son extrême efficacité et sa capacité rare d'intégration dans un système relationnel. Dans un premier temps, nous adaptons le schéma algorithmique de Buc pour le calcul des Cubes Émergents avec l'algorithme appelé E-Idea. Ensuite nous proposons la première solution dédiée, F-Idea, permettant d'obtenir les représentations par bordures dans un contexte multidimensionnel. Cet algorithme représente la première alternative de calcul des bordures dans un contexte multidimensionnel sans utiliser les approches binaires telles que Max-Miner (Jr., 1998), Mafia (Burdick *et al.*, 2005), GenMax (Gouda et Zaki, 2005). Pour montrer la faisabilité de notre approche, nous menons des expérimentations permettant d'évaluer la taille des différentes représentations : celle du Cube Émergent en guise de référence, celle des bordures classiques et bien sûr celle de nos nouvelles bordures. Les jeux de données utilisés sont spécialement variés afin de tenir compte de différents cas de figure. Les résultats obtenus confirment nos attentes. Tout d'abord, les représentations qui se veulent concises le sont réellement. Le gain, pour la représentation la moins réduite, est plus qu'intéressant puisqu'il se situe toujours vers un facteur 10. Une deuxième partie des évaluations est relative aux temps d'exécution. Le Cube Émergent étant un nouveau concept, il n'existe pas de point de comparaison. À défaut, nous avons choisi de comparer les temps d'exécution pour calculer d'une part le Cube Émergent avec E-Idea et d'autre part le cube de données avec Buc. Concernant

le calcul des bordures, nous pouvons utiliser, comme point de comparaison, un des algorithmes de calcul de bordures dans un contexte binaire. Nous avons choisi l'algorithme MAFIA pour son efficacité et l'avons adapté aux données multidimensionnelles. Les résultats obtenus confortent l'intérêt de notre démarche puisque les représentations étudiées sont réduites et efficacement calculables.

Dans ce chapitre, nous définissons le Cube Émergent après un bref rappel du contexte dans lequel il s'inscrit. Nous étudions ensuite les bornes classiques $[L; U]$ puis nous introduisons une nouvelle alternative avec $]U^{\sharp}; U]$. Le lien entre les couples de bordures est aussi solidement établi. Enfin, nous procédons à des expérimentations comparatives qui établissent l'efficacité de notre plateforme algorithmique IDEA.

5.2 Concepts

Dans ce paragraphe, nous présentons la problématique étudiée en proposant le concept de Cube Émergent. Ces cubes capturent les tendances non significatives pour l'utilisateur (parce que leur mesure est en dessous d'un certain seuil) mais qui croissent suffisamment pour le devenir ou au contraire des tendances avérées qui s'essoufflent sans nécessairement disparaître. Elles sont d'un intérêt certain, par exemple dans l'analyse de flots de données, car elles révèlent des renversements de tendances. Dans une application Web où les flots continus de données reçues décrivent de manière détaillée la navigation des utilisateurs (Han *et al.*, 2005), connaître l'engouement (ou au contraire le désintérêt) pour telle ou telle URL est particulièrement important pour l'administrateur afin d'allouer au mieux les ressources disponibles selon des besoins réels et fluctuants.

La caractérisation du Cube Émergent s'inscrit dans le contexte plus général du treillis cube de la relation r : $CL(r)$ (Casali *et al.*, 2003a). Ce dernier, décrit dans le chapitre 3, est un espace de recherche bien adapté au calcul du data cube de r. Il organise les tuples, solutions possibles du problème, selon un ordre de généralisation / spécialisation, noté $\preceq_g$ (Lakshmanan *et al.*, 2002). Ces tuples sont structurés selon les attributs dimensions de r et ces derniers peuvent prendre la valeur ALL (Gray *et al.*, 1997). De plus, nous ajoutons à ces tuples un tuple virtuel ne contenant que des valeurs vides afin de fermer la structure. Tout tuple du treillis cube généralise le tuple de valeurs vides. Pour manipuler les tuples de $CL(r)$, l'opérateur + est défini. À partir d'un couple de tuples, il retourne le tuple le plus spécifique de $CL(r)$ qui généralise les deux opérandes.

Exemple 5.1 - Considérons la relation DOCUMENT (*cf.* table 5.1) donnant les quantités de livres vendues par Type, Ville, Éditeur, Langue et Année. Dans CL(DOCUMENT), considérons les ventes de Nouvelles à Marseille quels que soient l'éditeur, la langue et l'année *i.e* le tuple (Nouvelles, Marseille, ALL, ALL, ALL). Ce tuple est spécialisé par les deux tuples suivants de la relation : (Nouvelles, Marseille, Collins, Français, 2010) et (Nouvelles, Marseille, Hachette, Anglais, 2009). De plus, (Nouvelles, Marseille, ALL, ALL, ALL) $\preceq_g$ (Nouvelles, Marseille, Collins, Français, 2010) illustre l'ordre de généralisation entre tuples. Enfin, nous avons (Nouvelles, Marseille, Hachette, Anglais, 2009) + (Nouvelles, Marseille, Collins, Français, 2010) = (Nouvelles, Marseille, ALL, ALL, 2010).

Dans la suite du mémoire, nous considérons seulement les fonctions agrégatives étudiées par Pei *et al.* (2004), comme par exemple COUNT et SUM.

Définition 5.1 (Fonction de mesure) **-** Soit f une fonction agrégative, r une relation et t un tuple (ou cellule) de $CL(r)$. Nous notons $f_{val}(t,r)$ la valeur de la fonction agrégative f associée

TABLE 5.1 – Relation exemple DOCUMENT

Type	Ville	Éditeur	Langue	Année	Quantité
Nouvelles	Marseille	Collins	Français	2009	100
Nouvelles	Marseille	Collins	Français	2010	300
Nouvelles	Marseille	Hachette	Anglais	2009	100
Pédagogie	Marseille	Collins	Anglais	2010	300
Pédagogie	Marseille	Hachette	Anglais	2009	100
Pédagogie	Marseille	Hachette	Français	2010	300
Pédagogie	Paris	Hachette	Anglais	2010	300
Pédagogie	Paris	Hachette	Français	2009	100
Essai	Marseille	Hachette	Français	2010	100
Essai	Paris	Collins	Français	2010	200
Essai	Paris	Hachette	Français	2009	600
Essai	Paris	Hachette	Français	2010	200

au tuple t dans $CL(r)$.

Exemple 5.2 - Si nous considérons les ventes de Nouvelles à Marseille, pour n'importe quels Éditeur, Langue et Année, *i.e.* le tuple (Nouvelles, Marseille, ALL, ALL, ALL) de CL(DOCUMENT) nous avons : SUM_{val}((Nouvelles, Marseille, ALL, ALL, ALL), DOCUMENT) = 500.

Un tuple émergent de r_1 vers r_2 a une valeur de mesure faible dans r_1 alors qu'elle est significative dans r_2.

Définition 5.2 (Tuple Émergent) **-** un tuple $t \in CL(r_1 \cup r_2)$ est dit émergent de r_1 vers r_2 si et seulement s'il satisfait les deux contraintes C_1 et C_2 :

$$\begin{cases} f_{val}(t,r_1) < MinThreshold_1 \ (C_1) \\ f_{val}(t,r_2) \geq MinThreshold_2 \ (C_2) \end{cases}$$

Exemple 5.3 - Imaginons que nous cherchions les renversements de tendances s'étant produits entre les années 2009 et 2010 pour la vente de livres repertoriées dans la relation DOCUMENT (*cf.* Table 5.1). La relation DOCUMENT_1 (*cf.* Table 5.2) correspond aux livres vendus en 2009 et DOCUMENT_2 (*cf.* Table 5.3) sélectionne les tuples de DOCUMENT pour lesquels *Année* = 2010. Soit $MinThreshold_1 = 200$ le seuil pour la relation DOCUMENT_1 et $MinThreshold_2 = 200$ celui pour la relation DOCUMENT_2, le tuple t_1 =(Pédagogie, Marseille, ALL, ALL) est émergent de DOCUMENT_1 vers DOCUMENT_2 car $SUM_{val}(\text{t}_1, \text{DOCUMENT}_1) = 100$ ($< \text{MinThreshold}_1$) et $SUM_{val}(\text{t}_1, \text{DOCUMENT}_2) = 600$ ($\geq \text{MinThreshold}_2$). En revanche, le tuple t_2 = (Essai, Marseille, ALL, ALL) n'est pas émergent car $SUM_{val}(\text{t}_2, \text{DOCUMENT}_2) = 100$.

Dans l'objectif de pouvoir quantifier clairement un renversement de tendances, nous introduisons la mesure suivante.

Définition 5.3 (Taux d'Emergence) **-** Soit r_1 et r_2 deux relations unicompatibles, $t \in CL(r_1 \cup r_2)$ un tuple et f une des fonctions agrégatives citées. Le taux d'émergence de t de r_1 vers r_2, noté

TABLE 5.2 – Relation DOCUMENT$_1$ correspondant aux ventes de livres de l'année 2009

Type	Ville	Éditeur	Langue	Quantité
Nouvelles	Marseille	Collins	Français	100
Nouvelles	Marseille	Hachette	Anglais	100
Pédagogie	Paris	Hachette	Français	100
Essai	Paris	Hachette	Français	600
Pédagogie	Marseille	Hachette	Anglais	100

TABLE 5.3 – Relation exemple DOCUMENT$_2$ correspondant aux ventes de livres de l'année 2010

Type	Ville	Éditeur	Langue	Quantité
Nouvelles	Marseille	Collins	Français	300
Pédagogie	Marseille	Collins	Anglais	300
Pédagogie	Marseille	Hachette	Français	300
Pédagogie	Paris	Hachette	Anglais	300
Essai	Marseille	Hachette	Français	100
Essai	Paris	Hachette	Français	200
Essai	Paris	Collins	Français	200

$ER(t)$, est défini par :

$$ER(t) = \begin{cases} 0 \text{ si } f_{val}(t,r_1) = 0 \text{ et } f_{val}(t,r_2) = 0 \\ \infty \text{ si } f_{val}(t,r_1) = 0 \text{ et } f_{val}(t,r_2) \neq 0 \\ \dfrac{f_{val}(t,r_2)}{f_{val}(t,r_1)} \text{ sinon.} \end{cases}$$

Le taux d'émergence varie en fonction des seuils $MinSeuil_1$ et $MinSeuil_2$ fixés par l'utilisateur. Nous distinguons trois cas possibles :

1. Si $MinSeuil_1 = 0$ et $MinSeuil_2 > 0$, alors le taux d'émergence des tuples solutions a pour valeur l'infini. On cherche donc des tuples ne généralisant aucun tuple de r_1. Ces tuples émergents correspondent aux nouvelles tendances de r_1 vers r_2.
2. Si $MinSeuil_1 \leq MinSeuil_2$, alors le taux d'émergence des tuples solutions est supérieur ou égal à 1. Les tuples résultats sont alors positivement émergents. Ils sont caractérisés par une nette augmentation de leur mesure entre r_1 et r_2.
3. Si $MinSeuil_1 > MinSeuil_2$, alors le taux d'émergence appartient à l'intervalle [0,1[. Les tuples solutions sont négativement émergents (ou immergents). Ces tuples ont pour caractéristique une diminution de leur mesure de r_1 vers r_2.

Observons que lorsque le taux d'émergence est supérieur à 1, il caractérise des tendances significatives dans r_2 mais insuffisamment marquée dans r_1. Au contraire, quand le taux d'émergence est inférieur à 1, il souligne les tendances immergentes, pertinentes dans r_1 mais pas dans r_2.

Exemple 5.4 - À partir des relations DOCUMENT$_1$ et DOCUMENT$_2$, nous calculons $ER((\text{ Pédagogie, Marseille, ALL, ALL})) = 600/100$. Bien sûr plus le taux d'émergence est élevé, plus le renversement de tendance est marqué. Ainsi le tuple cité signifie un bond pour la vente de livres pédagogiques à Marseille entre DOCUMENT$_1$ et DOCUMENT$_2$.

Proposition 5.1 - Soit $MinRatio = \frac{MinThreshold_2}{MinThreshold_1}$,

$$\forall t \in \text{EC}(r_1, r_2), \text{ nous avons } ER(t) > MinRatio$$

Démonstration.

$$f_{val}(t,r_1) \leq MinThreshold_1 \Rightarrow \frac{1}{f_{val}(t,r_1)} \geq \frac{1}{MinThreshold_1} \text{ou} f_{val}(t,r_2) \geq MinThreshold_2$$
$$\Rightarrow \frac{f_{val}(t,r_2)}{f_{val}(t,r_1)} \geq \frac{MinThreshold_2}{MinThreshold_1} \Rightarrow ER(t) \geq MinRatio$$

□

La conséquence de la proposition est que les tuples émergents ont un taux d'émergence *« intéressant »*. Cela constitue une première aide pour l'utilisateur afin de bien spécifier ses contraintes d'émergence. Par exemple considérons $MinThreshold_1 = 300$ et $MinThreshold_2 = 500$, alors tous les tuples émergents ont un taux d'émergence supérieur à $5/3$.

Définition 5.4 (Cube Émergent) **-** Nous appelons Cube Émergent l'ensemble de tous les tuples de $CL(r_1 \cup r_2)$ émergents de r_1 vers r_2. Le Cube Émergent, noté $\text{EC}(r_1, r_2)$, est défini par :

$$\text{EC}(r_1, r_2) = \{t \in CL(r_1 \cup r_2) \mid C_1(t) \wedge C_2(t)\}$$

avec $C_1(t) = f_{val}(t,r_1) < MinThreshold_1$ et $C_2(t) = f_{val}(t,r_2) \geq MinThreshold_2$.

Exemple 5.5 - La table 5.4 donne le Cube Émergent de la relation DOCUMENT$_1$ vers la relation DOCUMENT$_2$ avec les seuils $MinThreshold_1 = 200$ et $MinThreshold_2 = 200$.

Les coûts de calcul et de stockage du Cube Émergent dépendent du nombre de tuples émergents. Ce nombre est borné par la taille de l'espace de recherche (l'ensemble des tuples possibles). L'espace de recherche dans lequel s'inscrit le Cube Émergent est lié aux relations en entrée ($CL(r_1 \cup r_2)$). Le second paramètre qui va influer sur le nombre de solutions est la double contrainte d'émergence. C'est l'utilisateur qui fixe cette contrainte en donnant les seuils ($MinThreshold_1$ et $MinThreshold_2$) Plus la contrainte est forte plus le Cube Émergent associé est réduit. Le cas extrême étant atteint quand $MinThreshold_1 = 0^+$ et $MinThreshold_2 = +\infty$. Avec de tels seuils, aucun tuple n'est émergent et le résultat est toujours l'ensemble vide. Inversement, lorsque la contrainte est plus lâche moins de tuples sont filtrés. Le Cube Émergent est donc plus volumineux. La contrainte d'émergence la plus faible possible est obtenue pour $MinThreshold_1 = +\infty$ et $MinThreshold_2 = 0^+$. Dans ce cas, pour qu'un tuple soit émergent il faut que la valeur de sa mesure dans r_2 soit différente de 0. Le tuple doit donc appartenir au cube de données de r_2. Ainsi, dans le pire des cas (du point de vue de sa taille), le calcul du Cube Émergent se ramène au calcul du cube de données de r_2.

Cette discussion concernant l'impact des seuils sur la taille du Cube Émergent met en évidence l'importance du calibrage des contraintes pour obtenir un Cube Émergent d'un volume réellement exploitable. En ayant identifié les cas extrêmes, nous avons vu que le problème de calcul d'un Cube Émergent est au moins aussi difficile (du point de vue de la complexité) que

TABLE 5.4 – Cube Émergent de DOCUMENT_1 vers DOCUMENT_2

Tuple Émergent	ER
('ALL' , 'ALL' , 'ALL' , Anglais)	3
('ALL' , 'ALL' , Collins , 'ALL')	8
('ALL' , 'ALL' , Collins , Anglais)	∞
('ALL' , 'ALL' , Collins , Français)	5
('ALL' , 'ALL' , Hachette, Anglais)	1.5
('ALL' , Paris , 'ALL' , Anglais)	∞
('ALL' , Paris , Hachette, Anglais)	∞
('ALL' , Marseille, 'ALL' , Anglais)	1.5
('ALL' , Marseille, 'ALL' , Français)	7
('ALL' , Marseille, Collins , 'ALL')	6
('ALL' , Marseille, Collins , Anglais)	∞
('ALL' , Marseille, Collins , Français)	3
('ALL' , Marseille, Hachette, 'ALL')	2
('ALL' , Marseille, Hachette, Français)	∞
(Nouvelles, 'ALL' , 'ALL' , 'ALL')	1.5
(Nouvelles, 'ALL' , 'ALL' , Français)	3
(Nouvelles, 'ALL' , Collins , 'ALL')	3
(Nouvelles, 'ALL' , Collins , Français)	3
(Nouvelles, Marseille, 'ALL' , 'ALL')	1.5
(Nouvelles, Marseille, 'ALL' , Français)	3
(Nouvelles, Marseille, Collins , 'ALL')	3
(Nouvelles, Marseille, Collins , Français)	3
(Pédagogie, 'ALL' , 'ALL' , 'ALL')	4.5
(Pédagogie, 'ALL' , 'ALL' , Anglais)	6
(Pédagogie, 'ALL' , 'ALL' , Français)	3
(Pédagogie, 'ALL' , Collins , 'ALL')	∞
(Pédagogie, 'ALL' , Collins , Anglais)	∞
(Pédagogie, 'ALL' , Hachette, 'ALL')	3
(Pédagogie, 'ALL' , Hachette, Anglais)	3
(Pédagogie, 'ALL' , Hachette, Français)	3
(Pédagogie, Paris , 'ALL' , 'ALL')	3
(Pédagogie, Paris , 'ALL' , Anglais)	∞
(Pédagogie, Paris , Hachette, 'ALL')	3
(Pédagogie, Paris , Hachette, Anglais)	∞
(Pédagogie, Marseille, 'ALL' , 'ALL')	6
(Pédagogie, Marseille, 'ALL' , Anglais)	3
(Pédagogie, Marseille, 'ALL' , Français)	∞
(Pédagogie, Marseille, Collins , 'ALL')	∞
(Pédagogie, Marseille, Collins , Anglais)	∞
(Pédagogie, Marseille, Hachette, 'ALL')	3
(Pédagogie, Marseille, Hachette, Français)	∞

le problème de calcul d'un cube de données (*c.f* paragraphe 2.3 p.22). La spécification des contraintes est donc un élément clef pour que l'utilisateur puisse appréhender puis exploiter les résultats.

5.3 Bordures pour le Cube Émergent

Les bordures offrent une représentation des Cubes Émergents et ont des avantages supplémentaires. Tout d'abord, les renversements de tendances peuvent être isolés très efficacement et à un moindre coût car il n'est pas nécessaire de calculer et de stocker les deux cubes sous-jacents. Ainsi, nous économisons temps d'exécution et espace de stockage. De plus, nous pouvons répondre quasi immédiatement à des requêtes telles que : « Est-ce que cette tendance est émergente ? ». Afin de gérer systématiquement ce type de requête, nous pouvons concevoir des classifieurs efficaces pour décider si une tendance est émergente ou pas. De plus, en se focalisant sur des tuples particuliers, les bordures des Cubes Émergents offrent une point de départ intéressant pour la navigation à travers un cube de données. Enfin, nous tenons compte des bordures décrites afin de caractériser la taille exacte des Cubes Émergents.

Dans ce paragraphe, nous présentons les bordures classiques $[L;U]$ et introduisons une nouvelle représentation condensée : les bordures $]U^\sharp;U]$. Finalement nous établissons le lien entre les bordures $U^\sharp$ et L en utilisant le concept de transversal cubique (Casali *et al.*, 2003b).

5.3.1 Bordures *[L ;U]*

Étant donnée la définition des contraintes d'émergence C_1 et C_2, qui sont respectivement monotone et antimonotone, pour chaque tuple t satisfaisant la contrainte C_1, nous savons que tous les tuples plus spécifiques que t satisfont aussi la contrainte C_1. Cela implique que les tuples les moins spécifiques (les minimaux selon $\preceq_g$) satisfaisant la contrainte C_1 sont les plus importants car ils délimitent l'espace des solutions. Inversement, dans le cas de la contrainte C_2, ce sont les tuples les plus spécifiques (les maximaux selon $\preceq_g$) qui sont les plus importants. Ces deux ensembles de tuples forment les frontières du Cube Émergent.

Définition 5.5 (Bordures *[L ;U]*) **-** Le Cube Émergent peut être représenté par les bordures : U qui englobe les tuples émergents maximaux et L qui contient tous les tuples émergents minimaux selon l'ordre de généralisation $\preceq_g$.

$$\begin{cases} L = \min_{\preceq_g}(\{t \in CL(r_1 \cup r_2) \mid C_1(t) \wedge C_2(t)\}) \\ U = \max_{\preceq_g}(\{t \in CL(r_1 \cup r_2) \mid C_1(t) \wedge C_2(t)\}) \end{cases}$$

Proposition 5.2 - Les bordures $[L;U]$ sont une représentation du Cube Émergent.

Démonstration. $\forall\ t \in CL(r_1 \cup r_2)$, t est émergent de r_1 vers r_2 si et seulement si $C_1(t) \wedge C_2(t)$. Or par définition des contraintes, C_1 est monotone et C_2 antimonotone. Donc, $\exists l \in L$ et $\exists u \in U$ tels que $l \preceq_g t$ et $t \preceq_g u$. En d'autres termes, t est émergent si et seulement s'il est compris dans l'intervalle $[L;U]$. □

La figure 5.1 offre une illustration des bordures $[L;U]$ en les situant dans le treillis cube de $r_1 \cup r_2$. La partie hachurée correspond au Cube Émergent. Comme le montre cette figure, les bordures $[L;U]$ forment les frontières du Cube Émergent.

Exemple 5.6 - Avec nos relations exemples DOCUMENT_1 et DOCUMENT_2, la table 5.5 donne les bordures $[L;U]$ pour le Cube Émergent, en conservant une valeur similaire pour les seuils

FIGURE 5.1 – Illustration de la proposition 5.2

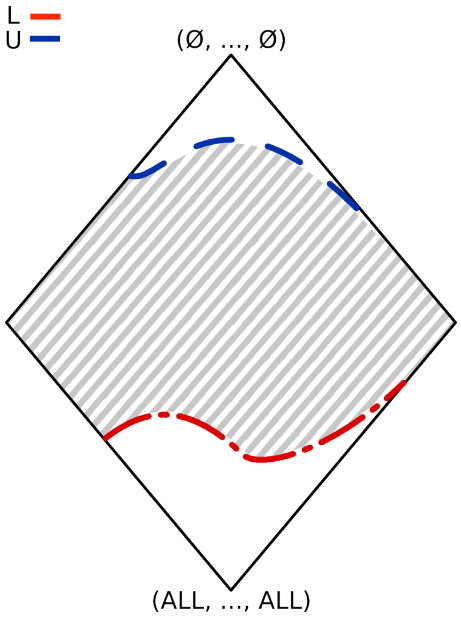


TABLE 5.5 – Bordures $[L; U]$ du Cube Émergent

U	(Nouvelles, Marseille, Collins, Français)
	(Pédagogie, Marseille, Collins, Anglais)
	(Pédagogie, Paris, Hachette, Anglais)
	(Pédagogie, Marseille, Hachette, Français)
L	(Nouvelles, ALL, ALL, ALL)
	(Pédagogie, ALL, ALL, ALL)
	(ALL, ALL, Collins, ALL)
	(ALL, ALL, ALL, Anglais)
	(ALL, Marseille, ALL, Français)
	(ALL, Marseille, Hachette, ALL)

($MinThreshold_1 = 200$, $MinThreshold_2 = 200$). À partir des bordures, nous savons que le tuple (Nouvelles, ALL, Collins, Français) est émergent car il spécialise le tuple (Nouvelles, ALL, ALL, ALL) qui appartient à la bordure L tout en généralisant le tuple (Nouvelles, Marseille, Collins, Français) de la bordure U. De plus, le tuple (ALL, Paris, Hachette, ALL) n'est pas émergent. Même s'il généralise le tuple (Pédagogie, Paris, Hachette, Anglais) qui appartient à la bordure U, il ne spécialise aucun tuple de la bordure L.

La figure 5.2 représente les bordures $[L;U]$ du Cube Émergent ($MinThreshold_1 = 200$, $MinThreshold_2 = 200$) sur le treillis cube de la relation exemple. Pour raison de lisibilité du dessin, nous restreignons le calcul de ce cube aux dimensions {Type, Ville, Éditeur} ; '*' est utilisé à la place de ALL ; les valeurs des différentes dimensions sont codées comme suit :

Type		Ville		Éditeur	
Nouvelles	= N	Marseille	= 1	Hachette	= H
Pédagogie	= P	Paris	= 2	Collins	= C
Essai	= E				

Le principe de représentation d'un ensemble convexe par des bordures Min/Max (comme $[L;U]$) est classique en apprentissage automatique (Raedt et Kramer, 2001) mais n'a encore que très peu été étudié pour la représentation des cubes de données contraints (Casali *et al.*, 2007). Pourtant comme nous allons le voir dans les expérimentations, ces bordures offrent un résumé qui est à la fois bien plus réduit et beaucoup moins coûteux à calculer que les Cubes Émergents qu'elles représentent.

5.3.2 Bordures $]U^\sharp ; U]$

Les bordures $[L;U]$ sont des bordures Min/Max, elles résument le Cube Émergent en utilisant ses frontières. En utilisant le même principe, nous allons dans ce paragraphe introduire une nouvelle représentation condensée : les bordures $]U^\sharp;U]$. Cette représentation remplace la bordure Min (L) par une nouvelle bordure $U^\sharp$. Celle-ci s'appuie sur les tuples maximaux satisfaisant la contrainte C_2 (tuples significatifs dans r_2) mais sans vérifier C_1 (donc ils sont aussi significatifs dans r_1 et ne sont pas émergents).

De surcroît nous apportons une optimisation de l'espace de recherche en considérant uniquement le treillis cube de r_2 ($CL(r_2)$), à la place de l'intégralité de l'espace de recherche (*i.e* treillis cube de $r_1 \cup r_2$). Effectivement, par définition (cf. Contrainte C_2) tout tuple émergent est nécessairement un tuple de $CL(r_2)$.

Définition 5.6 (Bordures $]U^\sharp ;U]$) **-** Le Cube Émergent peut être représenté par deux bordures : U (*cf. Définition 5.3*) et $U^\sharp$ contenant tous les tuples maximaux ne satisfaisant pas la contrainte monotone C_1 mais vérifiant la contrainte C_2. Nous avons donc :

$$\begin{cases} U^\sharp = \max_{\preceq_g}(\{t \in CL(r_2) \mid \neg C_1(t) \wedge C_2(t)\}) \\ U = \max_{\preceq_g}(\{t \in CL(r_2) \mid C_1(t) \wedge C_2(t)\}) \end{cases}$$

En considérant l'espace de recherche $CL(r_2)$, nous devons vérifier la double contrainte suivante pour les tuples de $U^\sharp$: ils ne satisfont pas C_1 mais C_2 est vérifiée.

Exemple 5.7 - Avec nos relations exemples Document_1 et Document_2, la table 5.6 présente les bordures $]U^\sharp;U]$ du Cube Émergent de Document_1 vers Document_2. Par exemple, le

FIGURE 5.2 – Représentation du Cube Émergent et de ses bordures $[L; U]$

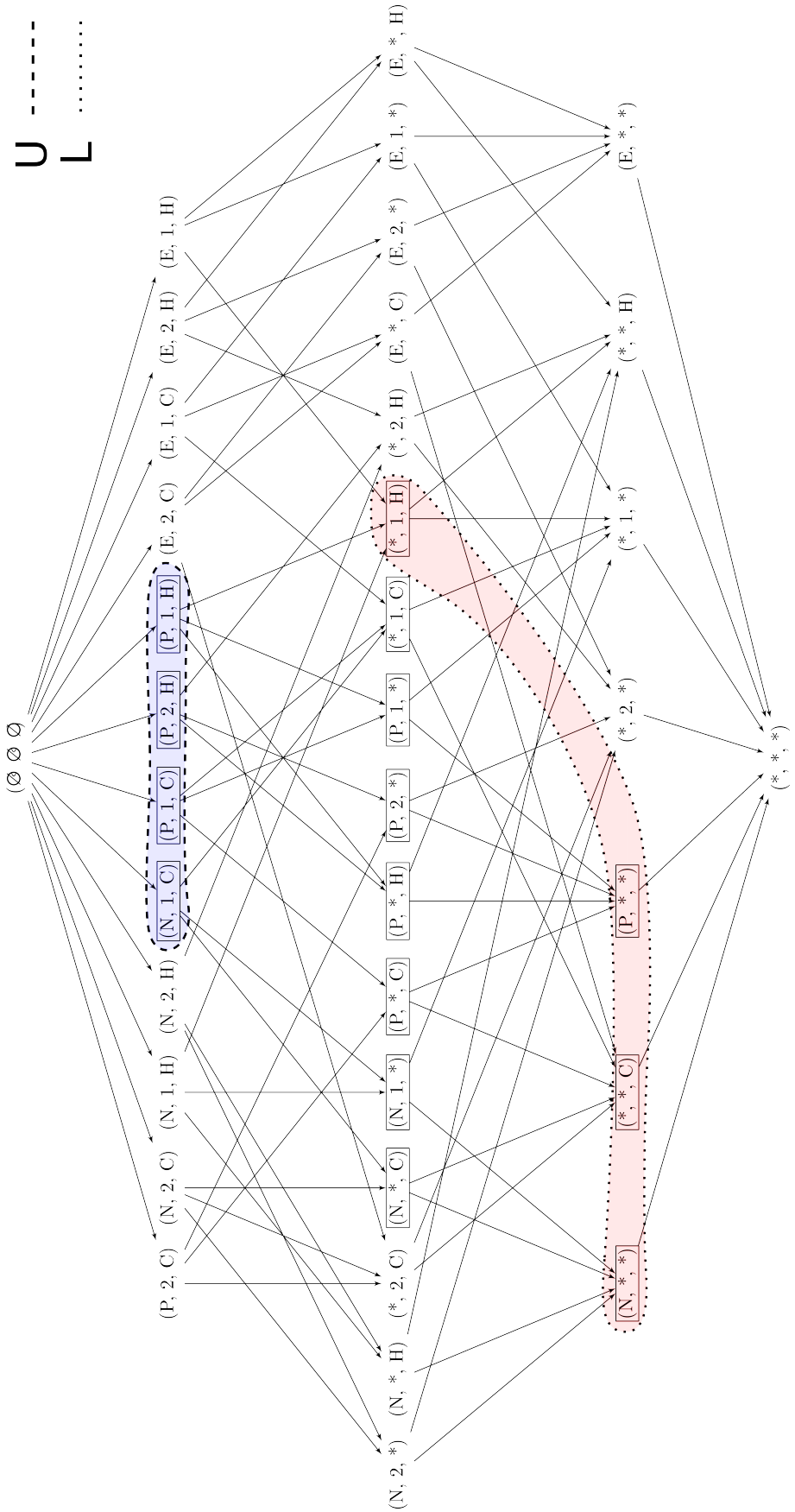


TABLE 5.6 – Bordures $]U^\sharp; U]$ du Cube Émergent

U	(Nouvelles, Marseille, Collins, Français) (Pédagogie, Marseille, Collins, Anglais) (Pédagogie, Paris, Hachette, Anglais) (Pédagogie, Marseille, Hachette, Français)
$U^\sharp$	(ALL, Paris, Hachette, ALL) (ALL, Marseille, ALL, ALL) (ALL, ALL, ALL, Français)

tuple (ALL, Paris, Hachette, ALL) appartient à $U^\sharp$ car le nombre de livres vendus à Paris pour l'éditeur Hachette est supérieur aux seuils donnés non seulement dans r_2 (C_2) mais aussi dans r_1 ($\neg C_1$) et tous les tuples spécialisant (ALL, Paris, Hachette, ALL) ne satisfont pas la contrainte C_1 mais vérifient la contrainte C_2. C'est pourquoi le tuple cité est maximal.

Avec la proposition suivante, nous disposons d'un mécanisme simple pour savoir si un tuple est émergent ou pas en utilisant les bordures $]U^\sharp; U]$.

Proposition 5.3 - Les bordures Max/Max sont un résumé du cube émergent : $\forall\ t \in CL(r_2)$, t est émergent de r_1 vers $r_2 \Leftrightarrow \forall\ l \in U^\sharp, l \not\preceq_g t$ et $\exists\ u \in U$ tel que $t \preceq_g u$. Autrement dit, un tuple t est émergent si et seulement s'il est dans l'intervalle $]U^\sharp\,;\,U]$.

Démonstration. D'après la définition 5.4, le cube émergent est l'ensemble des tuples satisfaisant la conjonction de contraintes monotone/antimonotone C_1 et C_2. Donc un tuple t est émergent si et seulement si t ne satisfait pas la contrainte antimonotone $\neg C_1$ et satisfait la contrainte antimonotone C_2. Donc d'après le théorème 3.7, t est émergent si et seulement si $\nexists t^{'} \in U^*$ tel que $t \preceq_g t^{'}$ ($C_1(t)$) et $\exists u \in U$ tel que $t \preceq_g u$ ($C_2(t)$). □

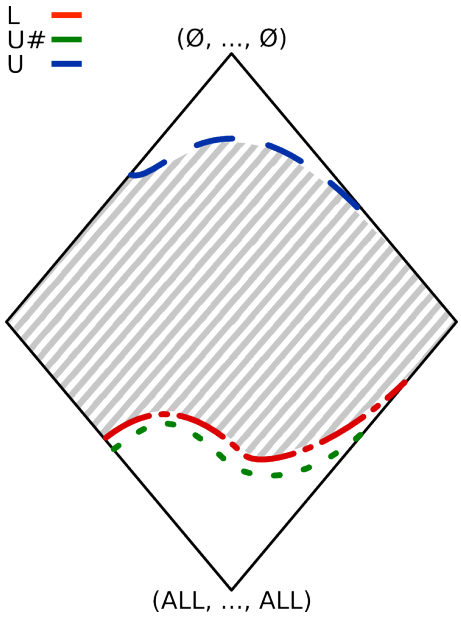
Dans la figure 5.3 nous ajoutons la bordure $U^\sharp$ qui se situe juste en dessous de la bordure L.

Exemple 5.8 - Le tuple (Nouvelles, ALL, Collins, Français) est émergent car il généralise le tuple (Nouvelles, Marseille, Collins, Français) qui appartient à la bordure U et ne généralise aucun tuple de la bordure $U^\sharp$. De plus le tuple (ALL, Paris, ALL, ALL) n'est pas émergent car il généralise le tuple (ALL, Paris, Hachette, ALL) de la bordure $U^\sharp$.

La figure 5.4 représente le couple de bordures $[U^\sharp; U]$ du Cube Émergent ($MinThreshold_1 = 200$, $MinThreshold_2 = 200$) sur le treillis cube de la relation exemple. Pour raison de lisibilité du dessin, nous restreignons le calcul de ce cube aux dimensions {Type, Ville, Éditeur} et utilisons le même codage que dans l'exemple 5.6.

Dans le paragraphe précédent, nous avons souligné le fait que le concept de bordures Min/Max est classique en fouille de données binaires mais nouveau pour la représentation de cubes de données. Les bordures Max/Max (comme $]U^\sharp; U]$) sont, à notre connaissance, complètement nouvelles et n'ont donc jamais fait l'objet d'expérimentation. Pourtant comme nous allons le voir à plusieurs reprises dans ce mémoire, elles ont, par rapport aux bordures $[L; U]$, à la fois des avantages directs (meilleur temps de calcul et taille encore plus réduite) et des avantages indirects (caractérisation exacte de la cardinalité des Cubes Émergents). Les points forts de ce concept en font un candidat particulièrement pertinent pour la représentation des Cubes Émergents ou plus généralement des cubes contraints.

FIGURE 5.3 – Illustration de la proposition 5.3



5.3.3 Lien entre les bordures

Dans le contexte de l'extraction de motifs fréquents, Mannila et Toivonen (1997) établissent le lien entre les bordures positive et négative. Dans un esprit similaire, nous caractérisons le lien entre les bordures $U^\sharp$ et L. La caractérisation est basée sur le concept de transversal cubique (Casali *et al.*, 2003b) et vise à donner un solide fondement aux bordures proposées ainsi qu'aux résultats énoncés. Ce lien peut être exploité pour tirer profit d'algorithmes existants et efficaces (Eiter et Gottlob, 1995; Gunopulos *et al.*, 1997; Casali *et al.*, 2003b) pour calculer la bordure L en utilisant les transversaux cubiques (*cf.* paragraphe 3.2.3 p.54). À travers la proposition suivante, nous établissons le lien entre la bordure L et la bordure $U^\sharp$.

Proposition 5.4 - $L = ccTr(U^\sharp, U)$. De plus, $\forall\ t \in CL(r_1 \cup r_2)$, t est émergent si et seulement si t est un transversal cubique de $U^\sharp$ et $\exists\ u \in U$ tel que $t \preceq_g u$.

Démonstration.

$$\begin{aligned} ccTr(U^\sharp, U) &= \\ &= ccTr(\max_{\preceq_g}(\{t \in CL(r_2) \mid \neg C_1(t) \wedge C_2(t)\}), U) \\ &= ccTr(\{t \in CL(r_2) \mid \neg C_1(t) \wedge C_2(t)\}, U) \\ &= \min_{\preceq_g}(\{t \in CL(r_2) \mid \neg(\neg C_1(t) \wedge C_2(t)) \wedge C_2(t)\}) \\ &= \min_{\preceq_g}(\{t \in CL(r_2) \mid (C_1(t) \wedge C_2(t)) \vee (C_2(t) \wedge \neg C_2(t))\}) \end{aligned}$$

Comme la deuxième contrainte $(C_2(t) \wedge \neg C_2(t))$ ne peut jamais être satisfaite, nous avons :

$$\begin{aligned} ccTr(U^\sharp, U) &= \min_{\preceq_g}(\{t \in CL(r_2) \mid (C_1(t) \wedge C_2(t))\} \\ &= L \end{aligned}$$

□

À partir d'un couple de bordures, il est facile d'obtenir l'autre couple de bordures en appliquant la proposition donnée ci-avant.

FIGURE 5.4 – Représentation du Cube Émergent et de ses bordures $[U^\sharp; U]$

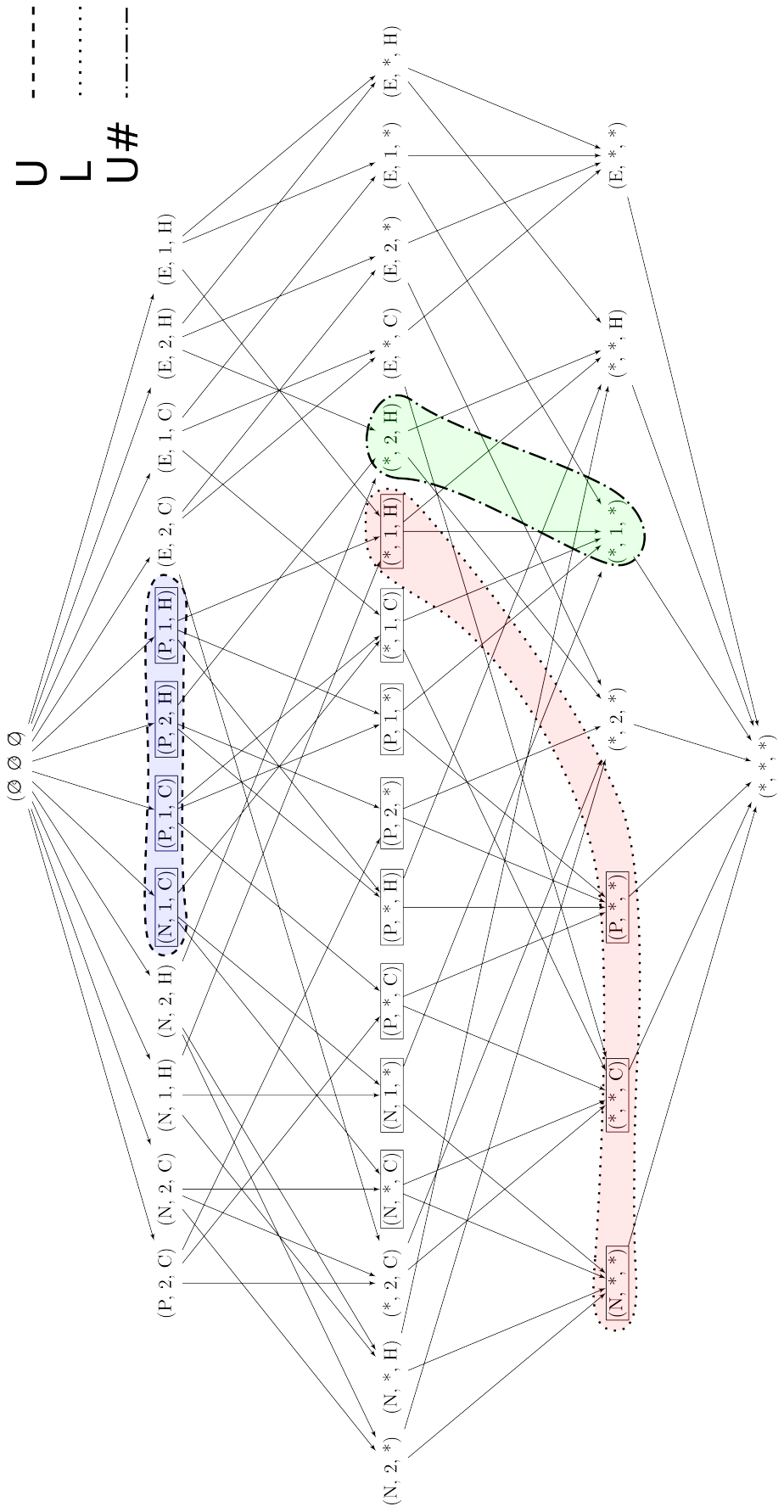


TABLE 5.7 – Transversaux cubiques minimaux de $U^\sharp$

$ccTr(U^\sharp)$
(Nouvelles, ALL, ALL, ALL)
(Pédagogie, ALL, ALL, ALL)
(Essai, ALL, ALL, ALL)
(ALL, ALL, Collins, ALL)
(ALL, ALL, ALL, Anglais)
(ALL, Marseille, Hachette, ALL)
(ALL, Marseille, ALL, Français)
(ALL, Paris, ALL, Français)
(ALL, ALL, Hachette, Français)

Exemple 5.9 - Si nous éliminons les tuples de la table 5.7 qui ne généralise aucun tuple de U ($cf.$ Table 5.5), nous obtenons la bordure L donnée dans la même table. Par exemple, le tuple (Essai, ALL, ALL, ALL) est éliminé de $ccTr(U^\sharp)$ car il ne généralise aucun tuple de U.

En fouille de données, les bordures couramment utilisées pour représenter un ensemble de solutions ayant la structure d'espace convexe sont les bordures Min/Max. Les bordures Max/Max n'ont, à notre connaissance, jamais été proposées ni expérimentées dans la littérature pourtant elles ont des avantages indéniables :

(i) l'étude expérimentale que nous avons menée pour valider cette nouvelle représentation ($cf.$ section 5.5) montre un gain très significatif de la taille de la bordure $U^\sharp$ par rapport à celle de L ;

(ii) le calcul des bordures est basé principalement sur des algorithmes de calcul des maximaux. Dériver la bordure $U^\sharp$ est une étape préliminaire au calcul de la bordure L. Ainsi, en utilisant les bordures Max/Max, non seulement nous économisons de la mémoire mais nous gagnons aussi le temps de calcul des transversaux cubiques pour calculer L ($cf.$ proposition 5.4).

Il faut noter que, dans un contexte de représentation sans perte d'information, les éléments de $U^\sharp$ sont des fermés cubiques non émergents. Cette représentation est donc homogène avec l'approche par fermeture cubique qui est introduite dans le chapitre 7.

5.3.4 Étude de la complexité

Le calcul des représentations par bordures du Cube Émergent se ramène à deux problèmes classiques en fouille de données binaires. Premièrement, le calcul des bordures Max (U et $U^\sharp$) peut se comparer à l'extraction des motifs maximaux fréquents dans une base de données binaires. Deuxièmement, le calcul de la bordure Min (L) à partir de $U^\sharp$ revient à un calcul des minimaux transversaux. Dans ce paragraphe, nous étudions la complexité de ces deux problèmes. Comme nous allons le voir, la complexité intrinsèque de ces problèmes est exponentielle par rapport au nombre de dimensions de la relation de base. Malgré ce résultat négatif, les relations utilisées en pratique ayant généralement un nombre de dimensions raisonnable ($0 < |Dim(r)| \leq 20$), les algorithmes de calcul sont performants.

Complexité de l'extraction des motifs maximaux fréquents

L'extraction des motifs δ-fréquents (*i.e.* trouver tous les tuples $t \in \mathcal{P}(\mathcal{I})$ tels que $freq(t) > \delta$) est un problème central pour la fouille de données. Le problème de décision associé est extrêmement simple à résoudre (il est linéaire dans le nombre d'éléments de $\mathcal{I}$). Pourtant, l'extraction de tous les motifs fréquents est un problème très difficile du point de vue de la complexité.

Extraire l'ensemble des motifs maximaux fréquents revient à les énumérer un par un. L'énumération est algorithmiquement au moins aussi difficile que le comptage (*i.e.* si on sait énumérer polynomialement les solutions, alors évidemment on sait aussi compter les solutions polynomialement). Dans l'exemple suivant, nous montrons que le nombre de motifs fréquents peut être exponentiel.

Exemple 5.10 - Soit $X = \{x_1,x_2,...,x_{2n-1},x_{2n}\}$ un ensemble de cardinalité $2n$. Nous allons construire la base de données binaires $\mathcal{D}$ avec $2n$ motifs, $t_1,t_2,...,t_{2n-1},t_{2n}$ tels que $t_i = X - \{x_i\}$ pour $1 \leq i \leq 2n$. Nous pouvons affirmer qu'il y a exactement $\binom{2n}{n}$ motifs n-fréquents. pour n'importe quel motif $I \subseteq X$, $\mathcal{D}(I) = \{t_i \mid x_i \notin I, x_i \in I\}$ si $|I| = k$ alors $freq_{\mathcal{D}}(I)$ est exactement $2n - k$. Un motif I est n-fréquent si et seulement si $|I| = n$. Le nombre de motifs de taille n est clairement $\binom{n}{2n}$. Il y a bien $\binom{n}{2n} = \frac{(2n)!}{n!.n!} \geq 2^n$ motifs n-fréquents. Le nombre de maximaux n-fréquents est exponentiel dans la taille de $\mathcal{D}$.

L'ensemble des motifs maximaux fréquents est une antichaîne. D'après le théorème de Sperner la taille de la plus grande antichaîne de $\langle \mathcal{P}(E), \subseteq \rangle$ avec $|E| = n$ est $\binom{n}{\lfloor n/2 \rfloor}$. Le nombre de motifs maximaux est donc (comme l'illustre l'exemple) exponentiel dans la taille des données.

Guizhen Yang démontre dans (Yang, 2004) que compter les motifs maximaux fréquents est #P-complet, le problème d'énumération est donc NP-difficile.

Complexité du calcul des minimaux transversaux

Avant de donner la complexité du problème de calcul des minimaux transversaux, nous allons donner quelques définitions utiles.

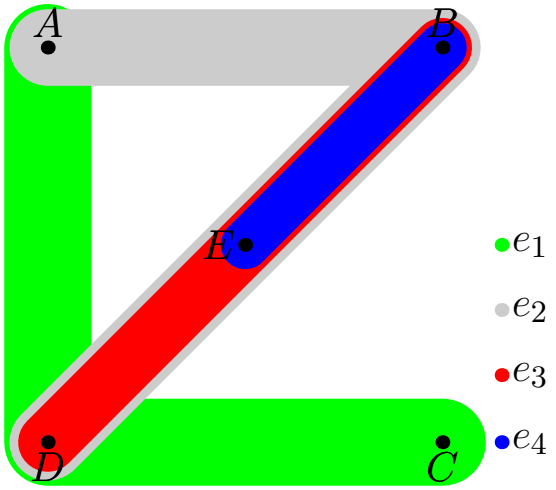
Définition 5.7 (Hypergraphe Dual) **-** Soit un hypergraphe $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ avec $\mathcal{V} = \{v_1, ..., v_n\}$ et $\mathcal{E} = \{e_1, ..., e_n\}$. L'hypergraphe dual $\mathcal{D} = dual(\mathcal{H}) = (\mathcal{V}',\mathcal{E}')$ de $\mathcal{H}$ est tel que à chaque arête $e_i \in \mathcal{E}$ correspond un sommet portant le même nom dans $\mathcal{V}'$ et à chaque sommet $v_i \in \mathcal{V}$ correspond une arête d_i de $\mathcal{D}$ telle que $d_i = \{e_i \mid e_i$ est adjacente au sommet $v_i\}$.

Si Tr est un transversal de $\mathcal{H}$ alors pour tout $e_i \in \mathcal{E}$, on a $e_i \cap \mathcal{S} \neq \emptyset$. En d'autres termes, chaque arête doit être *« touchée »* par un sommet de Tr. Dans le dual $\mathcal{D}$ de $\mathcal{H}$ on a donc chaque sommet de $\mathcal{D}$ couvert par une arête de $dual(Tr)$. De ce fait, un transversal est une couverture par des arêtes du dual.

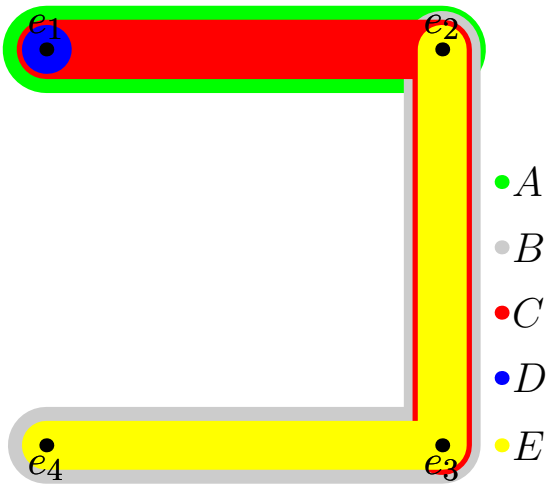
Exemple 5.11 -
Soit un hypergraphe $\mathcal{H} = (\mathcal{V},\mathcal{E})$ tel que $\mathcal{V} = \{A, B, C, D, E\}$ et $\mathcal{E} = \{e_1, e_2, e_3, e_4\}$ avec $e_1 = \{A, C, D\}$, $e_2 = \{A, B, C, E\}$, $e_3 = \{B, C, E\}$, $e_4 = \{B, E\}$ (*c.f.* figure 5.5a). Le dual de $\mathcal{H}$ est l'hypergraphe $\mathcal{D} = (\mathcal{V}',\mathcal{E}')$ tel que $\mathcal{V}' = \{e_1, e_2, e_3, e_4\}$ et $\mathcal{E}' = \{A, B, C, D, E\}$ avec $A = \{e_1, e_2\}$, $B = \{e_2, e_3, e_4\}$, $C = \{e_1, e_2, e_3\}$, $D = \{e_1\}$, $E = \{e_2, e_3, e_4\}$ (*c.f.* figure 5.5b).

$$Tr(H) = \{\{A, B\}, \{A, E\}, \{B, C\}, \{B, D\}, \{C, E\}, \{D, E\}\}$$

$\{A, B\}$ est un transversal et comme on le voit sur la figure 5.5b c'est aussi une couverture par des arêtes du dual.



(a) Représentation graphique de l'hypergraphe $\mathcal{H}$



(b) Représentation graphique du dual de l'hypergraphe $\mathcal{H}$

En d'autres termes, *Minimal Transversal* est équivalent à *Minimal Set-Cover*. Ce dernier étant NP-Complet, *Minimal Transversal* est aussi NP-Complet.

Comme pour les motifs maximaux fréquents, nous allons donner un exemple montrant que la taille de l'ensemble des minimaux transversaux n'est pas polynomiale dans la taille des données.

Exemple 5.12 -
Soit un hypergraphe $\mathcal{H}_n = (\mathcal{V}_n,\mathcal{E}_n)$, $\mathcal{V}_n = \{v_1, v_2, ..., v_{2n-1}, v_{2n}\}$ $\mathcal{E}_n = \{\{v_{2i-1},v_{2i}\} \mid 1 \leq i \leq n\}$. L'ensemble des transversaux minimaux de $\mathcal{H}$ est l'hypergraphe $Tr(\mathcal{H}) = \{\{x_1,...,x_n\} \mid x_i \in \{v_{2i},v_{2i-1}\}, 1 \leq i \leq n\}$. La taille de cet ensemble est $|Tr(\mathcal{H})| = 2^n$. Le nombre de transversaux est donc lui aussi exponentiel dans la taille des données.

Dans le cas des transversaux cubiques on connaît une borne($|\mathcal{D}|$) pour la taille des arêtes. La recherche des transversaux est dans ce cas polynomiale (Eiter et Gottlob, 1995). Malgré les résultats négatifs sur la complexité du problème général, le calcul des transversaux cubiques est un problème abordable.

5.4 Calcul du Cube Émergent et de ses bordures

Dans ce paragraphe nous étudions les méthodes permettant d'obtenir le Cube Émergent ainsi que ses bordures. Nous les abordons avec un double point de vue, bases de données et algorithmique. Pour l'approche base de données, nous tirons profit des possibilités de SQL et spécifions différentes requêtes de calcul du Cube Émergent. Outre leur intérêt pédagogique, elles peuvent facilement être mises en œuvre lorsque les volumes de données traités sont de taille raisonnable. En revanche, c'est l'efficacité que nous visons avec l'approche algorithmique.

5.4.1 Expression Sql du Cube Émergent

Dans l'objectif de donner une expression SQL la plus intuitive possible du Cube Émergent nous allons dans un premier temps donner la requête SQL calculant uniquement l'ensemble des tuples émergents. Nous utilisons dans notre requête l'opérateur *Cube by* ou *Group by cube* (implémenté dans plusieurs SGBD), pour calculer les cubes de données de r_1 et de r_2. Une fois

ces deux cubes calculés, nous en faisons la différence. Nous obtenons donc les tuples du cube de données de r_2, satisfaisant la seconde contrainte d'émergence C_2, privés des tuples du cube de données de r_1 ne satisfaisant pas la première contrainte d'émergence C_1.

```
Select  D_1, D_2, ..., D_n
From r_2
Group by Cube (D_1, D_2, ..., D_n)
Having f(M) <= MinThld_2
Minus
Select D_1, D_2, ..., D_n
From r_1
Group by Cube (D_1, D_2, ..., D_n)
Having f(M) <= MinThld_1
```

Malheureusement, avec cette simple requête, nous ne pouvons pas retrouver la valeur du taux d'émergence. En fait, pour les tuples émergents qui sont significatifs dans r_2 et qui n'existent pas dans r_1, nous ne pouvons pas calculer la valeur du taux d'émergence à cause de l'absence de mesure pour ce tuple dans r_1.
Pour contourner ce problème, nous proposons une autre requête qui calcule l'intégralité du Cube Émergent. Cette nouvelle requête est bien plus complexe que la précédente car elle fait une distinction entre deux cas (les deux opérandes de l'opérateur Union) :

1. les tuples émergents dont on connaît toutes les mesures car ils sont présents dans les cubes des deux relations r_1 et r_2. Ces tuples correspondent aux basculements de tendances entre ces cubes.
2. les tuples émergents qui n'existent pas dans r_1 ni dans son cube. Ils correspondent aux nouveautés apparaissant dans le cube de r_2.

```
Select C_2.D_1, C_2.D_2, ..., C_2.D_n, C_2.M_2/C_1.M_1 ER
From ( Select   D_1, D_2, ..., D_n, f({M|*}) M2
       From   r_2
       Group by cube (D_1, D_2, ..., D_n)
       Having   f({M|*}) <= MinThld_2) C_2,
              (Select   D_1, D_2, ..., D_n, f({M|*}) M1
               From   r_1
               Group by cube (D_1, D_2, ..., D_n)
               Having   f({M|*}) < MinThld_1) C_1
Where   NVL(C_1.D_1,'ALL') = NVL(C_2.D_1,'ALL') And
        NVL(C_1.D_2,'ALL') = NVL(C_2.D_2,'ALL') And ...
        NVL(C_1.D_n,'ALL') = NVL(C_2.D_n,'ALL')
Union
Select  D_1, D_2, ..., D_n, Inf
From ( Select    D_1, D_2, ..., D_n
       From   r_2
       Group by cube (D_1, D_2, ..., D_n)
       Having   f({M|*}) <= MinThld_2)
Where (NVL(D_1,'ALL'), ..., NVL(D_n,'ALL'))
       Not In (Select   NVL(D_1,'ALL'), ..., NVL(D_n,'ALL')
               From   r_1
```

```
GROUP BY CUBE(D_1, D_2, ..., D_n)
HAVING   f({M|*}) <= MinThld_1)
```

Dans les clauses *Where* nous avons utilisé la fonction NVL pour substituer la valeur $'ALL'$ à la valeur *nulle*, qui est généralement utilisée à la place de la valeur symbolique ALL.

Nous avons également imaginé une troisième démarche qui simplifie pour beaucoup l'expression de la requête. L'idée de base est de se dispenser du calcul des deux cubes en effectuant une « fusion » des deux relations opérandes (Diop *et al.*, 2002). Le résultat de cette fusion est une nouvelle relation r dont le schéma est le suivant : $\mathcal{R} = \mathcal{D} \cup \mathcal{M}_1 \cup \mathcal{M}_2$ où $\mathcal{M}_1$ est l'attribut mesure de r_1 et $\mathcal{M}_2$ celui de r_2. Tout tuple de r_1 est considéré dans r avec la valeur 0 pour l'attribut mesure M_2 et inversement tout tuple de r_2 se voit affecter la valeur 0 pour la mesure M_1. Le fait de dissocier les deux mesures en considérant deux attributs différents permet de réaliser l'agrégation de r en conservant les agrégats obtenus pour r_1 et ceux obtenus pour r_2 en quelque sorte en les concaténant. La requête SQL basée sur cette méthode et effectuant le calcul du Cube Émergent est donnée ci-après.

```
SELECT D_1, D_2, ..., D_n, SUM(M2)/NULLIF(SUM(M1),0) ER
FROM (D_1, D_2, ..., D_n, M M1, 0 M2
      FROM r_1
      UNION
      SELECT D_1, D_2,..., D_n, 0 M1, M M2
      FROM r_2)
GROUP BY CUBE(D_1, D_2,..., D_n)
HAVING sum(M1) <= MinThld_1 AND SUM(M2)> MinThld_2
```

Remarquons que la relation fusionnée r est une relation temporaire calculée dans la clause FROM en réalisant l'union des tuples de r_1 et de r_2. Il suffit alors, dans le premier bloc, de procéder au calcul du cube et de spécifier la double contrainte d'émergence dans la clause HAVING. Pour éviter le problème de la division par 0 lors du calcul du taux d'émergence, nous utilisons la fonction NULLIF qui substitue au dénominateur la valeur NULL lorsqu'il est égal à 0. Ainsi, nous obtenons la valeur NULL correspondant à un taux d'émergence infini.

Exemple 5.13 - Avec nos relations exemples DOCUMENT_1 et DOCUMENT_2, la requête par fusion est la suivante.

```
SELECT Type,Ville,Editeur,Langue, SUM(M2)/NULLIF(SUM(M1),0) ER
FROM (SELECT Type,Ville,Editeur,Langue, Quantite M1, 0 M2
      FROM Document1
      UNION
      SELECT Type,Ville,Editeur,Langue, 0 M1, Quantite M2
      FROM Document2)
GROUP BY CUBE(Type,Ville,Editeur,Langue)
HAVING sum(M1) <= 300 AND SUM(M2)> 300
```

Les résultats de cette requête sont donnés dans la table 5.8

Pour l'instant, il n'existe aucun moyen intégré aux SGBD pour calculer les bordures. Même à partir des ensembles de tuples retournés par les requêtes proposées, on ne sait pas extraire les maximaux/minimaux selon l'ordre de généralisation.

TABLE 5.8 – Cube Émergent calculé par la requête avec fusion des relations

Tuple Émergent	ER
(ALL, ALL, ALL, Anglais)	3
(ALL, ALL, Collins, ALL)	8
(ALL, ALL, Collins, Français)	5
(ALL, Marseille, ALL, ALL)	3.3
(ALL, Marseille, ALL, Français)	7
(ALL, Marseille, Collins, ALL)	6
(ALL, Marseille, Hachette, ALL)	2
(ALL, Marseille, Hachette, Français)	NULL
(Pédagogie, ALL, ALL, ALL)	4,5
(Pédagogie, ALL, ALL, Anglais)	6
(Pédagogie, ALL, Hachette, ALL)	3
(Pédagogie, Marseille, ALL, ALL)	6

5.4.2 L'algorithme de calcul du Cube Émergent : E-Idea

Pour que le Cube Émergent soit un opérateur directement exploitable par l'utilisateur, les algorithmes de calcul que nous proposons doivent être intégrables directement dans les SGBD. Avec une approche relationnelle intégrable, il est possible tirer pleinement partie des outils d'analyse ROLAP existants. Le Cube Émergent ne sera ainsi qu'un cube particulier et, comme on le fait déjà avec le cube de données originel, il sera possible de l'interroger, l'explorer, y naviguer.

L'algorithme BUC (Beyer et Ramakrishnan, 1999) et ses différentes variantes (Morfonios et Ioannidis, 2008) ont montré toute leur efficacité aussi bien pour le calcul d'iceberg-cubes que pour des datacubes complets. Plutôt que de développer une nouvelle approche algorithmique nous avons choisi de nous appuyer sur l'algorithme BUC pour proposer une solution logicielle homogène capable de calculer le Cube Émergent aussi bien que ses différentes représentations. L'algorithme BUC est présenté en détail dans le paragraphe 2.3.6 p. 30.

Dans ce paragraphe, nous nous intéressons à l'adaptation du schéma algorithmique de BUC pour le calcul du Cube Émergent. Cette approche tire profit de la structure de la relation « fusionnée » que nous avons décrite au paragraphe 5.4.1. C'est véritablement cette structure astucieuse de relation concaténant les différentes mesures qui nous permet d'adapter des algorithmes de calcul de cube à notre contexte de Cube Émergent.

Exemple 5.14 - Avec nos relations exemples DOCUMENT_1 et DOCUMENT_2, la relation « fusionnée » est donnée dans la table 5.9. La relation résultat a les même attributs dimensions que les relations d'entrée, mais possède deux attributs mesures. L'attribut mesure $Quantité_1$ correspond aux quantités vendues pour la relation DOCUMENT_1 et $Quantité_2$ correspond à celles vendues pour la relation DOCUMENT_2. C'est cette relation qui sert d'entrée aux algorithmes proposés.

L'algorithme E-IDEA est une solution algorithmique inspirée de l'approche de BUC prenant en compte la contrainte d'émergence. Cette dernière est la conjonction de deux contraintes de natures différentes : l'une est monotone (C_1) et l'autre antimonotone (C_2). Le parcours du treillis cube utilisé par BUC favorise la contrainte antimonotone. En effet, comme il traite les tuples des

TABLE 5.9 – Relation « fusionnée » DOCUMENT$_{Fus}$

Id	Type	Ville	Éditeur	Langue	Quantité$_1$	Quantité$_2$
1	Nouvelles	Marseille	Collins	Français	100	0
2	Nouvelles	Marseille	Collins	Français	0	300
3	Nouvelles	Marseille	Hachette	Anglais	100	0
4	Pédagogie	Marseille	Collins	Anglais	0	300
5	Pédagogie	Marseille	Hachette	Français	0	300
6	Pédagogie	Marseille	Hachette	Anglais	100	0
7	Pédagogie	Paris	Hachette	Anglais	0	300
8	Pédagogie	Paris	Hachette	Français	100	0
9	Essai	Marseille	Hachette	Français	0	100
10	Essai	Paris	Hachette	Français	600	0
11	Essai	Paris	Hachette	Français	0	200
12	Essai	Paris	Collins	Français	0	200

plus agrégés vers les moins agrégés, c'est la seconde contrainte qui impose l'arrêt de l'algorithme.

Au début de son traitement les tuples multidimensionnels considérés ne satisfont pas la contrainte C_1 mais vérifient bien C_2 ; ces tuples sont donc non émergents, ils doivent être ignorés. Une fois tous ces tuples traités, E-IDEA passe aux tuples Émergents qui eux devront être écrits sur disque. Puis, il sort de l'espace des résultats lorsque C_2 n'est plus satisfaite pour retomber sur des tuples non émergents. La contrainte C_2 n'étant plus vérifiée, tous les successeurs ne sont plus émergents, l'algorithme s'arrête. La figure 5.5 illustre sur un treillis les différentes étapes rencontrées par l'algorithme durant le parcours de l'espace de recherche.

Pour traiter les différents cas possible l'algorithme fonctionne par phases et pour chacune de ces phases il a un comportement spécifique. Comme le montre la figure 5.5, il y a en tout 3 phases ($\mathbf{P}_1$, $\mathbf{P}_2$ et $\mathbf{P}_3$), chacune correspondant à l'état (vérifié ou non vérifié) des contraintes C_1 et C_2. Le tableau ci-dessous donne la correspondance entre état des contraintes et la phase dans laquelle l'algorithme se trouve :

	C_2	$\neg C_2$
C_1	$\mathbf{P}_2$	$\mathbf{P}_3$
$\neg C_1$	$\mathbf{P}_1$	

Comme souligné ci-dessus, C_1 et C_2 ne sont pas de nature quelconque. Le parcours en profondeur de E-IDEA implique que la contrainte monotone n'est pas satisfaite au début de l'algorithme puis le deviendra plus tard. Inversement, la contrainte antimonotone est satisfaite au début puis par la suite ne l'est plus. Ainsi, les transitions de phases ne peuvent toutes se produire. Par exemple si l'algorithme est dans la phase P_3 (contrainte C_2 n'est plus satisfaite), il ne pourra plus revenir dans l'une ou l'autre des phases. L'automate donné dans la figure 5.6 décrit les différentes transitions possibles.

Décrivons le comportement de l'algorithme pour chacune de ces phases :

Phase initiale $\mathbf{P}_1$: c'est l'état dans lequel se trouve l'algorithme au démarrage. Durant cette phase, les tuples considérés ne sont pas émergents car la contrainte C_1 n'est pas encore vérifiée. L'algorithme se contente juste de partitionner la relation sans écrire aucun résultat. E-IDEA change de phase dès lors que l'une des deux contraintes change d'état.

FIGURE 5.5 – Illustration des différentes étapes rencontrées par E-IDEA

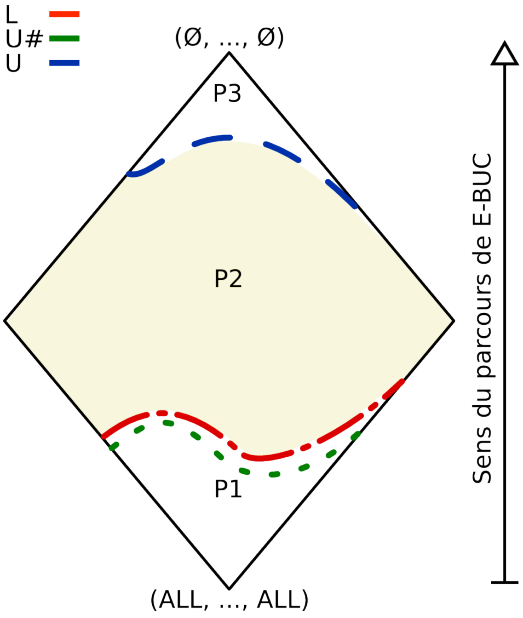
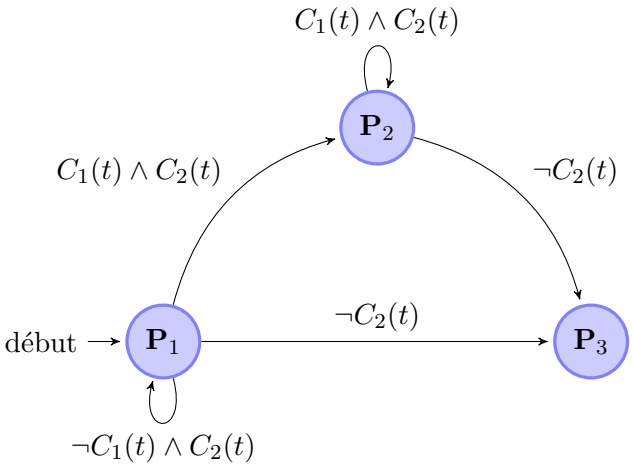


FIGURE 5.6 – Automate de transitions de phases de l'algorithme E-IDEA



Phase principale $\mathbf{P}_2$: durant cette phase tous les tuples rencontrés sont émergents. E-Idea doit donc écrire chacun d'entre eux sur disque. Ces opérations d'E/S étant très coûteuses, c'est l'étape la plus consommatrice en temps de l'algorithme. Cette étape se termine quand un tuple ne vérifiant plus la contrainte C_2 est rencontré. Dans ce cas, l'algorithme passe en phase 3.

Phase finale $\mathbf{P}_3$: L'algorithme arrive dans cette phase lorsque la contrainte C_2 n'est plus vérifiée pour le tuple courant t. Cette contrainte étant antimonotone elle ne peut plus être vérifiée pour tous les tuples plus spécifiques que t. Il n'y a donc plus besoin de parcourir l'espace de recherche, l'algorithme se termine.

Description de l'algorithme

E-Idea est une adaptation du scéma algorithmique de Buc pour calculer le Cube Émergent. Notre algorithme utilise le même découpage en sous-algorithmes que Buc (*cf.* paragraphe 2.3.6 p. 30) :

- Agréger (*cf.* algorithme 3) qui calcule la valeur de la mesure pour la partition courante.
- Buc (*cf.* algorithme 2)
- ÉcrireSuivants (*cf.* algorithme 4) qui permet d'écrire les résultats directement sur disque lorsque le cube de la partition courante est trivial à calculer.
- Partitionner (*cf.* algorithme 5) qui découpe efficacement la relation en fragment en fonction des valeurs de l'attribut courant. Cette opération est réalisée grâce à un tri par comptage de complexité linéaire.

Nous ne modifions que les deux premiers algorithmes. La modification apportée à Agréger est donnée dans l'algorithme 24. Notre relation fusionnée ayant deux attributs mesures, il faut calculer la valeur de la fonction agrégative sur chacun de ces attributs. À partir de ces deux valeurs, il est toujours possible de dériver le taux d'émergence associé au tuple multidimensionnel courant. L'algorithme E-Idea (*cf.* algorithme 25), quant à lui, reprend le même schéma algorithmique que Buc mais en adaptant son comportement en fonction de la phase dans laquelle il se trouve.

Algorithme 24 Algorithme Agréger

Entrée :
Une relation fusionnée r
La cellule courante $CelluleCour$
Une fonction agrégative additive f

1: **soit** r_1 la projection de r sur $\mathcal{D} \cup \mathcal{M}_1$
2: **soit** r_2 la projection de r sur $\mathcal{D} \cup \mathcal{M}_2$
3: **pour tout** $t \in r$ **faire**
4: $\quad CelluleCour.M_1 = CelluleCour.M_1 + f(t, r_1)$;
5: $\quad CelluleCour.M_2 = CelluleCour.M_2 + f(t, r_2)$;
6: **fin pour**

Élimination des duplicats : phase $\mathbf{P}_0$

Comme pour l'algorithme Buc la procédure ÉcrireSuivants permet de gagner énormément de temps en évitant de faire des appels récursifs inutiles. Cette procédure est appelée lorsque les partitions ne contiennent plus qu'un seul élément (aussi appelé *Base Single Tuple*). Si un même

Algorithme 25 Algorithme E-IDEA

Entrée :
Une relation fusionnée r
L'ensemble des dimensions $\mathcal{D} = \{d_1, \ldots, d_n\}$ restant à traiter.
La cellule courante $CelluleCour$
La phase courante de l'algorithme $\mathbf{P}_{Cour}$
Sortie :
Le Cube Émergent de r.
La phase sortie de l'algorithme.

1: $DimCour := d_1$;
2: AGRÉGER($r, CelluleCour, f$) ; //Calcule les valeurs de la fonction mesure et les écrit dans la cellule courante
3: **soit** $\mathbf{P}_{Suiv}$:= DÉTERMINERPHASESUIVANTE($\mathbf{P}_{Cour}$, $CelluleCour$) ; //*cf.* Figure 5.6
4: **si** $\mathbf{P}_{Suiv} = \mathbf{P}_3$ **alors**
5: **retourner** $\mathbf{P}_{Suiv}$;
6: **sinon si** $\mathbf{P}_{Suiv} = \mathbf{P}_2$ **alors**
7: **si** $|r| = 1$ **alors**
8: ÉCRIRESUIVANTS($r[0], \mathcal{D}, CelluleCour$) ; //La relation courante ne contient qu'un seul tuple, on peut écrire directement les résultats sans agréger
9: **retourner** $\mathbf{P}_3$; //Une fois tous les successeurs écrits, on passe dans la phase d'arrêt $\mathbf{P}_3$
10: **fin si**
11: ÉCRIRE($CelluleCour$) ;
12: **fin si**
13: **pour tout** $d_j \in \mathcal{D}$ **faire**
14: C := $|r(d_j)|$;
15: PARTITIONNER(r, d_j) ; //r est partitionnée suivant ses valeurs pour l'attribut d_j, en C fragments : $r_1, \ldots, r_C$
16: **pour** $i = 1, \ldots, C$ **faire**
17: $CelluleCour.d_j := r_i[0].d_j$; //On affecte à la cellule courante la valeur de la dimension d_j pour r_i
18: E-IDEA(r_i, $\{d_{j+1}, \ldots, d_n\}$, $CelluleCour$, $\mathbf{P}_{Suiv}$) ;
19: **fin pour**
20: $CelluleCour.d_j := ALL$; //La dimension d_j est entièrement traitée
21: **fin pour**
22: **retourner** $\mathbf{P}_{Suiv}$;

TABLE 5.10 – Relation « fusionnée » sans doublon DOCUMENT$^{+}_{Fus}$

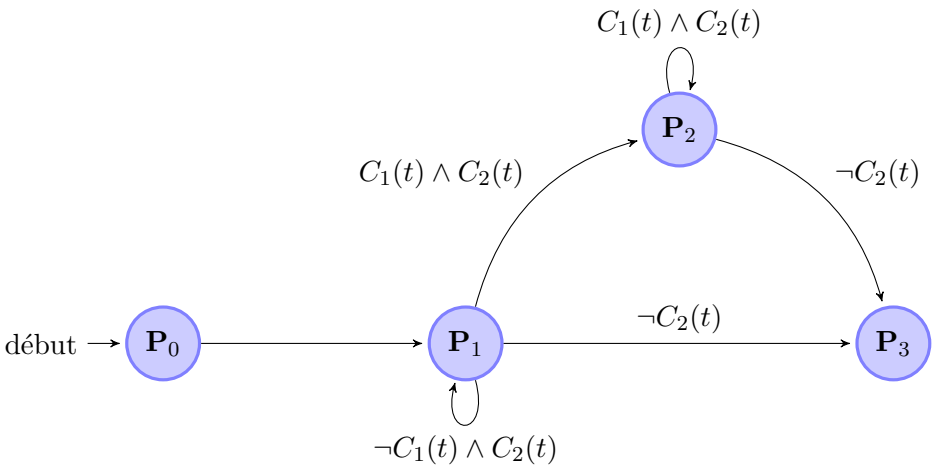
Id	Type	Ville	Éditeur	Langue	Quantité$_1$	Quantité$_2$
1	Nouvelles	Marseille	Collins	Français	100	300
2	Nouvelles	Marseille	Hachette	Anglais	100	0
3	Pédagogie	Marseille	Collins	Anglais	0	300
4	Pédagogie	Marseille	Hachette	Français	0	300
5	Pédagogie	Marseille	Hachette	Anglais	100	0
6	Pédagogie	Paris	Hachette	Anglais	0	300
7	Pédagogie	Paris	Hachette	Français	100	0
8	Essai	Marseille	Hachette	Français	0	100
9	Essai	Paris	Hachette	Français	600	200
10	Essai	Paris	Collins	Français	0	200

tuple est présent deux fois dans la relation d'entrée, la partition ne sera jamais de taille 1, cette optimisation ne pourra pas avoir lieu. Ce fait est peu gênant dans le cas du cube de données car les relations contiennent rarement des tuples dupliqués. Malheureusement, avec notre structure de relation fusionnée, il existe au sein de la relation d'entrée r un grand nombre de tuples dupliqués. Effectivement, lorsque l'on fusionne les deux relations, les tuples qui sont présents à la fois dans r_1 et r_2 se retrouvent deux fois dans la relation résutat r. La première occurence avec une valeur uniquement pour l'attribut mesure $\mathcal{M}_1$ et la seconde uniquement pour $\mathcal{M}_2$. Pour supprimer ces duplicats, il suffit de les pré-agréger en les combinant en un seul et même tuple ayant une mesure pour les deux attributs $\mathcal{M}_1$ et $\mathcal{M}_2$.

Exemple 5.15 - Avec les relations exemples, le tuple (Nouvelles, Marseille, Collins, Français) est présent au sein de DOCUMENT$_1$ et de DOCUMENT$_2$. Il l'est donc présent en double dans la relation « fusionnée » (*cf.* table 5.9). Pour supprimer ce doublon, il suffit de remplacer les tuples (Nouvelles, Marseille, Collins, Français)100 :0 et (Nouvelles, Marseille, Collins, Français)0 :300 par le tuple (Nouvelles, Marseille, Collins, Français)100 :300. La relation « fusionnée » sans duplicat est donnée dans la table 5.10.

Cette opération de « dédoublonnage » nécessite de trier toute la relation d'entrée. Elle a donc un coût additionnel non négligeable. Notre idée pour pouvoir supprimer les duplicats sans perdre trop de temps est de profiter du tri effectué lors du premier partitionnement pour « dédupliquer ». Ainsi, on peut considérer que l'on ajoute une phase suplémentaire à notre algorithme que nous nommons $\mathbf{P}_0$. Le passage de $\mathbf{P}_0$ vers $\mathbf{P}_1$ se fait sans condition. Le nouvel automate décrivant les transitions de phases est donné dans la figure 5.7. Cette nouvelle phase n'arrive que lors du premier appel de l'algorithme. Pour éviter de devoir rajouter un test inutile dans tous les appels sauf le premier, nous créons une nouvelle procédure nommée E-IDEA_$\mathbf{P}_0$. Cette procédure est pratiquement la même que la procédure E-IDEA mais PARTITIONNER est remplacée par PARTITIONNERETDÉDUPLIQUER qui comme son nom l'indique va à la fois partitionner et profiter du tri effectué pour éliminer les duplicats. La procédure E-IDEA_$\mathbf{P}_0$ est donnée dans l'algorithme 26.

FIGURE 5.7 – Automate de transitions de phases de l'algorithme E-IDEA avec élimination des duplicats



Algorithme 26 Algorithme E-IDEA_$\mathbf{P}_0$

Entrée :
Une relation fusionnée r
L'ensemble des dimensions $\mathcal{D} = \{d_1, \ldots, d_n\}$ restant à traiter.
La cellule courante *CelluleCour*
Sortie :
Le Cube Émergent de r.
La phase sortie de l'algorithme.

1: $DimCour := d_1$;
2: AGRÉGER($r, CelluleCour, f$) ; //Calcule les valeurs de la fonction mesure et les écrit dans la cellule courante
3: **si** $C_1((\text{ALL}, \ldots, \text{ALL}))$ **alors**
4: BUC(r, $\mathcal{D}$, $(\text{ALL}, \ldots, \text{ALL})$) ; //Si la contrainte monotone est satisfaite dès le premier tuple, le résultat se ramène au cube iceberg de r
5: **retourner** $\mathbf{P}_0$;
6: **fin si**
7: **pour tout** $d_j \in \mathcal{D}$ **faire**
8: C := $|r(d_j)|$;
9: PARTITIONNERETDÉDUPLIQUER(r, d_j) ; //r est partitionnée suivant ses valeurs pour l'attribut d_j, en C fragments : $r_1, \ldots, r_C$
10: **pour** $i = 1, \ldots, C$ **faire**
11: $CelluleCour.d_j := r_i[0].d_j$; //On affecte à la cellule courante la valeur de la dimension d_j pour r_i
12: E-IDEA(r_i, $\{d_{j+1}, \ldots, d_n\}$, $CelluleCour$, $\mathbf{P}_1$) ;
13: **fin pour**
14: $CelluleCour.d_j := ALL$; //La dimension d_j est entièrement traitée
15: **fin pour**
16: **retourner** $\mathbf{P}_0$;

5.4.3 L'algorithme générique de calcul des bordures du Cube Émergent : F-Idea

Les bordures ont largement été étudiées dans un contexte de fouille de données binaires. Des algorithmes performants ont été proposés pour calculer les bordures maximales (Burdick *et al.*, 2005; Gouda et Zaki, 2005; Jr., 1998). Cependant, il n'existe aucune solution dédiée pour calculer ce type de représentation dans le contexte multidimensionnel. Notre objectif ici est de concevoir une première solution logicielle générique pour obtenir toutes les bordures du Cube Émergent. Là encore, nous souhaitons tirer partie de l'efficacité d'algorithmes existants et à nouveau notre choix s'est porté sur l'algorithme Buc, toujours pour les mêmes raisons. Lors du calcul du Cube Émergent, E-Idea utilise la majeure partie du temps d'exécution pour réaliser les sorties, *i.e.* l'écriture du cube sur disque. Forts de cette observation, nous adaptons E-Idea de manière à n'écrire que les tuples des bordures. De plus, grâce à une structure de données adaptée, nous évitons l'écriture multiple des mêmes tuples. Dans un premier temps, nous présentons l'algorithme calculant la bordure U puis nous montrons les modifications à lui apporter pour obtenir la bordure $U^\sharp$ ainsi que L.

Sur la figure 5.5, on remarque que les différentes bordures délimitent les phases de l'algorithme E-Idea. En d'autres termes, les tuples des bordures correspondent à des tuples pour lesquels l'algorithme change de phases. Nous faisons plusieurs propositions pour démontrer à quel type de transition est associé chaque bordure. Mais au préalable, nous définissons plusieurs concepts qui nous permettent de mieux les formaliser.

Définition 5.8 (Ensemble de tuples en accord) **-** Soit $P \subseteq r$ un sous-ensemble de tuples de la relation r et $D \subseteq \mathcal{D}$ un ensemble de dimensions, on dit que P est en accord sur l'ensemble D si et seulement si $\forall t, t' \in P$ on a $t[D] = t'[D]$.

Définition 5.9 (Idea-Successeur) **-** Soit $D = \{d_1, \ldots, d_j\}$ un sous-ensemble ordonné de dimensions ($D \subseteq \mathcal{D}$) et soit P un ensemble de tuples de la relation r en accord sur D. Un ensemble de tuples P' est dit Idea-Successeur de P par rapport à D si et seulement si $P' \subseteq P$ et $\exists d_k \in \mathcal{D} \backslash D$ (avec $k > j$) tel que $\forall t,t' \in P'$, $t.d_k = t'.d_k$.

Autrement dit, P' est un Idea-Successeur de P par rapport à D si P' est un sous-ensemble de P qui est en accord sur un sur-ensemble direct de D dont la dimension supplémentaire d_k est plus grande (selon $\geq_{\mathcal{D}}$) que la plus grande des dimensions de D (d_j dans la définition ci-dessus).

Exemple 5.16 - En reprenant la relation exemple fusionnée et sans doublon Document^+_{Fus} (*cf.* table 5.10), soit $P = \text{Document}^+_{Fus} = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}\}$ un ensemble de tuples en accord sur $D = \emptyset$, l'ensemble de tuples $P' = \{t_1, t_2, t_3, t_4, t_5\}$ est en accord sur $D' = \{Ville\}$ un sur-ensemble direct de D. P' est donc un Idea-Successeur de P par rapport à D.

Définition 5.10 (Idea-Successeur maximal) **-** Soit $D = \{d_1, \ldots, d_j\}$ un sous-ensemble ordonné de dimensions ($D \subseteq \mathcal{D}$) et soit P un ensemble de tuples d'une relation r tels que $\forall t, t' \in P$ on a $t[D] = t'[D]$. Un fragment P' est dit Idea-Successeur maximal de P par rapport à D si et seulement si P' est un Idea-Successeur de P par rapport à D et $\nexists t \in P \backslash P'$ tel que $t \cup P'$ soit un Idea-Successeur de P par rapport à D.

Cette notion de Idea-Successeur maximal nous permet de formaliser clairement l'ordre dans lequel les algorithmes Buc et E-Idea parcourent les partitions de la relation d'entrée (*cf.* figures 2.10 et 2.11 du paragraphe 2.3.6). Ainsi, dire qu'un fragment P' est un Idea-Successeur maximal de P par rapport à D revient à dire que si l'on fait un appel à E-Idea avec comme

arguments P et D, alors on est certain que l'un des appels récursifs directs prend comme paramètre le fragment P' et l'ensemble de dimensions $D \cup d_k$. Si l'on se préoccupe de l'arbre des appels récursifs cela revient à dire qu'il existe une arête qui relie les deux appels de paramètre (P,D) et (P',$D \cup d_k$). Il est intéressant de noter que cette notion de IDEA-Successeur maximal a des liens de parenté évidents avec la notion de classe d'équivalence Dimension-Mesure présentée dans le paragraphe 3.4. De la même manière qu'à chaque classe-DM est associé un tuple multidimensionnel, nous pouvons faire de même pour chaque IDEA-Successeur maximal.

Exemple 5.17 - D'après l'exemple précédent, $P' = \{t_1, t_2, t_3, t_4, t_5\}$ est un IDEA-Successeur de $P = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}\}$ par rapport à $D = \emptyset$. P' n'est pas IDEA-Successeur maximal car t_8 partage les même valeurs que les tuples de P' sur D'. $P' \cup t_8$ est, quant à lui, un IDEA-Successeur maximal de P par rapport à D car il n'existe aucun tuple partageant les mêmes valeurs pour les attributs de D'. Le tuple multidimensionnel associé à P est (ALL, ALL, ALL, ALL). Celui associé à $P' \cup t_8$ est le tuple (ALL, Marseille, ALL, ALL).

Lors de l'exécution de E-IDEA, les transitions de phases ont lieu juste après un appel récursif à l'algorithme. Si pour un appel à E-IDEA avec comme arguments P et D, l'algorithme est en phase $\mathbf{P}_i$ et que lors du traitement de l'un de ses IDEA-Successeurs maximaux P', il est en phase $\mathbf{P}_j$ on dira que P' est issu d'une transition $\mathbf{P}_i \rightarrow \mathbf{P}_j$ et que P est source d'une transition $\mathbf{P}_i \rightarrow \mathbf{P}_j$.

Proposition 5.5 - Soit un tuple $t \in L$, l'ensemble de tuples P' ayant servi au calcul de t est issu d'une transition $\mathbf{P}_1 \rightarrow \mathbf{P}_2$

Démonstration.

$$
\begin{aligned}
& t \in L \\
\Leftrightarrow & t \in \min_{\preceq_g}(\{t \in CL(r_1 \cup r_2) \mid C_1(t) \wedge C_2(t)\}) \\
\Leftrightarrow & \forall t' \text{ tel que } t' \preceq_g t \text{ on a } \neg C_1(t') \wedge C_2(t') \\
\Rightarrow & \forall t' \text{ tel que } t' \preceq_g t, \text{ l'algorithme E-IDEA est en phase } \mathbf{P}_1 \text{ lorsqu'il génère } t' \qquad (5.1)
\end{aligned}
$$

De même

$$
\begin{aligned}
& t \in L \\
\Leftrightarrow & t \in \min_{\preceq_g}(\{t \in CL(r_1 \cup r_2) \mid C_1(t) \wedge C_2(t)\}) \\
\Rightarrow & t \text{ est un tuple émergent} \\
\Rightarrow & \text{l'algorithme E-IDEA est en phase } \mathbf{P}_2 \text{ lorsqu'il génère } t \qquad (5.2)
\end{aligned}
$$

D'après (5.1) et (5.2) la transition permettant de passer d'un tuple t' à t est bien de type $\mathbf{P}_1 \rightarrow \mathbf{P}_2$ □

Proposition 5.6 - Soit un tuple $t \in U$, l'ensemble de tuples P ayant servi au calcul de t n'est source que de transitions $\mathbf{P}_2 \rightarrow \mathbf{P}_3$.

Démonstration.

$$\begin{aligned}
& t \in U \\
\Leftrightarrow & t \in \max_{\preceq_g}(\{t \in CL(r_1 \cup r_2) \mid C_1(t) \wedge C_2(t)\}) \\
\Rightarrow & t \text{ est un tuple émergent} \\
\Rightarrow & \text{l'algorithme E-IDEA est en phase } \mathbf{P}_2 \text{ après avoir généré } t
\end{aligned} \tag{5.3}$$

Supposons qu'il existe une transition qui ne soit pas $\mathbf{P}_2 \rightarrow \mathbf{P}_3$, dans ce cas, d'après l'automate de transition de phase (*cf.* figure 5.6), cette transition est forcément $\mathbf{P}_2 \rightarrow \mathbf{P}_2$. Cela veut dire que le tuple t' généré lors de cette transition est un tuple émergent plus général que t. Il y a une contradiction avec le fait que $t \in U$. □

Proposition 5.7 - Soit un tuple $t \in U^\sharp$, l'ensemble de tuples P ayant servi au calcul de t n'est source d'aucune transition $\mathbf{P}_1 \rightarrow \mathbf{P}_1$.

Démonstration.

$$\begin{aligned}
& t \in U^\sharp \\
\Leftrightarrow & t \in \max_{\preceq_g}(\{t \in CL(r_1 \cup r_2) \mid \neg C_1(t) \wedge C_2(t)\}) \\
\Leftrightarrow & \forall t' \text{ tel que } t \preceq_g t' \text{ on a } C_1(t') \wedge C_2(t') \text{ ou } \neg\neg C_1(t) \wedge \neg C_2(t) \\
\Rightarrow & \forall t' \text{ tel que } t \preceq_g t', \text{ l'algorithme E-IDEA est en phase } \mathbf{P}_2 \text{ ou } \mathbf{P}_3 \text{ lorsqu'il génère } t'
\end{aligned} \tag{5.4}$$

De même

$$\begin{aligned}
& t \in U^\sharp \\
\Leftrightarrow & t \in \max_{\preceq_g}(\{t \in CL(r_1 \cup r_2) \mid \neg C_1(t) \wedge C_2(t)\}) \\
\Rightarrow & \text{l'algorithme E-IDEA est en phase } \mathbf{P}_1 \text{ lorsqu'il génère } t
\end{aligned} \tag{5.5}$$

D'après (5.4) et (5.5) toutes les transitions permettant de passer d'un tuple t à t' sont toutes de type $\mathbf{P}_1 \rightarrow \mathbf{P}_2$ ou $\mathbf{P}_1 \rightarrow \mathbf{P}_3$. S'il en existait une seule qui ne soit pas de ce type alors t ne serait pas maximal. □

Ces trois propositions sont fondamentales pour caractériser les algorithmes capables de dériver les différentes bordures. Intégrer l'une de ces propositions au sein de l'algorithme E-IDEA nous permet de calculer la bordure correspondante. L'algorithme découlant de cette intégration est appelé F-IDEA (pour *Frontier Integrable DatabasE Algorithm*). F-IDEA n'est pas seulement un algorithme capable de calculer une unique bordure, mais il apporte une solution algorithmique générique capable de calculer n'importe quelle bordure ou combinaison de bordures. Ainsi, disposer d'une ou de trois bordures à la fois est tout à fait possible dès l'intégration des propositions correspondantes.

Pour un souci de clarté, nous présentons la démarche d'adaptation de E-IDEA en F-IDEA uniquement pour la bordure $U^\sharp$. Les autres versions de F-IDEA peuvent être obtenues en suivant exactement la même démarche que celle décrite ci-dessous. La première des modifications à apporter est de supprimer toutes les opérations effectuées pour les tuples générer pendant la phase $\mathbf{P}_2$ car tous les tuples de $U^\sharp$ sont produits durant la phase $\mathbf{P}_1$. La deuxième des différences se situe au niveau de la gestion des *Base Single Tuples*. La partition de ses tuples est constituée

que d'un simple tuple, ainsi pour un tel éléments il n'est pas besoin de considéré tous ses successeurs mais juste le plus grand d'entre eux. La projection du tuple sur l'ensemble des dimensions restant à traiter est alors ajouter dans la bordure. La dernière des modifications à faire concerne la vérification de la condition donnée par la proposition 5.7. Après chaque appel récursif on vérifie que la transition n'est pas $\mathbf{P}_1 \rightarrow \mathbf{P}_1$. Si aucune telle transition n'est rencontrée, le tuple courant est testé et ajouté à la bordure $U^\sharp$. La procédure AjouterABordure se contente de vérifier la maximalité du tuple et de l'ajouter au résultat. Comme on le voit sur cette version de F-Idea, les adaptations à faire sont toutes induites par les propositions 5.5, 5.6 et 5.7. Il y a effectivement des optimisations plus spécifiques aux calcul de chaque bordure, mais l'algorithme F-Idea perdrait en généricité. Malgré cela, cet algorithme est, comme nous allons le voir dans le prochain paragraphe, vraiment très efficace.

5.5 Évaluations Expérimentales

Afin de valider nos représentations, nous avons procédé à des expérimentations en suivant une triple objectif : mesurer et comparer la taille des représentations par bordure et celle du Cube émergent, déterminer le temps de calcul d'un Cube émergent selon l'approche algorithmique et avec l'algorithme E-Idea puis mesurer le temps de calcul des bordures en utilisant l'algorithme F-Idea.

Les expérimentations sont menées sur des données issues d'un éventail large et varié de domaines similaires aux ensemble de données utilisés dans Xin *et al.* (2007). Il est bien connu que les données synthétiques sont faiblement corrélées alors que les données de bases réelles ou statistiques sont fortement corrélées. Pour les données synthétiques [5], nous utilisons les notations suivantes pour décrire les relations : $\mathcal{D}$ le nombre de dimensions, $\mathcal{C}$ la cardinalité de chaque dimension, $\mathcal{T}$ le nombre de tuples dans chaque relation, $\mathcal{M}_1$ ($\mathcal{M}_2$ respectivement) le seuil correspondant à la contrainte d'émergence C_1 (C_2 respectivement), et $\mathcal{S}$ le biais ou zipf des données. Quand $\mathcal{S}$ est égal à 0, les données sont uniformes. Quand $\mathcal{S}$ croît, les données sont plus biaisées. $\mathcal{S}$ est appliqué à toutes les dimensions dans chaque relation de la base de données. Pour les données réelles, nous utilisons les relations concernant la météorologie SEP83L.DAT et SEP85L.DAT utilisées dans Xin *et al.* (2006), qui ont 1.002.752 tuples avec 8 dimensions sélectionnées. Les attributs (avec leur cardinalités) sont les suivantes : année mois jour heure (238), latitude (5260), longitude (6187), nombre de stations (6515), temps actuel (100), code de changement (110), altitude solaire (1535) et luminance lunaire relative (155).

5.5.1 Taille des bordures

Un intérêt majeur des bordures est de réduire l'information requise. L'objectif, important dans le contexte Olap, est bien sûr un gain d'espace ; mais manipuler moins d'informations a des conséquences bénéfiques sur le temps d'exécution. Par exemple, savoir si tel renversement de tendance est avéré, peut être obtenu plus efficacement. Nous avons mesuré la taille du Cube émergent et celle de chacune des représentations par bordures. Nous effectuons différentes comparaisons expérimentales sur les données synthétiques en faisant varier :

- la taille des relations en entrée (*cf.* figure 5.8),
- la cardinalité de chaque dimension (*cf.* figure 5.9),
- le biais des données (*cf.* figure 5.10).

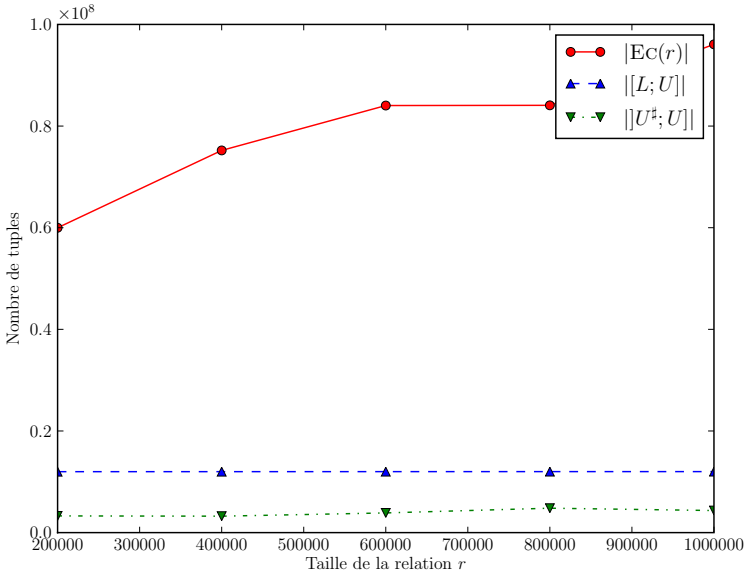
5. Le générateur de données synthétique est disponible à : http://illimine.cs.uiuc.edu/

Algorithme 27 Algorithme F-IDEA pour la bordure $U^\sharp$

Entrée :
Une relation fusionnée r courante
L'ensemble des dimensions $\mathcal{D} = \{d_1, \ldots, d_n\}$ restant à traiter.
La cellule courante *CelluleCour*
La phase courante de l'algorithme $\mathbf{P}_{Cour}$
Sortie :
La bordure $U^\sharp$ Cube Émergent de r.
La phase sortie de l'algorithme.

1: $DimCour := d_1$;
2: AGRÉGER($r, CelluleCour, f$) ; //Calcule les valeurs de la fonction mesure et les écrit dans la cellule courante
3: **soit** $\mathbf{P}_{Suiv}$:= DÉTERMINERPHASESUIVANTE($\mathbf{P}_{Cour}$, $CelluleCour$) ; //*cf.* Figure 5.6
4: **si** $(\mathbf{P}_{Suiv} = \mathbf{P}_3) \vee (\mathbf{P}_{Suiv} = \mathbf{P}_2)$ **alors**
5: **retourner** $\mathbf{P}_{Suiv}$; //On ne cherche que les tuples de $U^\sharp$ pas besoin de parcourir les tuples émergents
6: **sinon si** $|r| = 1$ **alors**
7: **soit** $t := r[0]$;
8: AJOUTERABORDURE($t[\mathcal{D}]$) ; //La relation courante ne contient qu'un seul tuple t, on peut l'ajouter directement dans $U^\sharp$ sans traiter tous ses prédécesseurs
9: **retourner** $\mathbf{P}_1$;
10: **fin si**
11: **pour tout** $d_j \in \mathcal{D}$ **faire**
12: C := $|r(d_j)|$;
13: PARTITIONNER(r, d_j) ; //r est partitionnée suivant ses valeurs pour l'attribut d_j, en C fragments : $r_1, \ldots, r_C$
14: **soit** $TestU^\sharp$: booléen permettant de tester si la partition courante vérifie la condition de la proposition 5.7 ;
15: $TestU^\sharp := (\mathbf{P}_{Suiv} == \mathbf{P}_1)$; //Teste si la source de la transition est bien $\mathbf{P}_1$
//Ce test est facultatif pour cette bordure mais devra être ajouté impérativement pour les autres
16: **pour** $i = 1, \ldots, C$ **faire**
17: $CelluleCour.d_j := r_i[0].d_j$; //On affecte à la cellule courante la valeur de la dimension d_j pour r_i
18: $\mathbf{P}_{Res}$:= F-IDEA(r_i, $\{d_{j+1}, \ldots, d_n\}$, $CelluleCour$, $\mathbf{P}_{Suiv}$) ;
19: $TestU^\sharp := TestU^\sharp \wedge (\mathbf{P}_{Res} \neq \mathbf{P}_1)$;
20: **fin pour**
21: **si** $TestU^\sharp$ **alors**
22: AJOUTERABORDURE($CelluleCour$) ;
23: **fin si**
24: $CelluleCour.d_j := ALL$; //La dimension d_j est entièrement traitée
25: **fin pour**
26: **retourner** $\mathbf{P}_{Suiv}$;

FIGURE 5.8 – Taille des bordures L et $U^\sharp$ avec $\mathcal{D} = 10$, $\mathcal{C} = 100$, $\mathcal{S} = 0$



Nous calculons aussi la taille des bordures L et $U^\sharp$ en faisant varier le seuil appliqué à la relation r_1 (*cf.* figure 5.11). Les résultats sont convaincants, les représentations par bordure apportent une réduction considérables de l'espace nécessaire. De plus, ces résultats expérimentaux soulignent l'intérêt de notre nouvelle bordure car dans tous les cas, le gain de taille de $U^\sharp$ comparé à L est réellement significatif. Ainsi nous pensons que notre bordure $U^\sharp$ est une alternative particulièrement intéressante à la classique bordure L très largement utilisée dans un contexte fouille de données.

5.5.2 Temps de calcul du Cube Émergent

Pour mesurer le temps de calcul des Cubes Émergents, nous avons utilisé la version la plus optimale de nos requêtes SQL sous ORACLE 10G et l'algorithme E-IDEA. Les mêmes jeux de données que précédemment ont été utilisés. Les différents calculs menés montrent l'efficacité de E-IDEA. Les résultats expérimentaux nous ont cependant réservé une surprise : le bon comportement de notre requête Sql. Celle-ci rend le Cube émergent, même lorsqu'il est obtenu à partir d'un ensemble de données initial volumineux, en temps parfaitement raisonnable.

5.5.3 Temps de calcul des bordures du Cube Émergent

Enfin, nous avons mesuré les temps d'exécution nécessaires pour obtenir les représentations par bordures et nous les avons comparé aux temps pour calculer le Cube émergent. Les mêmes jeux de données ont, là encore, été utilisés. Évidemment, le temps d'obtention des bordures est plus rapide que le calcul du Cube émergent. Il représente environ le dixième du temps nécessaire pour le Cube émergent.

FIGURE 5.9 – Taille des bordures L et $U^\sharp$ avec $\mathcal{D} = 10$, $\mathcal{T} = 1000$K, $\mathcal{S} = 0$

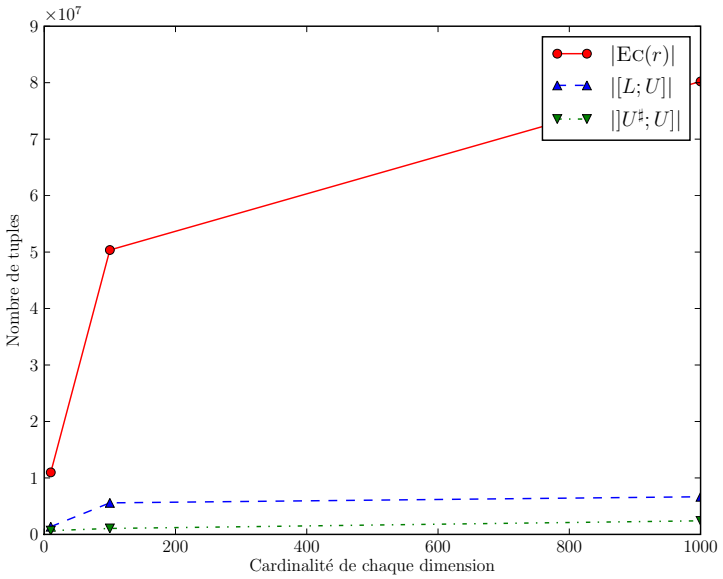


FIGURE 5.10 – Taille des bordures L et $U^\sharp$ avec $\mathcal{D} = 10$, $\mathcal{C} = 100$, $\mathcal{T} = 1000$K

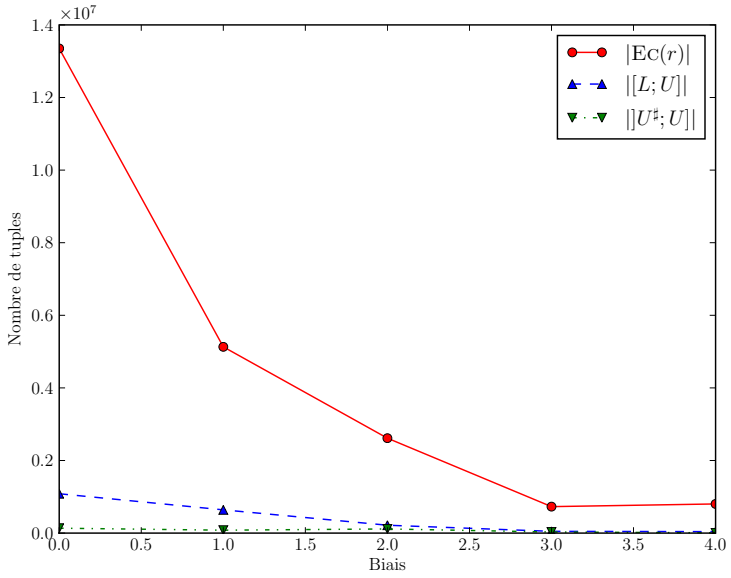


FIGURE 5.11 – Taille des bordures L et $U^\sharp$ pour les relations de données météorologiques

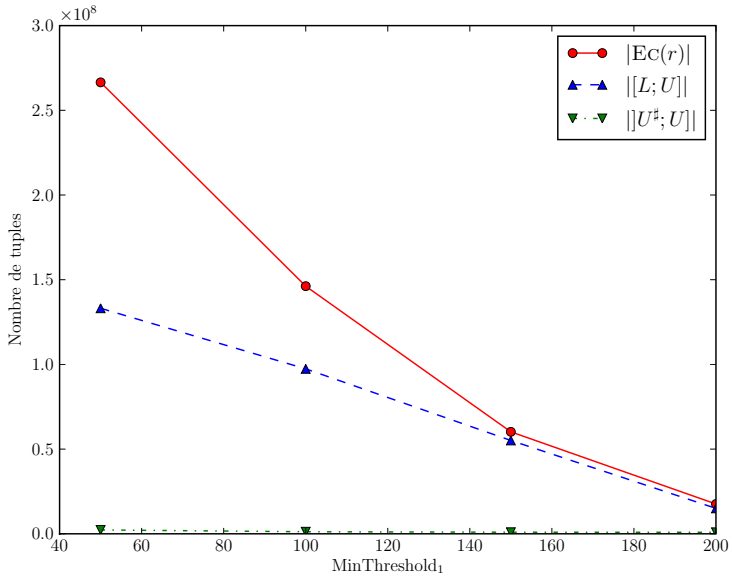


FIGURE 5.12 – Temps de calcul du Cube Émergent avec $\mathcal{D} = 10$, $\mathcal{C} = 100$, $\mathcal{S} = 0$

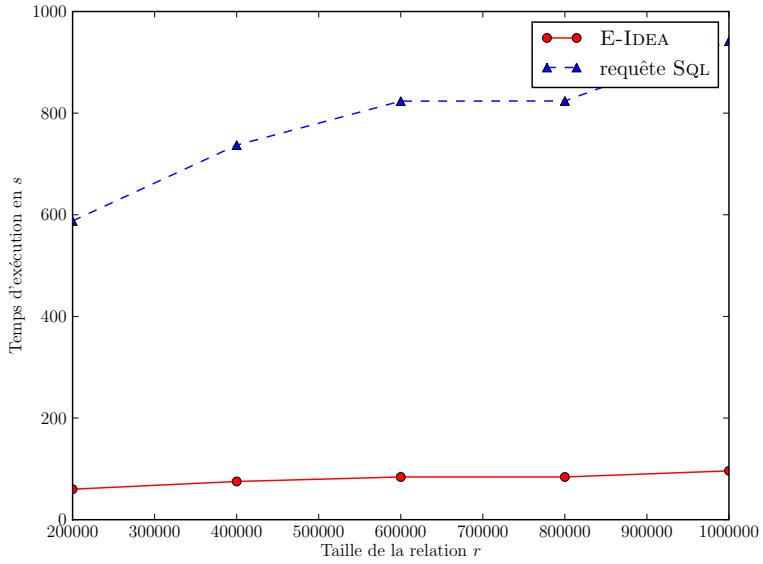


FIGURE 5.13 – Temps de calcul du Cube Émergent avec $\mathcal{D} = 10$, $\mathcal{T} = 1000$K, $\mathcal{S} = 0$

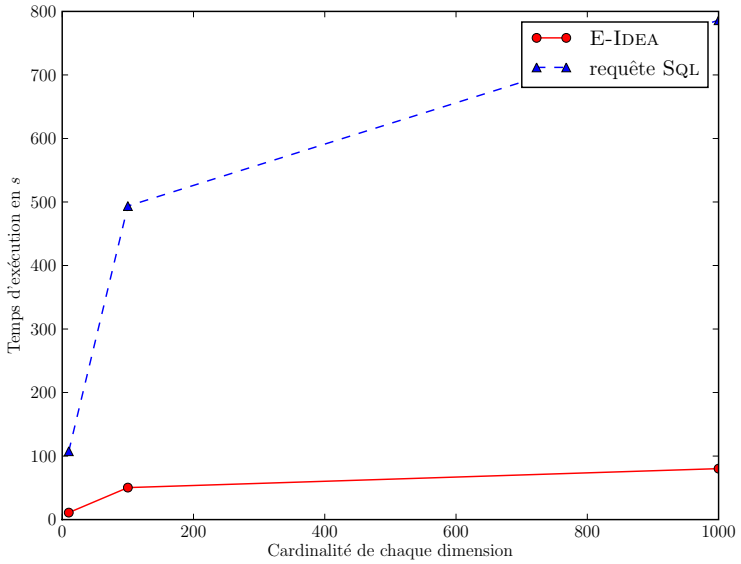


FIGURE 5.14 – Temps de calcul du Cube Émergent avec $\mathcal{D} = 10$, $\mathcal{C} = 100$, $\mathcal{T} = 1000$K

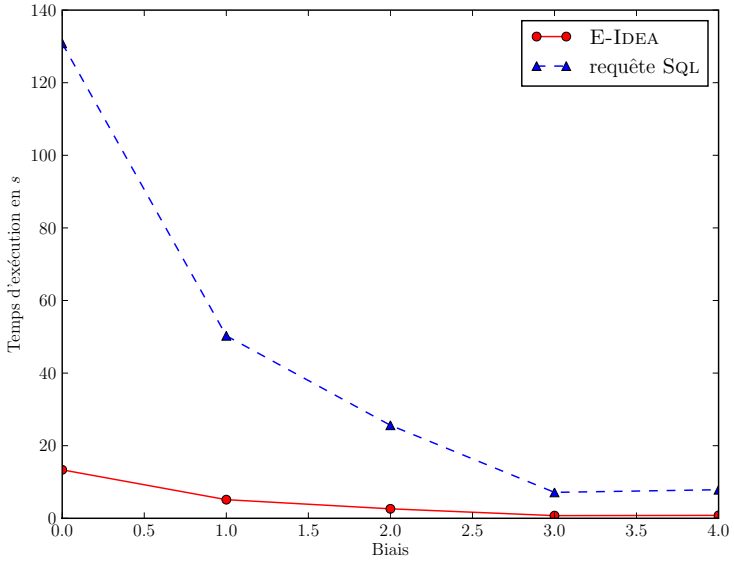


FIGURE 5.15 – Temps de calcul du Cube Émergent pour les relations de données météorologiques

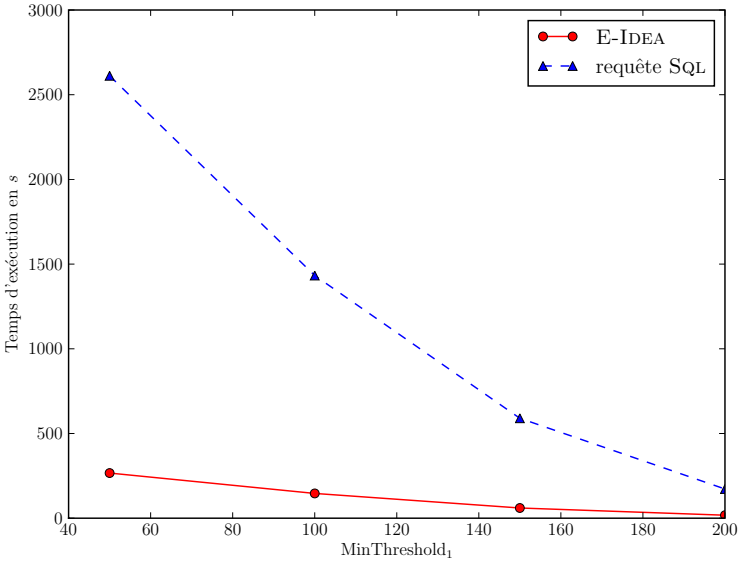


FIGURE 5.16 – Temps de calcul des bordures L et $U^\sharp$ avec $\mathcal{D} = 10$, $\mathcal{C} = 100$, $\mathcal{S} = 0$

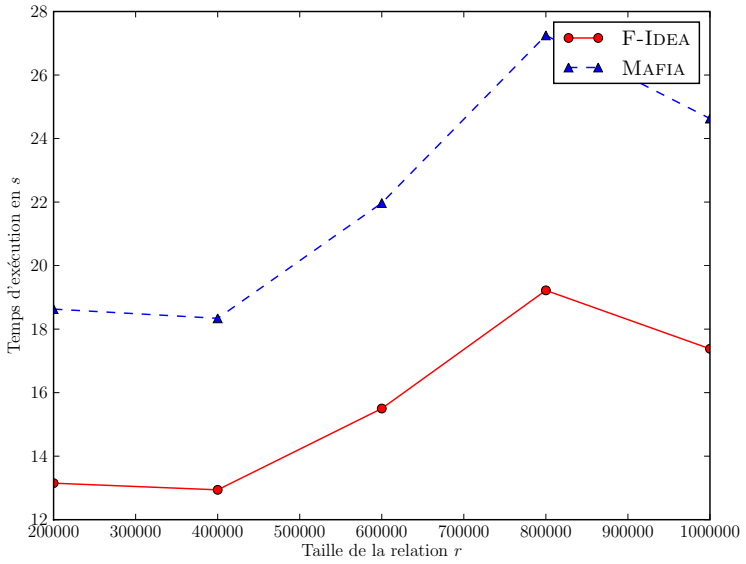


FIGURE 5.17 – Temps de calcul des bordures L et $U^\sharp$ avec $\mathcal{D} = 10$, $\mathcal{T} = 1000$K, $\mathcal{S} = 0$

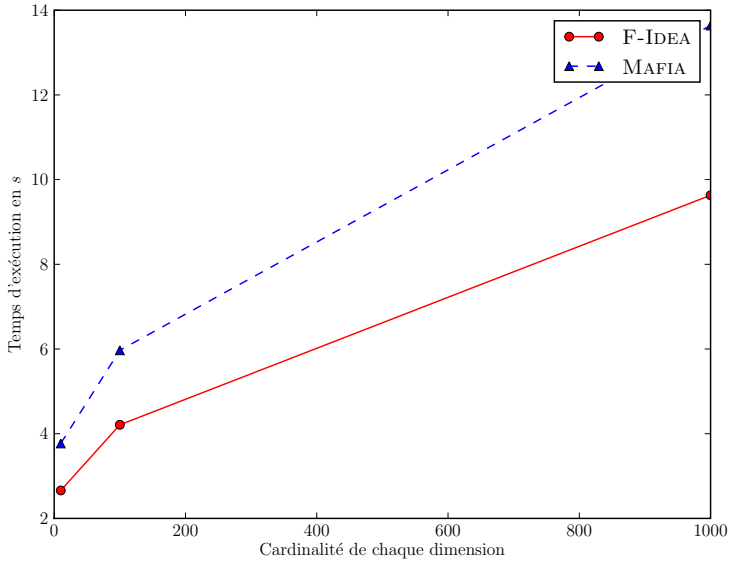


FIGURE 5.18 – Temps de calcul des bordures L et $U^\sharp$ avec $\mathcal{D} = 10$, $\mathcal{C} = 100$, $\mathcal{T} = 1000$K

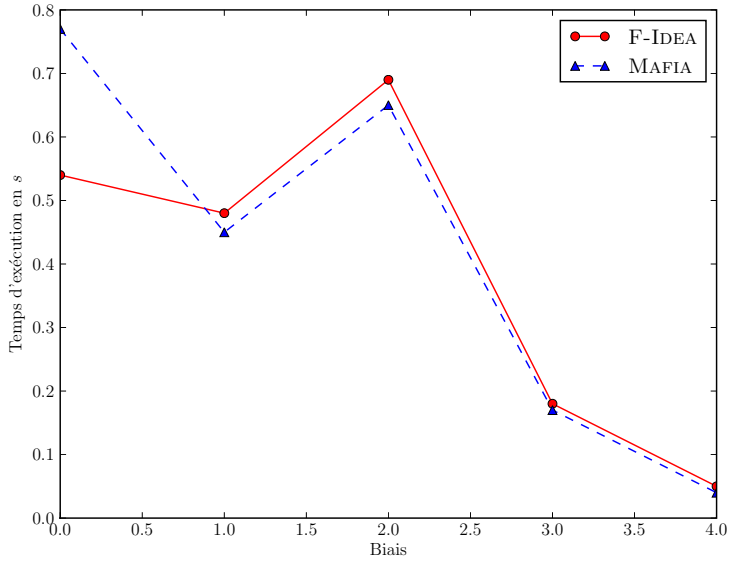
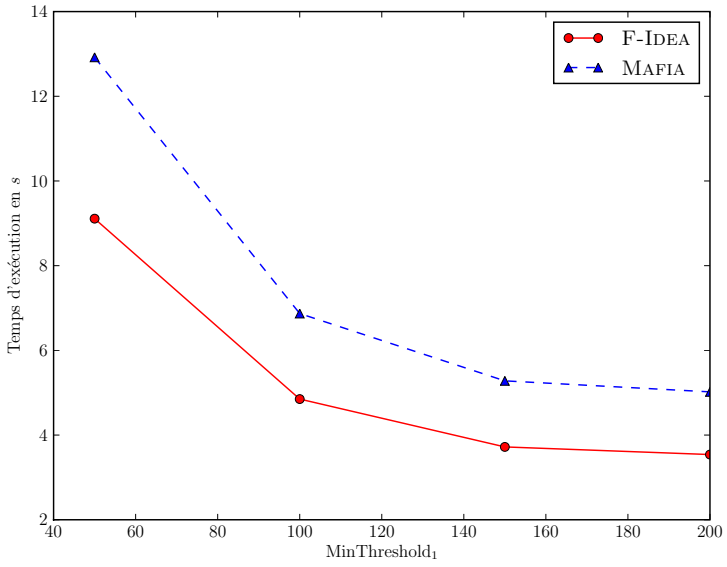


FIGURE 5.19 – Temps de calcul des bordures L et $U^\sharp$ pour les relations de données météorologiques



5.6 Conclusion

Dans ce chapitre, nous avons proposé le concept de Cube Émergent afin de capturer les renversements de tendances entre deux data cubes de deux relations comparables. Calculer un Cube Émergent est coûteux et les résultats peuvent s'avérer très volumineux. Aussi nous nous sommes intéressés aux représentations réduites de tels cubes. Les Cubes Émergents peuvent être représentés par les bordures classiques $[L; U]$. À ces dernières nous ajoutons de nouvelles bordures $]U^\sharp; U]$ qui ont bien sûr les mêmes avantages que les bordures classiques : éviter de calculer les deux cubes sous-jacents à comparer, permettre de savoir si un renversement de tendance est effectif, offrir un outil de classification de tendances et focaliser l'attention de l'utilisateur sur un ensemble particulier de tuples intéressants. Cet ensemble peut être le point de départ pour l'exploration des résultats. De plus, $U^\sharp$ a un autre avantage par rapport à L car il nous permet de solidement caractériser la taille des Cubes Émergents (*cf.* chapitre 6). Nous avons formalisé le lien entre les deux couples de bordures ce qui permet d'utiliser des algorithmes existants pour calculer L. Des approches permettant l'obtention du Cube Émergent et de ses bordures ont aussi été proposées. Elles englobent des solutions bases de données, via des requêtes SQL, et des solutions algorithmiques efficaces. Parmi les requêtes SQL proposées, la plus simple et la plus efficace s'appuie sur une relation qui fusionne les deux relations considérées pour le calcul du cube émergent. En proposant les solutions algorithmiques, notre idée était d'exploiter une approche existante, évidemment performante et intégrable au cœur d'un SGBD. Ces raisons expliquent notre choix de l'algorithme BUC. Nous avons proposé le premier algorithme de calcul des bordures d'un cube F-IDEA ainsi que celui du calcul de Cubes Émergents E-IDEA. Des expérimentations conséquentes ont été menées dans un double objectif, d'une part évaluer la taille du cube émergent et celle de ses bordures et d'autre part mesurer les temps d'exécution nécessaires pour obtenir les résultats précédents. Concernant le premier objectif, nos attentes sont

confirmées par une réduction considérable de la taille des deux couples de bordures par rapport à celle du cube émergent. De plus, la nouvelle bordure $U^\sharp$ se révèle en pratique nettement plus réduite que la bordure L. Concernant les temps d'exécution, trouver des approches de comparaison était plus délicat puisque, en dehors de nos propositions, il n'existe pas d'algorithme calculant le cube émergent ni d'algorithme de calcul de bordures d'un cube. Nous avons remédié à ce problème en comparant d'une part E-Idea et la plus performante des requêtes Sql proposées et d'autre part F-Idea et une adaptation de l'algorithme Mafia au contexte multidimensionnel. Les résultats obtenus montrent l'efficacité de E-Idea et le relativement bon comportement de la meilleure de nos requêtes, qui sur des jeux de données volumineux, rend le résultat en un temps acceptable. Comparé à l'adaptation de Mafia, F-Idea se révèle plus performant montrant ainsi qu'un algorithme conçu directement pour un contexte multidimensionnel est plus efficace.

6

Estimation de la taille du Cube Émergent

Sommaire

6.1 Introduction et Motivations

S'il est bien un domaine dans lequel connaître la taille du résultat, qui sera manipulé par l'utilisateur, est primordial, c'est celui des entrepôts de données. La raison en est évidente : les énormes volumes de données collectées, générées et stockées. Ainsi, connaître la taille d'un futur data cube c'est donner à l'administrateur du système une information clef pour en gérer l'organisation. Il peut alors savoir si l'espace libre disponible est suffisant pour stocker ce data cube avant même de lancer son calcul coûteux. Et, si ce n'est pas le cas, il peut savoir combien d'espace disque libérer ou, si la matérialisation complète est impossible, il dispose d'une information capitale pour effectuer le choix des cuboïdes à matérialiser. Connaître la taille du résultat, c'est aussi pouvoir en estimer le temps de calcul en déterminant en particulier le facteur le plus important, le temps d'écriture sur disque.

Pour un cube de données complet, borner la taille est bien connu. Mais, comme nous l'avons vu, l'application de la double contrainte d'émergence, avec ses seuils spécifiés par l'utilisateur, peut conduire aux deux cas extrêmes, soit un résultat vide, soit le data cube complet de la deuxième relation. Dans ce contexte, prévoir la taille du Cube Émergent est crucial. En effet, outre les raisons invoquées plus haut, il en est une encore plus importante. Avant même d'entreprendre le calcul du Cube Émergent, l'utilisateur va savoir s'il va crouler sous des monceaux d'informations tellement volumineux qu'ils en seront inexploitables ou, au contraire, s'il ne va

isoler que quelques phénomènes très exceptionnels. En d'autres termes, le décideur va savoir si sa double contrainte d'émergence est réellement adaptée aux données étudiées. En imaginant que l'information sur la taille du Cube Émergent puisse lui être communiquée de manière quasi instantanée, c'est un véritable processus de calibrage des contraintes qui lui est offert. En effet, en fonction du volume prévu pour le Cube Émergent et compte tenu du taux d'émergence (*cf.* Chapitre 5), l'utilisateur peut renforcer les contraintes d'émergence ou au contraire les relâcher jusqu'à ce que la taille du futur résultat semble convenable : assez large pour que des renversements de tendance pertinents soient capturés mais suffisamment réduite pour que la connaissance extraite soit effectivement exploitable. Enfin, une méthode alternative pour calculer les Cubes Émergents utilisant les bordures (sans appliquer directement la contrainte d'émergence) peut être mise en œuvre de façon fructueuse à condition cependant que les bordures réduisent significativement la taille des données manipulées. Une fois encore, prédire la taille du résultat est important pour savoir quelle sera la meilleure méthode de calcul.

Dans ce chapitre, nous développons une méthode permettant de prédire la taille des Cubes Émergents. Malheureusement, il n'est pas possible de prédire la taille exacte des Cubes Émergents à cause de la contrainte d'émergence qui bien sûr varie. C'est pourquoi le calcul d'une borne supérieure est tout d'abord proposé. Son intérêt est de donner une idée immédiate mais grossière du futur volume du résultat. Il correspond au pire cas, quand tous les tuples de r_2 révèlent des tendances émergentes, i.e. le data cube de r_2. Nous pouvons alors utiliser les méthodes analytiques décrites dans Feller (1971) pour évaluer la cardinalité des cubes. Nous améliorons une de ces méthodes afin de retourner très rapidement une borne supérieure pour la taille des Cubes Émergents. Toutefois, cette borne supérieure reste insensible aux contraintes et la problématique est maintenant de tenir compte de la contrainte d'émergence sans énumération des tuples. La méthode que nous proposons utilise les bordures comme une expression des contraintes. Pour chaque couple de bordures proposé, nous introduisons une caractérisation solidement fondée de la taille exacte du Cube Émergent, en nous appuyant sur le concept d'idéal d'un ordre (Ganter et Wille, 1999) et le principe d'inclusion-exclusion (Matousek et Nesetril, 2004). Malheureusement, le calcul de ces expressions nécessite un temps d'exécution coûteux (proche du temps de calcul d'un data cube). Il est donc impératif d'explorer une autre voie pour les évaluer. Puisque le temps de réponse est le paramètre le plus critique pour notre processus de calibrage, nous choisissons de relâcher le paramètre de l'exactitude en fournissant très rapidement à l'utilisateur une valeur approximative de la taille attendue avec une précision convenable. Notre nouvelle stratégie pour approximer la taille des Cubes Émergents s'appuie sur les bordures proposées et l'adaptation à notre contexte de travail de l'algorithme proche de l'optimal HyperLogLog Flajolet *et al.* (2007). Pour montrer la faisabilité de notre démarche, nous procédons à des évaluations expérimentales en exploitant les mêmes jeux de données que ceux utilisés au chapitre précédent (*cf.* paragraphe 5.5). Pour ces expérimentations, notre objectif est double : d'une part calculer la taille approximative du Cube Émergent et la comparer avec sa taille réelle de manière à quantifier la précision de la méthode d'approximation et d'autre part à évaluer le temps d'exécution nécessaire pour obtenir cette taille approximative. Pour le premier objectif, des résultats pertinents sont obtenus puisque l'erreur moyenne est systématiquement en dessous de 5%. Quant aux temps de réponse, ils sont parfaitement compatibles avec les attentes des utilisateurs.

Ce chapitre est organisé de la façon suivante. Nous proposons la caractérisation de la taille exacte des Cubes Émergents au paragraphe 6.2. Puis nous utilisons une méthode d'estimation statistique pour obtenir une borne supérieure de cette taille (Cf. paragraphe 6.3.1). Pour approximer la taille exacte des Cubes Émergents, nous utilisons une méthode de comptage probabiliste décrite au paragraphe 6.3.2. Les expérimentations menées sont décrites et commentées au paragraphe 6.4.

TABLE 6.1 – Relation DOCUMENT$_1$ correspondant aux ventes de livres de l'année 2009

Type	Ville	Éditeur	Langue	Quantité
Nouvelles	Marseille	Collins	Français	100
Nouvelles	Marseille	Hachette	Anglais	100
Pédagogie	Paris	Hachette	Français	100
Essai	Paris	Hachette	Français	600
Pédagogie	Marseille	Hachette	Anglais	100

6.2 Caractérisation de la taille exacte des Cubes Émergents

Dans ce paragraphe, nous caractérisons la taille exacte des Cubes Émergents en utilisant les couples de bordures introduits. Les bordures et les contraintes sont intrinsèquement liées : en effet les bordures sont caractérisées par leurs contraintes et les contraintes sont représentées par les bordures. La nature des contraintes, monotone et anti-monotone, permet de décrire l'ensemble des solutions en s'appuyant sur les concepts d'idéal et de filtre. Effectivement les tuples satisfaisant une contrainte anti-monotone sont tous ceux généralisant un tuple de la bordure maximale associée. Ils constituent l'idéal pour l'ordre de généralisation de cette bordure. Inversement les tuples respectant une contrainte monotone sont ceux spécialisant un tuple de la bordure minimale associée. Ils appartiennent au filtre de l'ordre considéré.
Nous commençons par rappeler les définitions d'idéal et de filtre d'un ordre (Stumme *et al.*, 2002).

Définition 6.1 (Idéal d'un Ordre) **-** Soit $T \subseteq CL(r)$ un ensemble de tuples. Un idéal d'ordre généré par T est noté $\downarrow T$ et comprend tous les tuples généralisant au moins un tuple de T. $\downarrow T$ est défini comme suit :

$$\downarrow T = \{t \in CL(r) \mid \exists t' \in T \mid t \preceq_g t'\}$$

Définition 6.2 (Filtre d'un Ordre) **-** Soit $T \subseteq CL(r)$ un ensemble de tuples. Un filtre d'ordre généré par T est noté $\uparrow T$ et comprend tous les tuples spécialisant au moins un tuple de T. $\uparrow T$ est défini comme suit :

$$\uparrow T = \{t \in CL(r) \mid \exists t' \in T \mid t' \preceq_g t\}$$

Exemple 6.1 - Pour faciliter la lecture, nous redonnons ci-dessous les deux relations utilisées DOCUMENT$_1$ (*cf.* table 6.1) et DOCUMENT$_2$ (*cf.* table 6.2) donnant les quantités de livres vendues par Type, Ville, Éditeur et Langue. Nous utilisons également le même Cube Émergent que dans le chapitre précédent (*cf.* table 6.3 et 6.4). L'idéal de la bordure $U^\sharp$ est donné dans la table 6.5

À partir des tuples maximaux satisfaisant la contrainte antimonotone, l'ensemble complet des solutions est l'idéal d'ordre généré par ces tuples maximaux. En utilisant cette caractéristique, nous introduisons une nouvelle définition du Cube Émergent à travers la proposition suivante.

Proposition 6.1 - Soit $]U^\sharp, U]$ les bordures du Cube Émergent EC(r_1,r_2). En utilisant les idéaux des bordures le Cube Émergent peut être exprimé de la manière suivante :

$$\text{EC}(r_1,r_2) = \downarrow U \setminus \downarrow U^\sharp$$

TABLE 6.2 – Relation exemple DOCUMENT$_2$ correspondant aux ventes de livres de l'année 2010

Type	Ville	Éditeur	Langue	Quantité
Nouvelles	Marseille	Collins	Français	300
Pédagogie	Marseille	Collins	Anglais	300
Pédagogie	Marseille	Hachette	Français	300
Pédagogie	Paris	Hachette	Anglais	300
Essai	Marseille	Hachette	Français	100
Essai	Paris	Hachette	Français	200
Essai	Paris	Collins	Français	200

TABLE 6.3 – Cube Émergent de DOCUMENT$_1$ vers DOCUMENT$_2$

Tuple Émergent	Tuple Émergent
('ALL' , 'ALL' , 'ALL' , Anglais)	('ALL' , 'ALL' , Collins , 'ALL')
('ALL' , 'ALL' , Collins , Anglais)	('ALL' , 'ALL' , Collins , Français)
('ALL' , 'ALL' , Hachette, Anglais)	('ALL' , Paris , 'ALL' , Anglais)
('ALL' , Paris , Hachette, Anglais)	('ALL' , Marseille, 'ALL' , Anglais)
('ALL' , Marseille, 'ALL' , Français)	('ALL' , Marseille, Collins , 'ALL')
('ALL' , Marseille, Collins , Anglais)	('ALL' , Marseille, Collins , Français)
('ALL' , Marseille, Hachette, 'ALL')	('ALL' , Marseille, Hachette, Français)
(Nouvelles, 'ALL' , 'ALL' , 'ALL')	(Nouvelles, 'ALL' , 'ALL' , Français)
(Nouvelles, 'ALL' , Collins , 'ALL')	(Nouvelles, 'ALL' , Collins , Français)
(Nouvelles, Marseille, 'ALL' , 'ALL')	(Nouvelles, Marseille, 'ALL' , Français)
(Nouvelles, Marseille, Collins , 'ALL')	(Nouvelles, Marseille, Collins , Français)
(Pédagogie, 'ALL' , 'ALL' , 'ALL')	(Pédagogie, 'ALL' , 'ALL' , Anglais)
(Pédagogie, 'ALL' , 'ALL' , Français)	(Pédagogie, 'ALL' , Collins , 'ALL')
(Pédagogie, 'ALL' , Collins , Anglais)	(Pédagogie, 'ALL' , Hachette, 'ALL')
(Pédagogie, 'ALL' , Hachette, Anglais)	(Pédagogie, 'ALL' , Hachette, Français)
(Pédagogie, Paris , 'ALL' , 'ALL')	(Pédagogie, Paris , 'ALL' , Anglais)
(Pédagogie, Paris , Hachette, 'ALL')	(Pédagogie, Paris , Hachette, Anglais)
(Pédagogie, Marseille, 'ALL' , 'ALL')	(Pédagogie, Marseille, 'ALL' , Anglais)
(Pédagogie, Marseille, 'ALL' , Français)	(Pédagogie, Marseille, Collins , 'ALL')
(Pédagogie, Marseille, Collins , Anglais)	(Pédagogie, Marseille, Hachette, 'ALL')
(Pédagogie, Marseille, Hachette, Français)	

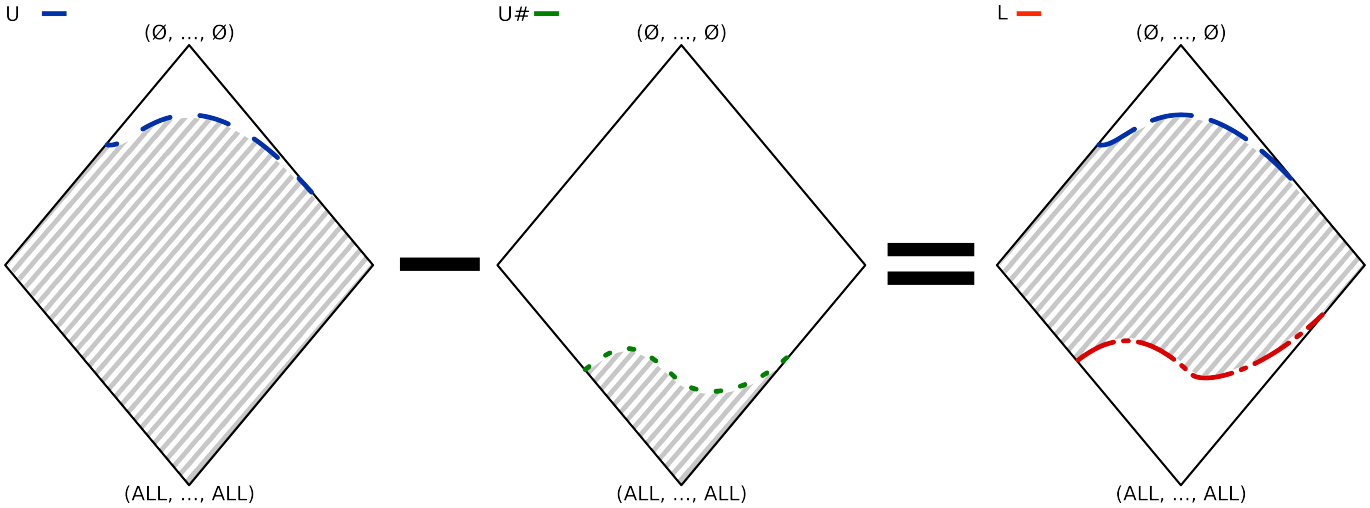
TABLE 6.4 – Bordures U, L et $U^\sharp$ du Cube Émergent

U	(Nouvelles, Marseille, Collins, Français)
	(Pédagogie, Marseille, Collins, Anglais)
	(Pédagogie, Paris, Hachette, Anglais)
	(Pédagogie, Marseille, Hachette, Français)
L	(Nouvelles, ALL, ALL, ALL)
	(Pédagogie, ALL, ALL, ALL)
	(ALL, ALL, Collins, ALL)
	(ALL, ALL, ALL, Anglais)
	(ALL, Marseille, ALL, Français)
	(ALL, Marseille, Hachette, ALL)
$U^\sharp$	(ALL, Paris, Hachette, ALL)
	(ALL, Marseille, ALL, ALL)
	(ALL, ALL, ALL, Français)

TABLE 6.5 – Idéal d'ordre de la bordure $U^\sharp$

(ALL, Paris, Hachette, ALL)
(ALL, ALL, Hachette, ALL)
(ALL, Paris, ALL, ALL)
(ALL, Marseille, ALL, ALL)
(ALL, ALL, ALL, Français)
(ALL, ALL, ALL, ALL)

FIGURE 6.1 – Illustration de la proposition 6.1



Démonstration.

$$\begin{aligned}\text{Ec}(r_1, r_2) = \\ &= \{t \in CL(r) \mid C_1(t) \wedge C_2(t)\} \\ &= \{t \in CL(r) \mid t \in{\downarrow} U \text{ et } t \notin{\downarrow} U^\sharp\} \\ &= \{t \in CL(r) \mid t \in{\downarrow} U\} \backslash \{t \in CL(r) \mid t \in{\downarrow} U^\sharp\}\end{aligned}$$

Les tuples de $U^\sharp$ satisfont la contrainte C_2 et donc $U^\sharp \subset{\downarrow} U$ et ${\downarrow} U^\sharp \subset{\downarrow} U$
Donc $\text{Ec}(r_1, r_2) = {\downarrow} U \backslash {\downarrow} U^\sharp$ □

La proposition 6.1 est illustrée dans la figure 6.1 où le premier losange représente, dans le treillis cube, la bordure maximale U et l'idéal d'ordre généré (zone hachurée). Le deuxième losange symbolise la bordure $U^\sharp$ et son idéal. Le troisième losange est obtenu en privant le premier idéal du second. Le résultat est le Cube Émergent.

Proposition 6.2 - Soit $]U^\sharp, U]$ les bordures du Cube Émergent $\text{Ec}(r_1,r_2)$. La taille de ce dernier peut être exprimée de la manière suivante selon le couple de bordure utilisée :

$$|\text{Ec}(r_1,r_2)| = |{\downarrow} U| - |{\downarrow} U^\sharp|$$

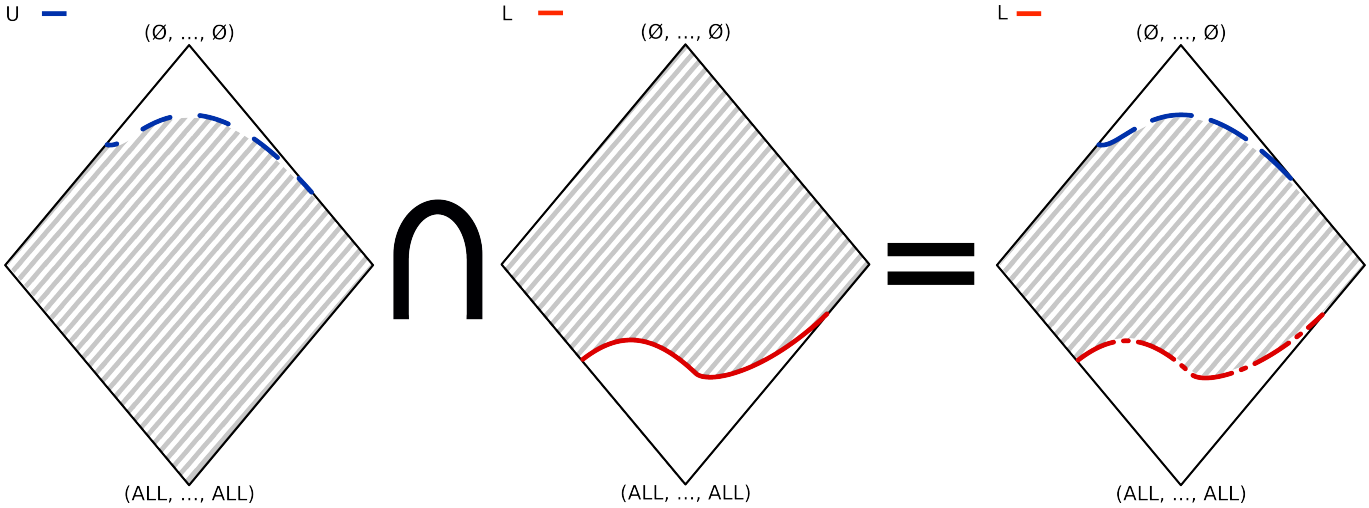
Démonstration.
Directe à partir de la proposition 6.1 □

Dualement, à partir des tuples minimaux satisfaisant la contrainte monotone, l'ensemble complet des solutions est le filtre d'ordre généré par ces tuples minimaux. En nous appuyant sur ce constat, nous proposons une caractérisation du Cube Émergent à travers la proposition suivante.

Proposition 6.3 - Soit $[L, U]$ les bordures du Cube Émergent $\text{Ec}(r_1,r_2)$. En utilisant l'idéal de la bordure U et le le filtre de L, le Cube Émergent peut être caractérisé de la manière suivante :

$$\text{Ec}(r_1,r_2) = {\downarrow} U \cap {\uparrow} L$$

FIGURE 6.2 – Illustration de la proposition 6.3



Démonstration.

$$\begin{aligned}\textsc{Ec}(r_1, r_2) = \\ &= \{t \in CL(r) \mid C_1(t) \wedge C_2(t)\} \\ &= \{t \in CL(r) \mid t \in{\downarrow} U \text{ et } t \in{\uparrow} L\}\end{aligned}$$

Donc $\textsc{Ec}(r_1, r_2) = {\downarrow} U \cap {\uparrow} L$ □

La figure 6.2 illustre la proposition 6.3. L'idéal de la bordure U et le filtre de L sont représentés par les deux premiers losanges. Leur intersection résulte dans le Cube Émergent.

Proposition 6.4 - Soit $[L, U]$ les bordures du Cube Émergent $\textsc{Ec}(r_1,r_2)$. La taille de ce dernier peut être exprimée de la manière suivante selon le couple de bordure utilisée :

$$|\textsc{Ec}(r_1,r_2)| = |{\downarrow} U \cap {\uparrow} L| = |{\downarrow} U| + |{\uparrow} L| - |{\downarrow} U \cup {\uparrow} L|$$

Démonstration.
Directe à partir de la proposition 6.3 □

Cette deuxième caractérisation a un double inconvénient par rapport à la première. D'abord, elle utilise les deux concepts d'idéal et de filtre ce qui nécessiterait de développer deux algorithmes de calcul alors que seul le calcul d'un idéal d'ordre est requis pour la première caractérisation. Ensuite elle s'appuie sur la cardinalité de l'union de deux ensembles, idéal et filtre, potentiellement de grande taille et dont rien ne dit qu'ils sont disjoints. Ces deux inconvénients nous font bien évidemment préférer la première caractérisation basée sur $U^\sharp$. Ainsi, outre sa taille plus réduite, son temps de calcul plus court, $U^\sharp$ a un autre avantage par rapport à L : permettre une caractérisation moins laborieuse que L.

Il convient à présent de s'intéresser au calcul d'un idéal. En adaptant le principe d'inclusion-exclusion (Matousek et Nesetril, 2004) au treillis cube, nous donnons une méthode pour calculer la cardinalité d'un idéal d'ordre généré par un ensemble de tuples $T = \{t_1, t_2, \ldots, t_n\}$. En préambule, nous définissons deux suites outils (E) et (I) :

TABLE 6.6 – Exécution des suites outils pour calculer l'idéal de $U^\sharp$

E_0	$\emptyset$		
E_1	(ALL, Paris, Hachette, ALL)	I_1	$\emptyset$
E_2	(ALL, Paris, Hachette, ALL) (ALL, Marseille, ALL, ALL)	I_2	(ALL, ALL, ALL, ALL)
E_3	(ALL, Paris, Hachette, ALL) (ALL, Marseille, ALL, ALL) (ALL, ALL, ALL, Français)	I_2	(ALL, ALL, ALL, ALL)

$$\begin{cases} E_0 &= \emptyset \\ E_1 &= \{t_1\} \\ &\vdots \\ E_n &= \{t_1, t_2, \ldots, t_n\} \end{cases} \qquad \begin{cases} I_1 &= \emptyset \\ I_2 &= \{t_2 + t_1\} \\ &\vdots \\ I_n &= \{t_n + t \mid t \in E_{n-1}\} \end{cases}$$

Proposition 6.5 - Soit $T \subseteq CL(r)$ un ensemble de tuples. La fonction récursive suivante calcule la taille de l'idéal d'ordre généré par T :

$$\begin{cases} |\downarrow T| &= |\downarrow E_n| \\ |\downarrow E_n| &= |\downarrow E_{n-1}| + |\downarrow t_n| - |\downarrow I_n| \\ |\downarrow t_n| &= 2^{|Attr(t)|} \end{cases}$$

Exemple 6.2 - La table 6.6 illustre le déroulement des suites (E) et (I) pour la bordure $U^\sharp$ de nos relations d'exemple. La suite est calculée comme suit :

$$\begin{aligned} |\downarrow U^\sharp| &= |\downarrow E_3| \\ |\downarrow E_3| &= |\downarrow E_2| + |\downarrow (\text{ALL, ALL, ALL, Français})| - |\downarrow (\text{ALL, ALL, ALL, ALL})| \\ &= |\downarrow E_2| + 2^1 - 1 = |\downarrow E_2| + 1 \\ |\downarrow E_2| &= |\downarrow E_1| + |\downarrow (\text{ALL, Marseille, ALL, ALL})| - |\downarrow (\text{ALL, ALL, ALL, ALL})| \\ &= |\downarrow E_1| + 2^1 - 1 = |\downarrow E_1| + 1 \\ |\downarrow E_1| &= |\downarrow (\text{ALL, Paris, Hachette, ALL})| = 2^2 = 4 \end{aligned}$$

En remontant on obtient $|\downarrow U^\sharp| = 6$ ce qui correspond au résultat obtenu dans la table 6.5

Dans le même esprit que les suites de Berge et Norris, nous avons proposé une méthode de calcul pour déterminer la taille exacte du Cube Émergent à partir de la caractérisation basée sur les idéaux des bordures (*cf.* Proposition 6.2). Cependant, la suite donnée souffre du même inconvénient que les autres suites. Malgré un caractère didactique indéniable, elle est peu efficace. En effet, elle requiert un temps d'exécution exponentiel. Le problème étant $\sharp P$-complet, il n'y a que peu d'espoir de trouver une méthode réellement exploitable en pratique. Néanmoins, la caractérisation de taille proposée est intéressante car elle pourra être approximée. Nous montrons la faisabilité d'une telle approximation au paragraphe 6.3.2 et de surcroît obtenons des résultats ayant une bonne précision.

6.3 Méthodes d'estimation

Dans ce paragraphe, nous présentons deux méthodes permettant d'approximer la taille des Cubes Émergents. La première est une adaptation directe d'une des approches proposées pour le data cube et donc elle ne prend pas en compte la contrainte d'émergence. Cette méthode nous donne une borne supérieure pour la taille qui nous servira de point de référence. La seconde méthode, quant à elle, vise à améliorer l'approximation en intégrant les contraintes grâce à la nouvelle caractérisation basée sur les bordures que nous avons proposée dans le paragraphe précédent.

6.3.1 Méthode d'estimation statistique

En considérant l'hypothèse suivant laquelle les données sont uniformément distribuées, un résultat standard en statistique pour estimer la taille d'un ensemble est le suivant (Feller, 1971) : Si P éléments sont choisis uniformément et au hasard à partir d'un ensemble de N éléments, le nombre attendu d'éléments distincts est $N - N(1 - 1/N)^P$.

Cette formule peut être utilisée pour donner une estimation de la taille d'un cuboïde et, en sommant toutes ces estimations, la taille du cube de données peut être approximée Shukla *et al.* (1996) :

$$|Datacube(r)| \leq \sum_{X \subseteq \mathcal{D}} N_X - N_X(1 - \frac{1}{N_X})^{|r|}$$

La formule peut être adaptée pour rapidement trouver la borne supérieure de la taille du Cube Émergent puisque, dans le pire des cas, le Cube Émergent est le data cube de r_2. Outre cette adaptation, nous affinons cette approximation. En effet, un cuboïde est un agrégat de la relation d'origine, sa taille est donc forcément plus réduite que celle de la relation. Or ce constat n'est pas considéré dans la formule précédente. Nous avons donc :

$$\begin{array}{lcl} |\text{EC}(r_1,r_2)| & \leq & |Datacube(r_2)| \\ |Datacube(r_2)| & \leq & \sum\limits_{X \subseteq \mathcal{D}} \min(N_X - N_X(1 - \frac{1}{N_X})^{|r_2|}, |r_2|) \\ |\text{EC}(r_1,r_2)| & \leq & \sum\limits_{X \subseteq \mathcal{D}} \min(N_X - N_X(1 - \frac{1}{N_X})^{|r_2|}, |r_2|) \end{array}$$

où $N_X = \prod_{D_i \in X} |r_2(D_i)|$ et $r_2(D_i)$ la projection de r_2 sur la dimension D_i et N_X est l'estimation de la taille du cuboïde selon l'ensemble de dimensions X, bien sûr $N_X \leq |r_2|$

Soulignons que le résultat présenté n'est qu'une borne supérieure sommaire car elle provient d'une approximation doublement imprécise. Tout d'abord l'hypothèse de données uniformément réparties correspond au pire cas pour la taille du data cube et ce dernier est le pire des cas pour le Cube Émergent.

6.3.2 Méthode d'estimation par comptage probabiliste

L'une des raisons pour laquelle la borne supérieure précédente est imprécise est que son calcul n'intègre pas la contrainte d'émergence. Il est impératif de la prendre en compte pour améliorer l'approximation. Notre idée de base est d'exploiter les bordures du Cube Émergent car elles sont les meilleurs représentations des contraintes (comme vu au chapitre 5, elles sont à la fois réduites et rapides à calculer). En s'appuyant sur la caractérisation de la taille exacte basée sur ces bordures (*cf.* Proposition 6.2), il suffit de savoir estimer la taille d'un idéal d'ordre pour en dériver celle du Cube Émergent. Il existe dans la littérature de nombreux algorithmes

pour estimer la taille de très grands ensembles d'éléments. Ces travaux ont des applications dans des domaines variés. Citons par exemple l'analyse du trafic internet par les routeurs pour traiter des aspects de sécurité aussi bien que de dimensionnement, ou encore l'optimisation de requètes en bases de données, pour trouver la stratégie algorithmique la plus adaptée. D'autres exemples sont la comparaison de documents, l'analyse du génome, la connectivité dans de grands graphes... Dans ces différents contextes, la problematique posée est la suivante : Étant donné une collection ou multi-ensemble d'éléments, comment estimer la cardinalité ou nombre d'éléments distincts de ce multi-ensemble, en une seule passe et en utilisant le moins de mémoire possible.

Une solution naïve à cette problèmatique est de parcourir l'ensemble des éléments et d'en conserver, en mémoire, les occurences distinctes. Malheureusement, dans le contexte qui nous intéresse où il s'agit de traiter des ensembles massifs de données, la liste des différents éléments ne rentre pas en mémoire, cette solution est donc inapplicable.

Pour répondre à cette problématique, les algorithmes proposés ont relâché la contrainte d'exactitude et développé des méthodes probabilistes donnant de bonnes estimations de la cardinalité (Flajolet *et al.*, 2007). Comme nous l'avons souligné l'estimation de la taille du Cube Émergent se raméne à l'estimation de la cardinalité d'un idéal d'ordre. Nous nous trouvons donc confrontés aux mêmes difficultés que les approches citées : parcourir un vaste ensemble de données, en déterminer efficacement la cardinalité tout en disposant d'une mémoire limitée. Il est alors naturel de choisir le meilleur algorithme, tant sur le plan de la rapidité que sur celui de la précision des résultats, et de l'adapter à notre contexte de travail.

Afin d'estimer la taille d'un multi-ensemble, Flajolet *et al.* (2007) propose un estimateur de cardinalité qualifié de quasi optimal. Des travaux ont procédé à des évaluations expérimentales, sur un large panel de jeux d'éssais variés, montrant la très grande rapidité de cet algorithme. De plus, l'erreur observée est typiquement inférieure à 5% (Flajolet *et al.*, 2007). HyperLogLog nécessite un seul balayage des données. Pour chaque tuple il effectue très peu d'opérations. Il requiert peu de mémoire et enfin il estime la cardinalité d'un multi-ensemble avec une très bonne précision. C'est pourquoi nous l'avons choisi.

Disposant d'un ensemble de tuples T, notre objectif est d'estimer la cardinalité de son idéal d'ordre. Générer un idéal d'ordre sans duplicat n'est pas envisageable car cette opération est aussi coûteuse que le calcul d'un cube de données. L'idée est alors de générer rapidement un sur-ensemble de $\downarrow T$ contenant des tuples dupliqués en s'épargnant le contrôle de l'unicité de ces tuples. Le multi-ensemble obtenu a alors autant d'éléments distincts que $\downarrow T$. Encore une fois la taille de cet ensemble pose problème, car même s'il peut être généré rapidement, il n'est pas possible de le garder en mémoire. Nous ne pouvons donc pas donner cet ensemble directement en entrée d'HyperLogLog. Pour remédier à ce problème, nous le modifions en intégrant l'étape de génération directement au cœur de l'algorithme Idea-LogLog, en utilisant la fonction Suivant. Cette fonction prend en paramêtre un tuple t et génére son suivant dans l'ordre lectique (*cf.* définition 3.25). Pour générer efficacement elle utilise la correspondance entre ordre lectique et ordre naturel des nombre binaire. Ainsi calculer le suivant d'un tuple revient à incrémenter un compteur et à effectuer un simple masquage. Les tuples sont ainsi directement comptabilisés sans avoir à être conservés en mémoire.

Expliquons l'intuition de cet algorithme. Une fonction de hachage est choisie de manière à générer pour chaque tuple une valeur binaire ayant l'apparence de l'aléa. Les valeurs hachées résultantes semblent pratiquement uniformes et indépendantes. Ces caractéristiques font qu'au sein d'un mot binaire obtenu, la probabilité de trouver le premier bit à 1, à la position n, est : 2^{-n}. En observant la valeur de cette position n pour tous les tuples, une idée grossière de la cardinalité peut être obtenue. En effet, la plus grande valeur de n, notée n_{max} et appelée observable, correspond à la moins probable des valeurs hachées. La rencontrer signifie vraisemblablement

Algorithme 28 SIGNATURE

Entrée : T ($T \subseteq CL(r)$)
Sortie : Une collection de registres contenant la valeur de l'observable pour les m expériences
define $m = 2^b$ avec $b = 6$
et $\alpha_m = 0.709$
let M une collection de m registres initialisés à 0
let $\rho(y)$ le rang du premier bit à 1 à partir de la gauche dans y
let $h : CL(r) \rightarrow \{0,1\}^{32}$ une bonne fonction de hachage
1: **pour tout** $t \in T$ **faire**
2: $t' := t$
3: **tant que** $t' \neq \emptyset$ **faire**
4: $x := h(t')$
5: $j := \langle x_1, x_2, \ldots, x_b \rangle_2$ //Les b premiers bits de x correspondant au numéro de l'expérience
6: $w := x_{b+1}, x_{b+2}, \ldots$ //w est le mot binaire sur lequel l'observation est réalisée
7: $M[j] := max(M[j], \rho(w))$ //$M[j]$ contient la plus grande valeur de la position du premier bit à 1 pour la j-ième expérience
8: $t' :=$ SUIVANT(t')
9: **fin tant que**
10: **fin pour**
11: **retourner** M

que l'ensemble a une cardinalité de l'ordre de $2^{n_{max}}$. Cependant le résultat obtenu manque de précision. L'idée est alors de simuler l'effet de m expériences de manière à raffiner la cardinalité jusqu'à obtenir une bonne approximation.

Pour simplifier la présentation d'IDEA-LOGLOG, nous l'avons découpé en deux algorithmes. Le premier, SIGNATURE (*cf.* algorithme 28), calcule la valeur de l'observable pour les m expériences simulées. Comme on peut le voir les opérations effectuées dans la boucle principale sont peu nombreuses et simples : hachage, découpage d'un mot binaire, affectation et appel de la fonction SUIVANT. Cette dernière retourne le prochain tuple à traiter. Elle aussi n'utilise que quelques opérations binaires peu coûteuses pour obtenir ce résultat.

Exemple 6.3 - La table 6.7 illustre le déroulement de l'algorithme SUIVANT pour le tuple (Nouvelles, Marseille, Collins).

Le deuxième sous-algorithme, ESTIM (*cf.* algorithme 29), calcule quant à lui l'estimation de la cardinalité à partir de la moyenne stochastique des valeurs des m observables. Cette dernière est ajustée pour tenir compte des caractéristiques de l'ensemble d'entrée. Cela permet d'obtenir des estimations satisfaisantes même lorsque l'ensemble a une cardinalité réduite.

Nous disposons des bordures qui condensent un Cube Émergent. En appliquant l'algorithme IDEA-LOGLOG (*cf.* algorithme 30) sur tous les tuples des bordures, nous pouvons estimer la cardinalité des idéaux d'ordre générés par U ou $U^\sharp$. L'algorithme retourne une cardinalité approximative avec une précision quasi optimale. Une fois cette cardinalité calculée, nous utilisons la proposition 6.2 pour dériver la cardinalité estimée du Cube Émergent.

La modification présentée d'HYPERLOGLOG est dédiée au calcul des idéaux et ne peut donc pas être utilisée pour estimer la taille du Cube Émergent à partir de la bordure L. Cependant elle peut être adaptée, pour calculer le filtre de L, en modifiant uniquement la fonction SUIVANT.

Algorithme 29 ESTIM

Entrée : M une collection de registres contenant la valeur de l'observable pour les m expériences
Sortie : Estimation de la cardinalité de $\downarrow T$

define $m = 2^b$ avec $b = 6$
et $\alpha_m = 0.709$

1: $E := \alpha_m m^2 . \left(\sum_{j=0}^{m-1} 2^{-M[j]}\right)^{-1}$
2: $E^* := E$
3: **si** $E \leq \frac{5}{2}m$ **alors**
4: **let** V le nombre de registres égaux à 0
5: **si** $V \neq 0$ **alors** $E^* := m.log(m/V)$
6: **fin si**
7: **si** $E \geq \frac{1}{30}2^{32}$ **alors**
8: $E^* := 2^{32} log(1 - E/2^{32})$
9: **fin si**
10: **retourner** E^*

Algorithme 30 IDEA-LOGLOG estime la cardinalité d'un idéal d'ordre généré par un ensemble de tuples émergents T

Entrée : T ($T \subseteq CL(r)$)
Sortie : Estimation de la cardinalité de $\downarrow T$

1: $M :=$ SIGNATURE(T)
2: $E :=$ ESTIM(M)
3: **retourner** E

Cette version retourne un tuple plus spécifique que son prédécesseur, il contient donc plus de valeurs différentes de ALL. L'ajout de valeurs demande des vérifications supplémentaires pour ne pas générer de tuples erronés. En conséquence, cette version est plus coûteuse que la précédente mais permet d'atteindre l'objectif visé si l'on ne dispose que des bordures $[L; U]$.

Comme nous l'avons souligné précédemment, l'estimation de la cardinalité du Cube Émergent nécessite non seulement de disposer des cardinalités de l'idéal et du filtre des bordures mais aussi de la cardinalité de leur union. Or ce qui pose problème dans le cas général peut être résolu facilement grâce à une propriété d'HYPERLOGLOG :

$$\begin{aligned} Signature(A \cup B) &= Max(Signature(A), Signature(B)) \\ |A \cup B| &= Estim(Signature(A \cup B)) \end{aligned}$$

où Max est la fonction qui calcule le maximum des signatures élément par élément.

Après avoir calculé la cardinalité du filtre et de l'idéal, la cardinalité de l'union est dérivée directement à partir des résultats intermédiaires précédents et la propriété ci-dessus. Ainsi en utilisant la proposition 6.4 nous obtenons une autre estimation de la cardinalité du Cube Émergent se basant sur les bordures $[L; U]$.

TABLE 6.7 – Exécutions de SUIVANT pour le tuple (Nouvelles, Marseille, Collins)

Tuple	Masque binaire
(Nouvelles, Marseille, Collins)	(1 1 1)
(Nouvelles, Marseille, ALL)	(1 1 0)
(Nouvelles, ALL, Collins)	(1 0 1)
(Nouvelles, ALL, ALL)	(1 0 0)
(ALL, Marseille, Collins)	(0 1 1)
(ALL, Marseille, ALL)	(0 1 0)
(ALL, ALL, Collins)	(0 0 1)
(ALL, ALL, ALL)	(0 0 0)

6.4 Évaluations Expérimentales

Afin de valider notre approche, nous avons effectué des différentes expérimentations. Nous calculons la taille approximative des Cubes Émergents ainsi que leur taille exacte. Leur comparaison montre la précision réellement convenable de la méthode d'approximation mise en œuvre. Ensuite, nous comparons les temps d'exécution nécessaires pour obtenir les deux précédents résultats, et, évidemment, la méthode d'approximation est plus efficace puisque l'obtention de la taille exacte nécessite le calcul, forcément plus coûteux, du Cube Émergent. Mais avant tout, notre intention est de montrer que notre stratégie pour calibrer la double contrainte d'émergence est exploitable car la taille approximative est obtenue particulièrement rapidement avec une bonne précision.

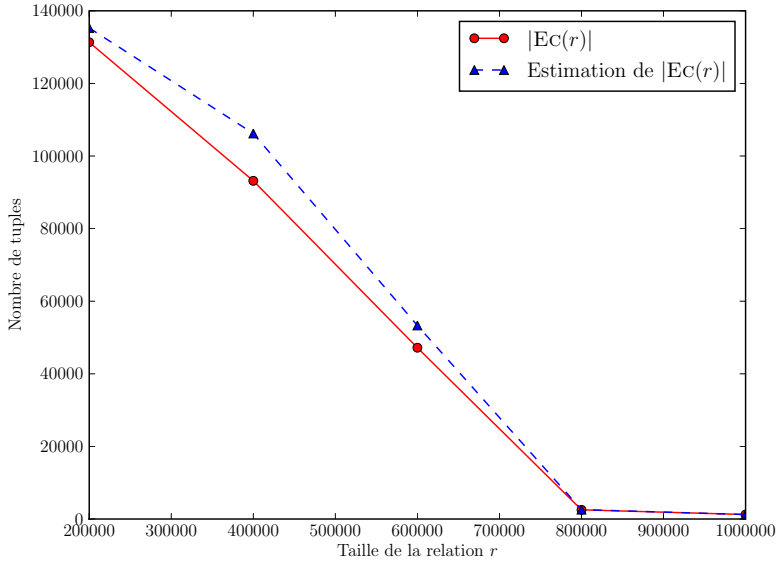
Les expérimentations sont menées sur des données issues d'un éventail large et varié de domaines similaires aux ensemble de données utilisés dans Xin *et al.* (2007). Il est bien connu que les données synthétiques sont faiblement corrélées alors que les données de bases réelles ou statistiques sont fortement corrélées. Pour les données synthétiques[6], nous utilisons les notations suivantes pour décrire les relations : $\mathcal{D}$ le nombre de dimensions, $\mathcal{C}$ la cardinalité de chaque dimension, $\mathcal{T}$ le nombre de tuples dans chaque relation, $\mathcal{M}_1$ ($\mathcal{M}_2$ respectivement) le seuil correspondant à la contrainte d'émergence C_1 (C_2 respectivement), et $\mathcal{S}$ le biais ou zipf des données. Quand $\mathcal{S}$ est egal à 0, les données sont uniformes. Quand $\mathcal{S}$ croît, les données sont plus biaisées. $\mathcal{S}$ est appliqué à toutes les dimensions dans chaque relation de la base de données. Pour les données réelles, nous utilisons les relations concernant la météorologie SEP83L.DAT et SEP85L.DAT utilisées dans Xin *et al.* (2006), qui ont 1.002.752 tuples avec 8 dimensions sélectionnées. Les attributs (avec leur cardinalités) sont les suivantes : année mois jour heure (238), latitude (5260), longitude (6187), nombre de stations (6515), temps actuel (100), code de changement (110), altitude solaire (1535) et luminance lunaire relative (155).

6.4.1 Estimation de la taille

Nos expérimentations concerne l'approximation de la taille des Cubes Émergents. Pour un couple de relations, nous calculons le Cube Émergent afin d'en obtenir la taille réelle. Nous utilisons l'algorithme IDEA-LOGLOG afin de déterminer sa taille approximative. Nous calculons également la borne supérieure analytique pour la taille du Cube Émergent.

6. Le générateur de données synthétique est disponible à : http://illimine.cs.uiuc.edu/

FIGURE 6.3 – Tailles exacte et approximative du Cube Émergent avec $\mathcal{D} = 10$, $\mathcal{C} = 100$, $\mathcal{S} = 0$



Pour les différentes relations, nous étudions l'influence, sur les taille ou borne calculées, de différents paramètres comme le nombre de tuples dans les relations originales, la cardinalité des dimensions, le biais des données et le seuil donné par l'utilisateur. Les résultats sont présentés dans les figures 6.3 à 6.6. Dans tous les cas, la taille exacte et la taille approximative sont très proches. Ces résultats renforcent notre idée qu'une bonne et rapide approximation de la taille fonctionne bien.

6.4.2 Temps d'exécution

Nous mesurons et comparons les temps d'exécution nécessaires pour d'une part calculer le Cube Émergent avec sa taille exacte et d'autre part obtenir la taille approximative de ce cube. Le facteur de gain varie de 300 à 500 (*cf.* figures 6.7 à 6.10).

FIGURE 6.4 – Tailles exacte et approximative du Cube Émergent avec $\mathcal{D} = 10$, $\mathcal{T} = 1000$K, $\mathcal{S} = 0$

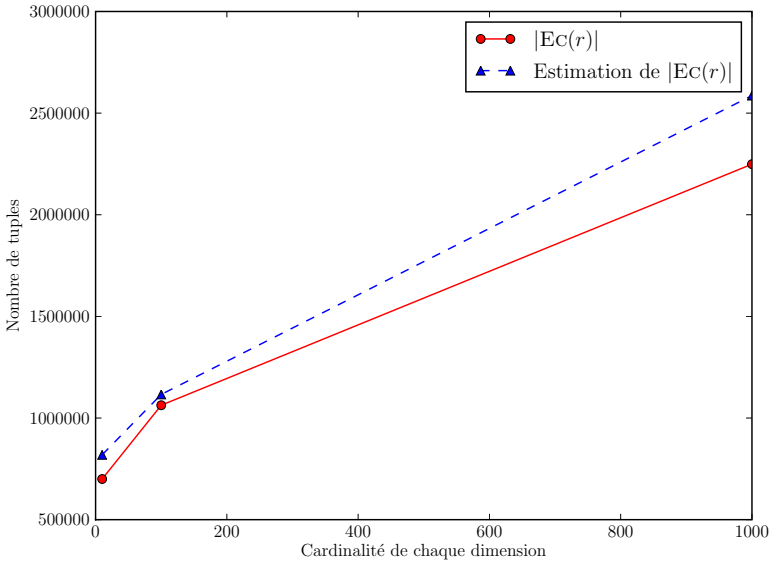


FIGURE 6.5 – Tailles exacte et approximative du Cube Émergent avec $\mathcal{D} = 10$, $\mathcal{C} = 100$, $\mathcal{T} = 1000$K

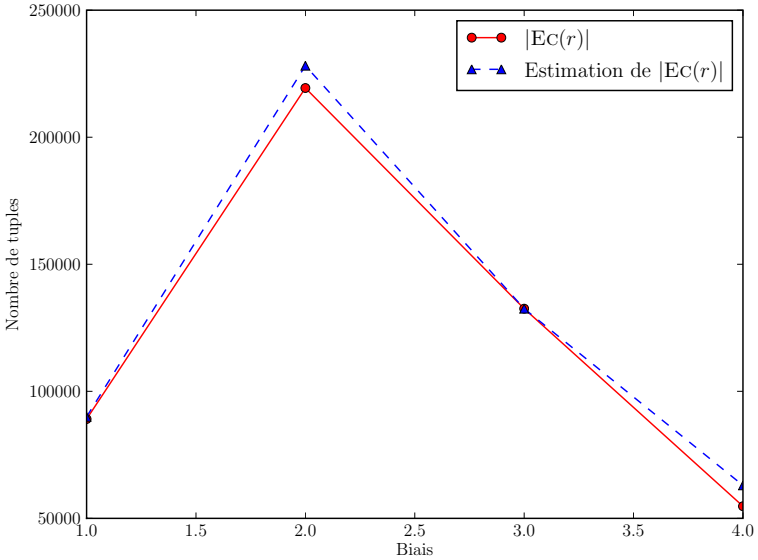


FIGURE 6.6 – Tailles exacte et approximative du Cube Émergent pour les données météorologiques

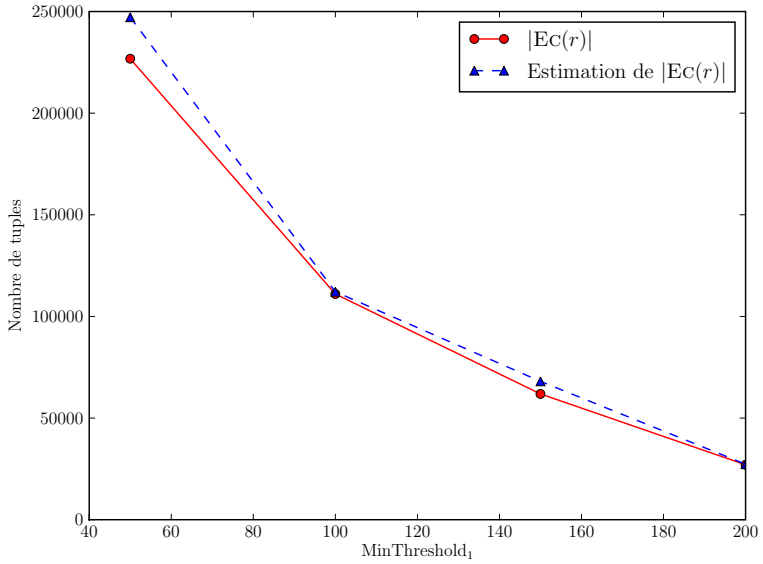


FIGURE 6.7 – Estimation et temps de calcul du Cube Émergent avec $\mathcal{D} = 10$, $\mathcal{C} = 100$, $\mathcal{S} = 0$

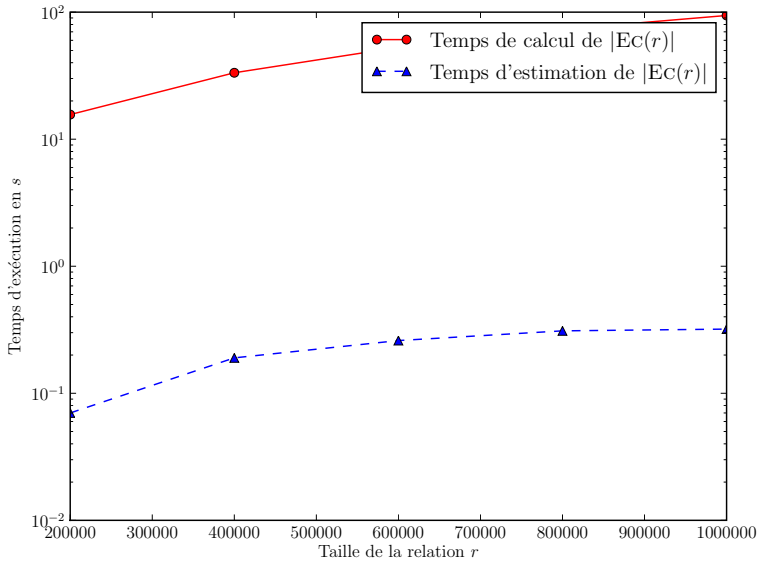


FIGURE 6.8 – Estimation et temps de calcul du Cube Émergent avec $\mathcal{D} = 10$, $\mathcal{T} = 1000$K, $\mathcal{S} = 0$

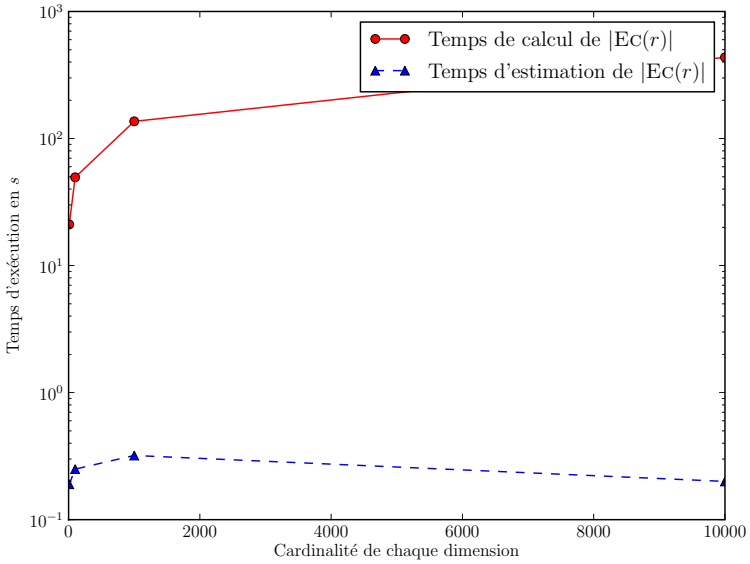


FIGURE 6.9 – Estimation et temps de calcul du Cube Émergent avec $\mathcal{D} = 10$, $\mathcal{C} = 100$, $\mathcal{T} = 1000$K

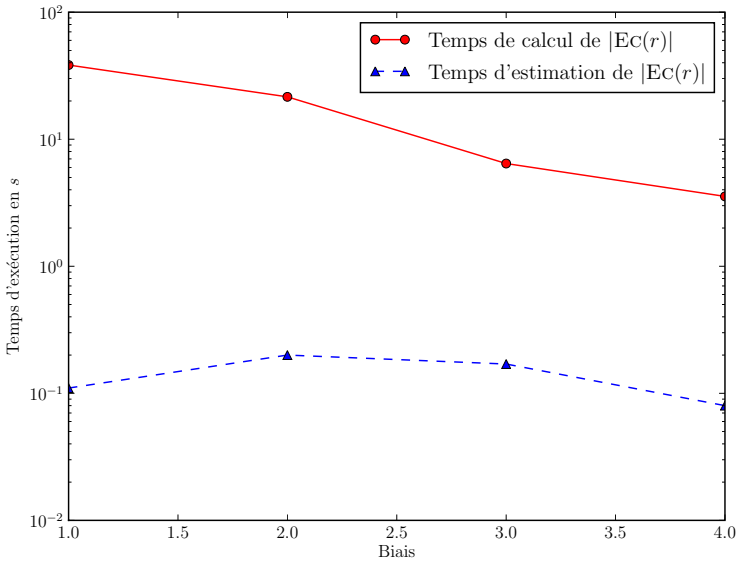
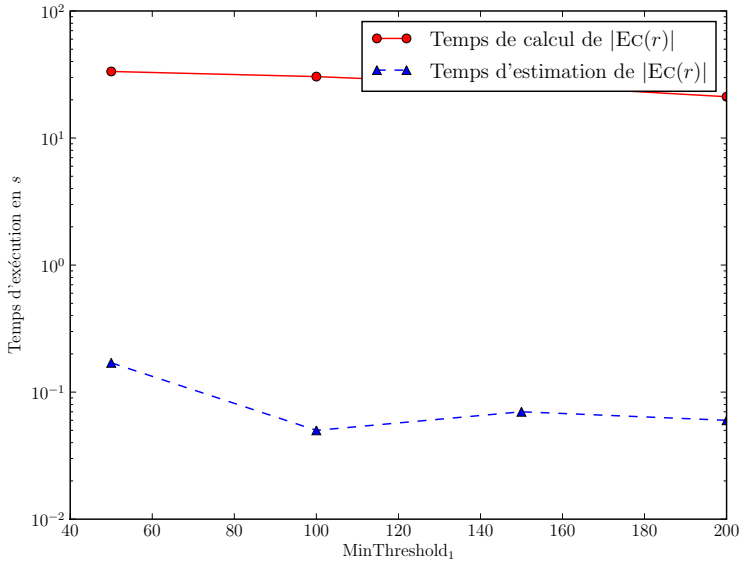


FIGURE 6.10 – Estimation et temps de calcul du Cube Émergent pour les données météorologiques



6.5 Conclusion

Pour que la découverte de nouveaux renversements de tendances soit fructueuse, il convient que le Cube Émergent ait une taille exploitable. Or cette taille dépend de la contrainte d'émergence spécifiée par l'utilisateur. Prédire la taille du Cube Émergent est donc particulièrement intéressant car doté de cette information, l'utilisateur peut ajuster la contrainte d'émergence et choisir des seuils réellement pertinents afin que les résultats soient manipulables. Mais une telle prédiction est vraiment exploitable à la condition qu'elle donne un résultat très rapidement. C'est seulement sous cette condition que les seuils pourront être calibrés au mieux. Afin de répondre à cet objectif, nous proposons une borne supérieure de la taille en question calculable quasi immédiatement. D'autre part, nous caractérisons solidement la taille exacte des Cubes Émergents. Une telle caractérisation s'appuie sur les bordures U, $U^\sharp$ et L qui sont une des meilleures représentations de la contrainte d'émergence, notamment de par leur taille réduite. Cependant, la classe de complexité à laquelle appartient notre problème laisse peu d'espoir de trouver un algorithme efficace pour calculer la taille recherchée. Néanmoins, les caractérisations proposées peuvent être approximées. Pour cela, nous adaptons l'algorithme HYPERLOGLOG qui implémente une méthode de comptage probabiliste. Son entrée est un couple de bordures et son résultat une taille approximative du Cube Émergent avec une précision quasi optimale. Enfin pour valider notre méthode, nous effectuons des expérimentations deux directions. Nous calculons et comparons les tailles exacte et approximative des Cubes Émergents et faisons de même pour les temps d'exécution. Les résultats obtenus sont convaincants : calculer la taille exacte est coûteux (même avec des algorithmes ayant prouvé leur efficacité) alors que la taille approximative est obtenue de manière particulièrement efficace et qu'elle est très proche de la taille exacte. La méthode d'approximation proposée peut s'appliquer directement pour estimer la taille de data cubes, de cubes icebergs, d'ensemble de motifs fréquents, d'ensemble de motifs

contraints Pei *et al.* (2004).

Une perspective intéressante des approches d'estimation présentées est de les utiliser comme un guide non seulement pour calibrer les seuils d'émergence mais aussi pour choisir l'algorithme de calcul de cube le mieux adapté. En effet, l'efficacité de ces algorithmes est fortement liée à la nature même des données (fortement ou faiblement corrélées) et la taille des futurs résultats est un paramètre crucial.

7

Représentations réduites du Cube Émergent

Sommaire

7.1 Introduction et Motivations

Offrir aux décideurs des Cubes Émergents est loin d'être trivial car deux data cubes, probablement volumineux donc coûteux à obtenir, à stocker et à gérer, doivent être calculés pour ensuite être comparés. En abordant cette problématique, notre idée est d'étudier des représentations réduites du Cube Émergent afin de minimiser chaque facette du problème. Des représentations réduites ont été largement étudiées en fouille de données et ont montré un véritable intérêt. Citons, par exemple, celles proposées pour les motifs fréquents (Pasquier *et al.*, 1999; Pei *et al.*, 2000; Zaki et Hsiao, 2002). Dans le contexte Olap, différentes approches visent la réduction de la taille des cubes de données, par exemple en éliminant les redondances intrinsèques au sein des cubes (Lakshmanan *et al.*, 2002; Casali *et al.*, 2003b; Morfonios et Ioannidis, 2006) ou en se focalisant sur les tendances les plus intéressantes à travers des cubes icebergs (Beyer et Ramakrishnan, 1999). Nous avons déjà présenté au chapitre 5 des représentations réduites pour

les Cubes Émergents qui s'appuient sur les bordures (Nedjar *et al.*, 2008, 2006a,b). Néanmoins, ces représentations ne permettent pas de retrouver les valeurs des mesures et donc ne peuvent pas être utilisées pour répondre à n'importe quelle requête OLAP.

Dans ce chapitre, nous proposons deux familles de représentations des Cubes Émergents qui éliminent les redondances, sont sans perte d'information et, bien sûr, évitent de calculer les deux cubes de données à comparer. La premiére s'appuie sur le cube fermé (*cf.* paragraphe 3.3.1 p.64) et la seconde sur le cube quotient (*cf.* paragraphe 3.3.2 p.67). Les représentations basées sur le cube fermé sont les plus réduites. Elles sont basées sur le concept de fermeture cubique (Casali *et al.*, 2003b). À partir de ces représentations, il est possible de retrouver les valeurs des mesures associées aux différentes tendances. Ainsi, le résultat de toute requête OLAP peut en être dérivé. Par exemple, nous pouvons répondre au type de questions suivant : « Comment évoluent les ventes de tel produit entre 2007 et 2008 par ville et par saison ? ». Ces représentations englobent l'ensemble des tuples fermés émergents auquel est ajoutée toute l'information nécessaire pour garantir une représentation sans perte. Cette information est soit la bordure L (*cf.* paragraphe 5.3.1 p.123), d'où le L-Cube Fermé Émergent (Nedjar *et al.*, 2010), soit la bordure $U^{\sharp}$ (*cf.* paragraphe 5.3.2 p.125) ce qui nous permet d'introduire le $U^{\sharp}$-Cube Fermé Émergent (Nedjar *et al.*, 2009, 2007b). Enfin nous réduisons la bordure $U^{\sharp}$ en éliminant les redondances. La nouvelle bordure introduite est appelée $U^{\sharp\sharp}$ et nous permet de proposer la représentation $U^{\sharp\sharp}$-Cube Fermé Émergent (Nedjar *et al.*, 2009, 2007b).
Certaines applications OLAP nécessitent non seulement de traiter des requêtes mais aussi de disposer de capacités de navigation au sein des cubes. En effectuant une telle navigation, l'utilisateur observe les données à différents niveaux de granularité. Par exemple, si un renversement de tendances très significatif apparaît à un niveau très agrégé, l'utilisateur peut vouloir cerner les origines du phénomène et donc « forer » dans le cube pour obtenir les données détaillées sous-jacentes. Néanmoins, la première famille de représentations n'offre pas de telles capacités de navigation. C'est pourquoi nous proposons une autre représentation qui, elle, les permet. Elle est basée sur la représentation par Cube Quotient (Lakshmanan *et al.*, 2002) mais elle n'est pas une simple adaptation de la structure citée à notre problématique de capture des tendances émergentes car, pour la caractériser, nous devons établir le lien entre le concept de fermeture cubique et le quotient cube. La représentation proposée est évidemment appelée Cube Quotient Émergent. De plus, nous établissons le lien existant entre certaines représentations proposées qui est un lien d'inclusion. Ce résultat analytique permet de guider le choix de la représentation la mieux adaptée au futur usage des Cubes Émergents. Les bordures minimale et maximale sont plus réduites que le L-Cube Fermé Émergent qui est lui-même plus réduit que le Cube Quotient Émergent. Évidemment, plus on souhaite de fonctionnalités, plus il est nécessaire de stocker de l'information et donc plus la taille de la représentation croît.

Enfin, nous effectuons des expérimentations afin de comparer la taille des représentations réduites et celle du Cube Émergent. Les résultats obtenus sont très positifs : pour les données réelles connues pour être fortement corrélées, les représentations apportent une importante réduction de taille.

Le premier paragraphe de ce chapitre est consacré aux représentations réduites s'appuyant sur le Cube Fermé. Avant de détailler la représentation réduite,le Cube Quotient Émergent(*cf.* paragraphe 7.3.2), nous rappelons ce qu'est un Cube Quotient mais en revisitant l'approche originale à la lumière du concept de fermeture (*cf.* paragraphe 7.3.1). Le paragraphe 7.6.3 décrit les évaluations expérimentales menées.

7.2 Cubes Fermés Émergents

Le Cube Fermé englobant l'ensemble des tuples fermés est actuellement une des représentations les plus réduites pour le data cube (Casali *et al.*, 2003b). Il est donc intéressant de proposer, pour le Cube Émergent, une structure s'appuyant sur les concepts associés au Cube Fermé. Malheureusement, une représentation uniquement basée sur les tuples fermés émergents est insuffisante pour être une représentation sans perte d'information. Nous montrons que, pour certains tuples, la valeur de la mesure ne peut pas être retrouvée. Pour éviter cet inconvénient, nous ajoutons l'information requise afin d'obtenir une représentation sans perte. Cette information est capturée soit par la bordure L, soit par la bordure $U^\sharp$, soit enfin par la bordure $U^\sharp$ réduite. Les trois représentations en découlant sont décrites dans les paragraphes suivants.

7.2.1 L-Cubes Fermés Émergents

Le **L**-Cube Fermé Émergent inclut à la fois *(i)* l'ensemble des tuples fermés émergents et *(ii)* la bordure L. Cette approche est conçue dans le même esprit que celle proposée par (Gollapudi et Sivakumar, 2004) dans le contexte de bases de données de transactions et qui englobe les motifs fermés contraints et la bordure minimale L.

Pour une raison de simplicité, nous utilisons, à partir de maintenant t au lieu de $(t, f_{val}(t,r))$ pour indiquer un tuple complet avec ses valeurs des dimensions et sa mesure.

L'idée de base de notre représentation est d'éliminer les redondances existant au sein des Cubes Émergents. Effectivement certains tuples véhiculent la même sémantique que d'autres qui sont plus agrégés. En fait, les uns et les autres sont construits en agrégeant exactement les mêmes tuples de la relation originale mais à des niveaux de granularité différents. Donc un seul tuple, le plus spécifique de tous, peut représenter tout l'ensemble. L'opérateur de Fermeture Cubique, présenté au paragraphe 3.3.1, est conçu pour calculer ce tuple représentatif. Nous rappelons ci-dessous les définitions dont nous allons avoir besoin pour définir nos représentations.

Définition 7.1 (Connexion cubique) **-** Soit $Rowid : r \to \mathbb{N}^*$ une fonction qui associe à chaque tuple un unique entier positif et $Tid(r) = \{Rowid(t) \mid t \in r\}$. Soit λ et σ deux fonctions définies comme suit :

$$\begin{array}{lccl} \lambda : & CL(r) & \to & \langle \mathcal{P}(Tid(r)), \subseteq \rangle \\ & t & \mapsto & \cup\{Rowid(t') \in Tid(r) \mid t \preceq_g t' \text{ et } t' \in r\} \end{array}$$

$$\begin{array}{lccl} \sigma : & \langle \mathcal{P}(Tid(r)), \subseteq \rangle & \to & CL(r) \\ & P & \mapsto & +\{t \in r \mid Rowid(t) \in P\} \end{array}$$

La fonction λ associe à tout tuple du treillis cube l'ensemble des identifiants des tuples qui le généralisent. La fonction σ quant à elle s'applique sur un ensemble d'identifiants de tuples et retourne la somme (*cf.* definition 3.4) de tous les tuples dotés de ces identifiants. En d'autres termes, elle renvoie le tuple le plus spécifique généralisant tous les tuples pourvus de ces identifiants.

Définition 7.2 (Fermeture Cubique) **-** Soit $T \subseteq CL(r)$ un ensemble de tuples, l'opérateur de Fermeture Cubique $\mathbb{C} : CL(r) \to CL(r)$ selon T peut être défini comme suit :

$$\mathbb{C}(t,T) = \sigma \circ \lambda(t) = (\emptyset, \ldots, \emptyset) + \sum_{\substack{t' \in T, \\ t \preceq_g t'}} t'$$

où l'opérateur $\sum$ a la même sémantique que l'opérateur $+$.

Considérons tous les tuples t' dans T. Agrégeons les ensemble en utilisant l'opérateur $+$. Nous obtenons un nouveau tuple qui généralise tous les tuples t' et qui est le plus spécifique. Ce nouveau tuple est la Fermeture Cubique de t.

Exemple 7.1 - Dans les exemples de ce chapitre, nous cherchons les renversements de tendances s'étant produits entre les années 2007 et 2008 pour la vente de produits repertoriées dans la relation VENTE (*cf.* table 7.1). La relation VENTE_{07} (*cf.* table 7.2) correspond aux produits vendus en 2007 et VENTE_{08} (*cf.* table 7.3) sélectionne les tuples de VENTE pour lesquels *Année* = 2008. Nous effectuons la Fermeture Cubique du tuple multidimensionnel (1, ALL, ALL) dans la relation VENTE_{07} en agrégeant tous les tuples qui le spécialisent grâce à l'opérateur $+$.

$$\begin{aligned}\mathbb{C}((1,\text{ ALL, ALL}),\text{VENTE}_{07}) &= (\emptyset,\ldots,\emptyset) + (1,\text{ Marseille, Printemps}) + (1,\text{ Marseille, Été})\\ &= (1,\text{ Marseille, ALL})\end{aligned}$$

TABLE 7.1 – Relation example VENTE

Produit	Ville	Saison	Année	Quantité
1	Marseille	Printemps	2007	100
1	Marseille	Été	2007	100
2	Paris	Été	2007	100
3	Paris	Été	2007	100
2	Marseille	Printemps	2008	200
2	Paris	Été	2008	100
1	Marseille	Printemps	2008	100
3	Paris	Été	2008	100
3	Paris	Automne	2008	300

TABLE 7.2 – Relation exemple VENTE_{07}

Produit	Ville	Saison	Quantité
1	Marseille	Printemps	100
1	Marseille	Été	100
2	Paris	Été	100
3	Paris	Été	100

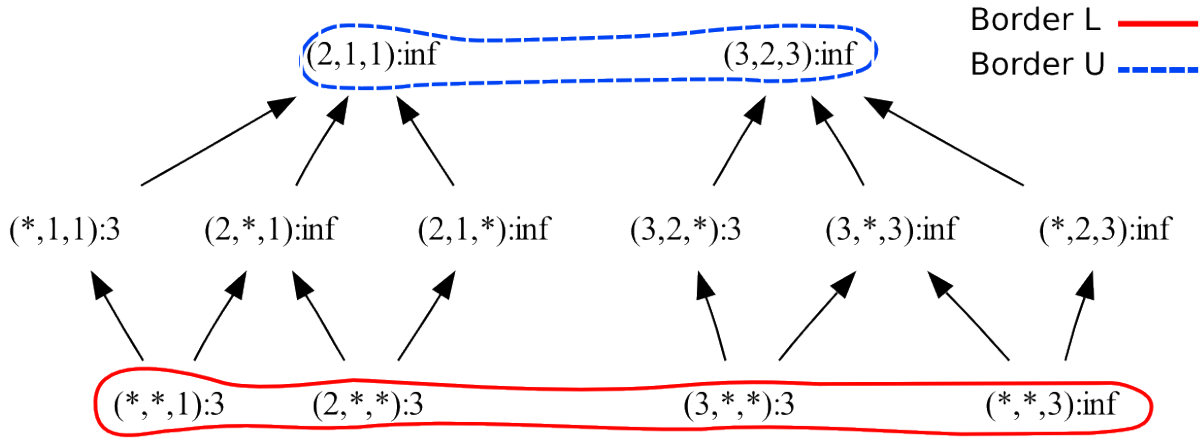
Définition 7.3 (Fonction Mesure compatible avec la Fermeture Cubique) **-** Une fonction mesure, f_{val}, relative à une fonction agrégative f, de $CL(r) \to \mathbb{R}$ est compatible avec l'opérateur de fermeture $\mathbb{C}$ sur T si et seulement si $\forall t,\ u \in CL(r)$, elle satisfait les trois propriétés suivantes :

1. $t \preceq_g u \Rightarrow f_{val}(t,T) \geq f_{val}(u,T)$ ou $f_{val}(t,T) \leq f_{val}(u,T)$,
2. $\mathbb{C}(t,T) = \mathbb{C}(u,T) \Rightarrow f_{val}(t,T) = f_{val}(u,T)$,

TABLE 7.3 – Relation example VENTE$_{08}$

Produit	Ville	Saison	Quantité
2	Marseille	Printemps	200
2	Paris	Été	100
1	Marseille	Printemps	100
3	Paris	Été	100
3	Paris	Automne	300

FIGURE 7.1 – Illustration du Cube Émergent avec ses bordures $[L; U]$



3. $t \preceq_g u$ et $f_{val}(t,T) = f_{val}(u,T) \Rightarrow \mathbb{C}(t,T) = \mathbb{C}(u,T)$.

Cette fonction est une adaptation, spécifique au cadre de travail du Treillis Cube, de la fonction poids introduite dans (Stumme *et al.*, 2002) pour n'importe quel système de fermeture de l'ensemble des parties. Par exemple, les fonctions mesures COUNT et SUM sont compatibles avec l'opérateur de Fermeture Cubique.

Donc, dans le même esprit que (Stumme *et al.*, 2002), nous pouvons donner une autre définition de l'opérateur de Fermeture Cubique en utilisant les fonctions mesures précédentes. L'opérateur de Fermeture Cubique selon T peut être défini comme suit :

$$\mathbb{C}(t,T) = t \bullet \{t' \in \mathcal{A}t(r)\} \text{ tels que } f_{val}(t,T) = f_{val}(t \bullet t',T).$$

Définition 7.4 (Tuple Fermé Émergent) **-** Soit $t \in CL(r)$ un tuple, t est un tuple fermé émergent si et seulement si :

1. t est un tuple émergent ;
2. $\mathbb{C}(t, r_1 \cup r_2) = t$.

Exemple 7.2 - Soit $MinThreshold_1 = 200$ le seuil pour la première contrainte d'émergence (C_1) et $MinThreshold_2 = 200$ le seuil pour la seconde (C_2). Le Cube Émergent de VENTE$_{07}$ vers VENTE$_{08}$ est donné dans la table 7.4. La figure 7.1 donne la représentation de ce cube sous

TABLE 7.4 – Cube Émergent de VENTE$_{07}$ vers VENTE$_{08}$

Tuples Émergents	ER
(ALL, ALL , Printemps)	3
(ALL, Marseille , Printemps)	3
(2 , ALL , ALL)	3
(3 , ALL , ALL)	3
(3 , Paris , ALL)	3
(ALL, ALL , Automne)	∞
(ALL, Paris , Automne)	∞
(2 , ALL , Printemps)	∞
(2 , Marseille , ALL)	∞
(2 , Marseille , Printemps)	∞
(3 , ALL , Automne)	∞
(3 , Paris , Automne)	∞

TABLE 7.5 – Bordure $[L;U]$

U	(2 , Marseille , Printemps) (3 , Paris , Automne)
L	(ALL, ALL , Printemps) (2 , ALL , ALL) (3 , ALL , ALL) (ALL, ALL , Automne)

forme de treillis. Sur ce treillis on voit aussi apparaître les bordures classique $[L;U]$; '*' est utilisé à la place de ALL ; les valeurs des différentes dimensions sont codées comme suit :

Ville		Saison	
Marseille	= 1	Printemps	= 1
Paris	= 2	Été	= 2
		Automne	= 3

Le tuple (2, Marseille, Printemps) est un tuple fermé émergent car :

1. (2, Marseille, Printemps) est un tuple émergent (*cf.* table 7.4).
2. $\mathbb{C}$((2, Marseille, Printemps), VENTE$_{07}\cup$ VENTE$_{08}$) = (2, Marseille, Printemps).

L'ensemble des tuples fermés émergents n'est pas une représentation sans perte du Cube Émergent car pour certains tuples il est impossible de décider s'ils sont émergents ou pas. Ce sont tous les tuples plus généraux que les tuples fermés émergents les plus généraux.

Exemple 7.3 - Considérons l'ensemble de tous les tuples émergents (T) dans la table 7.6. Les tuples (ALL, Paris, Automne) et (ALL, Paris, ALL) partagent la même fermeture sur T : (3, Paris, Automne) qui est émergent. Le premier tuple est aussi émergent alors que le second ne l'est pas.

FIGURE 7.2 – Illustration du L-Cube Fermé Émergent

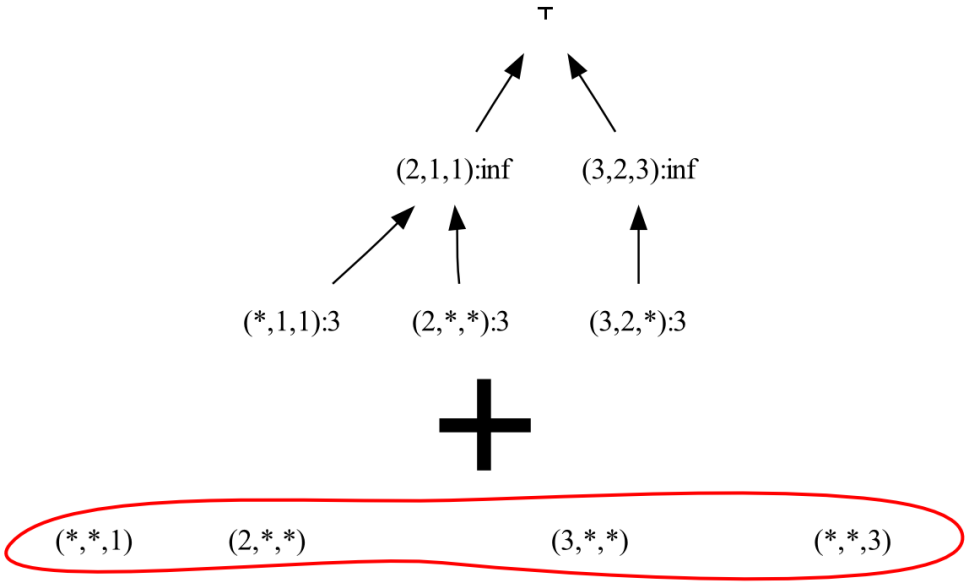


TABLE 7.6 – Ensemble des tuples fermés émergents

Tuple fermé émergent	ER
(ALL, Marseille, Printemps)	3
(2, ALL, ALL)	3
(3, Paris, ALL)	3
(2, Marseille, Printemps)	∞
(3, Paris, Automne)	∞

Afin d'obtenir une représentation sans perte, nous combinons d'une part l'ensemble des tuples fermés émergents à partir desquels les valeurs de la mesure peuvent être retrouvées et d'autre part les bordures qui délimitent l'espace des solutions. Cependant, la bordure U est déjà incluse dans l'ensemble des tuples fermés car les éléments de U sont les tuples émergents les plus détaillés (spécifiques). Ils sont donc obligatoirement des tuples fermés.

Définition 7.5 (L-Cube Fermé Émergent) **-**

$$\text{L-ECC}(r_1, r_2) = \{t \in CL(r_1 \cup r_2) \text{ tel que } t \text{ est un tuple fermé émergent}\} \cup L$$

Exemple 7.4 - Le L-Cube Fermé Émergent est représenté par la table 7.6 donnant l'ensemble des tuples fermés émergents et la table 7.5 qui propose la bordure L. Il est illustré par la figure 7.2

Afin de prouver que le L-Cube Fermé Émergent est une représentation sans perte d'information pour le Cube Émergent, nous introduisons deux propositions. La première montre que, pour tout tuple émergent, nous pouvons calculer sa fermeture cubique à partir soit de $r_1 \cup r_2$, soit du L-Cube Fermé Émergent et bien sûr obtenir le même résultat. La seconde montre que deux tuples ayant la même fermeture cubique ont le même taux d'émergence.

Proposition 7.1 - Pour tout tuple fermé émergent t, $\mathbb{C}(t,\text{L-Ecc}(r_1,r_2)) = \mathbb{C}(t, r_1 \cup r_2)$

Démonstration.

Soit t un tuple fermé émergent

$$\Rightarrow \mathbb{C}(t,r_1 \cup r_2) = (\emptyset,\ldots,\emptyset) + \sum t' = t \text{ tel que } t' \in r_1 \cup r_2 \text{ et } t \preceq_g t'$$
$$\Rightarrow \mathbb{C}(t,\text{L-Ecc}(r_1,r_2)) = (\emptyset,\ldots,\emptyset) + \sum t' \text{ tel que } t' = (\emptyset,\ldots,\emptyset) + \sum v,\ t' \preceq_g v \text{ et } t \preceq_g t'$$
$$\Rightarrow \mathbb{C}(t,\text{L-Ecc}(r_1,r_2)) = (\emptyset,\ldots,\emptyset) + \sum\sum v \text{ tel que } v \in r_1 \cup r_2 \text{ et } t \preceq_g v$$

En raison des propriétés de $\sum$, nous avons :

$$\mathbb{C}(t,\text{L-Ecc}(r_1,r_2)) = (\emptyset,\ldots,\emptyset) + \sum v = \mathbb{C}(t, r_1 \cup r_2) \text{ avec } v \in r_1 \cup r_2 \text{ et } t \preceq_g v$$

□

Proposition 7.2 - Soit t et u deux tuples de $\text{Ec}(r_1, r_2)$, si t et u ont la même fermeture cubique sur $r_1 \cup r_2$, alors leur taux d'émergence est le même. $\forall t, u \in \text{Ec}(r_1, r_2)$ tels que $\mathbb{C}(t, r_1 \cup r_2) = \mathbb{C}(u, r_1 \cup r_2)$, nous avons $ER(t) = ER(u)$.

Démonstration. Par hypothèse, la fonction aggrégative f est compatible avec la fermeture cubique. Donc pour tout couple de tuples t et u tels que $\mathbb{C}(t, r_1 \cup r_2) = \mathbb{C}(u, r_1 \cup r_2)$, nous avons :

$$f_{val}(t, r_1 \cup r_2) = f_{val}(u, r_1 \cup r_2) \text{ et } f_{val}(t, r_1 \cup r_2) = f_{val}(u, r_1 \cup r_2)$$

Comme les relations r_1 et r_2 sont distinctes, t et u généralisent le même ensemble de tuples dans r_1 et dans r_2, on a :

$$\nexists t', u' \in r_1 \text{ tels que } t \preceq_g t',\ u \preceq_g u'$$
$$\Rightarrow ER(t) = ER(u) = \infty$$

Sinon, nous avons :

$$f_{val}(t, r_1) = f_{val}(u, r_1) \text{ et } f_{val}(t, r_2) = f_{val}(u, r_2)$$
$$\Rightarrow \frac{f_{val}(t, r_2)}{f_{val}(t, r_1)} = \frac{f_{val}(u, r_2)}{f_{val}(u, r_1)}$$
$$\Leftrightarrow ER(t) = ER(u)$$

□

Afin de faire du L-Cube Fermé Émergent une représentation sans perte, nous devons calculer la fermeture cubique sur $r_1 \cup r_2$ car deux tuples peuvent avoir les mêmes fermetures cubiques sur r_1 et sur r_2, mais des fermetures cubiques différentes sur $r_1 \cup r_2$. La proposition suivante assure qu'avec le L-Cube Fermé Émergent nous pouvons dériver la mesure de tous les tuples émergents.

Proposition 7.3 - $\forall\ t \in CL(r_1 \cup r_2)$, t est un tuple émergent $\Rightarrow \mathbb{C}(t, \text{L-Ecc}(r_1, r_2))$ est un tuple fermé émergent.

TABLE 7.7 – Bordure $U^\sharp$

$U^\sharp$	(ALL , Marseille , ALL)
	(ALL , Paris , ALL)

Démonstration.

Si t est émergent :

$$\begin{aligned} &\text{Soit } t' = \mathbb{C}(t, r_1 \cup r_2), \text{ d'après la proposition 7.2} \\ \Rightarrow &ER(t) = ER(t') \end{aligned}$$

De plus comme t est émergent, t' l'est aussi.

$$\begin{aligned} &\Rightarrow t' \text{ est un tuple fermé émergent, d'après la proposition 7.1} \\ &\Leftrightarrow \mathbb{C}(t', r_1 \cup r_2) = \mathbb{C}(t', \text{L-ECC}(r_1, r_2)) = \mathbb{C}(t, \text{L-ECC}(r_1, r_2)) \\ &\Leftrightarrow \mathbb{C}(t, \text{L-ECC}(r_1, r_2)) \in \text{L-ECC}(r_1, r_2) \end{aligned}$$

□

Le L-Cube Fermé Émergent contient les bordures $[L; U]$, nous pouvons donc toujours savoir si un tuple est émergent ou pas. De plus quand un tuple est émergent grâce aux propositions précédentes, nous pouvons dériver sa mesure. Le L-Cube Fermé Émergent est donc une représentation bien sans perte du Cube Émergent.

Exemple 7.5 - Dérivons le taux d'émergence du tuple (ALL, Paris, Automne). Nous savons que ce tuple est émergent parce qu'il appartient à l'intervalle $[L; U]$ (*cf.* Figure 7.1). En calculant sa fermeture cubique sur L-ECC(VENTE$_{07}$, VENTE$_{08}$), nous obtenons le tuple (3, Paris, Automne). Puisque le taux d'émergence du tuple précédent est ∞, nous sommes sûrs que le taux d'émergence de (ALL, Paris, Automne) est ∞ et donc nous retrouvons le résultat présenté dans la table 7.4.

7.2.2 $U^\sharp$-Cube Fermé Émergent

Dans ce paragraphe, nous introduisons une nouvelle structure : le $U^\sharp$-Cube Fermé Émergent. Il inclut à la fois (i) l'ensemble des tuples émergents fermés et (ii) la bordure $U^\sharp$.

Définition 7.6 ($U^\sharp$-Cube Fermé Émergent) **-** Le $U^\sharp$-Cube Fermé Émergent est défini comme suit :

$$\text{U}^\sharp\text{-ECC}(r_1, r_2) = \{t \in CL(r_2) \text{ tel que } t \text{ est un tuple émergent fermé } \} \cup U^\sharp$$

Exemple 7.6 - Le $U^\sharp$-Cube Fermé Émergent est donné dans la table 7.6 donnant l'ensemble des tuples fermés émergents et la table 7.7 qui propose la bordure $U^\sharp$.

Afin de prouver que le $U^\sharp$-Cube Fermé Émergent est une couverture pour le Cube Émergent, nous introduisons une nouvelle proposition. Elle montre que, pour tout tuple émergent, nous pouvons calculer sa fermeture cubique à partir soit de $r_1 \cup r_2$ soit du $U^\sharp$-Cube Fermé Émergent et bien sûr obtenir le même résultat.

Proposition 7.4 - Pour tous les tuples émergents fermés t, $\mathbb{C}(t,\text{U}^\sharp\text{-Ecc}(r_1,r_2)) = \mathbb{C}(t,r_1 \cup r_2)$

Démonstration.

Soit t un tuple fermé émergent

$$\Rightarrow \mathbb{C}(t,r_1 \cup r_2) = (\emptyset,\ldots,\emptyset) + \sum t' = t \text{ tel que } t' \in r_1 \cup r_2 \text{ et } t \preceq_g t'$$
$$\Rightarrow \mathbb{C}(t,\text{U}^\sharp\text{-Ecc}(r_1,r_2)) = (\emptyset,\ldots,\emptyset) + \sum t' \text{ tel que } t' = (\emptyset,\ldots,\emptyset) + \sum v,\ t' \preceq_g v \text{ et } t \preceq_g t'$$
$$\Rightarrow \mathbb{C}(t,\text{U}^\sharp\text{-Ecc}(r_1,r_2)) = (\emptyset,\ldots,\emptyset) + \sum\sum v \text{ tel que } v \in r_1 \cup r_2 \text{ et } t \preceq_g v$$

En raison des propriétés de $\sum$, nous avons :

$$\mathbb{C}(t,\text{U}^\sharp\text{-Ecc}(r_1,r_2)) = (\emptyset,\ldots,\emptyset) + \sum v = \mathbb{C}(t,r_1 \cup r_2) \text{ avec } v \in r_1 \cup r_2 \text{ et } t \preceq_g v$$

□

Pour faire en sorte que le $U^\sharp$-Cube Fermé Émergent soit une couverture, nous devons calculer la fermeture cubique sur $r_1 \cup r_2$ car deux tuples peuvent avoir les mêmes fermetures cubiques sur r_1 et sur r_2, mais des fermetures différentes sur $r_1 \cup r_2$. Dans de tels cas de figure, il est impossible de calculer la fermeture cubique sur une seule relation. La proposition suivante assure que le $U^\sharp$-Cube Fermé Émergent est une couverture pour le Cube Émergent.

Proposition 7.5 - Le $U^\sharp$-Cube Fermé Émergent est une couverture pour le Cube Émergent : $\forall\ t \in CL(r_2)$, t est un tuple émergent si et seulement si :

$$\mathbb{C}(t,\text{U}^\sharp\text{-Ecc}(r_1,r_2)) \in \text{U}^\sharp\text{-Ecc}(r_1,r_2) \backslash U^\sharp$$

Démonstration. Si t est émergent, nous savons que $t' = \mathbb{C}(t,r_1 \cup r_2)$ a un taux d'émergence similaire à celui de t. Puisque t est émergent, t' l'est aussi. Donc $t' \in \text{U}^\sharp\text{-Ecc}(r_1,r_2) \backslash U^\sharp$ selon la proposition 7.1, $\mathbb{C}(t',r_1 \cup r_2) = \mathbb{C}(t',\text{U}^\sharp\text{-Ecc}(r_1,r_2)) = \mathbb{C}(t,\text{U}^\sharp\text{-Ecc}(r_1,r_2))$.
Nous avons donc $\mathbb{C}(t,\text{U}^\sharp\text{-Ecc}(r_1,r_2)) \in \text{U}^\sharp\text{-Ecc}(r_1,r_2)$. □
Si t n'est pas émergent alors $\exists u \in U^\sharp$ tel que $t \preceq_g u$ et donc $\forall t' \in \text{U}^\sharp\text{-Ecc}(r_1,r_2) \backslash U^\sharp\ t \not\preceq_g t'$. Puisque la fermeture d'un tuple est la somme de tous les tuples le spécialisant et t est uniquement spécialisé par des tuples de $U^\sharp$ alors $\mathbb{C}(t,\text{U}^\sharp\text{-Ecc}(r_1,r_2) \backslash U^\sharp) \notin \text{U}^\sharp\text{-Ecc}(r_1,r_2) \backslash U^\sharp$. □

De la même manière que L-Cube Fermé Émergent contient les bordures $[L;U]$, $U^\sharp$-Cube Fermé Émergent contient les bordures $U^\sharp$ et U. Nous pouvons donc aussi savoir si un tuple est émergent ou pas juste en vérifiant si le tuple est situé entre les bordures. Mais avec le $U^\sharp$-Cube Fermé Émergent, nous pouvons nous passer de ce test, car la proposition précédente garantit que la fermeture d'un tuple émergent est un fermé émergent. Ainsi juste en calculant la fermeture d'un tuple on peut savoir s'il est émergent et dériver la mesure. Le $U^\sharp$-Cube Fermé Émergent est donc une bien meilleure représentation sans perte d'information que le L-Cube Fermé Émergent.

Exemple 7.7 - Dérivons le taux d'émergence du tuple (ALL, Paris, Automne). En calculant sa fermeture cubique sur $\text{U}^\sharp\text{-Ecc}(\text{Vente}_{07}, \text{Vente}_{08})$, nous obtenons le tuple (3, Paris, Automne). Nous savons que ce tuple est émergent car c'est un tuple fermé émergent (*cf.* table 7.6). Puisque le taux d'émergence du tuple précédent est ∞, nous sommes sûrs que le taux d'émergence de (ALL, Paris, Automne) est ∞ et donc nous retrouvons le résultat présenté dans la table 7.4.

7.2.3 Cubes Émergents Fermés Réduits

Dans le paragraphe précédent, nous avons montré que la bordure $U^\sharp$ doit être ajouté aux tuples émergents fermés afin d'obtenir une représentation sans perte d'information des Cubes Émergents. La première question qui vient à l'esprit avec ce type de représentation composite est : « l'information ajoutée est-elle minimale ? ». Cette question est naturelle, nous allons montrer que dans le cas du $U^\sharp$-Cube Fermé Émergent, il est légitime de se la poser. Effectivement, la proposition 7.5 nous garantit que l'on peut savoir si un tuple est émergent seulement en calculant sa fermeture. C'est grâce à cette propriété que le $U^\sharp$-Cube Fermé Émergent est une représentation du Cube Émergent. Dans ce paragraphe, nous montrons qu'il existe des tuples que l'on peut éliminer sans pour autant perdre cette propriété. Pour nous de tels tuples sont considérés comme redondants. En utilisant la fermeture cubique, nous simplifions la bordure $U^\sharp$ en éliminant toutes les redondances qu'elle contient. De cette manière nous obtenons une nouvelle couverture : le $U^\sharp$-Cube Fermé Émergent Réduit.

Définition 7.7 (Tuple Fermé Émergent Redondant) **-** $\forall\, t \in\ U^\sharp$, t est un tuple fermé redondant si et seulement si $\mathbb{C}(t, \text{U}^\sharp\text{-Ecc}(r_1, r_2)\backslash\{t\}) = t$.

Soulignons que la définition précédente est proposée dans le même esprit que l'élimination des attributs redondants lors du calcul de couvertures minimales pour les dépendances fonctionnelles.

Définition 7.8 (Bordure Réduite $U^\sharp$) **-** La bordure $U^\sharp$ *Réduite*, notée $U^{\sharp\sharp}$, est composée des tuples de $U^\sharp$ qui ne sont pas redondants.

$$U^{\sharp\sharp} = \{t \in U^\sharp \text{ tel que } t \text{ n'est pas un tuple fermé redondant}\}$$

Rappelons que, comme les tuples de $U^\sharp$, tous les tuples de $U^{\sharp\sharp}$ satisfont la contrainte anti-monotone C_2.

Définition 7.9 ($U^\sharp$-Cube Fermé Émergent Réduit) **-** Le $U^\sharp$-Cube Fermé Émergent Réduit est défini comme suit :

$$\text{R-Ecc}(r_1, r_2) = \{t \in CL(r_2) \text{ tel que } t \text{ est un tuple émergent fermé}\} \cup U^{\sharp\sharp}$$

La proposition suivante montre que l'élimination des tuples fermés redondants de la bordure $U^\sharp$ n'altère pas le calcul de la fermeture pour les tuples émergents fermés. Donc le $U^\sharp$-Cube Fermé Émergent Réduit est une couverture pour le $U^\sharp$-Cube Fermé Émergent et par transitivité une couverture pour le Cube Émergent (*cf.* proposition 7.7).

Proposition 7.6 -

$$\forall\, t \in \text{R-Ecc}(r_1, r_2), \mathbb{C}(t, \text{R-Ecc}(r_1, r_2)) = \mathbb{C}(t, \text{U}^\sharp\text{-Ecc}(r_1, r_2))$$

Démonstration.
Le $U^\sharp$-Cube Fermé Émergent Réduit est par définition le $U^\sharp$-Cube Fermé Émergent duquel les tuples fermés redondants dans $U^\sharp$ ont été éliminés. La fermeture du tuple t est la somme de tous les tuples specialisant t. Si un tuple t est émergent et fermé alors il ne spécialise aucun tuple de $U^\sharp$. $\forall t \in \text{U}^\sharp\text{-Ecc}(r_1, r_2)\backslash U^\sharp$, nous savons qu'aucun tuple de $U^\sharp$ n'est impliqué dans le calcul de sa fermeture. Nous avons donc :

$$\mathbb{C}(t, \text{U}^\sharp\text{-Ecc}(r_1, r_2)) = \mathbb{C}(t, \text{R-Ecc}(r_1, r_2)) \quad \square$$

De plus les seuls éléments éliminés de $U^\sharp$ ne sont pas nécessaires pour le calcul de la fermeture. Donc par construction, $\forall t$ of $U^\sharp$, nous avons $\mathbb{C}(t, \text{U}^\sharp\text{-Ecc}(r_1, r_2)) = \mathbb{C}(t, \text{R-Ecc}(r_1, r_2))$. $\square$

Proposition 7.7 - Le $U^\sharp$-Cube Fermé Émergent Réduit est une couverture pour le Cube Émergent : $\forall\ t \in CL(r_2)$, t est un tuple émergent si et seulement si $\mathbb{C}(t, \text{R-Ecc}(r_1, r_2))$ est un tuple émergent fermé.

Démonstration.
Si t est émergent, le calcul de sa fermeture cubique ne requiert aucun élément de $U^\sharp$ (et bien sûr aucun tuple de $U^{\sharp\sharp}$) donc :

$$\mathbb{C}(t, \text{R-Ecc}(r_1, r_2)) = \mathbb{C}(t, \text{U}^\sharp\text{-Ecc}(r_1, r_2)) = \mathbb{C}(t, \text{U}^\sharp\text{-Ecc}(r_1, r_2) \backslash U^\sharp$$

En utilisant la proposition 7.5 nous avons $\mathbb{C}(t, \text{R-Ecc}(r_1, r_2))$ est un tuple émergent fermé. □
Si t n'est pas émergent, deux cas sont possibles :

1. Calculer la fermeture de t nécessite seulement des éléments de $U^{\sharp\sharp}$. Dans ce cas,

   $$\mathbb{C}(t, \text{R-Ecc}(r_1, r_2)) = \mathbb{C}(t, \text{U}^\sharp\text{-Ecc}(r_1, r_2)),$$

   ce qui est similaire à la proposition 7.5.
2. Le calcul de la fermeture de t utilise au moins un tuple u fermé et redondant. $\forall u \in U^\sharp \backslash U^{\sharp\sharp}$ on a :

   $$\mathbb{C}(u, \text{R-Ecc}(r_1, r_2)) = u \text{ et } u \notin \text{U}^\sharp\text{-Ecc}(r_1,r_2)\backslash U^\sharp$$
   $$\mathbb{C}(t, \text{R-Ecc}(r_1, r_2)) = \sum_{u \in U^\sharp \backslash U^{\sharp\sharp}} u + \sum_{v \in U^{\sharp\sharp}} v$$
   $$\text{or } \sum_{u \in U^\sharp \backslash U^{\sharp\sharp}} u \notin \text{R-Ecc}(r_1,r_2)\backslash U^{\sharp\sharp} \text{ et } \sum_{\in U^{\sharp\sharp}} v \notin \text{R-Ecc}(r_1,r_2)$$

   $\Rightarrow \mathbb{C}(t, \text{R-Ecc}(r_1, r_2))$ n'est pas un tuple émergent fermé.

□

7.3 Cubes Quotients Émergents

Un Cube Quotient (Lakshmanan *et al.*, 2002) offre un résumé d'un data cube pour certaines fonctions agrégatives comme Count, Sum, ... De plus le Cube Quotient préserve la sémantique des opérateurs Roll-Up/Drill-Down sur le data cube (Gray *et al.*, 1997). Avant de présenter le Cube Quotient Émergent nous allons revisiter les définitions du Cube Quotient avec les concepts associés aux cubes fermés.

7.3.1 Cubes Quotients et leur sémantique basée sur la fermeture

L'idée sous-tendant la représentation en question est d'éliminer les redondances en rassemblant ensemble les tuples véhiculant une information équivalente. Ceci résulte dans un ensemble de classes d'équivalence partitionnant les tuples du data cube. Un tel partitionnement peut être effectué de plusieurs manières. Mais, afin de préserver les capacités de navigation, il est nécessaire de gérer des classes convexes.

Définition 7.10 (Classes d'équivalence convexes) **-** Soit $\mathcal{C} \subseteq CL(r)$ une classe d'équivalence. Nous disons que $\mathcal{C}$ est convexe si et seulement si :

$$\forall t \in CL(r) \text{ si } \exists t', t'' \in \mathcal{C} \text{ tels que } t' \preceq_g t \preceq_g t'' \text{ alors } t \in \mathcal{C}.$$

Une partition $\mathcal{P}$ de $CL(r)$ qui comprend uniquement des classes d'équivalence convexes est appelée partition convexe.

La propriété de convexité rend possible la représentation de chaque classe d'équivalence à travers ses tuples minimaux et ses tuples maximaux. Les tuples intermédiaires ne sont, dès lors, plus utiles et la représentation sous-jacente est réduite. Pour être sûr que la partition est convexe, la relation d'équivalence suivante est utilisée.

Définition 7.11 (Relation d'équivalence quotient) **-** Soit f_{val} une fonction mesure. Nous définissons la relation d'équivalence $\equiv_f$ comme la fermeture réflexive transitive de la relation suivante τ : soit t, t' deux tuples, $t\ \tau\ t'$ est vrai si et seulement si (i) $f_{val}(t,r) = f_{val}(t',r)$ et (ii) t est soit un parent soit un enfant de t'.
La relation d'équivalence $\equiv_f$ est dite *relation d'équivalence quotient* si et seulement si elle satisfait la proprieté de congruence faible : $\forall t,t',u,u' \in CL(r)$, si $t \equiv_f t', u \equiv_f u', t \preceq_g u$ et $u' \preceq_g t'$, alors $t \equiv_f u$.

Nous notons $[t]_{\equiv_f}$ la classe d'équivalence de t ($[t]_{\equiv_f} = \{t' \in CL(r)$ tel que $t \equiv_f t'\}$). Alors le Cube Quotient est défini comme l'ensemble des classes d'équivalence, chacune d'entre elles étant pourvue de la valeur de la mesure.

Définition 7.12 (Cube Quotient) **-** Soit $CL(r)$ le treillis cube de la relation r et $\equiv_f$ une relation d'équivalence quotient. Le Cube Quotient de r, noté $QuotientCube(r, \equiv_f)$, est défini comme suit :

$$QuotientCube(r, \equiv_f) = \{([t]_{\equiv_f}, f_{val}(t,r)) \text{ tel que } t \in CL(r)\}.$$

Le Cube Quotient de r est une partition convexe de $CL(r)$.

Pour deux classes d'équivalence $\mathcal{C},\mathcal{C}' \in QuotientCube(r, \equiv_f)$, $\mathcal{C} \preceq_{QC} \mathcal{C}'$ quand $\exists t \in \mathcal{C}$ et $\exists t' \in \mathcal{C}'$ tels que $t \preceq_g t'$.

La construction d'un Cube Quotient dépend de la relation d'équivalence quotient choisie. Par conséquent pour deux relations d'équivalence quotients, leur Cubes Quotients associés peuvent différer. De plus, la relation d'équivalence quotient la plus utile est la relation d'équivalence de couverture. La couverture de tout tuple t est l'ensemble de tous les tuples agrégés ensemble pour obtenir t.

Définition 7.13 (Couverture) **-** Soit $t \in CL(r)$, La couverture de t est un ensemble de tuples de r qui sont généralisés par t (*i.e.* $cov(t,r) = \{t' \in r$ tel que $t \preceq_g t'\}$).

Deux tuples $t,t' \in CL(r)$ sont dits équivalents selon leur couverture sur r, $t \equiv_{cov} t'$, s'ils ont la même couverture, *i.e.* $cov(t,r) = cov(t',r)$.

La relation d'équivalence de couverture est une instance de $\equiv_f$ dans la définition 3.19, nous pouvons définir le *cube quotient de couverture*.

Nous montrons maintenant que le Cube Quotient de Couverture est fortement relié à la fermeture cubique. Deux tuples $t,t' \in CL(r)$ sont équivalents selon la fermeture cubique, $t \equiv_{\mathbb{C}} t'$, si et seulement si $\mathbb{C}(t,r) = \mathbb{C}(t',r)$.

Proposition 7.8 - Soit $t,t' \in CL(r)$, t est équivalent selon la couverture à t' sur r si et seulement si t est équivalent à t' selon la fermeture cubique.

Démonstration. Soit t,t' deux tuples de $CL(r)$.

⇒ Supposons que $t \equiv_{cov} t'$, dans ce cas le calcul de la fermeture cubique de t est le suivant :

$$t : \mathbb{C}(t,r) = \sum\{v \in r \text{ tel que } t \preceq_g v\} = \sum cov(t,r) = \sum cov(t',r) = \mathbb{C}(t',r)$$

Cela implique que si t et t' sont équivalents selon la couverture alors ils sont equivalents par fermeture.

⇐ Supposons que $t \equiv_{\mathbb{C}} t'$. Donc t et t' généralisent le même ensemble de tuples de r, ils sont donc équivalents selon la couverture.

□

La proposition ci-dessus établit le lien entre le Cube Quotient et les concepts associés à la fermeture cubique. De plus, elle montre qu'il est possible de définir un Cube Quotient de Couverture en utilisant toute fonction agrégative compatible avec la fermeture cubique.

7.3.2 Cubes Quotient Émergent

Dans le paragraphe précédent, nous avons rappelé la définition du Cube Quotient qui a été originellement proposé comme une représentation concise des data cubes (Lakshmanan *et al.*, 2002) préservant les opérateurs de navigation (Roll-Up / Drill-Down). Une adaptation directe de cette approche a été définie pour représenter des data cubes icebergs (Zhang *et al.*, 2007) avec des fonctions agrégatives monotones. Motivés par les propriétés pertinentes du Cube Quotient, nous voulons tirer profit d'une telle représentation pour condenser les Cubes Émergents. Comme le taux d'émergence n'est pas une fonction monotone, l'adaptation nécessaire est difficile à exprimer en utilisant les concepts originaux. C'est pourquoi nous établissons le lien entre le Cube Quotient et les concepts associés à la fermeture cubique. Nous avons souligné que ce lien nécessite une fonction mesure compatible avec l'opérateur de fermeture. Il est possible que deux tuples liés par l'ordre de généralisation aient tous deux un taux d'émergence infini. Néanmoins ces deux tuples peuvent avoir une fermeture différente. Par conséquent, le taux d'émergence n'est pas compatible avec la fermeture cubique car la troisième propriété de la définition 7.3 n'est pas vérifiée. Ainsi pour définir le Cube Quotient Émergent, il n'est pas possible d'utiliser le taux d'émergence comme fonction mesure. À la place, nous utilisons le couple $(f_{val}(t,r_1), f_{val}(t,r_2))$ car il est composé de deux fonctions qui elles mêmes sont compatibles avec la fermeture cubique.

Définition 7.14 (Cube Quotient Émergent) **-** Nous appelons Cube Quotient Émergent l'ensemble des classes d'équivalence de $CL(r_1 \cup r_2)$ émergeant de r_1 vers r_2 noté $\text{EQC}(r_1,r_2)$: $\text{EQC}(r_1,r_2) = \{([t]_{\equiv_f}, f_{val}(t,r_1), f_{val}(t,r_2))$ tel que $[t]_{\equiv_f} \in QuotientCube(r_1 \cup r_2, \equiv_f)$ et t est émergent de r_1 vers $r_2\}$.

Chaque classe d'équivalence du Cube Quotient Émergent est représenté par son élément maximal (selon l'ordre de généralisation) qui est un tuple fermé émergent et ses éléments minimaux qui sont les tuples clés cubiques associés aux tuples fermés cités. La proposition suivante montre que les bordures U et L sont incluses dans le Cube Quotient Émergent. Plus précisément, U contient l'élément maximal des classes maximales (qui sont des tuples fermés) tandis que L englobe les éléments minimaux des classes minimales (qui sont des tuples clés). Ainsi, naviguer dans le Cube Quotient Émergent est possible.

Proposition 7.9 - Les bordures classiques sont incluses dans le Cube Quotient Émergent. La caractérisation de ces bordures basée sur le Cube Quotient Émergent est la suivante :

1.

$$U = \max_{\preceq_g}(\{\max_{\preceq_{QC}}(\{[t]_{\equiv_f}\})\}) \text{ tel que } ([t]_{\equiv_f}, f_{val}(t,r_1), f_{val}(t,r_2)) \in \text{EQC}(r_1,r_2)$$

2. $$L = \min_{\preceq_g}(\{\min_{\preceq_{QC}}(\{[t]_{\equiv_f}\})\}) \text{ tel que } ([t]_{\equiv_f}, f_{val}(t,r_1), f_{val}(t,r_2)) \in \text{EQC}(r_1,r_2)$$

Démonstration.

1. $$\begin{aligned}&\max_{\preceq_g}(\{\max_{\preceq_{QC}}(\{[t]_{\equiv_f}\})\}) \text{ tel que } ([t]_{\equiv_f}, f_{val}(t,r_1), f_{val}(t,r_2)) \in \text{EQC}(r_1,r_2) = \\ =&\max_{\preceq_g}(\{[t]_{\equiv_f}\}) \text{ tel que } ([t]_{\equiv_f}, f_{val}(t,r_1), f_{val}(t,r_2)) \in \text{EQC}(r_1,r_2) \\ =&\max_{\preceq_g}(\{t \in \text{EC}(r_1,r_2)\}) \\ =&U\end{aligned}$$

2. $$\begin{aligned}&\min_{\preceq_g}(\{\max_{\preceq_{QC}}(\{[t]_{\equiv_f}\})\}) \text{ tel que } ([t]_{\equiv_f}, f_{val}(t,r_1), f_{val}(t,r_2)) \in \text{EQC}(r_1,r_2) = \\ =&\min_{\preceq_g}(\{[t]_{\equiv_f}\}) \text{ tel que } ([t]_{\equiv_f}, f_{val}(t,r_1), f_{val}(t,r_2)) \in \text{EQC}(r_1,r_2) \\ =&\min_{\preceq_g}(\{t \in \text{EC}(r_1,r_2)\}) \\ =&L\end{aligned}$$

□

La proposition suivante prouve que la représentation ci-dessus est correcte.

Proposition 7.10 - Le Cube Quotient Émergent est un résumé du Cube Émergent : $\forall\ t \in CL(r_1 \cup r_2)$, t est émergent si et seulement si $([t]_{\equiv_f}, f_{val}(t,r_1), f_{val}(t,r_2))$ appartient au Cube Quotient Émergent.

Démonstration. Par construction t est émergent si et seulement si$([t]_{\equiv_f}, f_{val}(t,r_1), f_{val}(t,r_2))$ appartient au cube quotient. D'après la proposition 7.9, les bordures L et U sont incluses dans le Cube Quotient Émergent.
De plus t est émergent si et seulement si$\exists l \in L$ et $\exists u \in U$ tels que $l \preceq_g t \preceq_g u$ (*cf.* proposition 5.2). Comme U (resp. L) est un sous-ensemble des maximaux (resp. minimaux) des classes d'équivalence du Cube Quotient Émergent, on obtient donc bien le résultat attendu :

$$min_{\preceq_g}([l]_{\equiv_f}) \preceq_g t \preceq_g max_{\preceq_g}([u]_{\equiv_f})$$

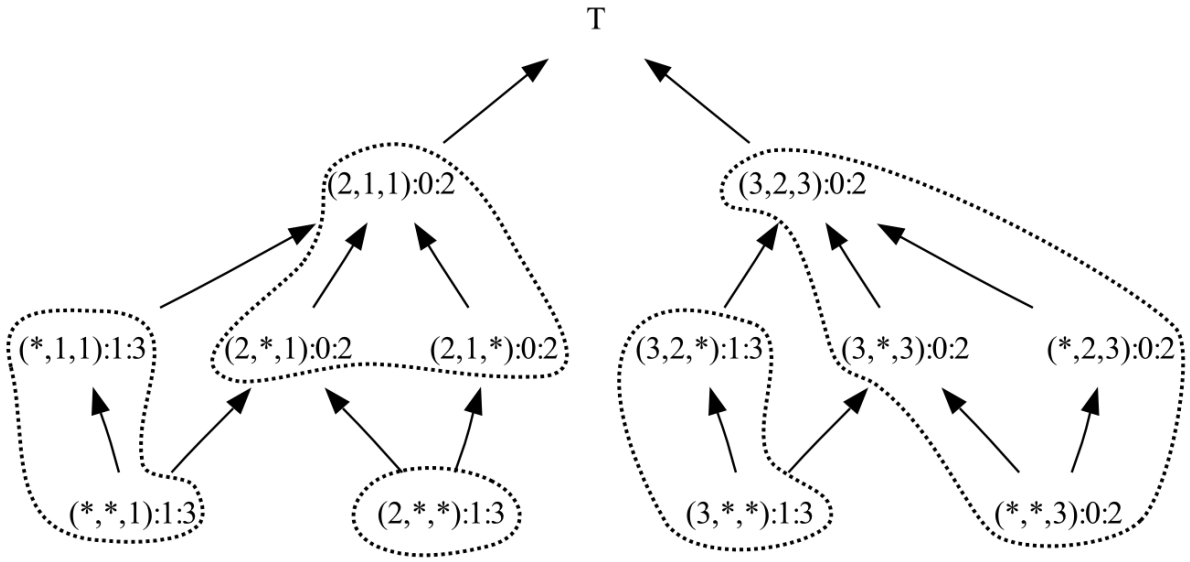
□

Exemple 7.8 - Avec les deux relations exemples VENTES_{07} et VENTES_{08}, la table 7.8 donne le Cube Quotient Émergent. Les deux premières colonnes correspondent respectivement au tuple maximal des classes d'équivalence et à leurs tuples minimaux. La dernière colonne représente le couple $(SUM_{val}(t,r_2), SUM_{val}(t,r_1))$ qui est utilisé pour calculer le taux d'émergence pour la classe d'équivalence concernée. La figure 7.3 illustre le Cube Quotient Émergent. Dans chaque nœud, le taux d'émergence est remplacé par le couple des mesures agrégées obtenues à partir de VENTES_{07} et VENTES_{08}. Les différentes classes d'équivalence sont encerclées. La classe la plus à droite est représentée avec ses tuples maximal et minimaux, les tuples intermédiaires qui sont redondants doivent être supprimés, ce qui est fait pour les autres classes.

TABLE 7.8 – Cube Quotient Émergent

Tuple Maximal	Tuples Minimaux	
(ALL, Marseille, Printemps)	(ALL, ALL, Printemps)	(3,1)
(2, ALL, ALL)	(2, ALL, ALL)	(3,1)
(3, Paris, ALL)	(3, ALL, ALL)	(3,1)
(2, Marseille, Printemps)	(2, ALL, Printemps) (2, Marseille, Printemps)	(2,0)
(3, Paris, Automne)	(ALL, ALL, Automne)	(2,0)

FIGURE 7.3 – Illustration du Cube Quotient Émergent



7.4 Liens et utilisations des différentes représentations

Nous ajoutons à la représentation réduite classique du Cube Émergent, les bordures L et U, deux nouvelles représentations sans perte d'information : le L-Cube Fermé Émergent et le Cube Quotient Émergent. Chaque structure a des usages particuliers pour la fouille de bases de données OLAP. Disposant simplement des bordures, l'utilisateur peut savoir si un tuple est émergent ou pas. Comme les motifs émergents qui offrent des classifieurs précis dans les bases de données de transactions (Dong et Li, 2005), les bordures citées peuvent être utilisées pour des tâches de classification dans les bases de données OLAP.

Néanmoins, à partir des bordures, il est impossible d'obtenir le taux d'émergence des tuples. Afin d'éviter cet inconvénient, nous proposons le Cube Fermé Émergent pour répondre à toutes les requêtes OLAP qu'il est possible d'exprimer sur le Cube Émergent (sans avoir besoin de le calculer). Enfin, afin d'offrir aux utilisateurs des outils pour naviguer au sein des Cubes Émergent, nous caractérisons le Cube Quotient Émergent. À travers le théorème suivant, nous établissons les liens d'inclusion entre les différentes représentations.

Théorème 7.11 - Soit $[L;U]$, L-ECC and EQC des représentations pour le Cube Émergent (EC) de deux relations r_1 et r_2. Nous avons alors :

$$[L;U] \subseteq \text{L-ECC} \subseteq \text{EQC} \subseteq \text{EC}$$

Démonstration. La proposition 7.9 montre que $[L;U] \subseteq$ EQC $\subseteq$ EC. La proposition 7.3, quant à elle, prouve que nous avons $[L;U] \subseteq$ L-ECC $\subseteq$ EC. Pour démontrer ce théorème, nous allons montrer que les maximaux de chaque classe d'équivalence du Cube Quotient Émergent sont tous des tuples fermés émergents.
Soit t un tel tuple. En appliquant la définition de l'opérateur de fermeture cubique sur t nous avons :

$$\mathbb{C}(t,r) = t \bullet \{t' \in CL(r)\} \text{ tel que } f_{val}(t,r_1) = f_{val}(t \bullet t',r_1) \text{ et } f_{val}(t,r_2) = f_{val}(t \bullet t',r_2)$$

Si $t' \preceq_g t \Rightarrow t = t \bullet t' \Rightarrow \mathbb{C}(t,r) = t$

Sinon $t' \not\preceq_g t \Rightarrow t \preceq_g (t \bullet t')$

Comme f est une fonction monotone nous avons deux cas possibles :

$$f_{val}(t,r_1) \leq f_{val}(t \bullet t',r_1) \text{ ou } f_{val}(t,r_2) \leq f_{val}(t \bullet t',r_2)$$

Si nous avons $f_{val}(t,r_1) = f_{val}(t \bullet t',r_1)$, d'après la définition de classe d'équivalence on a :

$$f_{val}(t,r_2) < f_{val}(t \bullet t',r_2) \Rightarrow \mathbb{C}(t,r) = t$$

Sinon $f_{val}(t,r_1) < f_{val}(t \bullet t',r_1) \Rightarrow \mathbb{C}(t,r) = t$

Nous pouvons donc déduire que les maximaux des classes d'équivalence du Cube Quotient Émergent sont tous des tuples fermés émergents. Nous avons donc bien L-ECC $\subseteq$ EQC. □

Toutes les représentations proposées sont réduites comparées au Cube Émergent lui-même excepté dans les deux cas extrèmes : *(i)* quand il n'existe aucun tuple émergent et *(ii)* quand tous les tuples émergents sont fermés et donc que le Cube Émergent ne contient aucune redondance.

À part la représentation basée sur les bordures, les représentations proposées permettent de répondre aux mêmes requêtes que le Cube Émergent lui-même. Par exemple, « Pour quels produits la quantité vendue a augmenté de manière significative entre 2007 et 2008 ? » ; « Pour quelle saison de 2008 de nouveaux produits ont ils suscité l'engouement des acheteurs ? ». Ce type de connaissances est difficile à obtenir via des requêtes OLAP classiques (*cf.* paragraphe 5.4.1 P. 132).

Rappelons que la nouvelle bordure $U^{\sharp}$ s'est avérée expérimentalement plus réduite que L. La bordure $U^{\sharp\sharp}$ est par définition une réduction de $U^{\sharp}$. Ainsi même si un lien d'inclusion ne peut être établi, le $U^{\sharp}$-Cube Fermé Émergent est le $U^{\sharp}$-Cube Fermé Émergent Réduit sont en pratique des représentations plus réduites que le L-Cube Fermé Émergent.

7.5 L'algorithme de calcul de représentations réduites du Cube Émergent : ℂ-Idea

Doter le Cube Émergent de représentations aussi réduites que possible est cruciale. Mais avoir une représentation réduite sans moyen de la calculer efficacement n'a que peu d'intérêt. Dans ce paragraphe, notre objectif est d'ajouter, à la plateforme logicielle générique IDEA, la possibilité de calculer les différentes représentations sans perte d'informations. D'après les propositions ci-dessus, toutes nos représentations de la famille du Cube Fermé Émergent utilisent au moins l'une des bordures et les tuples fermés émergents, c'est pour cela que nous nous basons sur F-IDEA pour contruire une nouvelle famille d'algorithmes dédiés au calcul de nos représentations : ℂ-IDEA (pour *Close Integrable DatabasE Algorithm*). Cet algorithme utilise certaines propriétés de F-IDEA pour calculer efficacement l'ensemble des tuples fermés émergents. Préalablement à la présentation de l'algorithme ℂ-IDEA, nous faisons quelques propositions qui justifient la robustesse de notre démarche.

Proposition 7.12 (Tuples Émergents non-fermés) **-** Soit P un ensemble de tuples de r en accord sur l'ensemble de dimensions $D \subseteq \mathcal{D}$ généré lors d'un partitionnement de l'algorithme F-IDEA et P' l'un de ses IDEA-Successeurs maximaux par rapport à D (P' est un ensemble en accord selon $D \cup d_k$). Si $|P| = |P'|$ alors le tuple t associé à P est un tuple non-fermé.

Démonstration. Soit P un ensemble de tuples émergents en accord sur l'ensemble de dimensions $D \subseteq \mathcal{D}$ et P' l'un de ses IDEA-Successeurs maximaux par rapport à D. t et t' sont les tuples associés respectivement à P et P'. P' étant l'un des IDEA-Successeurs maximaux de P, on sait que $t \prec_g t'$. Par définition de la relation de couverture $cov(t) = P$ et $cov(t') = P'$. $t \preceq_g t'$ implique que $cov(t') \subseteq cov(t)$. D'où, si $|P| = |P'|$ alors $cov(t) = cov(t') \Leftrightarrow t \equiv_{cov} t' \Leftrightarrow t \equiv_{\mathbb{C}} t'$. D'après la propriété d'extensivité de l'opérateur de fermeture :

$$(t \prec_g t') \wedge (t' \preceq_g \mathbb{C}(t',r)) \wedge (\mathbb{C}(t',r) = \mathbb{C}(t,r)) \Rightarrow \mathbb{C}(t,r) \neq t$$

Le tuple t est donc un tuple non-fermé. □

Cette proposition est la première à intégrer au sein de F-IDEA pour éliminer la majorité des tuples non-émergent. Malheureusement, cette proposition ne garantit pas l'élimination de tous les tuples non fermés. Les tuples non éliminés grâce à cette proposition sont appelés des

tuples fermés candidats. Pour ne pas générer de solutions erronées, nous ajoutons un second test permettant de garantir que nous sommes en présence d'un vrai tuple fermé.

Proposition 7.13 (Fermeture d'un IDEA fragment) **-** Soit P l'un des fragments non vides générés lors d'un partitionnement de l'algorithme F-IDEA. La fermeture cubique du tuple multidimensionnel t associé P peut être obtenue comme suit :

$$\mathbb{C}(t, r_1 \cup r_2) = \underset{t' \in P}{+} t'$$

Démonstration. Soit P l'un des fragments non vides générés lors d'un partitionnement de l'algorithme F-IDEA. P est donc l'IDEA-Successeur maximal d'une suite d'IDEA-Successeurs maximaux dont l'origine est $r_1 \cup r_2$. Cette propriété nous donne la garantie que $P = cov(t)$ avec $cov(t) = \{t' \in r \text{ tel que } t \preceq_g t'\}$ (*cf.* définition 7.13). D'après la définition de la fermeture cubique on a :

$$\begin{aligned}\mathbb{C}(t, r_1 \cup r_2) &= \underset{t' \in r \text{ tel que } t \preceq_g t'}{+} t' \\ &= \underset{t' \in cov(t)}{+} t' \\ &= \underset{t' \in P}{+} t'\end{aligned}$$

□

Cette proposition est très intéréssante car le coût de calcul d'une fermeture est largement minimisé. Ce calcul s'effectuant sur un fragment minimal de la relation d'origine, on évite ainsi de devoir balayer toute la relation inutilement. En plus grace à la proposition 7.13, le nombre de calcul est lui aussi réduit.

Proposition 7.14 (Test de fermeture) **-** Soit t un tuple fermé candidat et P le fragment de r correspondant. t est un fermé si et seulement si $+_{t \in P} t = t$.

Démonstration. Directe à partir de la proposition 7.13 et la définition 7.4. □

L'algorithme $\mathbb{C}$-IDEA, est une déclinaison de F-IDEA qui durant la phase $\mathbf{P}_2$ va utiliser les propositions ci-dessus pour calculer efficacement les tuples fermés émergents et ainsi obtenir les représentations réduites du Cube Émergent. Notre approche tire profit du mode de fonctionnement de l'algorithme par partitionnement successif, pour optimiser autant que possible le calcul des Cubes Fermés Émergent. En plus comme tous les autres algorithmes de la famille IDEA, cette approche reste directement intégrable au sein d'un SGBD. L'algorithme 31 donne le pseudo-code de $\mathbb{C}$-IDEA.

7.6 Évaluations Expérimentales

Nous avons effectué des évaluations expérimentales avec une double intention : d'une part confirmer le résultat analytique obtenu sur la taille des différentes représentations et d'autre part quantifier la réduction d'espace apportée par ces représentations quand elles sont comparées au Cube Émergent.
Pour réaliser les évaluations présentées, nous utilisons les mêmes relations de bases de données que celles exploitées dans (Xin *et al.*, 2007). Les expérimentations sont menées sur des données provenant d'un éventail large et varié de domaines. Il est bien connu que les données synthétiques

Algorithme 31 Algorithme ℂ-IDEA

Entrée :
la relation fusionnée courante r
L'ensemble des dimensions $\mathcal{D} = \{d_1, \ldots, d_n\}$ restant à traiter.
La cellule courante *CelluleCour*
La phase courante de l'algorithme $\mathbf{P}_{Cour}$

Sortie :
L'ensemble des tuples fermés émergents de r.
La phase sortie de l'algorithme.

1: $DimCour := d_1$;
2: AGRÉGER($r, CelluleCour, f$) ; //Calcule les valeurs de la fonction mesure et les écrit dans la cellule courante
3: **soit** $\mathbf{P}_{Suiv}$:= DÉTERMINERPHASESUIVANTE($\mathbf{P}_{Cour}$, *CelluleCour*) ; //*cf.* Figure 5.6
4: **si** $\mathbf{P}_{Suiv} = \mathbf{P}_3$ **alors**
5: **retourner** $\mathbf{P}_{Suiv}$;
6: **sinon si** $\mathbf{P}_{Suiv} = \mathbf{P}_2$ **alors**
7: **si** $|r| = 1$ **alors**
8: AJOUTERFERMÉ($r[0]$) ;
9: **retourner** $\mathbf{P}_3$;
10: **fin si**
11: **fin si**
12: **pour tout** $d_j \in \mathcal{D}$ **faire**
13: C := $|r(d_j)|$;
14: PARTITIONNER(r, d_j) ; //r est partitionnée suivant ses valeurs pour l'attribut d_j, en C fragments : $r_1, \ldots, r_C$
15: **pour** $i = 1, \ldots, C$ **faire**
16: **si** $(\mathbf{P}_{Suiv} = \mathbf{P}_2) \wedge (|r_i| \neq |r|)$ **alors**
17: t := SOMME(r) ;
18: **si** $t = CelluleCour$ **alors**
19: AJOUTERFERMÉ(*CelluleCour*) ;
20: **fin si**
21: **fin si**
22: $CelluleCour.d_j := r_i[0].d_j$; //On affecte à la cellule courante la valeur de la dimension d_j pour r_i
23: E-IDEA(r_i, $\{d_{j+1}, \ldots, d_n\}$, *CelluleCour*, $\mathbf{P}_{Suiv}$) ;
24: **fin pour**
25: $CelluleCour.d_j := ALL$; //La dimension d_j est entièrement traitée
26: **fin pour**
27: **retourner** $\mathbf{P}_{Suiv}$;

sont faiblement corrélées alors que dans de nombreuses bases réelles ou statistiques les données sont fortement corrélées (Pasquier *et al.*, 1999). Pour les données synthétiques [7], nous utilisons les notations suivantes pour décrire les relations : $\mathcal{D}$ le nombre de dimensions, $\mathcal{C}$ la cardinalité de chaque dimension, $\mathcal{T}$ le nombre de tuples de la relation, $\mathcal{M}_1$ (respectivement $\mathcal{M}_2$) le seuil correspondant à la contrainte d'émergence C_1 (respectivement C_2), et $\mathcal{S}$ le biais ou zipf des données. Quand $\mathcal{S}$ est égal à 0, les données sont uniformes. Quand $\mathcal{S}$ croît, les données sont de plus en plus biaisées. $\mathcal{S}$ est appliqué sur toutes les relations dans chacune des bases de données. Pour les données réelles, nous utilisons les relations de données météorologiques SEP83L.DAT et SEP85L.DAT utilisées par (Xin *et al.*, 2006), qui ont 1.002.752 tuples avec 8 dimensions sélectionnées. Les attributs dimensions (avec leur cardinalité) sont les suivantes : année mois jour heure (238), latitude (5260), longitude (6187), numéro de station (6515), météo actuelle (100), code de changement (110), altitude solaire (1535) et luminance lunaire relative (155).

7.6.1 Taille des Cubes Fermés Émergents

Dans les figures de 7.4 à 7.5, nous reportons les résultats obtenus en comparant les tailles des Cubes Fermés Émergents et du Cube Émergent pour les données synthétiques. Dans les deux premières figures, quelle que soit la taille de la relation r_2 ou la cardinalité de ses dimensions, la taille du Cube Fermé Émergent est comparable ou légèrement inférieure à la taille du Cube Émergent. La raison de cette très faible réduction de l'espace est la suivante : comme mentionné précédemment, les données synthétiques sont très faiblement corrélées et donc ne contiennent que très peu de redondances. De plus, nous fixons le biais des données à zéro et donc nous n'influons pas sur le taux d'émergence original. Comme attendu, en faisant varier le biais des données, nous observons que, comparé au Cube Émergent, le Cube Fermé Émergent a une taille de plus en plus réduite au fur et à mesure que le biais des données augmente. Le facteur de réduction varie de 7 à 11. Le phénomène est illustré par la figure 7.4.

De plus, nous utilisons des données réelles, connues pour être fortement corrélées, pour comparer le Cube Émergent et les Cubes Fermés Émergents. Le paramètre qui varie est le seuil minimal appliqué à la relation r_2. Nous observons dans la figure 7.5 que plus ce seuil s'accroît plus la taille du Cube Émergent et du Cube Fermé Émergent décroît. Effectivement quand le seuil minimal est élevé, il est logique que le nombre de tuples émergents soit moindre. Cependant, le Cube Fermé Émergent est toujours plus réduit que le Cube Émergent avec un gain appréciable.

7.6.2 Taille des Cubes Quotients Émergents

Lors de la comparaison des tailles du Cube Quotient Émergent et du Cube Émergent, nous considérons exactement les mêmes cas d'expérimentation que pour le Cube Fermé Émergent et, bien sûr, obtenons des résultats similaires. Pour les données synthétiques et non biaisées, le Cube Quotient Émergent et le Cube Émergent ont pratiquement la même taille comme illustré par les figures 7.8 et 7.9. En augmentant le facteur biais des données, la figure 7.10 montre que le Cube Quotient Émergent permet une réduction effective. Pour les données réelles, les résultats sont donnés dans la figure 7.11.

7.6.3 Comparaison des différentes représentations

Finalement, les quatre dernières figures regroupent ensemble, avec un seule échelle, le ratio de réduction de toutes les représentations. À nouveau, nous considérons les différents cas d'ex-

7. Le générateur de données synthétiques est disponible à l'adresse : `http://illimine.cs.uiuc.edu/`

FIGURE 7.4 – Taille du Cube Fermé Émergent avec $\mathcal{D} = 10$, $\mathcal{C} = 100$, $\mathcal{S} = 0$

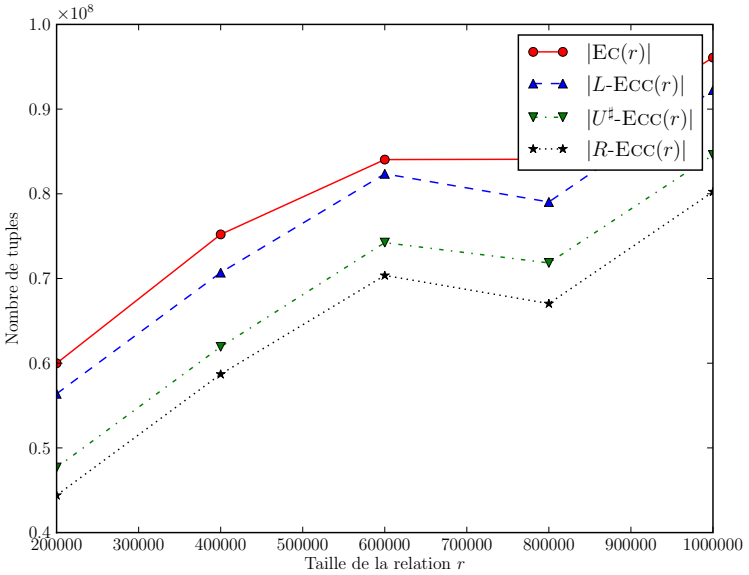


FIGURE 7.5 – Taille du Cube Fermé Émergent avec $\mathcal{D} = 10$, $\mathcal{T} = 1000$K, $\mathcal{S} = 0$

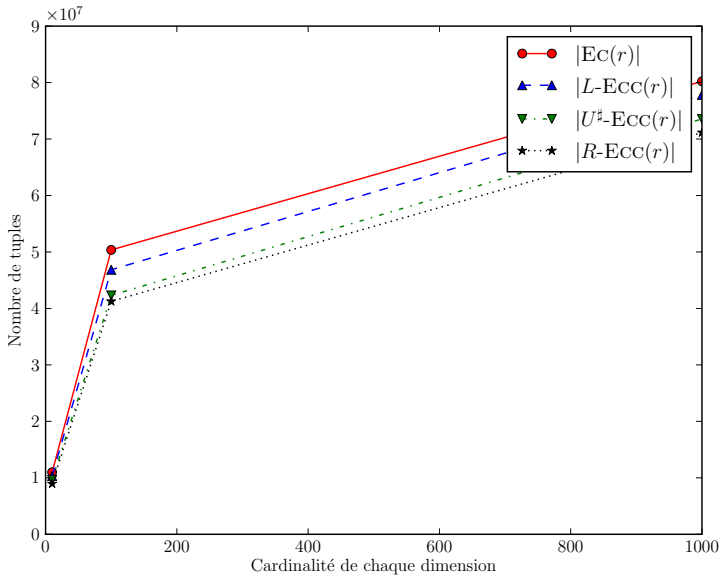


FIGURE 7.6 – Taille du Cube Fermé Émergent avec $\mathcal{D} = 10$, $\mathcal{C} = 100$, $\mathcal{T} = 1000$K

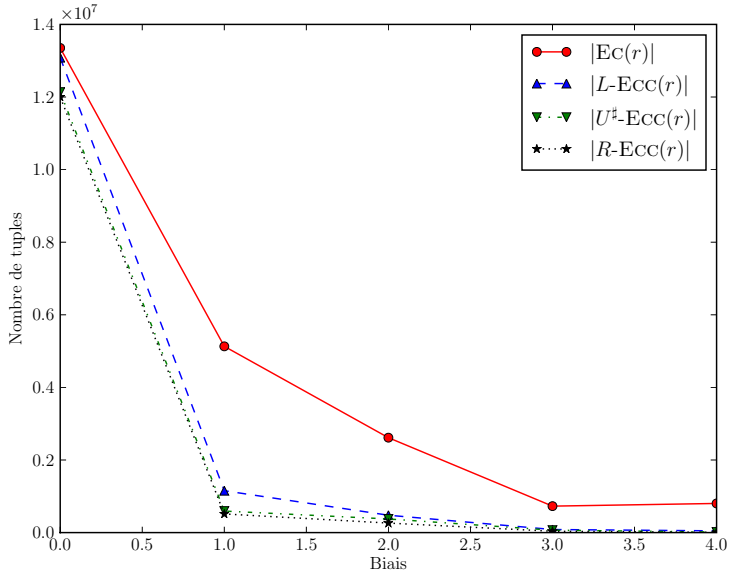


FIGURE 7.7 – Taille du Cube Fermé Émergent pour les relations de données météorologiques

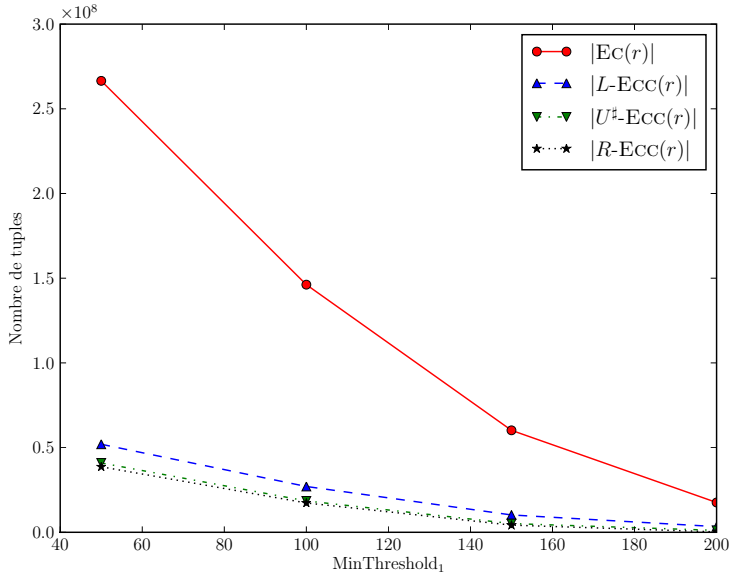


FIGURE 7.8 – Taille du Cube Quotient Émergent avec $\mathcal{D} = 10$, $\mathcal{C} = 100$, $\mathcal{S} = 0$

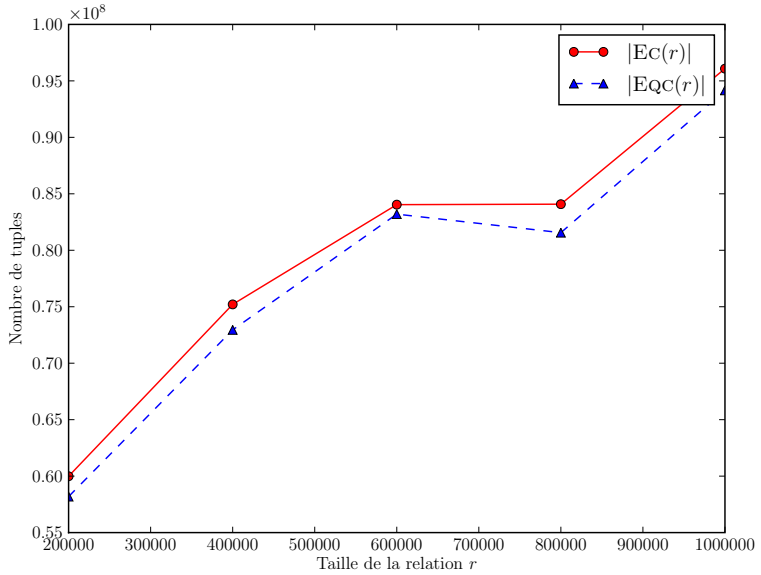


FIGURE 7.9 – Taille du Cube Quotient Émergent avec $\mathcal{D} = 10$, $\mathcal{T} = 1000$K, $\mathcal{S} = 0$

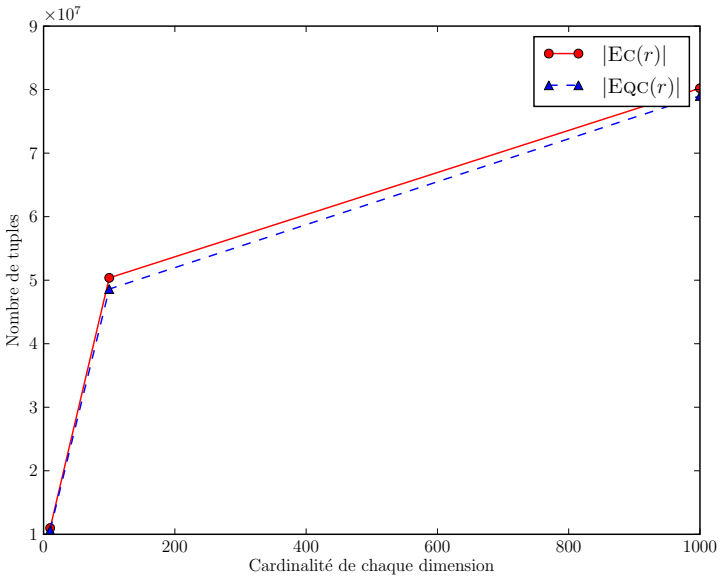


FIGURE 7.10 – Taille du Cube Quotient Émergent avec $\mathcal{D} = 10$, $\mathcal{C} = 100$, $\mathcal{T} = 1000$K

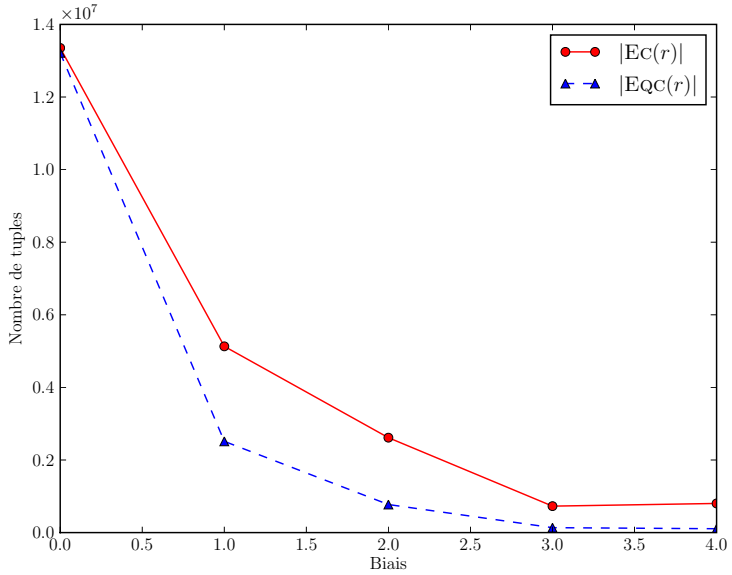


FIGURE 7.11 – Taille du Cube Quotient Émergent pour les relations de données météorologiques

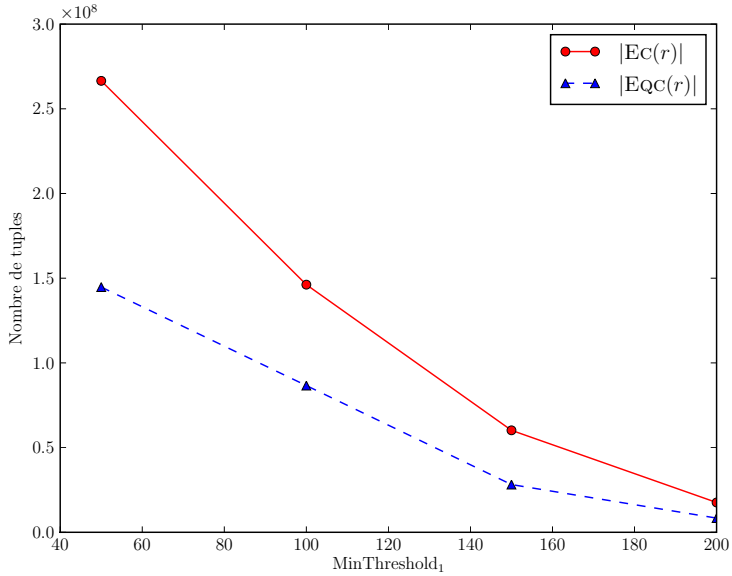
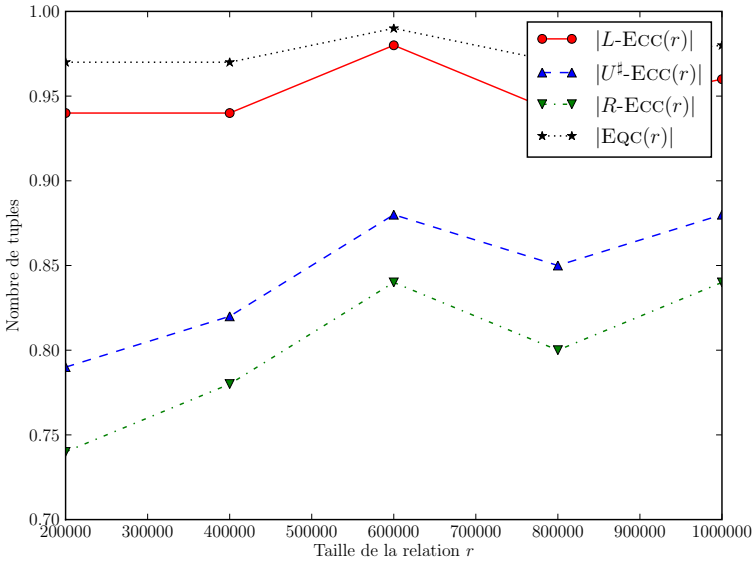


FIGURE 7.12 – Ratios de réduction avec $\mathcal{D} = 10$, $\mathcal{C} = 100$, $\mathcal{S} = 0$



périmentation : données synthétiques et non biaisées (Figures 7.12 et 7.13), données synthétique et biaisées (Figure 7.14), et données réelles (Figure 7.15). Les deux premières figures illustrent la taille particulièrement réduites des bordures quand les conditions sont particulièrement défavorables aux autres représentations. Les deux dernières figures illustrent clairement le résultat analytique concernant l'inclusion des représentations donné par le théoreme 7.11. Elles montrent que lorque le Cube Émergent englobe des redondances, les représentations étudiées ou proposées dans ce chapitre sont effectivement des représentations condensées.

FIGURE 7.13 – Ratios de réduction avec $\mathcal{D} = 10$, $\mathcal{T} = 1000$K, $\mathcal{S} = 0$

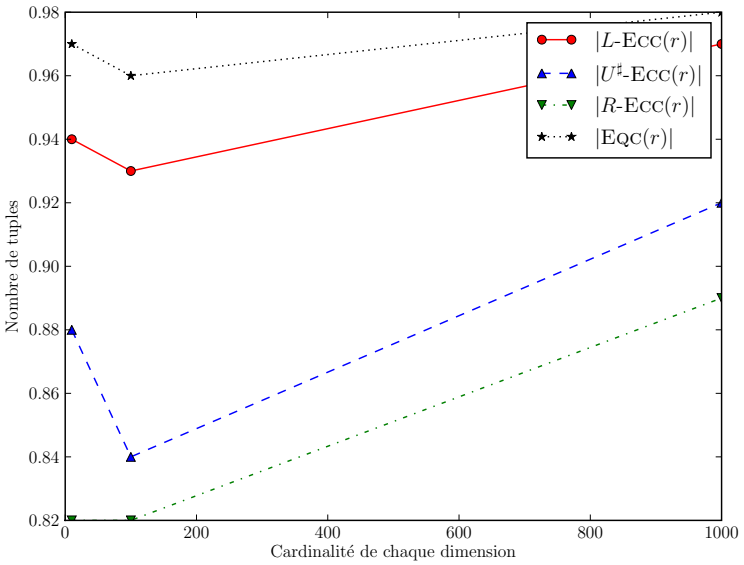


FIGURE 7.14 – Ratios de réduction avec $\mathcal{D} = 10$, $\mathcal{C} = 100$, $\mathcal{T} = 1000$K

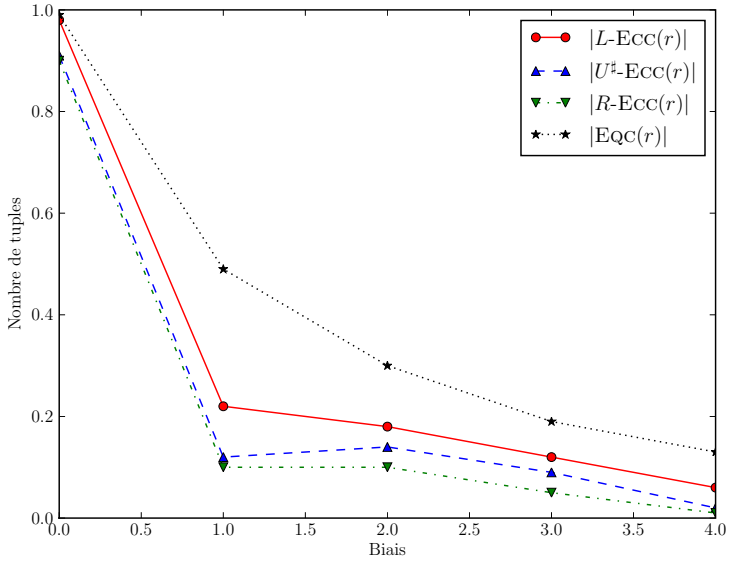
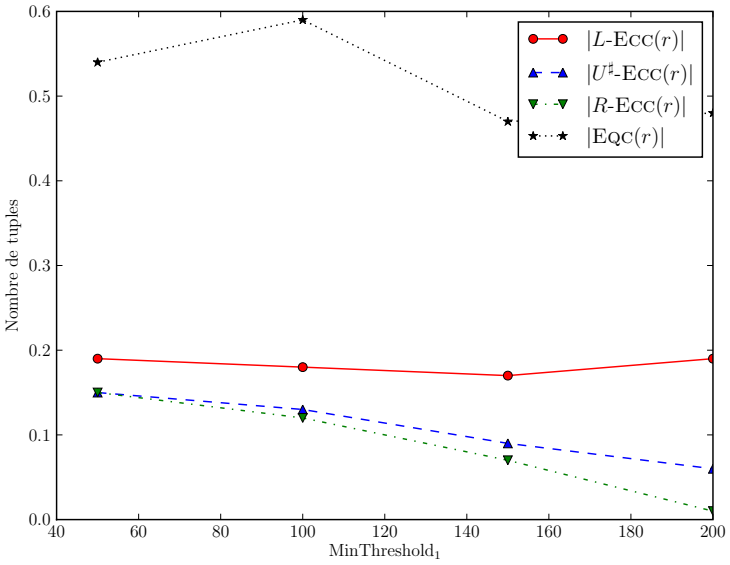


FIGURE 7.15 – Ratios de réduction pour les relations de données météorologiques



7.7 Conclusion

Complétant les représentations par bordures extrêmement concises mais ne permettant pas de dériver les valeurs des mesures, nous proposons differentes variantes du Cube Fermé Émergent qui sont toutes des représentations plus réduites que le Cube Émergent et ne génèrent pas de perte d'information. Bien sûr, ces représentations sont plus volumineuses que les bordures mais en contrepartie elles permettent de répondre à toute requête sur le Cube Émergent tout en stockant le minimum d'information. Donc si l'utilisation du Cube Émergent est la formulation de requêtes alors les variantes du Cube Fermé Émergent sont les meilleures représentations pour optimiser l'espace de stockage. Néanmoins, ces représentations n'offrent pas la capacité de naviguer dans les données à différents niveaux de granularité. Une telle fonctionnalité est possible avec le Cube Quotient qui est une représentation réduite d'un data cube. Ainsi, il est pertinent de l'adapter et de l'étendre afin d'intégrer la contrainte d'émergence. Ceci résulte dans la proposition du Cube Quotient Émergent qui est plus volumineux que le Cube Fermé Émergent mais permet de naviguer dans le cube. Selon le futur usage du Cube Émergent, il existe une « meilleure » représentation plus réduite que le Cube Émergent lui-même. Nous avons effectué des évaluations expérimentales en utilisant différents jeux de données. Dans les cas les plus défavorables quand les données n'englobent que peu de redondances, les Cubes Fermés Émergents et le Cube Quotient Émergent, qui écartent les redondances afin de réduire la taille de la représentation, ne sont pas favorisés. Heureusement, les cas cités concernent les données synthétiques. Dans les situations les plus courantes, quand des données réelles sont gérées, les Cubes Fermés Émergents et le Cube Quotient Émergent sont des représentations effectivement réduites du Cube Émergent avec une réduction appréciable de l'espace de stockage nécessaire.

8

Conclusion

Sommaire

Dans cette thèse, nous avons développé une méthode, qui se veut globale, est centrée sur le concept de Cube Émergent et permet à l'utilisateur final, le décideur, d'effectuer l'analyse des renversements de tendances. Comme l'analyse de tendances offerte par les cubes de données ou l'analyse de préférences permise par le calcul de SkyCubes, notre ambition est d'offrir une méthode permettant véritablement l'analyse des renversements de tendances qui met en évidence les phénomènes qui deviennent, au fil du temps, importants dans une population étudiée alors qu'ils étaient jusqu'alors négligeables, les caractéristiques différentes et marquées entre deux populations, les différences sensibles dans leur comportement ou, de manière générale, les écarts significatifs qui existent entre deux data cubes calculés à partir de deux relations comparables (ou deux sous-ensembles de tuples extraits d'une même relation).

Motivés par l'intérêt de ce concept qui véhicule un nouveau type de connaissances, pertinent dans de nombreux domaines d'application, nous lui avons accordé toute son importance en développant une approche globale dont nous résumons le bilan dans le premier paragraphe de ce chapitre de conclusion.

8.1 Bilan des contributions

Nous avons proposé, pour le Cube Émergent, différentes représentations réduites n'ayant pas les mêmes qualités. Nous les rappelons ci-dessous.

– Les bordures du Cube Émergent : elles offrent les représentations les plus réduites possibles mais au prix de la perte des valeurs de la mesure. Ainsi, elles permettent de savoir si une tendance est émergente mais pas d'en quantifier le taux d'émergence. Pour traiter de manière systématique ce type de questions, elles procurent des classifieurs efficaces. De plus, elles focalisent l'attention du décideur sur certains tuples particuliers qui peuvent être le point de départ d'une navigation dans le cube. En plus, des bordures classiques (L, U), nous avons défini un nouveau couple de frontières ($U^\sharp$, U) et constaté les qualités de $U^\sharp$ par rapport à L. En pratique $U^\sharp$ s'avère significativement plus réduite que L. De plus, elle nous a permis de caractériser la taille exacte des Cubes Émergents.

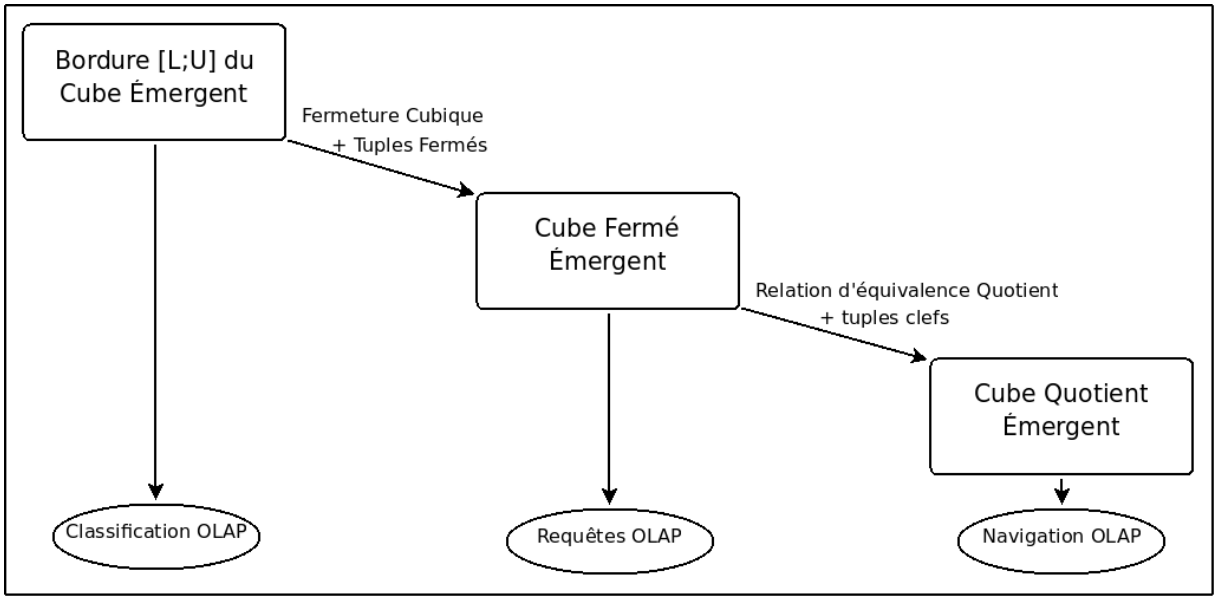


FIGURE 8.1 – Liens entre et utilisations des différentes représentations

- La famille des Cubes Fermés Émergents : en utilisant les tuples fermés émergents comme des représentants de sous-ensembles de tuples redondants, les représentations proposées éliminent la redondance contenue dans les Cubes Émergents. À cet ensemble de tuples fermés émergents, elles ajoutent l'une des bordures L, U ou $U^{\sharp}$ et sont ainsi sans perte d'information, dans le sens où les valeurs de la mesure peuvent être retrouvées. Dans ce cadre, nous avons montré que la bordure $U^{\sharp}$ peut encore être raffinée en éliminant les tuples redondants. Le résultat $U^{\sharp\sharp}$ a une taille encore plus restreinte. Les représentations basées sur le Cube Fermé Émergent permettent de répondre à toute requête qui serait formulée sur le Cube Émergent.
- Le Cube Quotient Émergent : dans le même esprit que le Cube Fermé Émergent, cette représentation élimine les redondances au sein du Cube Émergent mais de façon moins drastique que ne le fait le Cube Fermé Émergent. La conséquence, évidemment, est que la réduction apportée est moindre que celle des représentations précédentes. En revanche, le Cube Quotient Émergent préserve la capacité de naviguer dans le Cube Émergent. Avec cette structure, l'utilisateur peut non seulement formuler des requêtes mais aussi trouver les raisons d'un phénomène émergent en revenant aux tuples plus détaillés.

La figure 8.1 reprend et illustre à la fois les utilisations et les liens entre les différentes structures. Elle est une illustration de l'intuition selon laquelle plus l'utilisateur a besoin de fonctionnalités, plus le volume d'informations à préserver est important.

Pour disposer des représentations introduites, nous avons proposé deux approches. La première est une solution bases de données sous forme de plusieurs requêtes SQL. La plus efficace d'entre elles utilise une structure astucieuse pour fusionner les deux relations à comparer. La seconde est une solution algorithmique. Nous avons développé une plateforme logicielle homogène permettant d'obtenir les représentations proposées. Les algorithmes développés sont des adaptations de BUC qui calcule des cubes complets ou icebergs de manière très efficace et sans avoir recours à des structures de données sophistiquées et difficilement intégrables au sein d'un

SGBD. Afin de pouvoir utiliser BUC pour calculer et comparer les cubes de deux relations, nous avons à nouveau eu recours à la structure de relation fusionnée. La plateforme englobe :

- l'algorithme F-IDEA qui est utilisé pour calculer efficacement une, deux ou toutes les bordures du Cube Émergent en fonction de la demande des utilisateurs.
- l'algorithme E-IDEA qui restitue le Cube Émergent.

L'algorithme $\mathbb{C}$-IDEA a aussi été proposé et, toujours pour les mêmes raisons, suit le schéma algorithmique de E-IDEA. Mais, il n'est pas pour l'instant intégré à la plateforme logicielle. Son objectif est de calculer les représentations réduites sans perte basées sur le Cube Fermé Émergent.

Nous avons enfin proposé une approche d'estimation de la taille des Cubes Émergents. Elle permet, avant de lancer le moindre calcul, de savoir quel sera le volume des résultats et donc de calibrer au mieux la double contrainte d'émergence. En nous appuyant sur le concept d'idéal d'un ordre et le principe d'inclusion/exclusion, nous proposons une caractérisation de la taille exacte du Cube Émergent. Néanmoins, cette taille exacte est coûteuse à calculer. Mais, il est possible d'en donner une approximation avec une bonne précision et de manière quasi immédiate.

Nous avons mené de nombreuses évaluations expérimentales pour confirmer et quantifier nos résultats en choisissant des jeux d'essai variés. Ces expérimentations portent sur :

- la taille des représentations proposées. Nous avons mesuré la taille du Cube Émergent et de toutes les représentations introduites. Nous avons mis en œuvre les algorithmes F-IDEA et E-IDEA. Pour calculer les Cubes Fermés et Quotients Émergents, nous avons utilisé l'algorithme STARCUBING (Xin *et al.*, 2007) en utilisant la contrainte d'émergence pour élaguer l'espace de recherche. Les résultats obtenus sont conformes à nos attentes. Les bordures apportent une réduction considérable par rapport au Cube Émergent lui-même et la nouvelle bordure $U^\sharp$ est toujours plus restreinte que L et ce de manière significative. Les représentations par Cube Fermé Émergent et le Cube Quotient Émergent sont aussi très intéressantes lorsque les cubes englobent des redondances, i.e. pour les données réelles qui sont fortement corrélées. En revanche, ils sont très défavorisés lorsque des données synthétiques faiblement corrélées sont traitées. Elles n'incluent que peu de redondances et par conséquent les réductions de taille sont faibles ;
- les temps de calcul pour obtenir ces représentations. En l'absence d'algorithme calculant le Cube Émergent ou ses bordures, nous avons établi la comparaison entre E-IDEA et la requête SQL la plus efficace. En fait, notre objectif n'était pas réellement de comparer les temps de réponse des deux solutions mais plutôt de nous assurer de l'efficacité d' E-IDEA. Sur tous les jeux d'essai testés, notre algorithme s'est révélé performant. La bonne surprise révélée par les résultats pratiques est le bon comportement de la requête SQL qui, même sur des données très volumineuses, rend le Cube Émergent en un temps tout à fait acceptable (et sans aucun investissement de développement) ;
- concernant notre approche d'estimation de la taille du Cube Émergent, nous avons mesuré à la fois la précision des résultats et les temps de réponse. Pour quantifier le premier paramètre, nous avons comparé la taille exacte des Cubes Émergents et leur taille approximative. Les résultats sont réellement concluants puisque le taux d'erreur n'excède jamais 5%. Concernant les temps de réponse, il semble évident que le calcul du Cube Émergent nécessaire pour obtenir la taille exacte n'est pas réellement comparable au calcul de la taille approximative. Néanmoins, il nous importait de montrer la très grande rapidité de l'obtention de la taille approximative car, comme nous l'avons souligné, le temps de réponse est primordial pour prétendre offrir au décideur un véritable processus de calibrage des contraintes exprimées. Là encore, les résultats des expérimentations sont à la mesure de nos attentes : la taille approximative est rendue quasi instantanément.

Notre objectif dans cette thèse était, autour du concept de Cube Émergent, de proposer une approche aussi complète que possible permettant l'analyse des renversements de tendances. Il reste néanmoins des aspects à creuser davantage, des extensions possibles et des perspectives à explorer au delà de ce concept. Nous en donnons un aperçu dans le paragraphe suivant.

8.2 Perspectives

Le Cube Émergent peut être vu comme une instance d'un cube contraint (Nedjar *et al.*, 2006b) mais c'est l'instance la plus "complexe" dans la mesure où une double contrainte est exprimée mais sur deux relations différentes. Il en résulte que toutes les propositions que nous avons faites peuvent être directement données pour tout cube contraint. Il est, par exemple, immédiat de proposer des représentations réduites pour les cubes iceberg ou intervallaire, à travers les bordures, les variantes du Cube Fermé Émergent ou encore le Cube Quotient Émergent.
Mais les extensions possibles vont au delà des cubes contraints. Les nouvelles bordures introduites $]U^\sharp; U]$ peuvent être utilisées à la place de $[L; U]$ avec les mêmes avantages que nous avons observés. Ainsi, en classification automatique ou en fouille de données binaires pour déterminer si un motif contraint est une solution, les bordures $]U^\sharp; U]$ peuvent être exploitées plus efficacement que $[L; U]$.

Notre approche d'estimation de la taille d'un Cube Émergent peut être appliquée à n'importe quel cube, contraint ou non. Évidemment, si le cube est contraint, l'entrée donnée à notre algorithme Idea-LogLog peut être la bordure U ou $U^\sharp$. Dans le cas d'un data cube complet, c'est la relation originelle elle-même qui est considérée comme une bordure donnée à l'algorithme. Ce dernier est alors un véritable d'outil d'aide à l'administration de l'entrepôt. Une autre perspective intéressante de l'approche d'estimation de taille est de l'utiliser comme un guide non seulement pour le calibrage de seuil mais aussi pour choisir l'algorithme de calcul de cube le mieux adapté. En effet, l'efficacité de ces algorithmes est fortement liée à la nature même des données (fortement ou faiblement corrélées) et la taille des futurs résultats est un paramètre crucial.

Une suite du travail présenté concerne l'évaluation des requêtes. En effet, nous n'avons pas exploré et testé l'évaluation des requêtes Olap à partir des structures des Cubes Fermés et Quotients Émergents. Il faudrait, en particulier, comparer le temps de réponse des requêtes formulées sur ces structures par rapport au temps de construction des cuboïdes, nécessaires pour répondre aux mêmes requêtes, à partir de la relation originelle.
D'autre part, le graphe des classes réprésentant le Cube Quotient permet, en préservant la propriété RollUp/DrillDown du cube, la navigation au sein des données. Nous pensons que cette fonctionnalité est encore plus pertinente dans le contexte des Cubes Émergents pour rechercher les causes d'un renversements de tendances. Il est donc intéressant de concevoir une architecture logicielle combinant des techniques issues de la théorie des graphes, des interfaces homme-machine et des bases de données, pour implémenter ces capacités originales de navigation.

Concernant l'algorithme $\mathbb{C}$-Idea, le travail d'implémentation doit être réalisé. Il existe, à l'heure actuelle, deux algorithmes Close et NextClosure permettant de calculer n'importe quel opérateur de fermeture dans le contexte du treillis des parties d'un ensemble de valeurs. Notre ambition pour $\mathbb{C}$-Idea est d'atteindre le même objectif dans l'espace de recherche qu'est le treillis cube. De plus, comme il existe un plongement d'ordre du treillis cube vers le treillis des

parties, nous envisageons d'étudier le comportement de $\mathbb{C}$-Idea lors de l'extraction des motifs fermés contraints et de le comparer aux algorithmes existants dédiés. La base de transactions, utilisée dans ce contexte, est bien sûr représentable sous forme d'une relation bases de données où les domaines des attributs sont $\{0, 1\}$. La comparaison entre les différentes approches viserait non seulement les temps d'exécution et l'espace mémoire mais aussi la faisabilité de son intégration au sein d'un Sgbd. Soulignons que ce paramètre d'intégrabilité a été passé sous silence dans tous les algorithmes efficaces, mais hors du contexte bases de données.

Nous avons introduit la nouvelle structure du Cube Émergent dans le domaine assez mature de l'Olap. Une autre perspective intéressante est de tirer profit des travaux antérieurs sur les data cubes pour le Cube Émergent, par exemple en intégrant des hiérarchies de dimensions (Hurtado et Mendelzon, 2002), dans le même esprit que Cure for Cubes Morfonios et Ioannidis (2006), ou en concevant des méthodes d'indexage (Han et Kamber, 2006). Enfin, comme le souligne (Morfonios et Ioannidis, 2008), calculer un data cube n'est pas une fin en soi car son cycle de vie ne s'arrête pas là. La réponse efficace aux requêtes ainsi que la maintenance incrémentale sont aussi des problématiques à étudier. Cette remarque s'applique aussi aux représentations proposées pour le Cube Émergent. Ainsi une large perspective est de suivre ces représentations tout au long de leur cycle de vie.

Index

U

Bibliographie

Agarwal, S., Agrawal, R., Deshpande, P., Gupta, A., Naughton, J. F., Ramakrishnan, R. et Sarawagi, S. (1996). On the computation of multidimensional aggregates. *In* Vijayaraman *et al.* (1996), pages 506–521.

Agrawal, R., Imielinski, T. et Swami, A. N. (1993). Mining association rules between sets of items in large databases. *In* Buneman et Jajodia (1993), pages 207–216.

Agrawal, R., Mannila, H., Srikant, R., Toivonen, H. et Verkamo, A. I. (1996). Fast discovery of association rules. *In Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI/MIT Press.

Barbará, D. et Kamath, C., éditeurs (2003). *Proceedings of the Third SIAM International Conference on Data Mining, San Francisco, CA, USA, May 1-3, 2003*. SIAM.

Bastide, Y., Taouil, R., Pasquier, N., Stumme, G. et Lakhal, L. (2000). Mining frequent patterns with counting inference. *SIGKDD Explorations*, 2(2):66–75.

Beckmann, N., Kriegel, H.-P., Schneider, R. et Seeger, B. (1990). The r*-tree : An efficient and robust access method for points and rectangles. *In* Garcia-Molina et Jagadish (1990), pages 322–331.

Beeri, C., Dowd, M., Fagin, R. et Statman, R. (1984). On the structure of armstrong relations for functional dependencies. *J. ACM*, 31(1):30–46.

Berchtold, S., Böhm, C., Keim, D. A. et Kriegel, H.-P. (1997). A cost model for nearest neighbor search in high-dimensional data space. *In* Mendelzon et Özsoyoglu (1997), pages 78–86. Chairman-Mendelzon, Alberto and Chairman-Özsoyoglu, Z. Meral.

Berchtold, S., Keim, D. A. et Kriegel, H.-P. (1996). The x-tree : An index structure for high-dimensional data. *In* Vijayaraman *et al.* (1996), pages 28–39.

Berge, C. (1989). *Hypergraphs : combinatorics of finite sets*. North-Holland, Amsterdam.

Beyer, K. S. et Ramakrishnan, R. (1999). Bottom-up computation of sparse and iceberg cubes. *In* Delis *et al.* (1999), pages 359–370.

Birkhoff, G. (1970). *Lattice Theory*, volume XXV de *AMS Colloquium Publications*. American Mathematical Society, third (new) édition.

Böhm, K., Jensen, C. S., Haas, L. M., Kersten, M. L., Larson, P.-Å. et Ooi, B. C., éditeurs (2005). *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*. ACM.

Bonchi, F. et Lucchese, C. (2004). On closed constrained frequent pattern mining. *In* Morik et Rastogi (2004), pages 35–42.

Börzsönyi, S., Kossmann, D. et Stocker, K. (2001). The skyline operator. *In* ICDE2001 (2001), pages 421–430.

Boucelma, O., Hacid, M.-S., Libourel, T. et Petit, J.-M., éditeurs (2007). *23èmes Journées Bases de Données Avancées, BDA 2007, Marseille, 23-26 Octobre 2007, Actes (Informal Proceedings)*.

Buneman, P. et Jajodia, S., éditeurs (1993). *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*. ACM Press.

Burdick, D., Calimlim, M., Flannick, J., Gehrke, J. et Yiu, T. (2005). Mafia : A maximal frequent itemset algorithm. *IEEE Trans. Knowl. Data Eng.*, 17(11):1490–1504.

Carey, M. J. et Schneider, D. A., éditeurs (1995). *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*. ACM Press.

Casali, A. (2004). Mining borders of the difference of two datacubes. *In* Kambayashi *et al.* (2004), pages 391–400.

Casali, A., Cicchetti, R. et Lakhal, L. (2003a). Cube lattices : A framework for multidimensional data mining. *In* Barbará et Kamath (2003).

Casali, A., Cicchetti, R. et Lakhal, L. (2003b). Extracting semantics from data cubes using cube transversals and closures. *In* Getoor *et al.* (2003), pages 69–78.

Casali, A., Cicchetti, R. et Lakhal, L. (2005). Essential patterns : A perfect cover of frequent patterns. *In* Tjoa et Trujillo (2005), pages 428–437.

Casali, A., Nedjar, S., Cicchetti, R. et Lakhal, L. (2007). Convex cube : Towards a unified structure for multidimensional databases. *In* Wagner *et al.* (2007), pages 572–581.

Casali, A., Nedjar, S., Cicchetti, R., Lakhal, L. et Novelli, N. (2009). Lossless reduction of datacubes using partitions. *IJDWM*, 5(1):18–35.

Cercone, N., Lin, T. Y. et Wu, X., éditeurs (2001). *Proceedings of the 2001 IEEE International Conference on Data Mining, 29 November - 2 December 2001, San Jose, California, USA*. IEEE Computer Society.

Chaudhuri, S. et Dayal, U. (1997). An overview of data warehousing and olap technology. *SIGMOD Record*, 26(1):65–74.

Chen, M. C. et McNamee, L. (1989). On the data model and access method of summary data management. *IEEE Trans. Knowl. Data Eng.*, 1(4):519–529.

Chen, W., Naughton, J. F. et Bernstein, P. A., éditeurs (2000). *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*. ACM.

Cheng, H., Yu, P. S. et Han, J. (2006). Ac-close : Efficiently mining approximate closed itemsets by core pattern recovery. *In* Shi et Clifton (2006), pages 839–844.

Chomicki, J., Godfrey, P., Gryz, J. et Liang, D. (2005). Skyline with presorting : Theory and optimizations. *In* Klopotek *et al.* (2005), pages 595–604.

Chrisment, C., éditeur (2003). *19èmes Journées Bases de Données Avancées, BDA '03, 20-23 octobre 2003, Lyon, Actes (Informal Proceedings)*.

Dang, T. K. (2006). The sh-tree : A novel and flexible super hybrid index structure for similarity search on multidimensional data. *IJCSA*, 3(1):1–25.

Dayal, U., Whang, K.-Y., Lomet, D. B., Alonso, G., Lohman, G. M., Kersten, M. L., Cha, S. K. et Kim, Y.-K., éditeurs (2006). *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*. ACM.

Delis, A., Faloutsos, C. et Ghandeharizadeh, S., éditeurs (1999). *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*. ACM Press.

Diop, C. T., Giacometti, A., Laurent, D. et Spyratos, N. (2002). Composition of mining contexts for efficient extraction of association rules. *In* Jensen *et al.* (2002), pages 106–123.

Dogac, A., Ozsu, T. et Sellis, T., éditeurs (2007). *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, April 15-20, 2007, The Marmara Hotel, Istanbul, Turkey*. IEEE.

Dong, G. et Li, J. (1999). Efficient mining of emerging patterns : Discovering trends and differences. *In KDD*, pages 43–52.

Dong, G. et Li, J. (2005). Mining border descriptions of emerging patterns from dataset pairs. *Knowl. Inf. Syst.*, 8(2):178–202.

Eder, H. (2009). On Extending PostgreSQL with the Skyline Operator. Mémoire de D.E.A., Vienna University of Technology.

Eiter, T. et Gottlob, G. (1995). Identifying the minimal transversals of a hypergraph and related problems. *SIAM J. Comput.*, 24(6):1278–1304.

Feller, W. (1971). *An Introduction to Probability Theory and Its Applications*, volume 2. John Wiley & Sons.

Flajolet, P., Fusy, E., Gandouet, O., et Meunier, F. (2007). Hyperloglog : the analysis of a near-optimal cardinality estimation algorithm. *In Proceedings of the Conference on Analysis of Algorithms, AofA'07*, pages 127–146.

Flouvat, F., Marchi, F. D. et Petit, J.-M. (2005). A thorough experimental study of datasets for frequent itemsets. *In* Han et Wah (2005), pages 162–169.

Ganascia, J.-G. (1993). Tdis : an algebraic formalization. *In IJCAI*, pages 1008–1015.

Ganter, B. et Wille, R. (1999). *Formal Concept Analysis : Mathematical Foundations*. Springer.

Garcia-Molina, H. et Jagadish, H. V., éditeurs (1990). *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*. ACM Press.

Geerts, F., Goethals, B. et den Bussche, J. V. (2001). A tight upper bound on the number of candidate patterns. *In* Cercone *et al.* (2001), pages 155–162.

Getoor, L., Senator, T. E., Domingos, P. et Faloutsos, C., éditeurs (2003). *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 24 - 27, 2003*. ACM.

Gollapudi, S. et Sivakumar, D. (2004). Framework and algorithms for trend analysis in massive temporal data sets. *In* Grossman *et al.* (2004), pages 168–177.

Gouda, K. et Zaki, M. J. (2005). Genmax : An efficient algorithm for mining maximal frequent itemsets. *Data Min. Knowl. Discov.*, 11(3):223–242.

Grahne, G., Lakshmanan, L. V. S. et Wang, X. (2000). Efficient mining of constrained correlated sets. *In ICDE*, pages 512–521.

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

Grossman, D., Gravano, L., Zhai, C., Herzog, O. et Evans, D. A., éditeurs (2004). *Proceedings of the 2004 ACM CIKM International Conference on Information and Knowledge Management, Washington, DC, USA, November 8-13, 2004*. ACM.

Grossman, R. L., Han, J., Kumar, V., Mannila, H. et Motwani, R., éditeurs (2002). *Proceedings of the Second SIAM International Conference on Data Mining, Arlington, VA, USA, April 11-13, 2002*. SIAM.

Grust, T., Kröger, J., Gluche, D., Heuer, A. et Scholl, M. H. (1997). Query evaluation in croque - calculus and algebra coincide. *In* Small *et al.* (1997), pages 84–100.

Gunopulos, D., Khardon, R., Mannila, H. et Toivonen, H. (1997). Data mining, hypergraph transversals, and machine learning. *In* Mendelzon et Özsoyoglu (1997), pages 209–216. Chairman-Mendelzon, Alberto and Chairman-Özsoyoglu, Z. Meral.

Gupta, A., Shmueli, O. et Widom, J., éditeurs (1998). *Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*. Morgan Kaufmann.

Guttman, A. (1984). R-trees : A dynamic index structure for spatial searching. *In* Yormark (1984), pages 47–57.

Haas, L. M. et Tiwary, A., éditeurs (1998). *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*. ACM Press.

Hacid, M.-S., Ras, Z. W., Zighed, D. A. et Kodratoff, Y., éditeurs (2002). *Foundations of Intelligent Systems, 13th International Symposium, ISMIS 2002, Lyon, France, June 27-29, 2002, Proceedings*, volume 2366 de *Lecture Notes in Computer Science*. Springer.

Halevy, A. Y., Ives, Z. G. et Doan, A., éditeurs (2003). *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*. ACM.

Han, J., Chen, Y., Dong, G., Pei, J., Wah, B. W., Wang, J. et Cai, Y. D. (2005). Stream cube : An architecture for multi-dimensional analysis of data streams. *Distributed and Parallel Databases*, 18(2):173–197.

Han, J. et Kamber, M. (2006). *Data Mining : Concepts and Techniques*. Morgan Kaufmann.

Han, J., Pei, J., Dong, G. et Wang, K. (2001). Efficient computation of iceberg cubes with complex measures. *In SIGMOD Conference*, pages 1–12.

Han, J., Pei, J. et Yin, Y. (2000). Mining frequent patterns without candidate generation. *In* Chen *et al.* (2000), pages 1–12.

Han, J. et Wah, B., éditeurs (2005). *Proceedings of the 5th IEEE International Conference on Data Mining (ICDM 2005), 27-30 November 2005, Houston, Texas, USA*. IEEE Computer Society.

Harinarayan, V., Rajaraman, A. et Ullman, J. D. (1996). Implementing data cubes efficiently. *In* Jagadish et Mumick (1996), pages 205–216.

Hjaltason, G. R. et Samet, H. (1999). Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318.

Hurtado, C. A. et Mendelzon, A. O. (2002). Olap dimension constraints. *In* Popa (2002), pages 169–179.

ICDE2001 (2001). *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*. IEEE Computer Society.

Jagadish, H. V. et Mumick, I. S., éditeurs (1996). *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*. ACM Press.

Jarke, M., Carey, M. J., Dittrich, K. R., Lochovsky, F. H., Loucopoulos, P. et Jeusfeld, M. A., éditeurs (1997). *Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*. Morgan Kaufmann.

Jensen, C. S., Jeffery, K. G., Pokorný, J., Saltenis, S., Bertino, E., Böhm, K. et Jarke, M., éditeurs (2002). *Advances in Database Technology - EDBT 2002, 8th International Conference on Extending Database Technology, Prague, Czech Republic, March 25-27, Proceedings*, volume 2287 de *Lecture Notes in Computer Science*. Springer.

Jr., R. J. B. (1998). Efficiently mining long patterns from databases. *In* Haas et Tiwary (1998), pages 85–93.

Jr., R. J. B. et Agrawal, R. (1999). Mining the most interesting rules. *In KDD*, pages 145–154.

Jr., R. J. B., Agrawal, R. et Gunopulos, D. (2000). Constraint-based rule mining in large, dense databases. *Data Min. Knowl. Discov.*, 4(2/3):217–240.

Kambayashi, Y., Mohania, M. K. et Tjoa, A. M., éditeurs (2000). *Data Warehousing and Knowledge Discovery, Second International Conference, DaWaK 2000, London, UK, September 4-6, 2000, Proceedings*, volume 1874 de *Lecture Notes in Computer Science*. Springer.

Kambayashi, Y., Mohania, M. K. et Wöss, W., éditeurs (2004). *Data Warehousing and Knowledge Discovery, 6th International Conference, DaWaK 2004, Zaragoza, Spain, September 1-3, 2004, Proceedings*, volume 3181 de *Lecture Notes in Computer Science*. Springer.

Kim, W., Kohavi, R., Gehrke, J. et DuMouchel, W., éditeurs (2004). *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Seattle, Washington, USA, August 22-25, 2004*. ACM.

Klopotek, M. A., Wierzchon, S. T. et Trojanowski, K., éditeurs (2005). *Intelligent Information Processing and Web Mining, Proceedings of the International IIS : IIPWM'05 Conference held in Gdansk, Poland, June 13-16, 2005*, Advances in Soft Computing. Springer.

Koch, C., Gehrke, J., Garofalakis, M. N., Srivastava, D., Aberer, K., Deshpande, A., Florescu, D., Chan, C. Y., Ganti, V., Kanne, C.-C., Klas, W. et Neuhold, E. J., éditeurs (2007). *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*. ACM.

Kossmann, D., Ramsak, F. et Rost, S. (2002). Shooting stars in the sky : An online algorithm for skyline queries. *In* Lochovsky et Shan (2002), pages 275–286.

Kotsis, N. et McGregor, D. R. (2000). Elimination of redundant views in multidimensional aggregates. *In* Kambayashi *et al.* (2000), pages 146–161.

Kung, H. T., Luccio, F. et Preparata, F. P. (1975). On finding the maxima of a set of vectors. *J. ACM*, 22(4):469–476.

Lakshmanan, L. V. S., Pei, J. et Han, J. (2002). Quotient cube : How to summarize the semantics of a data cube. *In* Lochovsky et Shan (2002), pages 778–789.

Laporte, M., Novelli, N., Cicchetti, R. et Lakhal, L. (2002). Computing full and iceberg datacubes using partitions. *In* Hacid *et al.* (2002), pages 244–254.

Laurent, D., éditeur (2006). *22èmes Journées Bases de Données Avancées, BDA 2006, Lille, 17-20 octobre 2006, Actes (Informal Proceedings)*.

Li, W., Han, J. et Pei, J. (2001). Cmar : Accurate and efficient classification based on multiple class-association rules. *In* Cercone *et al.* (2001), pages 369–376.

Liu, B., Hsu, W. et Ma, Y. (1998). Integrating classification and association rule mining. *In KDD*, pages 80–86.

Liu, L., Reuter, A., Whang, K.-Y. et Zhang, J., éditeurs (2006). *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*. IEEE Computer Society.

Lochovsky, F. H. et Shan, W., éditeurs (2002). *Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*. Morgan Kaufmann.

Lopes, S., Petit, J.-M. et Lakhal, L. (2002). Functional and approximate dependency mining : database and fca points of view. *J. Exp. Theor. Artif. Intell.*, 14(2-3):93–114.

Mannila, H. et Toivonen, H. (1996). Multiple uses of frequent sets and condensed representations (extended abstract). *In KDD*, pages 189–194.

Mannila, H. et Toivonen, H. (1997). Levelwise search and borders of theories in knowledge discovery. *Data Min. Knowl. Discov.*, 1(3):241–258.

Matousek, J. et Nesetril, J. (2004). *Introduction aux mathématiques discrètes*. Springer.

Mendelzon, A. et Özsoyoglu, Z. M., éditeurs (1997). *PODS '97 : Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, New York, NY, USA. ACM. Chairman-Mendelzon, Alberto and Chairman-Özsoyoglu, Z. Meral.

Mitchell, T. M. (1982). Generalization as Search. *In Artificial Intelligence*, volume 18(2), pages 203–226.

Mitchell, T. M. (1997). *Machine learning*. MacGraw-Hill Series in Computer Science.

Morfonios, K. et Ioannidis, Y. E. (2006). Cure for cubes : Cubing using a rolap engine. *In* Dayal *et al.* (2006), pages 379–390.

Morfonios, K. et Ioannidis, Y. E. (2008). Supporting the data cube lifecycle : the power of rolap. *VLDB J.*, 17(4):729–764.

Morfonios, K., Konakas, S., Ioannidis, Y. E. et Kotsis, N. (2007). Rolap implementations of the data cube. *ACM Comput. Surv.*, 39(4).

Morik, K. et Rastogi, R., éditeurs (2004). *Proceedings of the 4th IEEE International Conference on Data Mining (ICDM 2004), 1-4 November 2004, Brighton, UK*. IEEE Computer Society.

Morse, M. D., Patel, J. M. et Jagadish, H. V. (2007). Efficient skyline computation over low-cardinality domains. *In* Koch *et al.* (2007), pages 267–278.

Nasr, G. E. et Badr, C. (2003). Buc algorithm for iceberg cubes : Implementation and sensitivity analysis. *In* Russell et Haller (2003), pages 255–260.

Nebel, B., éditeur (2001). *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*. Morgan Kaufmann.

Nedjar, S. (2009). Exact and approximate sizes of convex datacubes. *In* Song *et al.* (2009).

Nedjar, S., Casali, A., Cicchetti, R. et Lakhal, L. (2006a). Cocktail de cubes. *In* Laurent (2006).

Nedjar, S., Casali, A., Cicchetti, R. et Lakhal, L. (2006b). Cubes convexes. *Ingénierie des Systèmes d'Information*, 11(6):11–31.

Nedjar, S., Casali, A., Cicchetti, R. et Lakhal, L. (2007a). Emerging cubes for trends analysis in olap databases. *In* Song *et al.* (2007), pages 135–144.

Nedjar, S., Casali, A., Cicchetti, R. et Lakhal, L. (2007b). Résumés du cube emergent. *In* Boucelma *et al.* (2007).

Nedjar, S., Casali, A., Cicchetti, R. et Lakhal, L. (2008). Upper borders for emerging cubes. *In* Song *et al.* (2008), pages 45–54.

Nedjar, S., Casali, A., Cicchetti, R. et Lakhal, L. (2009). Emerging cubes : Borders, size estimations and lossless reductions. *Information Systems*, 34(6):536–550.

Nedjar, S., Casali, A., Cicchetti, R. et Lakhal, L. (2010). Reduced representations of emerging cubes for olap database mining. *IJBIDM*, 5(1):268–300.

Ng, R. T., Lakshmanan, L. V. S., Han, J. et Pang, A. (1998). Exploratory mining and pruning optimizations of constrained association rules. *In* Haas et Tiwary (1998), pages 13–24.

Norris, E. M. (1978). An algorithm for computing the maximal rectangles in a binary relation. *Revue Roumaine de mathematiques Pures et Appliquees*, 23(2):243–250.

Novelli, N. et Maabout, S. (2003). Algorithme efficace de calcul du produit de partitions et ses applications. *In* Chrisment (2003).

Papadias, D., Tao, Y., Fu, G. et Seeger, B. (2003). An optimal and progressive algorithm for skyline queries. *In* Halevy *et al.* (2003), pages 467–478.

Pasquier, N., Bastide, Y., Taouil, R. et Lakhal, L. (1999). Efficient mining of association rules using closed itemset lattices. *Information Systems*, 24(1):25–46.

Pei, J., Fu, A. W.-C., Lin, X. et Wang, H. (2007). Computing compressed multidimensional skyline cubes efficiently. *In* Dogac *et al.* (2007), pages 96–105.

Pei, J., Han, J. et Lakshmanan, L. V. S. (2004). Pushing convertible constraints in frequent itemset mining. *Data Min. Knowl. Discov.*, 8(3):227–252.

Pei, J., Han, J. et Mao, R. (2000). Closet : An efficient algorithm for mining frequent closed itemsets. *In ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 21–30.

Pei, J., Yuan, Y., Lin, X., Jin, W., Ester, M., Liu, Q., Wang, W., Tao, Y., Yu, J. X. et Zhang, Q. (2006). Towards multidimensional subspace skyline analysis. *ACM Trans. Database Syst.*, 31(4):1335–1381.

Popa, L., éditeur (2002). *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*. ACM.

Preparata, F. P. et Shamos, M. I. (1985). *Computational Geometry - An Introduction*. Springer.

Raedt, L. D. et Kramer, S. (2001). The levelwise version space algorithm and its application to molecular fragment finding. *In* Nebel (2001), pages 853–862.

Ross, K. A. et Srivastava, D. (1997). Fast computation of sparse datacubes. *In* Jarke *et al.* (1997), pages 116–125.

Roussopoulos, N., Kelley, S. et Vincent, F. (1995). Nearest neighbor queries. *In* Carey et Schneider (1995), pages 71–79.

Russell, I. et Haller, S. M., éditeurs (2003). *Proceedings of the Sixteenth International Florida Artificial Intelligence Research Society Conference, May 12-14, 2003, St. Augustine, Florida, USA*. AAAI Press.

Sellis, T. K., Roussopoulos, N. et Faloutsos, C. (1987). The r+-tree : A dynamic index for multi-dimensional objects. *In* Stocker *et al.* (1987), pages 507–518.

Shi, Y. et Clifton, C. W., éditeurs (2006). *Proceedings of the 6th IEEE International Conference on Data Mining (ICDM 2006), 18-22 December 2006, Hong Kong, China*. IEEE Computer Society.

Shoshani, A. (1997). Olap and statistical databases : Similarities and differences. *In* Mendelzon et Özsoyoglu (1997), pages 185–196. Chairman-Mendelzon, Alberto and Chairman-Özsoyoglu, Z. Meral.

Shukla, A., Deshpande, P. et Naughton, J. F. (1998). Materialized view selection for multidimensional datasets. *In* Gupta *et al.* (1998), pages 488–499.

Shukla, A., Deshpande, P., Naughton, J. F. et Ramasamy, K. (1996). Storage estimation for multidimensional aggregates in the presence of hierarchies. *In* Vijayaraman *et al.* (1996), pages 522–531.

Small, C., Douglas, P., Johnson, R. G., King, P. J. H. et Martin, G. N., éditeurs (1997). *Advances in Databases, 15th British National Conferenc on Databases, BNCOD 15, London, United Kindom, July 7-9, 1997, Proceedings*, volume 1271 de *Lecture Notes in Computer Science*. Springer.

Song, I.-Y., Eder, J. et Nguyen, T. M., éditeurs (2007). *Data Warehousing and Knowledge Discovery, 9th International Conference, DaWaK 2007, Regensburg, Germany, September 3-7, 2007, Proceedings*, volume 4654 de *Lecture Notes in Computer Science*. Springer.

Song, I.-Y., Eder, J. et Nguyen, T. M., éditeurs (2008). *Data Warehousing and Knowledge Discovery, 10th International Conference, DaWaK 2008, Turin, Italy, September 2-5, 2008, Proceedings*, volume 5182 de *Lecture Notes in Computer Science*. Springer.

Song, I.-Y., Eder, J. et Nguyen, T. M., éditeurs (2009). *Data warehousing and knowledge discovery, 9th international conference, dawak 2009, linz, austria, september 1-4, 2009, proceedings*, Lecture Notes in Computer Science. Springer.

Spyratos, N. (1987). The partition model : A deductive database model. *ACM Trans. Database Syst.*, 12(1):1–37.

Srikant, R. et Agrawal, R. (1996). Mining quantitative association rules in large relational tables. *In* Jagadish et Mumick (1996), pages 1–12.

Stocker, P. M., Kent, W. et Hammersley, P., éditeurs (1987). *Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*. Morgan Kaufmann.

Stumme, G., Taouil, R., Bastide, Y., Pasquier, N. et Lakhal, L. (2002). Computing iceberg concept lattices with titanic. *Data Knowl. Eng.*, 42(2):189–222.

Tjoa, A. M. et Trujillo, J., éditeurs (2005). *Data Warehousing and Knowledge Discovery, 7th International Conference, DaWaK 2005, Copenhagen, Denmark, August 22-26, 2005, Proceedings*, volume 3589 de *Lecture Notes in Computer Science*. Springer.

Vel, M. (1993). *Theory of Convex Structures*. Numéro 50 de North-Holland Mathematical Library. North-Holland, Amsterdam.

Vijayaraman, T. M., Buchmann, A. P., Mohan, C. et Sarda, N. L., éditeurs (1996). *Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*. Morgan Kaufmann.

Wagner, R., Revell, N. et Pernul, G., éditeurs (2007). *Database and Expert Systems Applications, 18th International Conference, DEXA 2007, Regensburg, Germany, September 3-7, 2007, Proceedings*, volume 4653 de *Lecture Notes in Computer Science*. Springer.

Wang, J., Han, J. et Li, C. (2007). Frequent closed sequence mining without candidate maintenance. *IEEE Trans. Knowl. Data Eng.*, 19(8):1042–1056.

Xin, D., Han, J., Li, X., Shao, Z. et Wah, B. W. (2007). Computing iceberg cubes by top-down and bottom-up integration : The starcubing approach. *IEEE Trans. Knowl. Data Eng.*, 19(1):111–126.

Xin, D., Shao, Z., Han, J. et Liu, H. (2006). C-cubing : Efficient computation of closed cubes by aggregation-based checking. *In* Liu *et al.* (2006), page 4.

Yang, G. (2004). The complexity of mining maximal frequent itemsets and maximal frequent patterns. *In* Kim *et al.* (2004), pages 344–353.

Yang, G. (2006). Computational aspects of mining maximal frequent patterns. *Theor. Comput. Sci.*, 362(1-3):63–85.

Yormark, B., éditeur (1984). *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*. ACM Press.

Yuan, Y., Lin, X., Liu, Q., Wang, W., Yu, J. X. et Zhang, Q. (2005). Efficient computation of the skyline cube. *In* Böhm *et al.* (2005), pages 241–252.

Zaki, M. J. et Hsiao, C.-J. (2002). Charm : An efficient algorithm for closed itemset mining. *In* Grossman *et al.* (2002).

Zaki, M. J. et Hsiao, C.-J. (2005). Efficient algorithms for mining closed itemsets and their lattice structure. *IEEE Trans. Knowl. Data Eng.*, 17(4):462–478.

Zhang, X., Chou, P. L. et Dong, G. (2007). Efficient computation of iceberg cubes by bounding aggregate functions. *IEEE Trans. Knowl. Data Eng.*, 19(7):903–918.

Résumé

Découvrir des renversements de tendances entre deux cubes de données offre aux utilisateurs une connaissance nouvelle et intéressante lors des fluctuations de l'univers réel modélisé : quelles sont les nouveautés ? Quelle tendance apparaît ou disparaît ? Nous introduisons le nouveau concept de Cube Émergent. Il capture les renversements de tendances en mettant en œuvre une contrainte d'émergence (conjonction de contrainte monotones et antimonotones). Les bordures, classiques en fouille de données, sont reprises pour le Cube Émergent. Dans un second temps, nous proposons un nouveau couple de bordures pour optimiser à la fois l'espace de stockage et le temps de calcul. Cette nouvelle représentation fournit une caractérisation simple de la taille du Cube Émergent aussi bien que des outils de classification et de navigation dans les cubes. La connexion entre les bordures classiques et celles proposées est formellement établie en utilisant le concept de cube transversal. Connaître la taille du Cube Émergent est d'un grand intérêt, en particulier pour ajuster au mieux la contrainte d'émergence sous-jacente. Cette problématique est traitée en étudiant une borne supérieure et en caractérisant la taille exacte du Cube Émergent. Deux stratégies sont proposées pour estimer rapidement cette taille : la première est basée sur une estimation analytique, sans accès à la base de données, la seconde s'appuie sur un comptage probabiliste utilisant les bordures proposées comme entrée de l'algorithme proche de l'optimal HYPERLOGLOG. Grâce à la particulière efficacité de cet algorithme, plusieurs itérations peuvent être réalisées pour calibrer au mieux la contrainte d'émergence. De plus, des nouvelles représentations réduites et sans perte d'information du Cube Émergent sont proposées en utilisant le concept de fermeture cubique.

Mots-clés: OLAP, bases de données, fouille de données multidimensionnelles, cube de données, treillis cube, bordures, cube fermé, cube quotient, représentation réduite.