Optimisation des requêtes skyline multidimensionnelles

Patrick Kamnang Wanko

▶ **To cite this version:**

Patrick Kamnang Wanko. Optimisation des requêtes skyline multidimensionnelles. Autre [cs.OH]. Université de Bordeaux, 2017. Français. NNT : 2017BORD0010 . tel-01507468

HAL Id: tel-01507468
https://theses.hal.science/tel-01507468

Submitted on 13 Apr 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Patrick Kamnang Wanko**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Optimisation des requêtes skyline multidimensionnelles

Soutenue publiquement le : 09 Février 2017

Devant la commission d'examen composée de :

M.	Bernd AMANN . .	Professeur des universités	Rapporteur
M.	Mohand-Saïd HACID	Professeur des universités	Rapporteur
Mme.	Alessia MILANI . .	Maître de conférences .	Examinatrice
M.	Patrick VALDURIEZ	Directeur de recherches	Examinateur
M.	Nicolas HANUSSE .	Directeur de recherches	Co-directeur de thèse
M.	Sofian MAABOUT .	Maître de conférences .	Co-directeur de thèse

– 2017 –

Résumé Dans le cadre de la sélection de meilleurs éléments au sein d'une base de données multidimensionnelle, plusieurs types de requêtes ont été définies. L'opérateur skyline présente l'avantage de ne pas nécessiter la définition d'une fonction de score permettant de classer lesdits éléments. Cependant, la propriété de monotonie que cet opérateur ne présente pas, rend non seulement (i) difficile l'optimisation de ses requêtes dans un contexte multidimensionnel, mais aussi (ii) presque imprévisible la taille du résultat des requêtes. Ce travail se propose, dans un premier temps, d'aborder la question de l'estimation de la taille du résultat d'une requête skyline donnée, en formulant des estimateurs présentant de bonnes propriétés statistiques (sans biais ou convergeant). Ensuite, il fournit deux approches différentes à l'optimisation des requêtes skyline. La première reposant sur un concept classique des bases de données qui est la dépendance fonctionnelle. La seconde se rapprochant des techniques de compression des données. Ces deux techniques trouvent leur place au sein de l'état de l'art comme le confortent les résultats expérimentaux. Nous abordons enfin la question de requêtes skyline au sein de données dynamiques en adaptant l'une de nos solutions précédentes dans cet intérêt.

Title Optimization of multidimensional skyline queries

Abstract As part of the selection of the best items in a multidimensional database, several kinds of query were defined. The skyline operator has the advantage of not requiring the definition of a scoring function in order to classify tuples. However, the property of monotony that this operator does not satify, (i) makes difficult to optimize its queries in a multidimensional context, (ii) makes hard to estimate the size of query result. This work proposes, first, to address the question of estimating the size of the result of a given skyline query, formulating estimators with good statistical properties (unbiased or convergent). Then, it provides two different approaches to optimize multidimensional skyline queries. The first leans on a well known database concept: functional dependencies. And the second approach looks like a data compression method. Both algorithms are very interesting as confirm the experimental results. Finally, we address the issue of skyline queries in dynamic data by adapting one of our previous solutions in this goal.

Keywords Skyline, Cardinality, Size, Optimization, Skycube, Skycuboid, Functional Dependency

Mots-clés Skyline, Cardinalité, Taille, Optimisation, Skycube, Skycuboid, Dépendance Fonctionnelle

Laboratoire d'accueil Laboratoire Bordelais de Recherche en Informatique (LaBRI), Unité Mixte de Recherche CNRS (UMR 5800); 351, cours de la Libération F-33405 Talence cedex

Remerciements

Je remercie l'Université de Bordeaux et le Laboratoire Bordelais de Recherche en Informatique de m'avoir accueilli dans le cadre de la réalisation de cette thèse.

J'exprime toute ma gratitude envers l'entreprise AT Internet et le Programme d'Investissements d'Avenir (PIA) qui, en supportant le projet SpeedData ont permis le financement de cette thèse.

Je remercie les rapporteurs Mohand-Saïd Hacid et Bernd Amann ainsi que les examinatrice et examinateur Alessia Milani et Patrick Valduriez pour l'attention qu'ils ont bien voulu porter à ce travail.

Je ne saurais poursuivre sans témoigner ma sincère gratitude envers mes directeurs de thèse Nicolas Hanusse et Sofian Maabout qui m'ont suivi tout le long de ce travail. Je leur dis merci pour l'encadrement, le soutien, les conseils et surtout l'encouragement qu'ils ont su m'apporter afin de parvenir à l'accomplissement de ce travail.

Ma gratitude va également à l'endroit des membres des groupes de travail EVA-DoMe et Masses de données (BigData) pour nos discussions et échanges édifiants.

Merci, à tous ceux que j'ai oublié de citer, qu'ils m'en excusent.

Mes remerciements vont enfin à l'endroit de mes proches qui m'ont toujours apporté réconfort : mon beau frère Voltaire, mon ami Paulin, mes parents Clément et Pauline, mes sœurs Marcelle, Marlyse, Josline, Paulette, Ghislaine, Vanessa et Audrey, mon épouse Michèle, sans oublier, un grand clin d'œil à mes filles Abigail et Emilie à qui je dédie ce travail.

Table des matières

Table des matières vii

Liste des tableaux ix

Table des figures xi

1 Introduction Générale 1

2 Estimation de la Taille du Skyline 13
Introduction . . . 13
2.1 Travaux relatifs . . . 14
2.2 Estimation de la taille du skyline . . . 16
2.2.1 Parcours des données . . . 18
2.2.2 Espérance de la taille du skyline . . . 24
2.3 Expérimentations . . . 28
2.3.1 Variation du nombre de tuples . . . 30
2.3.2 Variation du nombre de dimensions . . . 32
2.3.3 Variation du nombre de valeurs distinctes par dimension . . . 34
2.3.4 Données réelles . . . 35
Conclusion . . . 37

3 Matérialisation Partielle pour Optimiser les Requêtes Skyline 39
Introduction . . . 39
3.1 Travaux relatifs . . . 41
3.2 Matérialisation Partielle des Skycubes . . . 43
3.2.1 Monotonie des Motifs Skyline . . . 44
3.2.2 Matérialisation basée sur le topmost Skyline . . . 46
3.2.3 Le sous-skycube Minimal Complet en Information (MICS) 48
3.2.4 Dépendances Fonctionnelles et Inclusion entre skylines . . . 49
3.2.5 Recherche des Sous-espaces Clos . . . 52
3.2.6 Calcul du Sous-skycube Minimal à Information Complète (MICS) . . . 55
3.2.7 Analyse du nombre d'Espaces Clos . . . 57
3.2.8 Analyse de la Taille du Skyline . . . 58

3.3 Evaluation des requêtes . . . 61
3.3.1 Calcul de $Sky(Sky(X^+), X)$ . . . 62
3.3.2 Utilisation des Partitions . . . 63
3.3.3 Exécution de l'opération $Sky(Sky(Y), X)$ . . . 65
3.3.4 Calcul du Skycube Complet . . . 65
3.4 Mise en commun des composants . . . 67
3.5 Expérimentations . . . 68
3.5.1 Les données . . . 69
3.5.2 Calcul en Parallèle . . . 70
3.5.3 Construction du Skycube complet . . . 70
3.5.4 La Matérialisation Partielle du Skycube . . . 78
3.5.5 Exécution des requêtes . . . 84
Conclusion . . . 85

4 Calculer et Résumer le Skycube **87**
Introduction . . . 87
4.1 Le Skycube Négatif . . . 89
4.1.1 Réduction de la taille de NSC . . . 93
4.1.2 Exécution des requêtes skyline . . . 102
4.2 Évaluations expérimentales . . . 105
4.2.1 Données réelles . . . 105
4.2.2 Données synthétiques . . . 107
4.2.3 Comparaison à Hashcube . . . 111
4.2.4 Construction du skycube . . . 112
4.2.5 Comparaison avec le Skycube Partiel (MICS) . . . 112
Conclusion . . . 113

5 Le Skycube Négatif : Extensions **115**
Introduction . . . 115
5.1 Les requêtes k-dominant Skyline . . . 115
5.1.1 Travaux relatifs . . . 116
5.1.2 NSC et le k-dominant skyline . . . 116
5.1.3 Expérimentations . . . 118
5.2 Requêtes du skycube et données dynamiques . . . 119
5.2.1 Travaux Relatifs . . . 120
5.2.2 Maintenance du Skycube Négatif . . . 120
Conclusion . . . 129

6 Conclusion et Perspectives **131**

Bibliographie **135**

Liste des tableaux

1.1 Les hôtels 2
1.2 Les skycuboids formant le skycube (table des hôtels) 5
1.3 Notations 7

2.1 Notations 16
2.2 Les hôtels : données brutes 17
2.3 Les hôtels : données *normalisées* 18
2.4 Les fonctions de densité cumulée 19
2.5 Estimation de l'espérance de la taille du skyline 28

3.1 Table de données. 40
3.2 L'ensemble de tous les skylines. 41
3.3 Comparaison des algorithmes relativement aux tâches 69
3.4 Les données réelles 70
3.5 Temps d'exécution (sec.) sur des données réelles de petite taille. FMC est exécuté avec 1 ou 12 threads 76
3.6 Temps d'exécution (sec.) pour les données artificiellement augmentées FMC est exécuté avec 1 ou 12 threads 77
3.7 MICS vs Orion 78
3.8 Skycube Partiel vs CSC 80
3.9 Skycube Partiel vs Hashcube 81
3.10 Temps d'exécution des requêtes en seconde : optimisé/non optimisé et (la proportion de skycuboids matérialisés) 85

4.1 Les hôtels 88
4.2 Une table de données T 89
4.3 NSC relatif à T 92
4.4 Liste des paires synthétisant les ensembles de dominance 97
4.5 Organisation orientée sous-espace de NSC 104
4.6 Les caractéristiques des données réelles 105

Table des figures

1.1 Skycube représenté sous forme de treillis 5

2.1 Distribution des variables Y 26
2.2 Temps d'exécution par rapport à n (Taille de l'échantillon = 5% n) 29
2.3 Erreur relative au regard de n 30
2.4 Temps d'exécution au regard de n 31
2.5 Erreur relative par rapport à d 32
2.6 Temps d'exécution par rapport à d 33
2.7 Erreur relative au regard de k 34
2.8 Temps d'exécution par rapport à k 35
2.9 Erreur relative au regard de d 35
2.10 Temps d'exécution par rapport à d 36

3.1 Treillis des motifs skycube 45
3.2 Inclusions entre skycuboids revelées par les dépendances fonctionnelles 50
3.3 Espaces élagués au regard des attributs A et B : BCD est élagué par A et ne fait partie d'aucun des espaces cherchés associés à B, C et D. Par conséquent, $\pi_{BCD}(T)$ ne sera pas calculé. ACD est élagué par B. De ce fait, $\pi_{ACD}(T)$ ne sera pas calculé. $\pi_{AC}(T)$ est élagué par B car $A \rightarrow B$ est satisfaite et AC ne fait pas partie de $2^{D \setminus A}$ mais peut être testé avec D. De même que AD qui peut être testé avec C. 53
3.4 Comportements des clés et des espaces clos 58
3.5 L'évolution de la taille du skyline et du nombre de clos par rapport à la cardinalité k 62
3.6 Le Skycube Partiel 62
3.7 Speedup par rapport à la dimensionalité d ($n = 100K$) 72
3.8 Speedup par rapport à la taille des données n ($d = 16$) 73
3.9 Calcul du skycube complet sur des données réelles 75
3.10 Matérialisation complète pour MBL10 78
3.11 Analyse quantitative des espaces clos 82

3.12 Évolution de la taille du skyline relativement à celles de k et d avec $n = 10^6$ 84

4.1 Données réelles 106
4.2 Données synthétiques corrélées 108
4.3 Données synthétiques indépendantes 109
4.4 Données synthétiques anticorrélées 110
4.5 Construction du Skycube : NSC vs QSkyCube 112
4.6 Données synthétiques indépendantes 114

5.1 Temps de calcul du k-dominant skycube 119

Liste des Algorithmes

1 S_W, estimation de la taille du skyline . . . 22
2 S_S, estimation de la taille du skyline . . . 23
3 S_E, estime la taille du skyline . . . 27

4 Sky_X_from_Sky_Y . . . 45
5 SemiNaïvePartialSkyCube . . . 47
6 MaxNFD . . . 55
7 ClosedSubspaces . . . 56
8 MICS . . . 56
9 FMC algorithm . . . 66
10 Skycube Materialization . . . 67
11 Skyline Query Evaluation . . . 68

12 BUILDNSC . . . 91
13 EVALUATESKYLINE . . . 92
14 $ApproxMSP$ . . . 96
15 $listPairsTuple$, creates a reduced list of pairs associated to a tuple . . . 96
16 $MinimalSP$, minimal set of pairs covering $\mathcal{DOM}(t)$ . . . 99
17 NSC_SKYLINE from tuples . . . 103
18 BUILD_NSC . . . 104
19 NSC_SKYLINE from subspaces . . . 104

20 NSC_k-DOMINANT_SKYLINE from subspaces . . . 118
21 $ApproxMST$ . . . 122
22 ListTriplets . . . 123
23 BUILD_NSC . . . 123
24 NSC_SKYLINE from subspaces . . . 123
25 getStatusOfNewTuple . . . 124
26 Insert new tuple . . . 126
27 CheckDominanceDelete . . . 126
28 DeleteTupleFromNSC . . . 128

Chapitre 1

Introduction Générale

Récemment, les requêtes de «préférence» ont reçu un grand intérêt de la part de la communauté de chercheurs, notamment en bases de données et fouille de données. Intuitivement, étant donné un ensemble d'objets décrits chacun par un ensemble de variables (attributs), une requête de préférence permet à l'utilisateur d'exprimer un critère (généralement, une fonction sur les attributs) permettant de décrire une relation d'ordre sur les objets qu'il s'agira ensuite d'exploiter pour ne retourner que les « meilleurs » objets selon ce critère. L'exemple simple est le résultat d'une recherche par mots clés sur un moteur de recherche tel que Google. Le résultat est trié en respectant une mesure de pertinence qui est la combinaison de plusieurs critères. Les requêtes *Skyline* en sont une autre illustration : Soit un ensemble de véhicules décrits chacun par le nombre de kilomètres parcourus, la consommation moyenne et l'année d'immatriculation. Sélectionner les meilleurs véhicules à partir de cet ensemble en utilisant une combinaison des 3 critères n'est pas aisé. Par contre, on peut savoir quels sont ceux qui ne feront certainement pas partie des meilleurs, c'est-à-dire, ceux pour lesquels il existe un autre véhicule avec moins de kilométrage, moins de consommation et une année d'immatriculation plus récente. On parle également de *Vecteur optimal*, de *Front de Pareto* ou encore d'*Optimum au sens de Pareto* dans ce sens que dans le résultat, passant d'un élément à un autre lorsqu'on améliore la situation au regard d'un critère, on la dégrade forcément relativement à un autre critère.

Déterminer le skyline d'un ensemble d'objets est une requête survenant très couramment dans le quotidien :

— le choix d'un itinéraire pour un voyage en avion. On pourrait vouloir, à la fois, minimiser le coût du billet, le nombre de correspondances et le temps total du trajet ;
— le choix d'une assurance (santé, voiture, habitation, vie, accident) qui est une opération dans laquelle on cherche à ce que le maximum de risques soient pris en charge tout en minimisant le coût ;
— le choix d'une chambre d'hôtel où les critères de choix peuvent être : la distance par rapport à notre lieu d'intérêt, la classe de d'hôtel (nombre d'étoiles),

Id	(P)rix	(D)istance	(S)urface
r_1	10	10	12
r_2	20	5	12
r_3	10	11	10

TABLE 1.1 – Les hôtels

le prix de la chambre, la surface de la chambre, la présence de certaines commodités telles que le Wifi, la préparation du petit déjeuner, etc

Depuis son introduction à la communauté base de données par Börzsönyi *et al.* [2001], l'opérateur skyline fait l'objet d'un grand intérêt. La propriété lui permettant de se démarquer de beaucoup d'autres types de requêtes de préférence est le fait de ne pas reposer sur la définition d'une fonction de score (souvent difficile à définir) permettant de classer les tuples, comme c'est le cas pour les requêtes de type *Top-K*. La seule contrainte étant l'existence d'une relation d'ordre partiel au sein des valeurs de chacune des dimensions. Nous décrivons le concept de skyline et les notions qui lui sont relatives à partir d'un exemple pratique.

Exemple illustratif Considérons la table 1.1 reprenant certaines caractéristiques des chambres d'hôtels.

Ces chambres sont décrites par leur prix, la distance à la plage et leur surface. Un client potentiel rechercherait les *meilleures* chambres, c'est-à-dire, celles minimisant à la fois le prix et la distance et maximisant la surface. La chambre r_3 ne pourrait pas faire partie du résultat parce que la chambre r_1 est meilleure que celle-ci (on dit que r_1 *domine* r_3). En effet, r_1 est plus proche de la plage, plus grande et pas plus coûteuse que r_3. r_1 et r_2 sont cependant *incomparables* car r_1 est meilleure que r_2 au regard du prix tandis que r_2 plus près de la plage que r_1. Le skyline de T au regard de l'ensemble des trois attributs P, D et S est l'ensemble $\{r_1, r_2\}$ car il s'agit des éléments qui ne sont pas dominés au regard de l'ensemble des attributs considérés.

Travaux Relatifs au Calcul du Skyline Plusieurs algorithmes implémentant le calcul du skyline ont été proposés dans la littérature, par exemple Chomicki *et al.* [2003]; Papadias *et al.* [2005]; Bartolini *et al.* [2008].

La complexité de la plupart d'entre eux est analysée en terme de consommation mémoire, par exemple, Godfrey *et al.* [2007]; Bartolini *et al.* [2008]; Lee et won Hwang [2010]. Certains algorithmes ont été spécialement conçus pour le cas où les dimensions possèdent une faible cardinalité, par exemple, Morse *et al.* [2007b]. Tous ces algorithmes ont une complexité dans le pire des cas de l'ordre de $O(n^2)$ où n représente le nombre de tuples (taille de la table T). Le travail pionnier à ce sujet Börzsönyi *et al.* [2001], considère les données stockées sur disque et montre l'inadéquation de requêtes SQL pour efficacement exécuter les requêtes skyline. Toutefois, les algorithmes qu'il propose *souffrent* également de la complexité polynomiale en

temps. Récemment, Sheng et Tao [2012] a proposé un algorithme prenant en considération l'optimisation du nombre d'entrées/sorties et garantissant dans le pire des cas un nombre polylogarithmique d'accès disque. Sarma *et al.* [2009] a proposé un algorithme requérant $O(\log(n))$ parcours des données afin de déterminer un résultat approximatif du skyline avec grande probabilité. D'autres travaux font usage de certains outils tels que les index multidimensionnels. Par exemple, Papadias *et al.* [2005] a proposé d'utiliser les R-arbres (R-trees) pour optimiser la recherche des points skyline de manière progressive.

Au regard de nos lectures, l'algorithme BSkyTree de Lee et won Hwang [2010] est considéré comme étant l'algorithme le plus efficace pour une exécution séquentielle. Nous présentons ci-dessous cet algorithme ainsi que l'algorithme LESS mis pour *Linear Elimination Sort for Skyline* de Godfrey *et al.* [2005].

L'algorithme LESS combine les principes de deux précédentes approches à savoir (i) le traitement par block des données inspiré de l'algorithme BNL (*Block Nested Loop*) de Börzsönyi *et al.* [2001] et le tri des tuples élaboré par l'algorithme SFS (*Sort and Filter Skyline*) de Chomicki *et al.* [2003]. Il s'agit de conserver une fenêtre des tuples skyline de la partie traitée des données. Contrairement à l'approche SFS, un tuple est inséré dans cette fenêtre s'il fera partie du skyline final (il n'en sort plus), donc il peut déjà être retourné comme résultat. L'algorithme commence par le tri des tuples suivant le score obtenu en appliquant la fonction d'entropie $f(t) = \sum_{j=1}^{d} \ln(t[j])$ aux tuples t de la table. Ensuite, les tuples d'entropie la plus faible sont comparés entre eux et leur skyline est chargé dans la fenêtre (ceux-ci feront partie du skyline final), ils permettent alors de réduire considérablement la tâche à effectuer car ils élimineront les tuples qu'ils dominent dans le reste des données. Progressivement les blocs des données dont traités en comparant leurs tuples à ceux de la fenêtre courante. Les tuples qui ne sont pas dominés seront rajoutés à la fenêtre.

L'algorithme BSkyTree quant à lui, procède par le partitionnement des données. Il commence par choisir dans les données un tuple $p^v = (v_1, v_2, \ldots, v_d)$ que l'on appelle le pivot. Les valeurs de ce pivot définissent un partitionnement de l'espace complet en 2^d parties. La valeur v_j divise la dimension D_j en deux parties : la partie des points présentant une valeur inférieure à v_j pour la dimension D_j et celle des points présentant une valeur supérieure à v_j pour la dimension D_j. Chaque tuple de la table est alors associé à l'une des 2^d parties créées. Une relation de dominance (induites par celle des tuples qu'ils contiennent) peut alors être établie entre les différentes parties. Si on code par un vecteur $(a_1, a_2, \ldots, a_d)$ les différentes parties créées, de telles sorte que le j^{em} élément du vecteur présente la valeur 0 si tous les points de cette partie présentent une valeur inférieure à v_j pour la dimensions D_j et 1 sinon. Alors, dans le cas de deux dimensions :

— tout tuple de la partie $(0, 0)$ domine tout tuple de la partie $(1, 1)$ et ne nécessitent plus d'être comparés entre eux ;
— tout tuple de $(0, 1)$ est incomparable à tout tuple de $(1, 0)$ et ne nécessitent

plus d'être comparés ;

— les tuples de $(0, 0)$ dominent partiellement ceux de $(0, 1)$ et ceux de $(1, 0)$. Ils ne seront comparés qu'en les dimensions pour lesquelles un doute subsiste (respectivement en A d'une part et B d'autre part).

Cette relation entre les parties permet alors de réduire considérablement le nombre de comparaisons entre les tuples. Après avoir comparé un tuple au pivot, il est associé à sa classe (sa partie) et ne sera comparé qu'à ceux de sa partie et ceux des parties entretenant une relation de *dominance partielle* avec la sienne. Le processus peut également se poursuivre de manière récursive au sein des parties.

Des algorithmes en parallèle ont également été proposés, par exemple Chester *et al.* [2015]. Certaines propositions considèrent des distributions particulières des données, par exemple, Morse *et al.* [2007b] propose un algorithme très efficace dans le cas de dimensions présentant un nombre très faible de valeurs distinctes, tandis que Shang et Kitsuregawa [2013] présente un algorithme ayant de bonnes performances lorsque les données sont anti-corrélées. D'autres extensions introduisent un caractère probabiliste Pei *et al.* [2007] ou incertain Groz et Milo [2015] des données. Liu *et al.* [2015] étend le concept de dominance aux groupes de tuples et propose une technique d'extraction de ces groupes skyline. D'autres travaux considèrent le problème de réduction de la taille du résultat d'une requête skyline en ne conservant que les tuples satisfaisant certaines contraintes, par exemple, les tuples appartenant à un nombre minimum de skyline de sous-espaces Chan *et al.* [2006b] ou en sélectionnant les points skyline qui dominent un nombre maximal d'autres points Lin *et al.* [2007].

Les Skycuboids et le Skycube Partant du même exemple (table 1.1), considérons à présent, un riche touriste qui ne se soucie pas de la dépense. Celui-ci chercherait les meilleurs chambres prenant uniquement en considération la distance et la surface. Dans ce cas, r_2 est meilleure que r_1 et r_3, donc, le skyline de la table T, au regard de D et S est $\{r_2\}$. Nous aurions également pu considérer un autre profil d'utilisateur moins nanti (un jeune étudiant par exemple), qui ne se focaliserait que sur le prix pour choisir sa chambre d'hôtel ; les chambres qui attireraient son attention (skyline au regard du prix) seraient alors $\{r_1, r_3\}$. Cet exemple montre que, en fonction de l'ensemble d'attributs considérés (les préférences de l'utilisateur), nous obtenons différents résultats pour le choix des *meilleures* chambres d'hôtel. Le résultat d'une requête skyline sur un sous-ensemble de critères (par exemple $\{P\}$, $\{D\}$ ou $\{P, S\}$) est appelé *skycuboid*. Si d est le nombre de critères de notre table, alors il existe $2^d - 1$ skycuboids au total ; ce nombre correspond au cardinal de l'ensemble des parties d'un ensemble possédant d éléments (les critères), ôté de l'élément vide. L'ensemble formé de tous les skycuboids, c'est-à-dire, l'ensemble des requêtes susceptibles d'être posées par les utilisateurs, est appelé le *skycube* Pei *et al.* [2005]; Yuan *et al.* [2005]. Le tableau 1.2 présente les skycuboids relatifs à notre exemple des hôtels.

On pourrait préférer une représentation sous forme de treillis (Figure 1.1), mar-

Ensemble de critères	Skycuboid correspondant
$\{P\}$	$\{r_1, r_3\}$
$\{D\}$	$\{r_2\}$
$\{S\}$	$\{r_1, r_2\}$
$\{P, D\}$	$\{r_1, r_2\}$
$\{P, S\}$	$\{r_1\}$
$\{D, S\}$	$\{r_2\}$
$\{P, D, S\}$	$\{r_1, r_2\}$

TABLE 1.2 – Les skycuboids formant le skycube (table des hôtels)

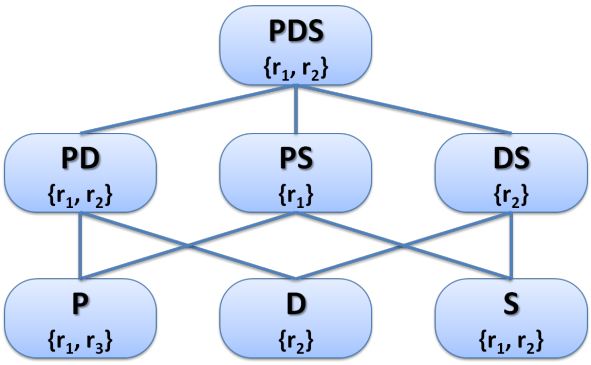


FIGURE 1.1 – Skycube représenté sous forme de treillis

quant en plus la relation d'inclusion entre les espaces.

Skyline et monotonie Nous observons qu'il n'y a pas une relation d'inclusion entre les skycuboids induite par l'inclusion entre les espaces : bien que nous ayons l'inclusion $\{P\} \subseteq \{P, D, S\}$, le skyline au regard de P n'est pas inclus dans celui de $\{P, D, S\}$ et réciproquement (c'est-à-dire le skyline au regard de l'espace $\{P, D, S\}$ n'est pas non plus inclus dans celui de P). Cette *non monotonie* des requêtes skyline est également pointée du doigt dans de précédents travaux Pei *et al.* [2006]; Xia *et al.* [2012]; Pei *et al.* [2005]; Yuan *et al.* [2005], ce qui rend l'optimisation de cette requête plus difficile que l'agrégation de requêtes dans le contexte des cubes de données. Plus précisement, nous ne pouvons pas calculer le skyline au regard de P à partir de celui obtenu au regard de l'espace $\{P, D, S\}$ (et vice versa) sans avoir à accéder à la table de données initiale.

Notations Nous introduisons plus formellement dans cette section, les principaux concepts et notations qui seront utilisés tout le long de ce document. Soit T une table dont l'ensemble d'attributs $Att(T)$ est composé de deux sous-ensembles $\mathcal{D}$ et

$Att(T) \setminus \mathcal{D}$. $\mathcal{D}$ est l'ensemble d'attributs (dimensions) qui peuvent être utilisés pour classer les tuples. Dans la littérature skyline, $\mathcal{D}$ est appelé un *espace multidimensionnel*. Si $X \subseteq \mathcal{D}$, alors X est un *sous-espace*.

$t[X]$ denote la projection du tuple t sur le sous-espace X. On note d le nombre de dimensions. Sans nuire à la généralité, pour chaque $D_i \in \mathcal{D}$, nous supposons qu'il existe un ordre partiel $<$ au sein des valeurs du domaine de D_i.

On dira que t' domine t au regard de l'espace X, ou que t' X-domine t, noté $t' \prec_X t$, si et seulement si pour tout $X_i \in X$, nous avons $t'[X_i] \leq t[X_i]$ et il existe $X_j \in X$ tel que $t'[X_j] < t[X_j]$.

Le skyline de T relativement à $X \subseteq \mathcal{D}$ est défini comme $Sky(T, X) = \{t \in T | \nexists t' \in T$ tel que $t' \prec_X t\}$. Dans le but de simplifier la notation, et lorsque T est compris comme étant la base de référence, il arrive d'omettre T et nous écrivons $Sky(X)$ à la place de $Sky(T, X)$.

Soient n le nombre de tuples appartenant à T (la taille de T) et k_j le nombre de valeurs distinctes de la dimension D_j, c'est-à-dire, $k_j = |\pi_{D_j}(T)|$. k_j est aussi appelé la cardinalité de la dimension D_j.

Le *skycube* de T, noté $\mathcal{S}(T)$ ou plus simplement $\mathcal{S}$ est l'ensemble de tous les skylines $Sky(T, X)$ où $X \subseteq \mathcal{D}$ et $X \neq \emptyset$. Formellement, $\mathcal{S}(T) = \{Sky(T, X) | X \subseteq \mathcal{D}$ et $X \neq \emptyset\}$. Tout skyline $Sky(T, X)$ est appelé un *skycuboid*. Il existe $2^d - 1$ skycuboids dans $\mathcal{S}(T)$. S est un *sous-skycube* du skycube $\mathcal{S}$ si $S \subseteq \mathcal{S}$.

La table 1.3 résume les différentes notations utilisées tout le long de ce document.

Motivations Mise à part la question de l'optimisation du calcul de la requête skyline, plusieurs autres problématiques ont été développées autour de l'opérateur skyline, parmi lesquelles on retrouve :

❶ **l'optimisation des requêtes du skycube (les skycuboids)**. En raison du fait que l'opérateur skyline ne présente pas la propriété de monotonie, l'optimisation (en temps et en espace) des requêtes du skycube est un challenge qui a motivé plusieurs travaux de la littérature. En effet, deux reflexions extrêmes peuvent être formulées en guise de solution à ce problème. D'une part, calculer à la demande chaque requête skyline en utilisant un des algorithmes de l'état de l'art ; cette opération possède dans le pire des cas, une complexité de l'ordre de $O(n^2)$ ce qui est loin d'être satisfaisant. D'autre part, pré-calculer l'ensemble du skycube, approche abordée par certains travaux de la littérature (Lee et won Hwang [2014]; Raïssi *et al.* [2010]). Bien que cette dernière approche présente l'avantage de fournir les résultats des requêtes de manière instantanée, une fois que le pré-calcul a été effectué, elle souffre néanmoins d'une part, de la complexité en temps de la phase de pré-calcul, car aucune garantie n'est formulée quant à la nature de la complexité des algorithmes. En effet, ces approches ont des complexité de l'ordre de $O(2^d n^2)$. D'autre part, l'espace du skycube construit et à stocker peut être grand, de l'ordre de $O(2^d n)$. En guise de compromis, un ensemble de tech-

Notation	Définition
$T, \mathcal{U}$	tables de données, instances de relation, ensembles de tuples
$\mathcal{D}$	l'ensemble d'attributs utilisés pour les requêtes
d	$\lvert\mathcal{D}\rvert$, nombre de dimensions
D_j, A_j	j^{eme} dimension
n, m	nombre de tuples
$X, Y \ldots$	sous-ensembles de dimensions, sous-espaces
XY	$X \cup Y$
$\lvert\lvert X\rvert\rvert$	nombre d'attributs de X
k	nombre de valeurs distinctes par dimension
t, t', t_i, u_j, v	des tuples
$t[X]$	projection du tuple t sur X
$\pi_X(T)$	projection de T sur X
$\lvert X\rvert$	la cardinalité de $\pi_X(T)$
$t_1 \prec_X t_2$	$t_1[X]$ domine $t_2[X]$ ou t_1 X-domine t_2
$t' \prec t$	t' domine t
$t' \not\prec t$	t' ne domine pas t
$Sky(T, X)$, $Sky_T(X)$ ou simplement $Sky(X)$	le skyline de T relativement à X
$Topmost$	$Sky_T(\mathcal{D})$, le skyline de T au regard de $\mathcal{D}$
$\lvert Sky(X)\rvert$	la taille de $Sky(X)$ en nombre de tuples
$\mathcal{S}(T)$ ou simplement $\mathcal{S}$	skycube de T
S	un sous-skycube de $\mathcal{S}$
$\arg\min_{\alpha \in Ens} Expr(\alpha)$	l'élément $\alpha \in Ens$ minimisant l'expression $Expr()$
$\arg\max_{\alpha \in Ens} Expr(\alpha)$	l'élément $\alpha \in Ens$ maximisant l'expression $Expr()$

TABLE 1.3 – Notations

niques ont été développées (Xia *et al.* [2012]; Bøgh *et al.* [2014]; Pei *et al.* [2005]), pré-calculant, en un temps (en théorie) inférieur à celui du calcul du skycube, une structure de données intermédiaire permettant de répondre, non plus de manière instantanée, mais moyennant un effort supplémentaire, aux requêtes posées.

❷ **l'estimation de la taille du résultat d'une requête skyline**. Il s'agit de fournir, avec une grandeur de complexité inférieure à celle du calcul exact, $O(n^2)$, une estimation de la taille du skyline. Les solutions à ce problème présentent plusieurs cas d'application ; partant de l'optimisation de requêtes plus ou moins complexes au sein de systèmes de gestion de bases de données (SGBD), à la sélection de meilleures vues (skycuboids) pour la matérialisa-

tion partielle du skycube. Plusieurs travaux ont traité cette problématique, faisant des hypothèses différentes sur les données. Certains (Bentley *et al.* [1978]; Buchta [1989]; Godfrey *et al.* [2004]) considèrent les dimensions indépendantes et les valeurs toutes distinctes par dimension et fournissent des formules asymptotiques estimant la taille du skyline. D'autres travaux (Chaudhuri *et al.* [2006]; Zhang *et al.* [2009]; Luo *et al.* [2012]), se passent de ces hypothèses et formulent des algorithmes d'estimation de la taille. Leurs inconvénients étant soit la complexité en temps des calculs (pas toujours meilleure que celle du calcul exact) ou encore la précision peu satisfaisante du résultat, par exemple l'algorithme LS (Log Sampling) de Chaudhuri *et al.* [2006].

❸ **l'optimisation des requêtes k-dominant skyline**. Étant donné qu'une requête skyline peut retourner l'ensemble des tuples de la table comme résultat (lorsque tous les tuples sont incomparables entre eux), des extensions ont été développées pour réduire la taille du skyline. Extensions parmi lesquelles on retrouve la requête k-dominant skyline. Elle se distingue de la requête skyline par le fait que, dans le résultat, les tuples, en plus de ne pas être dominés dans l'espace considéré, ne doivent pas être dominés dans tout sous-espace de k dimensions. Les techniques développées jusque là (Chan *et al.* [2006a]; Siddique et Morimoto [2009]; Md. Anisuzzaman et Yasuhiko [2010]; Dong *et al.* [2010]; Siddique et Morimoto [2012]) sont toutes dédiées, soit au calcul du k-dominant skyline à la demande, soit au pré-calcul de l'ensemble des k-dominant skylines en faisant varier le paramètre k. Aucune de ces approches n'opère un compromis (en effectuant un pré-calcul intermédiaire) comme évoqué dans le cas des skycuboids.

❹ **la recherche des groupes skyline**. Une autre variante de requête skyline est la notion de *group-based skyline* de Liu *et al.* [2015]. Il s'agit de déterminer des ensembles de k points qui ne sont pas dominés (pris ensembles). La notion de dominance, propre à un tuple se trouve de ce fait étendue aux k-uplets de tuples. Cette notion trouve son intérêt lorsque l'utilisateur recherche un ensemble de k éléments distincts dans la base de données ; elle devient autant plus pertinente lorsque la taille du skyline classique est petite (par exemple inférieure à k). Si le skyline classique est très grand alors le nombre de k group-based skylines peut être supérieur à la taille du skyline classique (car tout sous ensemble de k éléments du skyline est un résultat potentiel).

❺ **le calcul des skylines/skycuboids en environnement distribué**. Plusieurs travaux (Chester *et al.* [2015]; Afrati *et al.* [2015]; Im *et al.* [2011]; Mullesgaard *et al.* [2014]) ont abordé la question du calcul du skyline en distribué. L'intérêt de cette problématique réside dans les situations où les données sont très volumineuses pour être stockées sur une seule machine. Lorsque l'ensemble des données peut être supporté par une seule machine, il est difficile d'imaginer un scénario où on gagnerait à les répartir sur plusieurs ma-

chines pour améliorer les calculs. En effet, le fait de devoir comparer les tuples entre eux rend le calcul distribué assez coûteux en termes de transferts de données d'une machine à l'autre. Il est alors question de concevoir des procédés de calcul du skyline minimisant ses transferts.

❻ **les requêtes skylines/skycuboids en présence de données dynamiques**. La problématique du calcul du skyline au sein de données dynamiques a été plusieurs fois abordée dans des travaux antérieurs (Wu *et al.* [2007]; Hsueh *et al.* [2008, 2011]), certains de ces travaux (Tao et Papadias [2006]; Morse *et al.* [2007a]; Alexander *et al.* [2016]) formulent des hypothèses simplificatrices sur les données ; par exemple : la suppression des tuples s'effectue dans l'ordre de l'insertion. En revanche le problème des requêtes du skycube en présence de données dynamiques est très peu traité dans la littérature. En effet, à notre connaissance, seul Xia *et al.* [2012] propose la maintenance de la structure de données *Compressed Skycube* pour prendre en considération des insertions et suppressions de tuples dans le cadre de l'optimisation du calcul des skycuboids.

L'objectif de notre travail est d'apporter de nouvelles approches dans la résolution de quelques-uns de ces problèmes. Approches qui, améliorent les performances des traitements (temps, espace) ou des résultats (temps, précision) pour la résolution des questions abordées.

Résumé des contributions et organisation du manuscrit Nous avons abordé quelques questions à savoir l'estimation de la taille du résultat d'une requête skyline, l'optimisation (en temps et en espace) de l'exécution des requêtes skyline et k-dominant skyline et enfin l'exécution des requêtes skyline en présence de données dynamiques.

Dans le cadre du premier point, la difficulté liée à l'estimation de la taille du résultat d'une requête skyline tient du fait que le nombre de tuples dans le résultat n'est pas toujours proportionnel à la taille (nombre de tuples) des données en entrée. En effet, cette valeur peut être égale à 1 (un tuple domine tous les autres) comme elle peut être égale au nombre total de tuples des données (n) dans le cas où ils sont tous incomparables. C'est pourquoi, estimer la taille du skyline est une question cruciale à laquelle plusieurs travaux se sont consacrés ces dernières années (par exemple Bentley *et al.* [1978]; Buchta [1989]; Zhang *et al.* [2009]; Luo *et al.* [2012]). Ce problème est particulièrement important dans l'optique de l'intégration de l'opérateur skyline au sein de systèmes de gestion de bases de données. S'appuyant sur la distribution des valeurs, nous proposons (chapitre 2) deux estimateurs présentant de bonnes propriétés : le premier est sans biais et nécessite un parcours de l'ensemble des données (nous proposons également une variante de cet estimateur ne nécessitant le parcours que d'un échantillon des données), cet estimateur présente une complexité sous quadratique ($O(n \log n)$). Tandis que le second est convergent en loi et repose uniquement sur la distribution connue des valeurs, pour cela il présente une complexité linéaire dans le pire des cas. Ces estimateurs sont

simples à mettre en œuvre et montrent leur précision et leur efficacité à travers les résultats des expérimentations qui ont été faites. Ce travail a donné lieu à la publication Hanusse *et al.* [2016b] à la conférence *International Database Engineering & Applications Symposium* (IDEAS).

Relativement à la seconde problématique énumérée, dans le but d'optimiser l'exécution de chacune des requêtes skyline, les solutions proposées jusque là dans la littérature soit (i) pré-calculent tous les skylines (le skycube) ou alors (ii) font usage des techniques de compression de telle sorte que la déduction d'un skyline soit effectuée moyennant un moindre effort. Même si ces solutions (i) sont attrayantes parce que l'exécution des requêtes skyline requiert un temps optimal, elles souffrent du problème de scalabilité en temps et en espace car le nombre de skylines à matérialiser est exponentiel du nombre de dimensions d. D'autre part, ces solutions (ii) sont attrayantes en terme de consommation mémoire mais, comme montré, elles ont aussi une complexité en temps élevée. Dans ce sens, nous avons développé deux approches complètement différentes pour l'optimisation des requêtes du skycube. Notre but étant de proposer un compromis en temps c'est-à-dire renoncer à l'instantanéité dans le retour du résultat d'une requête, pour réduire le temps de précalcul ; cette approche ayant le mérite, dans certains cas, tout simplement de rendre le problème soluble étant donnés un jeu de données et une unité de calcul.

Notre première approche (chapitre 3) fait appel aux dépendances fonctionnelles, un concept fondamental en base de données. Les dépendances fonctionnelles nous ont permis d'établir une forme de *monotonie* entre les skylines. À partir de cette propriété, nous avons proposé des procédures ayant à dessein la matérialisation partielle ou complète du skycube. Des expérimentations étendues sur des données réelles et synthétiques montrent que nos propositions généralement surpassent les algorithmes de l'état de l'art. Ce travail a donné lieu à une publication Hanusse *et al.* [2016c] au journal *Transactions On Database Systems* (TODS).

Notre seconde approche (chapitre 4) dans l'optimisation des requêtes du skycube s'incrit plutôt dans le cadre de la compression du skycube et part de l'observation selon laquelle les précédents travaux avaient pour objectif de calculer ou de résumer pour chaque tuple t : "la liste des skylines auxquels t appartient" (par exemple Raïssi *et al.* [2010]; Xia *et al.* [2012]). Notre méthode explore alors l'information complémentaire, à savoir, "pour chaque tuple t, la liste des skylines auxquels t *n'appartient pas*". C'est pourquoi nous l'appelons le *Skycube Négatif*. En dépit de l'apparente équivalence entre ces deux informations, nos analyses et expérimentations montrent que ces deux points de vue ne conduisent pas à des comportements similaires des algorithmes associés. Plus spécifiquement, nos travaux montrent que (i) le résumé du *Skycube Négatif* peut être obtenu plus rapidement que les méthodes de l'état de l'art travaillant sur le *Skycube Positif* (complexité de $O\left([min(2^d, n)]^2 n\right)$ dans le pire des cas au lieu de $O(2^d n^2)$), (ii) en général, le *Skycube Négatif* requiert moins d'espace (complexité de $O\left(min(2^d, n)n\right)$ dans le pire des cas au lieu de $O(2^d n)$), (iii) les requêtes sont plus rapides suivant cette optique, (iv) le *Skycube Positif* peut être obtenu plus rapidement en calculant d'abord le *Skycube Négatif* (v)

cette technique peut être exploitée par d'autres types de requêtes notamment les requêtes k-dominant skyline. Ce travail a donné lieu à une publication Hanusse *et al.* [2016a] à la conférence *Conference on Information and Knowledge Management* (CIKM).

Dans le chapitre 5, nous évoquons quelques extensions relatives à la structure de données Skycube Négatif présentée au chapitre précédent. Nous y décrivons par exemple la simplicité avec laquelle cette structure permet également d'exécuter des requêtes k-dominant skyline, autre type de requête dont le skyline est un cas particulier. Nous y abordons également la question de la maintenance de cette structure de données ; en bref il s'agit de montrer comment le Skycube Négatif peut être modifié pour qu'il survive aux insertions/suppressions de tuples.

Chapitre 2

Estimation de la Taille du Skyline

Introduction

Dans ce chapitre, nous analysons le problème de l'estimation de la taille du skyline. Ce problème est de grande importance dans l'éventualité de l'intégration de l'opérateur skyline au sein des systèmes de gestion de bases de données. En effet, l'optimisation de requêtes requiert parfois la connaissance d'une estimation de la taille des résultats intermédiaires afin de proposer les meilleurs plans d'exécution ; cette idée est mise en exergue dans Chaudhuri [1998]. Par exemple, estimer le nombre de tuples retournés par une requête de sélection peut être possible si on connait le nombre de valeurs distinctes. Cette estimation peut être plus précise si nous connaissons également la distribution de ces valeurs (l'histogramme des valeurs). De la même façon, nous prenons en considération la connaissance de la distribution des données dont on veut estimer la taille du skyline. En outre, certains travaux reposent sur l'estimation de la taille du skyline. Par exemple, Xia *et al.* [2012] traite de l'optimisation des requêtes skyline multidimensionnelles dans le cadre du skycube. Les auteurs proposent une solution pour la sélection du meilleur ensemble de *skycuboids* à matérialiser pour optimiser les autres requêtes skyline. Cette solution suppose que les tailles des $2^d - 1$ skylines sont connues c'est-à-dire estimées avec précision.

Au regard des solutions proposées dans la littérature, pour l'estimation de la taille du skyline, nous apportons deux contributions majeures. D'une part, nous améliorons la complexité des calculs dans ce sens que le calcul ne dépend pas de la taille réelle du skyline comme c'est le cas pour certains travaux dont nous parlerons dans la partie 2.1. D'autre part, nos propositions ne reposent pas sur des calculs complexes tels que les méthodes à noyaux (Zhang *et al.* [2009]), le calcul de skylines d'échantillons (Chaudhuri *et al.* [2006]; Zhang *et al.* [2009]; Luo *et al.* [2012]) ou encore le calcul de multiples intégrales (Chaudhuri *et al.* [2006]) ; ce qui en facilite l'implémentation. Par exemple, lorsque les données sont parcourues pour estimer la taille du skyline, notre proposition consiste en le calcul d'une simple somme de produits de termes.

Organisation du chapitre et résumé des contributions Dans la section suivante, nous résumons quelques travaux précédents relatifs à l'estimation de la cardinalité du skyline. Ensuite, nous présentons nos propositions reposant sur la connaissance, mise à disposition de la distribution des valeurs des données. Nous faisons l'hypothèse d'indépendance entre les dimensions mais considérons que les valeurs peuvent se répéter au sein des dimensions. Nos estimateurs ($\hat{S}_W$, $\hat{S}_S$ et $\hat{S}_E$) se distinguent de ceux de l'état de l'art, à savoir *Purely Sampling* (PS) de Luo *et al.* [2012], *Log Sampling* (LS) de Chaudhuri *et al.* [2006] et Kernel-Based (KB) de Zhang *et al.* [2009], dans ce sens qu'ils permettent de réduire la complexité des calculs (complexité sous quadratique donc meilleure) tout en maintenant le niveau de précision dans les résultats. Dans la dernière section, sont présentées les évaluations expérimentales, montrant la précision de nos propositions et confirmant leur rapidité d'exécution ; par exemple, l'algorithme S_S est 1000 fois plus rapide que PS sur un jeu de données présentant des dimensions indépendantes, 10 millions de tuples, 15 dimensions et les valeurs distinctes au sein des dimensions.

2.1 Travaux relatifs

Plusieurs algorithmes ont été proposés pour calculer les requêtes skyline, en guise d'exemple, BNL et DC de Börzsönyi *et al.* [2001], SFS de Chomicki *et al.* [2003], LESS de Godfrey *et al.* [2005], LS de Morse *et al.* [2007b], SaLSa de Bartolini *et al.* [2008], ou encore BSkyTree de Lee et won Hwang [2010]. Bien que ces algorithmes reposent sur différents principes, ils possèdent tous une complexité dans le pire des cas de l'ordre de $\mathcal{O}(n^2)$. Donc, afin de présenter un intérêt pratique, tout algorithme dédié à l'estimation de la taille du skyline devrait avoir une complexité moindre. Nous évoquons alors quelques précédents travaux à ce sujet.

Les premiers travaux à ce sujet considèrent que les dimensions sont indépendantes et que les valeurs apparaissant dans chacune d'elles sont distinctes. Dans ce sens, Bentley *et al.* [1978] montre que l'espérance de la taille du skyline est de l'ordre de $\mathcal{O}\left((\ln n)^{d-1}\right)$. Dans Buchta [1989], la formule précédente a été améliorée en montrant que le nombre de vecteurs maximum est de l'ordre de $\Theta\left(\frac{(\ln n)^{d-1}}{(d-1)!}\right)$. L'avantage de ces résultats est de donner une idée asymptotique de la taille du skyline mais ils restent très imprécis en pratique. Sous les mêmes hypothèses, Godfrey *et al.* [2004] montre que la taille du skyline peut être estimée par le $(d+1)^{iem}$ ordre de la série harmonique en n, $H_{d+1,n} = \sum_{i=1}^{n} \frac{H_{d,i}}{i}$ où $H_{0,n} = 1$ et d est le nombre de dimensions.

D'autres travaux ce sont attardés sur la même problématique, relaxant cette fois les deux hypothèses faites (indépendance entre les dimensions et valeurs distinctes par dimension).

Chaudhuri *et al.* [2006] présente alors des formules analytiques donnant l'espérance exacte de la taille du skyline. Même si cela semble attrayant, ces formules

contiennent d intégrales simples ou une intégrale sur un espace à d dimensions ce qui les rend difficiles à implémenter. Plus précisement, la probabilité qu'un tuple appartienne au skyline est donnée par [1] $p = \int_{[0,1]^d} f(x).(1 - F(x))^{n-1}dx$ où $f(x)$ est la fonction de densité jointe des valeurs des tuples et $F(x)$ est la fonction de densité cumulative correspondant à f. La taille du skyline est alors estimée par $n \times p$. Les mêmes auteurs (Chaudhuri *et al.* [2006]) présentent une autre approche pour cette estimation faisant l'hypothèse que la taille du skyline suit le modèle $s = A\,(\log n)^B$, A et B étant deux paramètres à estimer. Le calcul s'effectue alors en calculant la taille exacte du skyline au sein de deux échantillons de tailles n_1 et n_2. Nous obtenons alors deux équations à deux inconnues (A et B). La résolution de ce système d'équations donne le modèle de la taille du skyline et en spécifiant la taille des données, on obtient une estimation de la taille du skyline.

Zhang *et al.* [2009] propose une approche basée sur l'estimation par noyau de la fonction de densité des tuples. L'association de cette méthode au calcul exact de la taille du skyline d'un échantillon des données, permet d'obtenir une estimation de la probabilité p qu'un tuple appartienne au skyline et de ce fait d'estimer tout comme Chaudhuri *et al.* [2006] la taille du skyline. En dépit de ces avantages, c'est-à-dire aucune hypothèse faite sur la nature des données, cette méthode souffre de deux problèmes : (i) elle requiert le calcul du skyline d'un échantillon, ce qui peut être couteux (la complexité en temps de l'estimation reste quadratique comme celle du calcul exact) ; (ii) intuitivement, les méthodes à noyau font un usage intensif du calcul de distances afin de trouver les plus proches voisins. Lorsque le nombre de dimensions augmente, il est connu, en raison du *fléau de la dimensionnalité*, que les tuples ont tendance à être difficilement distinguables les uns des autres du point de vue de la distance. Ce qui nécessitera alors des échantillons de grande taille pour garantir une moindre précision des résultats.

Comme la méthode basée sur le noyau, Luo *et al.* [2012] propose une méthode non paramétrique. Leur approche consiste à calculer le skyline d'un échantillon de données que l'on appelera le skyline intermédiaire. Ensuite, les données d'origine sont parcourues dans le but d'éliminer du skyline intermédiaire les tuples qui sont dominés par un tuple de l'ensemble des données. Si m est le nombre de tuples restants et s la taille de l'échantillon, alors $p = \frac{m}{s}$ est une estimation de la probabilité qu'un tuple appartienne au skyline. Une fois encore, la taille du skyline global est estimée par $n \times p$. Le principal inconvénient de cette approche est le fait que, bien qu'il s'agisse d'une approche basée sur l'échantillonnage, l'ensemble des données est requis et il est possible que chaque point soit comparé à tous les autres ce qui, dans ce cas est plus coûteux que de calculer directement le skyline et obtenir sa taille.

1. Sans nuire à la généralité, les valeurs des données sont considérées appartenant à l'intervalle $[0, 1]$.

Notation	Définition
$\mathcal{T}$	l'ensemble de tous les tuples possibles
k_j	cardinalité de la j^{ieme} dimension
$f_j(x)$	la fonction de distribution de la j^{ieme} dimension
$F_j(x)$	la fonction de distribution cumulative de la j^{ieme} dimension
$Prob(\mathcal{E})$	la probabilité que l'événement $\mathcal{E}$ se produise
$P^{\prec}(v)$	la probabilité que v soit dominé par un tuple aléatoire de $\mathcal{T}$
q_{t_ℓ}	la probabilité que le tuple t_ℓ appartienne à T
p_{t_ℓ}	la probabilité que le tuple $t_\ell \in T$ appartienne au skyline $Sky(\mathcal{D})$
$E()$	la fonction *espérance mathématique*
$\hat{P}$	estimation de la grandeur P

TABLE 2.1 – Notations

2.2 Estimation de la taille du skyline

Dans cette section, nous présentons nos principales contributions. Nous formulons également l'hypothèse d'indépendance entre les dimensions, mais pas celle des valeurs distinctes au sein des dimensions. Afin de simplifier la compréhension de nos procédés, nous commençons par définir quelques concepts liés à nos approches.

Définition des concepts et notations

En plus des notations présentées (table 1.3) en introduction du document, nous présentons (voir table 2.1) quelques concepts et notations spécifiques à ce chapitre.

Exemple 1. Le tableau 2.2 présente le jeu de données que nous prendrons comme exemple dans ce chapitre. De ces données, l'utilisateur cherche les meilleurs hôtels au regard des différents critères apparaissant dans le tableau. Ces critères sont le prix de la chambre d'hôtel, la distance jusqu'à l'arrêt bus, la surface de la chambre et le classement de l'hôtel en termes de nombre d'étoiles.

L'hôtel h_{12} ne fait pas partie du skyline parce qu'il est dominé par l'hôtel h_6. En effet, l'hôtel h_6 est meilleur que l'hôtel h_{12} en termes de surface (40 pour h_6 contre 20 pour h_{12}) et de classement (l'hôtel h_6 est un cinq-étoiles tandis que l'hôtel h_{12} est un trois-étoiles) ; ces deux hôtels sont néanmoins équivalents au regard des autres attributs (à savoir le prix et la distance). Dans ce cas, k_j vaut 3 pour tous les attributs D_j, j allant de 1 à 4. L'ensemble skyline des hôtels au regard de toutes les dimensions est constitué de h_4, h_6 et h_{10}. Donc la taille exacte du skyline est de 3 ($|Sky(T)| = 3$).

Notons que l'utilisateur cherche à minimiser le prix tandis qu'il veut maximiser la surface. Maximiser la surface équivaut à minimiser la différence entre la surface maximale et la surface de la chambre d'hôtel courante. L'ensemble peut être transformé en un problème de minimisation de chacun des critères en remplaçant chaque

Hôtels	Prix	Dist.	Surf.	Clas.
$h1$	150	100	10	****
$h2$	150	1000	10	***
$h3$	150	100	20	****
$h4$	40	100	40	***
$h5$	300	100	10	*****
$h6$	150	5000	40	*****
$h7$	40	100	10	***
$h8$	300	1000	10	***
$h9$	150	100	20	*****
$h10$	40	100	20	*****
$h11$	150	1000	20	****
$h12$	150	5000	20	***

TABLE 2.2 – Les hôtels : données brutes

valeur par son *rang* dans le tri obtenu en utilisant la relation d'ordre $<_j$ comme critère de tri. De cet exemple, le rang de la valeur 40 dans la dimension surface est 1 (préférence pour les valeurs élevées). La valeur 40 a également le rang 1 pour l'attribut prix mais cette fois parce qu'il s'agit de la plus petite valeur (préférence pour les valeurs faibles en ce qui concerne le prix). Le fait de remplacer les valeurs par leur rang selon l'ordre naturel des préférences pour chacune des dimensions ne change pas l'ensemble skyline. En guise d'illustration, le tableau 2.3 présente la normalisation (transformation) du tableau 2.2 suivant ce principe.

Donc, sans nuire à la généralité et par simplicité de présentation, nous considèrerons désormais une préférence pour les valeurs faibles.

Nous rappelons la définition de quelques notions statistiques sur lesquelles repose la compréhension de ce travail.

Définition 2.1 (Indépendance)**.** Soient X et Y deux variables aléatoires. X et Y sont dites indépendantes si et seulement si le processus de génération de X est indépendant de celui de Y.

Définition 2.2 (Espérance)**.** Soit X une variable aléatoire. On suppose que X peut prendre la valeur x_1 avec la probabilité p_1, la valeur x_2 avec la probabilité p_2, et ainsi de suite jusqu'à x_k avec la probabilité p_k. Alors, l'espérance de cette variable aléatoire X, notée $E(X)$ s'écrit

$$E(X) = x_1 \times p_1 + x_2 \times p_2 + \cdots + x_k \times p_k$$

Définition 2.3 (Estimateur d'Horvitz-Thompson)**.** Horvitz et Thompson [1952] Formellement, soit $\mathcal{T} = \{t_i, i = 1, 2, \ldots, N\}$ la population constituée de N individus distincts, soit $Y(t_i)$ la valeur observée du tuple t_i pour la variable aléatoire Y.

T	D_1	D_2	D_3	D_4
$h1$	2	1	3	2
$h2$	2	2	3	3
$h3$	2	1	2	2
$h4$	1	1	1	3
$h5$	3	1	3	1
$h6$	2	3	1	1
$h7$	1	1	3	3
$h8$	3	2	3	3
$h9$	2	1	2	1
$h10$	1	1	2	1
$h11$	2	2	2	2
$h12$	2	3	2	3

TABLE 2.3 – Les hôtels : données *normalisées*

Soit $\mathcal{S}$ l'échantillon indépendant tiré avec remise de $\mathcal{T}$ tel que $|\mathcal{S}| = n$. On suppose en plus que π_i est la probabilité qu'un tuple $t_i \in \mathcal{T}$ appartienne à l'échantillon $\mathcal{S}$, $\pi_i = Prob(t_i \in \mathcal{S})$. L'estimateur de Horvitz-Thompson $\hat{Y}_{HT}$ du total $\sum_{t_i \in \mathcal{T}} Y(t_i)$ est donné par

$$\hat{Y}_{HT} = \sum_{t_i \in S} \pi_i^{-1} Y(t_i)$$

Définition 2.4 (Estimateur sans biais)**.** Soit s un paramètre d'une population et soit $\hat{s}$ un estimateur de s. $\hat{s}$ est qualifié d'estimateur sans biais de s si la différence entre l'espérance de l'estimateur et la valeur exacte du paramètre estimé est égale à zero ; formellement, $E(\hat{s}) - s = 0$.

Définition 2.5 (Théorème Central Limite)**.** Soit $\{U_1, \ldots, U_d\}$ un échantillon aléatoire de taille d. C'est une séquence de variables aléatoires indépendantes et identiquement distribuées suivant une distribution d'espérance μ_U et de variance finie notée σ_U^2. Lorsque $d \to \infty$, la distribution de $Y = \sum_{j=1}^{d} U_j$ converge vers la distribution normale $\mathcal{N}(d \times \mu_U, d \times \sigma_U^2)$.

Nous commençons par la méthode qui requiert l'accès aux données, que ce soit le parcours de l'ensemble des données ou juste celui d'un échantillon pour ce faire.

2.2.1 Parcours des données

Accéder à toutes les données

Nous montrons ici comment, un parcours des données permet de calculer une statistique estimant sans biais la taille du skyline. Il est nécessaire de mentionner que

x	$F_1(x)$	$F_2(x)$	$F_3(x)$	$F_4(x)$
1	1/4	7/12	1/6	1/3
2	5/6	5/6	7/12	7/12
3	1	1	1	1

TABLE 2.4 – Les fonctions de densité cumulée

cette procédure possède une complexité de l'ordre de $\mathcal{O}(n \log n)$. Nous présentons tout d'abord quelques notations.

Considérons les tuples de T comme étant des lignes d'une matrice, i.e., le i^{iem} tuple t_i de T est égal à $\langle T[i,1], T[i,2], \ldots, T[i,d] \rangle$. Soient t_j et t_i tels que $t_j \prec t_i$. Ceci signifie que $t_i[\ell] \leq t_j[\ell]$ pour $1 \leq \ell \leq d$. Étant donné que nous considérons les données normalisées, le domaine de chaque dimension D_ℓ, noté Dom_ℓ, est égal à $\{1, 2, \ldots, k_\ell\}$ où k_ℓ est le nombre de valeurs distinctes de la dimension D_ℓ. Soit $c \in Dom_\ell$ et soit t un tuple aléatoire de T. Notons $f_\ell(c) = Prob(t[\ell] = c)$ et $F_\ell(c) = Prob(t[\ell] \leq c)$ respectivement la probabilité que $t[\ell] = c$ et la probabilité que $t[\ell] \leq c$. $f_\ell(x)$ est la fonction de distribution des valeurs de D_ℓ et $F_\ell(x)$ est la fonction de distribution cumulée associée à f_ℓ.

Exemple 2. De notre exemple courant, les fonctions de distribution cumulées des différentes valeurs des attributs sont présentées dans le tableau 2.4.

On désigne par $\mathcal{T}$ l'ensemble des tuples possibles en considérant l'ensemble des valeurs possibles de chaque dimension. Soit $v \in \mathcal{T}$ un tuple aléatoire et soit $P^{\prec}(v)$ la notation représentant la probabilité que v soit dominé par un tuple aléatoire suivant la loi décrite par les distributions F_ℓ. Alors $P^{\prec}(v)$ est donné par le résultat suivant.

Lemme 2.1. *Soit $v \in \mathcal{T}$ et soit t un tuple choisi de façon aléatoire et suivant la loi décrite par les distributions F_ℓ. Alors,*

$$Prob(t \prec v) = P^{\prec}(v) = \prod_{\ell=1}^{d} F_\ell(v[\ell]) - \prod_{\ell=1}^{d} f_\ell(v[\ell])$$

Démonstration. $t \prec v \Leftrightarrow t[\ell] \leq v[\ell]$ pour tout $1 \leq \ell \leq d$. Puisque $Prob(t[\ell] \leq v[\ell]) = F_\ell(v[\ell])$, et les dimensions sont indépendantes, alors $Prob(t \prec v \vee t = v) = \prod_{\ell=1}^{d} F_\ell(v[\ell])$. D'autre part, $Prob(t = v) = \prod_{\ell=1}^{d} f_\ell(v[\ell])$, donc

$$\begin{aligned} Prob(t \prec v) &= Prob(t \prec v \vee t = v) - Prob(t = v) \\ &= \prod_{\ell=1}^{d} F_\ell(v[\ell]) - \prod_{\ell=1}^{d} f_\ell(v[\ell]) \end{aligned} \tag{2.1}$$

□

Lemme 2.2. *Soit $v \in \mathcal{T}$ un tuple et soit $\mathcal{U} = \{u_1, u_2, \ldots, u_m\}$ un ensemble de m tuples indépendants et choisis aléatoirement et suivant les distributions F_ℓ, alors, la probabilité $Prob(u_i \not\prec v, \forall u_i \in \mathcal{U})$ que v ne soit dominé par aucun tuple appartenant à $\mathcal{U}$ est donnée par $(1 - P^{\prec}(v))^m$.*

Démonstration. À partir du lemme précédent, $Prob(u_i \prec v) = P^{\prec}(v)$, $\forall u_i \in \mathcal{U}$, alors $Prob(u_i \not\prec v) = 1 - P^{\prec}(v)$. En raison de l'indépendance des u_i, la probabilité que v ne soit dominé par aucun $u_i \in \mathcal{U}$ peut être écrite comme suit

$$\begin{aligned} Prob(u_i \not\prec v, \forall u_i \in \mathcal{U}) &= Prob(u_1 \not\prec v, \wedge u_2 \not\prec v \cdots \wedge u_m \not\prec v) \\ &= \prod_{u_i \in \mathcal{U}} Prob(u_i \not\prec v) \\ &= \prod_{u_i \in \mathcal{U}} (1 - P^{\prec}(v)) \\ &= (1 - P^{\prec}(v))^m \end{aligned}$$

□

Du lemme précédent, nous déduisons la probabilité qu'un tuple de T appartienne à son skyline.

Lemme 2.3. *Soit T un ensemble de tuples tel que $|T| = n$ et soit $t \in T$. Alors $Prob(t \in Sky(T, \mathcal{D})) = (1 - P^{\prec}(t))^{n-1}$*

Démonstration. Pour que t appartienne au skyline, il est nécessaire qu'il ne soit dominé par aucun des $n - 1$ autres tuples. Chacun de ces derniers possède une probabilité $P^{\prec}(t)$ de dominer t, par conséquent une probabilité égale à $1 - P^{\prec}(t)$ de ne pas dominer t. La probabilité que tous les $n - 1$ tuples satisfassent cette condition est égale au produit des probabilités que chacun d'eux la satisfasse ; ce qui prouve le lemme. □

À présent, nous pouvons construire notre premier estimateur de la taille du skyline. Celui-ci dépend de la probabilité pour un tuple d'appartenir au skyline et de la taille de T.

Proposition 1.

$$\hat{S}_W = \sum_{i=1}^{n} \left(1 - \prod_{\ell=1}^{d} F_\ell(t_i[\ell]) + \prod_{\ell=1}^{d} f_\ell(t_i[\ell]) \right)^{n-1}$$

est un estimateur sans biais de l'espérance de la taille du skyline d'une table contenant n tuples de d attributs dont les valeurs suivent les distributions cumulées F_l.

Démonstration. Nous devons montrer que $\hat{S}_W$ est égal à l'estimateur de Horvitz-Thompson (Définition 2.3) qui est connu pour être sans biais (Horvitz et Thompson

[1952]). Ici, la population est l'ensemble des tuples possibles $\mathcal{T}$ tandis que l'échantillon de tuples est l'ensemble des tuples $t \in T$. La variable observée est pour un tuple $t \in \mathcal{T}$, la probabilité pour celui-ci d'appartenir au skyline. Pour qu'un tuple $t \in \mathcal{T}$ appartienne au skyline, il doit tout d'abord faire partie des tuples présents au sein de la table T et ensuite faire partie du skyline de T. Soient q_t et p_t représentant respectivement ces probabilités. Alors $Prob(t \in Sky(T, \mathcal{D}))$ est égale à

$$Prob(t \in T) \times Prob(t \in Sky(T, \mathcal{D}) | t \in T) = q_t \times p_t$$

L'espérance de la variable aléatoire $|Sky(T, \mathcal{D})|$ est égale à $E(|Sky(T, \mathcal{D})|) = \sum_{t \in \mathcal{T}} q_t \times p_t$. Alors, l'estimateur de Horvitz-Thompson de $E(|Sky(T, \mathcal{D})|)$ (dans le cas de la somme) est donné par $\hat{E}(|Sky(T, \mathcal{D})|)_{HT} = \sum_{i=1}^{n} \pi_i^{-1} q_{t_i} \times p_{t_i}$ où π_i est la probilité que le tuple t_i appartienne à l'échantillon. Dans notre cas, l'échantillon est T d'où $\pi_i = q_{t_i}$. Alors,

$$\begin{aligned}
\hat{E}(|Sky(T, \mathcal{D})|)_{HT} &= \sum_{i=1}^{n} q_{t_i}^{-1} \times q_{t_i} \times p_{t_i} \\
&= \sum_{i=1}^{n} p_{t_i} \\
&= \sum_{i=1}^{n} \left(1 - \prod_{\ell=1}^{d} F_\ell(t_i[\ell]) + \prod_{\ell=1}^{d} f_\ell(t_i[\ell]) \right)^{n-1} \\
&= \hat{S}_W
\end{aligned}$$

□

L'algorithme 1 décrit comment implémenter cette estimation.

Exemple 3. En appliquant cet algorithme à notre exemple courant et en utilisant les fonctions de densité cumulative résumées dans le tableau 2.4, nous obtenons $\hat{S}_W = 3,67$ tandis que la taille exacte du skyline est de 3.

Nous concluons cette section en montrant que $\hat{S}_W$ peut être obtenu en $\mathcal{O}(n \log n)$. En effet, calculer les fonctions de distribution cumulative requiert un simple parcours des données. Le facteur $\log n$ provient du fait d'élever la probabilité $(1 - p)$, de ne pas être dominé par un tuple, à la puissance $n - 1$. Ensuite, nous effectuons la somme de n termes. Il est nécessaire de rappeler que le calcul du skyline est de l'ordre de $\mathcal{O}(n^2)$.

Échantillonnage

Lorsque les données sont volumineuses, il est possible que, même une estimation de la taille du skyline en $\mathcal{O}(n \log n)$ soit considérée comme chronophage. De ce

Algorithme 1 : S_W, estimation de la taille du skyline

Input :
- number of tuples : n
- number of attributes : d
- data table : $T[1 \dots n, 1 \dots d]$

Output : Skyline estimation : S_W

```
1 begin
2     for each ℓ in 1 ... d do
3         Compute f_ℓ from T
4         Compute F_ℓ from f_ℓ
5     S_W ← 0
6     for each i in 1 ... n do
7         S_W ← S_W + (1 − ∏_{ℓ=1}^{d} F_ℓ(T[i, ℓ]) + ∏_{ℓ=1}^{d} f_ℓ(T[i, ℓ]))^{n−1}
8     return S_W
```

fait, il serait intéressant, au lieu de mobiliser l'ensemble des données, que juste un échantillon puisse permettre de fournir une estimation sans biais. Le résultat suivant répond à cette problématique.

Proposition 2. Soit $M \subseteq T$ un échantillon aléatoire de tuples tels que $|M| = m$ et $|T| = n$. Alors

$$\hat{S}_S = \frac{n}{m} \sum_{t_i \in M} \left(1 - \prod_{\ell=1}^{d} F_\ell(t_i[\ell]) + \prod_{\ell=1}^{d} f_\ell(t_i[\ell]) \right)^{n-1}$$

est un estimateur sans biais de la taille du skyline.

Démonstration. La probabilité (π_i) qu'un tuple $t_i \in \mathcal{T}$ appartienne à l'échantillon est égale au produit des probabilités que t_i appartienne à la table T (probabilité notée q_i) et que t_i appartienne à l'échantillon sachant qu'il appartient à la table $\frac{\binom{n-1}{m-1}}{\binom{n}{m}} = \frac{m}{n}$, où $\binom{n}{m}$ est le nombre total d'échantillons possibles constitués de m tuples tirés de la table T, et $\binom{n-1}{m-1}$ est le nombre de ces échantillons dans lesquels t_i apparaît. Alors, $\pi_i = q_i \times \frac{m}{n}$.

L'estimateur de Horvitz-Thompson (Définition 2.3) donnant la taille du skyline est alors

$$
\begin{aligned}
\hat{E}(|Sky(T,\mathcal{D})|)_{HT} &= \sum_{t_i \in M} (\pi_i)^{-1} \times q_{t_i} \times p_{t_i} \\
&= \sum_{t_i \in M} (q_{t_i} \times \frac{m}{n})^{-1} \times q_{t_i} \times p_{t_i} \\
&= \frac{n}{m} \sum_{t_i \in M} p_{t_i} \\
&= \frac{n}{m} \sum_{t_i \in M} \left(1 - \prod_{\ell=1}^{d} F_\ell(t_i[\ell]) + \prod_{\ell=1}^{d} f_\ell(t_i[\ell])\right)^{n-1} \\
&= \hat{S}_S
\end{aligned}
$$

□

En pratique, f et F peuvent respectivement être estimés par f^M et F^M qui représentent la distribution obtenue au sein de l'échantillon M.

L'algorithme 2 montre comment l'estimation de la taille du skyline est déterminée en ne parcourant qu'un échantillon des données.

Algorithme 2 : S_S, estimation de la taille du skyline

Input :
- number of tuples : n
- size of sample : m
- number of attributes : d
- sample : $T[1 \dots m, 1 \dots d]$

Output : Skyline estimation : S_S

```
1 begin
2     for each ℓ in 1 ... d do
3         Compute f_ℓ from T
4         Compute F_ℓ from f_ℓ
5     S_S ← 0
6     for each i in 1 ... m do
7         S_S ← S_S + (1 − Π_{ℓ=1}^{d} F_ℓ(T[i,ℓ]) + Π_{ℓ=1}^{d} f_ℓ(T[i,ℓ]))^{n−1}
8     S_S ← S_S × n/m
9     return S_S
```

Exemple 4. Supposons, à partir de notre exemple courant, que nous sélectionnions un échantillon M de 4 tuples $\{h_5, h_8, h_{11}, h_{12}\}$. Nous calculons la distributions des données au sein de M. À partir de ce résultat intermédiaire et de l'échantillon nous

estimons la taille du skyline et obtenons $\hat{S}_S = 3,44$ tandis que la taille exacte du skyline est 3, rappelons qu'en faisant usage de l'ensemble des données nous avions obtenu $\hat{S}_W = 3,67$.

2.2.2 Espérance de la taille du skyline

Dans cette partie, nous présentons une manière d'estimer la taille du skyline lorsqu'on ne dispose que de la connaissance de la distribution des valeurs des données. Une fois de plus, nous faisons recours à l'estimation de l'espérance de la taille du skyline. Plus précisement, soit $\mathcal{T}(n, \{F_\ell\}, d)$ l'ensemble de toutes les tables T constituées de tuples dont les valeurs de chaque dimension D_ℓ suivent la loi définie par F_ℓ. Alors, la valeur moyenne des tailles de skyline des éléments de $\mathcal{T}(n, \{F_\ell\}, d)$ est donnée par $\frac{1}{|\mathcal{T}(n,\{F_\ell\},d)|}\sum_{T\in\mathcal{T}(n,\{F_\ell\},d)} |Sky(T, \mathcal{D})|$. Cette information donne une idée de la taille du skyline des tables présentant ce profil.

A présent, soit $Sky_\mathcal{T}$ la variable aléatoire correspondant à la taille du skyline des tables $T \in \mathcal{T}(n, \{F_\ell\}, d)$. Alors, l'espérance de $Sky_\mathcal{T}$ représente la taille moyenne que nous cherchons. Nous présentons une formule précise pour les cas où d est grand.

Nous nous restreignons au cas où l'ensemble des dimensions présentent la même distribution, c'est-à-dire $\forall i, j, k_i = k_j$ que nous noterons désormais k et $F_i = F_j$ qui sera noté F.

En raison du fait que $\hat{S}_W$ est un estimateur sans biais de $E(Sky_\mathcal{T})$, nous avons $E(\hat{S}_W) = E(Sky_\mathcal{T})$. De ce fait, estimer $E(\hat{S}_W)$ est équivalent à estimer $E(Sky_\mathcal{T})$; ce que nous developpons par la suite.

Nous considérons la table T comme une matrice où chaque cellule $t_i[j]$ est vue comme la réalisation d'une variable aléatoire T_{ij} dont la fonction de densité cumulée est F. Soient $X_i = \prod_{j=1}^{d} F(T_{ij})$ et $Z_i = (1 - X_i)^{n-1}$ deux ensembles de variables aléatoires.

Notre principal résultat peut être énoncé comme suit :

Proposition 3. Lorsque $d \to \infty$, la variable aléatoire $Z_i = (1 - X_i)^{n-1}$ converge en loi vers $(1 - X)^{n-1}$ où X est une variable aléatoire telles que $Y = \log(X)$ suit la loi normale $\mathcal{N}(\mu, \sigma^2)$. Alors, $E(Sky_\mathcal{T})$ peut être estimé par $\hat{S}_E = n \times E\left[(1 - X)^{n-1}\right]$ où

$$\mu = d\sum_{\ell=1}^{k} f(\ell) log(F(\ell))$$

et

$$\sigma^2 = d\left[\sum_{\ell=1}^{k} f(\ell)\left(log(F(\ell))\right)^2 - \left(\sum_{\ell=1}^{k} f(\ell) log(F(\ell))\right)^2\right]$$

Démonstration. Soient $U_j = \log(F(T_{ij}))$ des variables aléatoires définies sur l'ensemble $\{\log(F(1)), \log(F(2)), \ldots, \log(F(k))\}$. Puisque les variables U_j sont indépendantes et identiquement distribuées, nous notons μ_U leur espérance et σ_U^2 leur variance. L'espérance de U_j est égale à $\mu_U = E(U_j) = \sum_{\ell=1}^{d} f(\ell) \times \log(F(\ell))$ tandis que la variance de U_j est égale à :

$$\begin{aligned}\sigma_U^2 &= V(U_j) \\ &= E(U - E(t)) = E(U^2) - E(U)^2 \\ &= \sum_{\ell=1}^{k} f(\ell)\,(log(F(\ell)))^2 - (\mu_U)^2\end{aligned}$$

Soit $Y_i = \log(X_i)$ une variable aléatoire. Puisque $X_i = \prod_{j=1}^{d} F(T_{ij})$ alors $Y_i = \prod_{j=1}^{d} \log(F(T_{ij})) = \sum_{j=1}^{d} U_j$. Etant donné que les variables aléatoires Y_i sont indépendantes et identiquement distribuées, nous notons μ_Y et σ_Y^2 respectivement leur espérance et leur variance. L'espérance de Y_i est égale à $\mu_Y = E(Y_i) = \sum_{j=1}^{d} E(U_j) = d \times \mu_U$ et la variance de Y_i est égale à $\sigma_Y^2 = V(Y_i) = \sum_{j=1}^{d} V(U_j) = d \times \sigma_U^2$.

Étant donné que Y_i représente une somme de variables aléatoires indépendantes et identiquement distribuées, nous lui appliquons le *Théorème Central Limite* (Définition 2.5). Alors, lorsque d devient grand, Y_i converge vers la loi normale $Y_i \sim \mathcal{N}(\mu_Y, \sigma_Y^2)$. Cette convergence est illustrée dans la figure 2.1 où, dépendant des valeurs de k et de d, différents cas sont représentés. Pour k et d fixés, l'histogramme de 10000 valeurs de Y_i est tracé, cet histogramme est comparé à la courbe de la densité de la loi normale correspondante. La figure 2.1 montre qu'au fur et à mesure que k et/ou d augmentent, la convergence se précise.

D'après le *Théorème Central Limite*, lorsque d augmente, $Y_i = \log(X_i)$ converge vers la loi normale $\mathcal{N}(\mu_Y, \sigma_Y^2)$.

Soit Y une variable aléatoire telle que Y_i converge en loi vers Y lorsque $d \to \infty$ ce qui se note $Y_i \xrightarrow{\mathcal{L}} Y$. Il est établi que $Y \sim \mathcal{N}(d\mu_U, d\sigma_U^2)$. Soit X une variable aléatoire telle que $Y = \log(X)$. Alors, lorsque $d \to \infty$, X_i converge vers X. D'où $Z_i = (1 - X_i)^{n-1}$ converge vers $(1 - X)^{n-1}$.

Partant de la définition de $\hat{S}_W$ (voir Proposition 1), nous avons

$$\hat{S}_W = \sum_{i=1}^{n} \left(1 - \prod_{\ell=1}^{d} F_\ell(t_i[\ell]) + \prod_{\ell=1}^{d} f_\ell(t_i[\ell])\right)^{n-1}$$

Le terme $\prod_{\ell=1}^{d} f_\ell(t_i[\ell])$ correspond à la probabilité pour deux tuples d'être égaux, mais

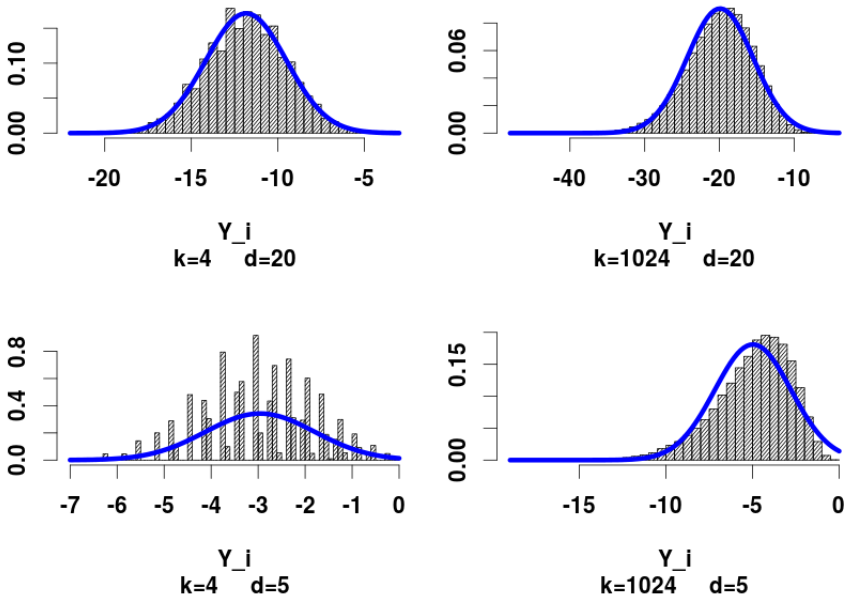


FIGURE 2.1 – Distribution des variables Y

au fur et à mesure que d augmente, cette probabilité tend très rapidement à s'annuler, comparée à $\prod_{\ell=1}^{d} F_\ell(t_i[\ell])$; alors $\hat{S}_W = \sum_{i=1}^{n} \left(1 - \prod_{\ell=1}^{d} F_\ell(t_i[\ell])\right)^{n-1} = \sum_{i=1}^{n} Z_i$, cependant, Z_i converge en loi vers $(1-X)^{n-1}$, de ce fait, $E(\hat{S}_W) = \sum_{i=1}^{n} E((1-X)^{n-1}) = n \times E\left((1-X)^{n-1}\right)$.

$E(Sky_{\mathcal{T}}) = E(\hat{S}_W)$ puisque $\hat{S}_W$ est sans biais ; alors, lorsque d devient grand, $\hat{S}_E = n \times E\left[(1-X)^{n-1}\right]$ est une une *bonne* estimation de l'espérance de la taille du skyline $E(Sky_{\mathcal{T}})$. □

Nous n'avons jusque-là uniquement montré que $E(Sky_{\mathcal{T}})$ pourrait, sous certaines conditions, être assimilé à $nE[(1-X)^{n-1}]$. D'un point de vue pratique, ce n'est pas suffisant pour obtenir une estimation de la taille du skyline. En effet, il est nécessaire d'estimer l'espérance de la variable aléatoire $Z = (1-X)^{n-1}$. Par définition, cette espérance est égale à $E(Z) = \int_0^1 z f_Z(z) \cdot dz$ où f_Z est la fonction de densité associée à Z.

Nous utilisons la *méthode des rectangles* pour estimer $E(Z)$. Soit N le nombre d'intervalles utilisés pour estimer $E(Z)$. L'estimation de $E(Z)$ est alors

$$\hat{E}(Z) = \sum_{\alpha=1}^{N} \frac{\alpha - \frac{1}{2}}{N} \left[F_Z\left(\frac{\alpha}{N}\right) - F_Z\left(\frac{\alpha - 1}{N}\right) \right]$$

où $F_Z(z) = Prob(Z \leq z)$ est la fonction de distribution cumulée relative à Z.

Nous devons déterminer F_Z.

$$\begin{aligned} F_Z(z) &= Prob(Z \leq z) = Prob\left[(1-X)^{n-1} \leq z\right] \\ &= Prob\left[X \geq 1 - \sqrt[n-1]{z}\right] \\ &= 1 - Prob\left[Y < log\left(1 - \sqrt[n-1]{z}\right)\right] \\ &= 1 - F_Y\left[log\left(1 - \sqrt[n-1]{z}\right)\right] \end{aligned}$$

Où F_Y est la fonction de densité cumulée relative à Y. Étant donné que Y suit une loi normale, la fonction F_Y peut être facilement accessible puisqu'elle est largement implémentée en informatique.

L'algorithme 8 montre une façon d'implémenter le calcul de l'estimation de l'espérance de la taille du skyline conformement à la méthode présentée. Cet algorithme a une complexité de l'ordre de $\mathcal{O}(d \times k + N)$, sachant que N est choisi par l'utilisateur.

Algorithme 3 : S_E, estime la taille du skyline

Input :
- number of tuples : n
- number of attributes : d
- number of distinct values per dimension : k
- distribution function : $f(quantile)$
- cumulative distribution functions : $F(quantile)$
- number of intervals : N
- normal cumulative distribution functions : $F_Z(quantile, \mu, \sigma^2)$

Output : Skyline estimation : S_E

begin

$\mu \leftarrow d \times \sum_{j=1}^{k} f(j) log(F(j))$

$\sigma^2 \leftarrow d \left[\sum_{j=1}^{k} f(j) \left(log(F(j))\right)^2 - \left(\sum_{j=1}^{k} f(j) log(F(j)) \right)^2 \right]$

$S_E \leftarrow 0$

for ***each*** α ***in*** $1 \ldots N$ **do**

$S_E \leftarrow S_E + \frac{\alpha - \frac{1}{2}}{N} \left[F_Z\left(\frac{\alpha}{N}, \mu, \sigma^2\right) - F_Z\left(\frac{\alpha-1}{N}, \mu, \sigma^2\right)\right]$

$S_E \leftarrow S_E \times n$

return S_E

Exemple 5. Toujours à partir de notre exemple courant, nous calculons l'estimation de l'espérance de la taille du skyline pour différentes valeurs de N en suivant l'algorithme présenté précédemment. Nous avons également calculé l'estimation de l'espérance de la taille du skyline en réalisant la moyenne des tailles de skyline de

$N = 8$	$N = 16$	$N = 32$	$N = 32768$	10000 jeux de données
$3,00$	$2,89$	$2,85$	$2,82$	$3,59$

TABLE 2.5 – Estimation de l'espérance de la taille du skyline

10000 jeux de données générés suivant le même format que celui de notre exemple ($k = 3$, $d = 4$, $n = 12$) ; cette méthode est proche de la méthode *bootstrap* de Efron [1979] utilisée pour l'estimation l'espérance d'un paramétre au sein d'une population. Le résultat est présenté dans le tableau 2.5.

2.3 Expérimentations

Dans cette section, nous comparons nos méthodes (notée S_W, S_S et S_E pour $N = \sqrt{n}$) avec l'algorithme Purely Sampling (PS) de Luo *et al.* [2012]. Nous ne considérons pas l'algorithme Kernel Based (KB) proposé par Zhang *et al.* [2009] parce qu'il a été montré dans Luo *et al.* [2012] que KB est plus lent et moins précis que PS. Dans leurs expérimentations, Luo *et al.* [2012] choisissent des échantillons de taille 5 %, 10 % et 15 % de n. Dans nos configurations, nous considérons plutôt des échantillons de taille $\sqrt{n}$ parce que PS possède une complexité dans le pire des cas de l'ordre de $\mathcal{O}(m^2 + s_1 n)$ où m est la taille de l'échantillon et s_1 la taille du skyline de l'échantillon ; tandis que la complexité de l'algorithme S_W est de l'ordre de $O(n \log n)$. En prenant $m = \sqrt{n}$, les deux algorithmes deviennent de ce fait comparables du point de vue du temps d'exécution. D'autre part, comme cela apparaît dans la figure 2.2, lorsque le nombre de tuples n est grand ($n \geq 10^5$), PS requiert plus de temps pour estimer la taille du skyline qu'il n'en faut pour calculer la valeur exacte à l'aide de l'algorithme BSkyTree de Lee et won Hwang [2010] (noté XT) qui, à notre connaissance, est le meilleur algorithme de l'état de l'art pour le calcul du skyline. Notons que la même figure montre que tous les algorithmes proposés ici requièrent moins de temps (S_S utilise le même echantillon que PS). Donc, un échantillon de taille 5% (le plus petit proposé par les auteurs) est déjà trop grand pour obtenir une estimation dans un temps raisonnable (plus rapidement que le calcul exact). Les données utilisées pour ce test sont constituées de 15 dimensions dont les valeurs ont été uniformément distribuées (chaque dimension possède $\sqrt{n}$ valeurs distinctes).

Tous les algorithmes comparés ont été implémentés en C++ en utilisant l'outil Code : :Blocks pour l'implémentation. Lesdites expérimentations ont été faites sur un ordinateur possédant un processeur Intel Core i7 - 2600 CPU @ 3,4 GHz x 8, une mémoire RAM de 7,8 GB et 500 GB de disque dur ; le tout sous le système d'exploitation Ubuntu 64 bit.

Dans la suite, nous analysons la précision et le temps d'exécution de ces algorithmes au regard de l'évolution de trois paramètres :

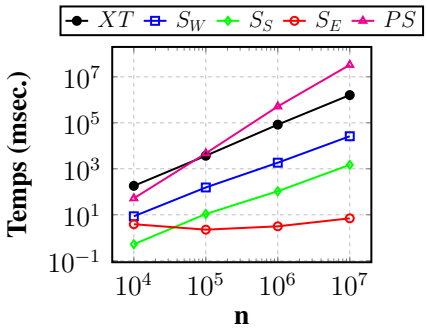


FIGURE 2.2 – Temps d'exécution par rapport à n (Taille de l'échantillon = 5% n)

— n (le nombre de tuples),
— d (le nombre de dimensions), et
— k (la cardinalité des dimensions).

Pour chaque configuration, nous analysons quatre distributions de données différentes :

— PROPORTIONAL : $k = \sqrt{n}$ et le nombre d'occurences de chaque valeur est proportionnel à la valeur en question (les petites valeurs apparaissent moins souvent que les plus grandes).
— INVERSE : $k = \sqrt{n}$ et le nombre d'occurences de chaque valeur est inversement proportionnel à la valeur en question (les petites valeurs apparaissent plus souvent que les plus grandes).
— UNIFORM : $k = \sqrt{n}$ et en moyenne, toutes les valeurs ont la même fréquence d'apparition.
— DISTINCT : $k = n$ c'est-à-dire les valeurs sont toutes distinctes au sein de chaque attribut.

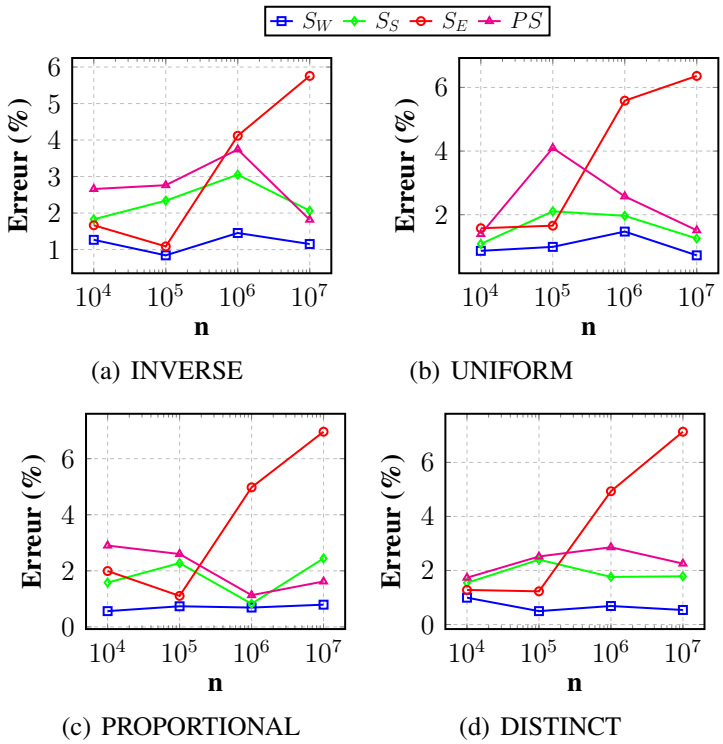
Ces distributions ont été choisies pour considérer autant que possible divers cas de figure des données. Nous voulons capter, autant que faire se peut, le comportement des estimateurs lorsque varie la nature de la distribution des valeurs au sein des dimensions. C'est pourquoi, nous avons considéré les cas où les valeurs préférées (valeurs faibles) sont prédominantes (INVERSE) par rapport aux autres, plus rares que les autres (PROPORTIONAL), aussi fréquentes que les autres (UNIFORM). Nous avons également voulu observer le comportement dans la situation extrême où toutes les valeurs sont distinctes au sein de chaque dimension (DISTINCT).

Le taux d'erreur relative est obtenu à partir de l'expression suivante

$$\frac{|\widehat{Sky} - EXACT|}{EXACT} \times 100$$

où $\widehat{Sky}$ est l'un des estimateurs.

Chaque expérimentation a été réalisée 5 fois. La valeur reportée pour le temps d'exécution ou pour le taux d'erreur relative est la moyenne des 5 valeurs obtenues pour le test correspondant.



(a) INVERSE (b) UNIFORM

(c) PROPORTIONAL (d) DISTINCT

FIGURE 2.3 – Erreur relative au regard de n

2.3.1 Variation du nombre de tuples

Nous avons généré 16 jeux de données de $d = 15$ dimensions chacun, en considérant chaque combinaison des paramètres suivants :

— $n \in \{10^4, 10^5, 10^6, 10^7\}$

— la distribution étant l'une des distributions précédemment mentionnées (PROPORTIONAL, INVERSE, UNIFORM, DISTINCT).

Précision

La figure 2.3 présente les taux d'erreur relative des algorithmes S_W , S_S, S_E et PS par rapport à la taille exacte du skyline. Dans l'ensemble, S_W apparaît comme ayant la plus petite erreur moyenne (moins de 2%) suivi de S_S. Quant à S_E, son taux d'erreur semble généralement augmenter avec le nombre de tuples. Ce comportement pourrait s'expliquer par le fait que cette méthode n'est pas sans biais. Lorsque n augmente, tous les algorithmes (mis à part S_E) semblent fournir des estimations plus précises. Ils présentent tous le même comportement indépendamment de la distribution des données considérée même si S_W possède une volatilité (variance de l'erreur) plus faible.

Comparons à présent ces méthodes en termes de temps d'exécution.

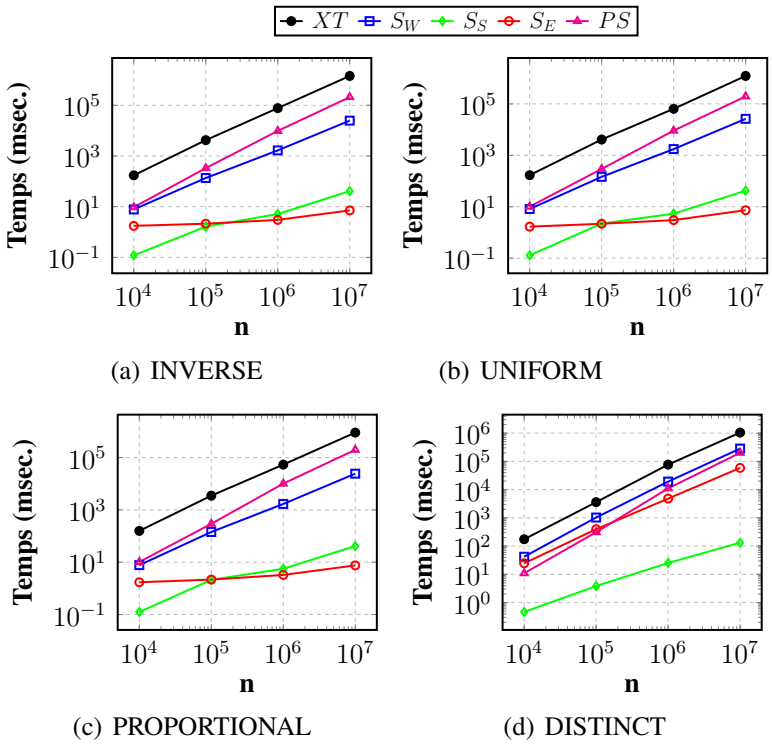


FIGURE 2.4 – Temps d'exécution au regard de n

Temps d'exécution

La figure 2.4 illustre les chiffres des temps d'exécution requis par les différents estimateurs. Nous avons utilisé l'implémentation de l'algorithme BSkyTree de Lee et won Hwang [2010] afin d'obtenir les valeurs exactes (XT dans la figure) à comparer aux estimations.

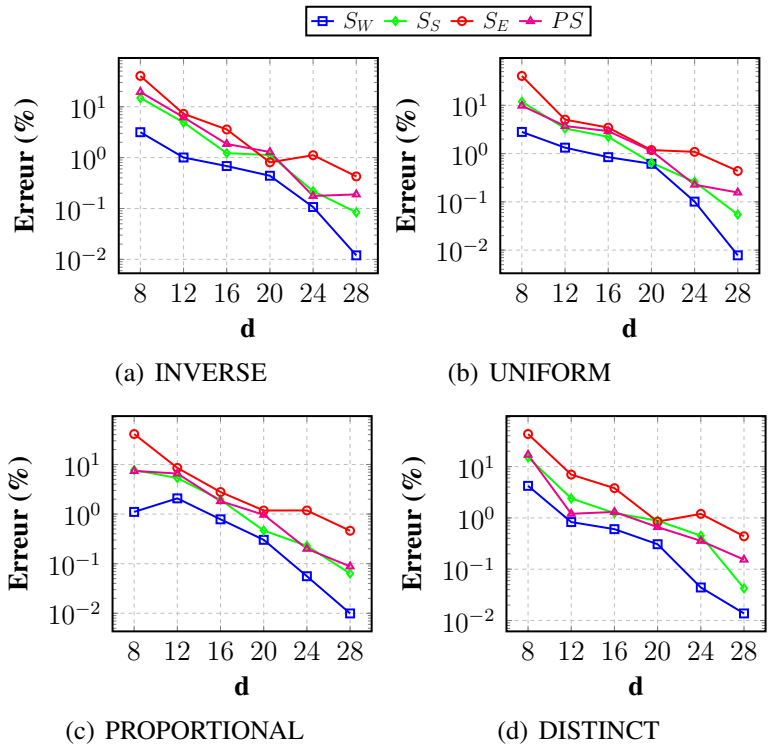
La première remarque est qu'en général, PS requiert un temps pratiquement comparable à celui de BSkyTree (juste environ 10 fois plus rapide) indépendamment de la distribution des données. Ce qui rend cet estimateur difficilement utilisable en pratique sachant que nous utilisons un échantillon plus petit que celui recommandé par les auteurs ($\sqrt{n}$ au lieu de $5\%n$).

D'autre part, S_W qui est l'estimateur le plus précis (voir section précédente) semble d'un ordre de magnitude plus rapide que BSkyTree excepté dans le cas où toutes les valeurs sont distinctes par dimension ($k = n$).

Rappelons que, dans cette situation, il existe dans la littérature (voir par exemple Bentley *et al.* [1978]) des formules analytiques donnant instantanément des estimations de la taille du skyline.

Il est important de préciser que S_S semble être un compromis intéressant entre le temps d'exécution et la précision dans les résultats. En effet, étant un peu moins précis que S_W, il est cependant bien plus rapide que celui-ci.

Enfin, nous observons que S_E, l'estimateur le moins précis, est par contre le



(a) INVERSE (b) UNIFORM

(c) PROPORTIONAL (d) DISTINCT

FIGURE 2.5 – Erreur relative par rapport à d

plus rapide tant qu'il ne s'agit pas de valeurs distinctes par dimension. Il présente un comportement différent dans le cas de la distribution dont les valeurs sont distinctes par dimension parce que sa complexité dépend de la cardinalité k.

2.3.2 Variation du nombre de dimensions

À présent, nous analysons l'impact du nombre de dimensions à la fois sur la précision des résultats d'estimation et sur le temps nécessaire pour les produire. Nous avons fixé le nombre de tuples n à 10^6 et nous avons fait varier le nombre de dimensions d en considérant les valeurs de l'ensemble $\{8, 12, 16, 20, 24, 28\}$. Pour chaque paires (n, d), nous avons considéré les quatre distributions de données présentées : PROPORTIONAL, INVERSE, UNIFORM et DISTINCT.

Précision

La figure 2.5 montre les taux d'erreur relative par rapport à la taille exacte du skyline. Globalement, lorsque le nombre de dimensions augmente, toutes les méthodes présentent le même comportement indépendamment de la distribution considérée : les taux d'erreur relative décroissent. Contrairement à l'observation faite précédemment (en faisant varier n pour d fixé), S_E fournit des meilleures estimations

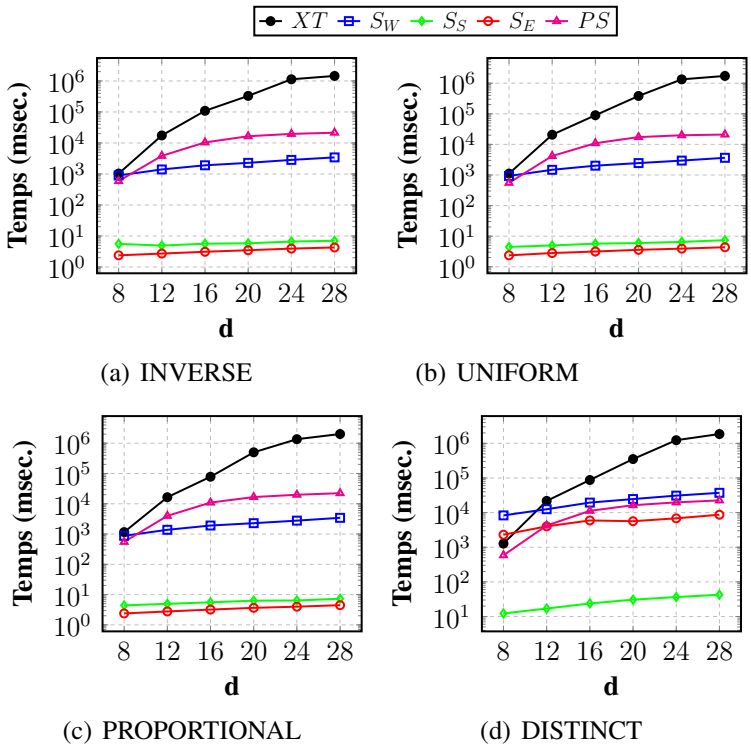
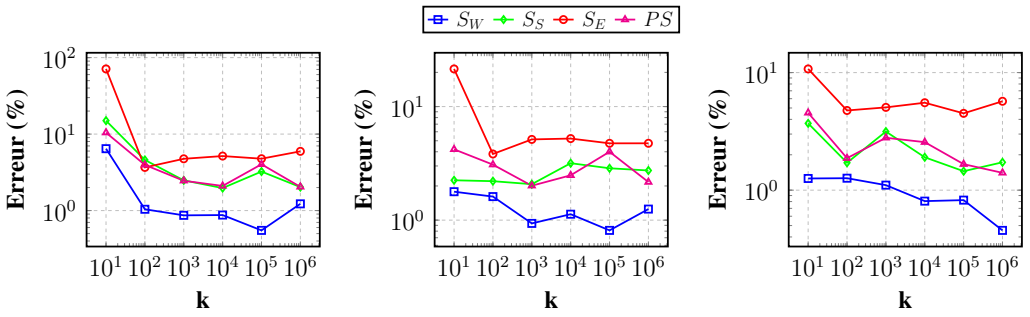


FIGURE 2.6 – Temps d'exécution par rapport à d

lorsque le nombre de dimensions augmente. Cette observation est en cohérence avec la proposition 3 stipulant que S_E converge (en loi) lorsque le nombre de dimensions augmente. Les deux approches opérant sur les échantillons (PS et S_S) présentent le même comportement se situant entre S_E (le moins précis) et S_W (le plus précis).

Temps d'exécution

Comme nous pouvons l'observer dans la figure 2.6 présentant les temps d'exécution des estimateurs lorsque le nombre de dimensions augmente, S_S et P_S ne sont pas vraiment équivalents (contrairement à ce que semblait suggérer la section précédente). En effet, nous observons que S_S est au moins 1000 fois plus rapide que PS. PS étant encore l'algorithme requérant le plus de temps pour fournir une estimation. Il est important de noter que le temps de calcul du résultat exact (XT) augmente exponentiellement avec d : $10^3 msec$ pour $d = 8$ et $10^6 msec$ pour $d = 28$ indépendamment de la distribution considérée. D'autre part, S_S et S_E ont presque un temps d'exécution constant. Mis à part le cas de la distribution DISTINCT des données, les deux estimateurs (S_S et S_E) requièrent moins de 10msec pour fournir une estimation. Enfin, nous constatons que dans le cas de la distribution DISTINCT, le temps d'exécution de la méthode S_E est comparable à celui de S_W et plus élevé que celui de S_S. Une fois encore, nous précisons qu'il existe des approches intéres-



(a) INVERSE (b) UNIFORM (c) PROPORTIONAL

FIGURE 2.7 – Erreur relative au regard de k

santes spécialisées dans cette distribution de données.

2.3.3 Variation du nombre de valeurs distinctes par dimension

Dans cette section, nous étudions l'influence de la cardinalité des dimensions sur la précision et les temps d'exécution des algorithmes proposés. Pour cela, nous fixons le nombre de tuples n à 10^6 et le nombre de dimensions d à 15. Nous faisons alors varier la cardinalité des dimensions en considérant les valeurs suivantes : $\{10^1, 10^2, 10^3, 10^4, 10^5, 10^6\}$. Enfin, pour chaque triplet (n, d, k), nous considérons les trois distributions des données suivantes : PROPORTIONAL, INVERSE et UNIFORM [2]. Ici également, les mesures reportées représentent la moyenne obtenue pour 5 expériences dans les mêmes conditions.

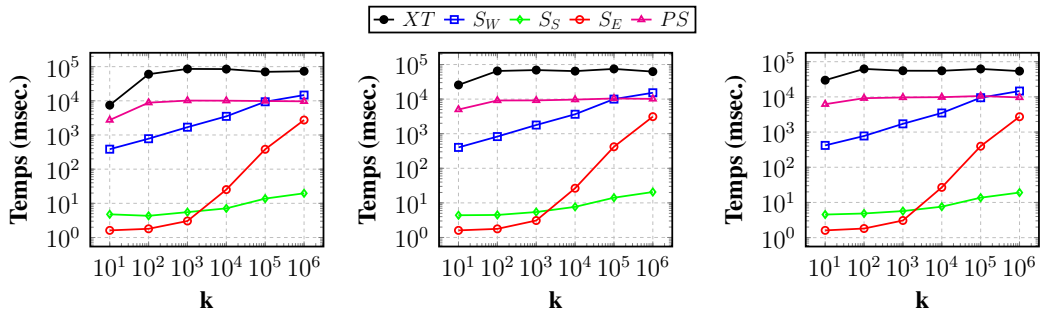
Précision

La figure 2.7 montre les taux d'erreur par rapport à la cardinalité des dimensions. Globalement, la précision des quatre estimateurs semble ne pas être affectée par la valeur de k. Cependant, une analyse approfondie suggère que lorsque k augmente, tous les estimateurs deviennent meilleurs. Enfin, aussitôt que k devient plus grand que 10^2 c'est-à-dire $n/10^4$, le taux d'erreur passe en dessous de 5% dans tous les cas. Notons cependant que S_W est la méthode la plus précise quelle que soit la valeur de k et la distribution des données considérée.

Temps d'exécution

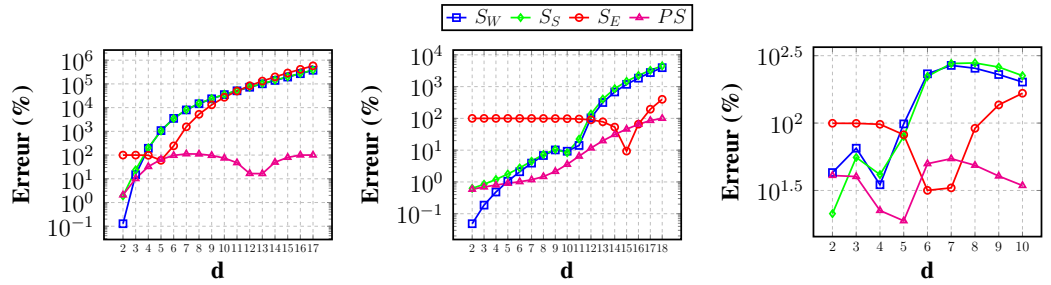
Comme présenté dans la figure 2.8 et sans surprise, l'algorithme exact aussi bien que l'algorithme PS ont des temps d'exécution presque constants (une légère croissance passant de $k = 10^1$ à $k = 10^2$). Nos méthodes présentent une croissance des temps de calcul (S_E plus que les autres) lorsque le nombre de valeurs distinctes

2. La distribution DISTINCT est prise en compte dans chaque cas lorsque $k = 10^6$.



(a) INVERSE (b) UNIFORM (c) PROPORTIONAL

FIGURE 2.8 – Temps d'exécution par rapport à k



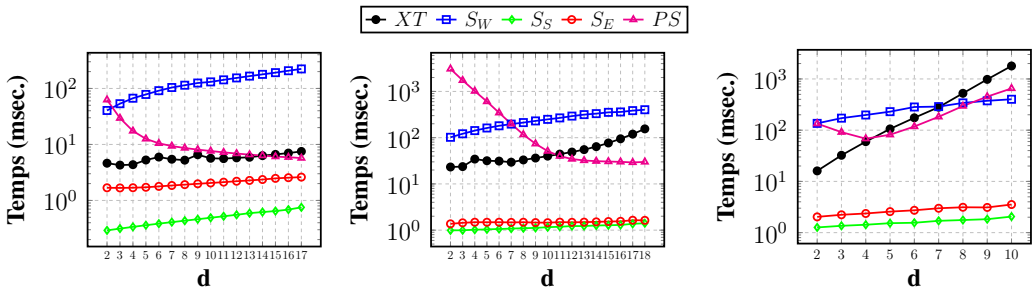
(a) NBA (b) BASEBALL (c) IPUMS

FIGURE 2.9 – Erreur relative au regard de d

par dimension augmente. Mais ces méthodes restent tout de même, en général, plus rapide que PS et ce, indépendamment de la distribution.

2.3.4 Données réelles

Dans cette section, nous montrons le comportement des différents estimateurs sur des données réelles. Les données réelles présentent en général des corrélations, positives ou négatives. Cependant, il est vraiment peu probable que la majorité des combinaisons d'un petit nombre de dimensions soient corrélées. Nos expérimentations confirment cette assertion. Pour ce faire, nous avons considéré des données bien connues de la communauté skyline : NBA, BASEBALL et IPUMS. Tous ces jeux de données sont corrélées. Plus précisément, nous considérons chaque paire de dimensions et calculons le coefficient de corrélation linéaire associé. Ce coefficient prend ses valeurs dans l'intervalle $[-1, 1]$. Une valeur proche de 1 (respectivement de -1) signifie que les deux dimensions sont positivement corrélées (respectivement négativement corrélées) et une valeur proche de 0 signifie que les dimensions ont tendance à être linéairement indépendantes. Pour chaque jeu de données, nous avons calculé la moyenne de cet indicateur. Le jeu de données NBA est constitué de 20493 tuples et 17 attributs ; c'est le jeu de données le plus corrélé avec une



(a) NBA (b) BASEBALL (c) IPUMS

FIGURE 2.10 – Temps d'exécution par rapport à d

moyenne du coefficient de corrélation entre les dimensions de 0.55 et une taille de skyline égale à 3. Le fichier BASEBALL est constitué de 92797 tuples et 18 dimensions ; pour un coefficient de corrélation moyen de 0.23 et une taille de skyline égale à 78. Enfin, IPUMS présente 75836 enregistrements et 10 variables ; c'est le jeu de données le moins corrélé, avec un coefficient moyen de 0.17 et une taille de skyline égale à 3852.

En raison de la taille faible de ces jeux de données réels, nous les avons repliqué 10 fois chacun ; rendant les temps d'exécution des méthodes plus différenciables tout en conservant les mêmes taux d'erreur. Pour chaque valeur d_j ($d_j \in \{2, 3, \ldots, d\}$) du nombre de dimensions, nous avons calculé à la fois la moyenne de la taille du skyline (pour chaque approche incluant la méthode exacte) et le temps d'exécution ; et ce, en considérant tous les sous espaces de d_j dimensions.

La figure 2.9 montre, pour chaque jeu de données, les taux d'erreur des estimateurs lorsque le nombre de dimensions augmente. Il apparaît que lorsque la distribution des données s'éloigne considérablement de nos hypothèses (indépendance entre les dimensions), nos estimations deviennent pires que PS. En effet, les taux d'erreur sont globalement meilleurs pour IPUMS dont le niveau de corrélation présente la valeur la plus basse parmi ces jeux de données. Lorsque le nombre de dimensions augmente, les taux d'erreur de S_W et S_S augmentent. Au regard de la précision, PS semble être le meilleur choix. Cependant, comme présenté dans la figure 2.10, le calcul exact du skyline requiert en général moins de temps que ce qu'il est nécessaire à PS pour fournir son estimation. Ceci est particulièrement vrai lorsque les dimensions présentent des corrélations (NBA et BASEBALL) ou lorsque le nombre de dimensions considérées est faible, ce qui à notre avis est très fréquent en pratique. En somme, même pour les données réelles, nos estimateurs présentent des résultats intéressants pour des requêtes se rapportant à peu de dimensions.

Conclusion

Cette partie du travail propose des estimateurs sans biais pour le calcul de la taille du skyline basés sur la connaissance de la distribution des données. L'idée maitresse consiste au parcours des données (en totalité ou en partie) pour estimer la probabilité p qu'un tuple appartienne au skyline. Une fois cette probabilité estimée, la taille du skyline se déduit par le produit $p \times n$. Plus encore, connaissant la distribution des données, nous proposons également un estimateur de l'espérance de la taille du skyline pour toutes les tables respectant la distribution en question. Bien que le dernier estimateur soit biaisé, nous avons montré que c'est un bon estimateur de l'espérance de la taille du skyline.

Nous avons comparé nos propositions à la méthode PS de l'état de l'art. Les expérimentations faites montrent que nos solutions offrent de meilleurs résultats aussi bien du point de vue précision que temps d'exécution. Plus précisement, configurer PS afin qu'il soit plus compétitif en termes de temps d'exécution diminuera la précision de ses résultats.

Chapitre 3

Matérialisation Partielle pour Optimiser les Requêtes Skyline

Introduction

L'analyse de bases de données multidimensionnelles est un des principaux domaines de recherches de la dernière décennie. Le pré-calcul est une solution ordinaire à l'optimisation de requêtes multidimensionnelles. Une proposition récente entrant dans ce cadre est la notion de cube de données Gray *et al.* [1997] qui, intuitivement, représente l'ensemble des requêtes d'agrégation considérant tous les sous-ensembles d'attributs. Après une série de travaux se focalisant sur les façons d'entièrement matérialiser le cube de données, voir Agarwal *et al.* [1996]; Zhao *et al.* [1997], il a été rapidement établi que cette solution est irréaliste en pratique en raison de la grande quantité d'espace mémoire requise aussi bien que le temps de calcul nécessaire puisque le nombre de requêtes est exponentiel du nombre de dimensions. De ce fait, la question qui en découlait était de savoir comment matérialiser juste un sous-ensemble de requêtes tout en satisfaisant certaines contraintes. Ce problème a été largement étudié dans la littérature et différentes solutions ont été proposées dépendant des objectifs et contraintes fixés. Par exemple, Harinarayan *et al.* [1996]; Agrawal *et al.* [2000]; Li *et al.* [2005] considèrent l'espace mémoire comme ressource limitée et essayent de trouver une partie du cube de données qui peut y être stockée tout en minimisant le temps global de réponse aux requêtes ; tandis que Hanusse *et al.* [2009] considère le problème différemment : étant donné un temps de réponse aux requêtes, déterminer la plus petite partie du cube de données garantissant cette performance.

Dans la littérature, la structure de données *skycube* a été proposée indépendamment dans Pei *et al.* [2005] et Yuan *et al.* [2005]. Cette structure a pour objectif d'aider les utilisateurs à pouvoir naviguer à travers l'espace multidimensionnel en demandant les requêtes skyline sur les ensembles d'attributs spécifiés. Dans le but d'optimiser les temps de réponse, plusieurs algorithmes ayant pour objectif de matérialiser le skycube, c'est-à-dire pré-calculant tous les skylines possibles, ont été

Id	A	B	C	D
t_1	1	3	6	8
t_2	1	3	5	8
t_3	2	4	5	7
t_4	4	4	4	6
t_5	3	9	9	7
t_6	5	8	7	7

TABLE 3.1 – Table de données.

proposés. Voir par exemple Pei *et al.* [2006] et Lee et won Hwang [2014]. Contrairement aux cubes de données, très peu de solutions ont été proposées pour la matérialisation partielle du skycube. Au meilleur de nos connaissances, il existe deux propositions : Raïssi *et al.* [2010] a introduit le concept de *closed skycube*. Les auteurs y proposent un algorithme identifiant les skylines équivalents, dans le but de réduire l'espace mémoire nécessaire en ne stockant qu'une seule copie des résultats identiques. La seconde contribution est appelée *compressed skycube* (CSC), proposée dans Xia *et al.* [2012]. Au lieu de raisonner à l'echelle des skylines comme c'est le cas pour *closed skycube*, cette technique opère au niveau des tuples, c'est-à-dire, un tuple appartenant à plusieurs skylines pourrait ne pas être stocké dans chacun de ces skylines. Nous apportons de plus amples détails concernant ces deux solutions dans la section 3.1 dédiée aux travaux relatifs à ce domaine.

Exemple 6. Nous empruntons la table 3.1 à partir de l'article Raïssi *et al.* [2010] et l'utiliserons comme exemple illustratif tout le long de ce travail.

$Att(T) = \{Id, A, B, C, D\}$. Soit $\mathcal{D} = ABCD$ et soit $X = ABCD$, alors $t_2 \prec_X t_1$. En effet, $t_2[X_i] \leq t_1[X_i]$ pour tout $X_i \in \{A, B, C, D\}$ et $t_2[C] < t_1[C]$. Dans cet exemple $d = 4$. Les skylines relativement à chacun des sous-espaces de $\mathcal{D}$, c'est-à-dire, l'ensemble de tous les skycuboids sont présentés dans la table 3.2.

Cet exemple montre que les skylines résultants ne sont pas *monotones*, c'est-à-dire, $X \subseteq X' \not\Rightarrow Sky(X) \subseteq Sky(X')$ et $X \subseteq X' \not\Rightarrow Sky(X) \supseteq Sky(X')$. Par exemple, $Sky(ABD) \not\subseteq Sky(ABCD)$ et $Sky(D) \not\supseteq Sky(AD)$. Cette propriété rend la matérialisation partielle des skycubes plus difficile que celle des cubes de données.

Organisation du chapitre et résumé des contributions La section 3.1 est dédiée à une étude des références de la littérature relatives à l'exécution des requêtes skylines multidimensionnelles. Ensuite, nous présentons notre contribution à cette problématique. Nous commençons par une première solution à la matérialisation partielle du skyline basée sur le topmost skyline (section 3.2.2). Cette solution exploite la propriété de monotonie des motifs de skyline (présentée en section 3.2.1) et n'est efficace que lorsque le topmost skyline est petit. En effet, nous montrons

Subspace	Skyline	Subspace	Skyline
$ABCD$	$\{t_2, t_3, t_4\}$	ABC	$\{t_2, t_4\}$
ABD	$\{t_1, t_2, t_3, t_4\}$	ACD	$\{t_2, t_3, t_4\}$
BCD	$\{t_2, t_4\}$	AB	$\{t_1, t_2\}$
AC	$\{t_2, t_4\}$	AD	$\{t_1, t_2, t_3, t_4\}$
BC	$\{t_2, t_4\}$	BD	$\{t_1, t_2, t_4\}$
CD	$\{t_4\}$	A	$\{t_1, t_2\}$
B	$\{t_1, t_2\}$	C	$\{t_4\}$
D	$\{t_4\}$		

TABLE 3.2 – L'ensemble de tous les skylines.

que dans ce cas, le topmost skyline peut répondre à n'importe quelle autre requête skyline sans avoir à accéder à la table de données ; et sa taille (petite), nous garantit une efficacité dans l'exécution en suivant ce procédé. Dans la partie suivante (section 3.2.4), nous fournissons une solution complémentaire (lorque le topmost skyline n'est pas forcément petit), basée sur les dépendances fonctionnelles. Pour cela, nous montrons que l'existence des dépendances fonctionnelles implique des inclusions entre les skylines (Théorème 3.3). En effet, ce concept nous permet de définir une hiérarchie d'inclusion entre les skycuboids, ce qui nous permet de formuler une solution reposant sur la matérialisation partielle, analogue à celle des cubes de données, pour optimiser en temps et en espace le calcul des skycuboids. Nous analysons par la suite l'exécution des requêtes à partir de la partie matérialisée du skycube (section 3.3). Enfin, nous comparons nos propositions à quelques solutions de l'état de l'art (Xia *et al.* [2012]; Bøgh *et al.* [2014]; Pei *et al.* [2005]) à travers une série d'expérimentations ayant pour objectif de confirmer l'efficacité de notre approche.

3.1 Travaux relatifs

Dans le cadre restreint du calcul des skylines multidimensionnels, plusieurs travaux ont considéré la matérialisation complète du skycube : Lee et won Hwang [2014, 2010]; Pei *et al.* [2005]; Yuan *et al.* [2005]. Afin d'éviter la solution naïve consistant à calculer chaque skyline à partir de la table de base, ces algorithmes essayent de mutualiser certains calculs en utilisant (i) le partage de la structure (*shared structures*) simplifiant la propagation/élimination des points skyline de $Sky(X)$ à/de $Sky(Y)$ lorsque $Y \subseteq X$ ou (ii) le partage de calculs (*shared computations*). Plus précisément, l'idée ici consiste à élaborer des règles de déduction qui permettent de déterminer $Sky(X)$ en utilisant une partie de $Sky(Y)$. Par exemple, si $Y \subset X$ alors un tuple t_i ne peut appartenir à $Sky(Y)$ s'il ne correspond pas en Y à un tuple t_j de $Sky(X)$. Dans le but d'exploiter cette propriété, on doit conserver

$Sky(X)$ en mémoire. Puisque plusieurs skylines nécessitent d'être gardés en mémoire de cette façon, l'utilisation de la mémoire pourrait rapidement devenir un problème lorsque les données sont de grande taille. En raison du nombre exponentiel de skycuboids, certains travaux essayent de définir des techniques de compressions Xia *et al.* [2012]; Raïssi *et al.* [2010], pour réduire de ce fait l'espace utilisé pour le stockage du skycube entier.

Pei *et al.* [2005] a proposé le concept de *skyline group lattice* qui peut être vu comme une technique de matérialisation partielle du skycube. Cette structure consiste en le stockage pour chaque tuple t, d'un ensemble de paires de la forme $\langle$max-subspace, min-subspaces$\rangle$. Les éléments de chaque paire agissent comme des bordures : le premier est une borne supérieure et le second une bordure inférieure. La sémantique de ces paires est que le tuple t appartient à tous les skylines relatifs aux espaces inclus dans la bordure supérieure qui incluent au moins un élement de la bordure inférieure. Par exemple, supposons que la paire $p_1 = \langle ABCD, \{AC, AD\}\rangle$ soit associée au tuple t. Ceci signifie que t appartient aux skylines relatifs aux espaces $ABCD$, ABC, ACD, AC et AD mais pas à ceux relatifs aux espaces BCD, BC ou BD car aucun d'entre eux ne contient AC ou AD (la frontière inférieure). t n'appartient pas aux skylines relatifs aux espaces A, C et D (sous-espaces des frontières inférieures) et encore moins aux espaces de la forme $ABCDE_i$, c'est-à-dire les sur-espaces de la borne supérieure $ABCD$. Tous les tuples partageant une paire et ayant la même valeur pour l'espace spécifié par la borne supérieure de la paire forment un groupe skyline et l'ensemble des groupes skyline forme un treillis. Cette technique fait partie des approches raisonnant au niveau des tuples dans ce sens qu'elle essaye de réduire le nombre de fois qu'un tuple sera stocké tandis que d'autres approches (en occurrence celle que nous présentons dans ce chapitre) raisonnent à l'échelle des espaces en minimisant plutôt le nombre de skycuboids stockés. D'un autre point de vue, les avantages de cette structure de données viennent avec un coût : il est plus coûteux de la calculer que de construire le skycube complet.

Raïssi *et al.* [2010] a proposé le concept de *closed skycube* en tant que manière de résumer le skycube. Brièvement, cette méthode partitionne les $2^d - 1$ skycuboids en classes d'équivalence : deux skycuboids étant équivalents si leurs skylines respectifs sont égaux (contiennent exactement les mêmes tuples). Par conséquent, le nombre de skylines matérialisés est égal au nombre de classes d'équivalence. Une fois que les classes d'équivalence sont identifiées et matérialisées, l'exécution des requêtes est immédiate : pour chaque requête $Sky(T, X)$, on retourne le skyline associé à la classe d'équivalence de X. Même si cette solution présente un temps optimal de réponse aux requêtes, la recherche des classes d'équivalence requiert beaucoup de temps. D'autre part, la taille du *closed skycube* pourrait être égale à celle du skycube dans le cas où tous les skylines sont différents d'un espace à un autre.

Xia *et al.* [2012] a proposé la structure de données Compressed SkyCube (CSC). CSC peut être décrit comme suit. Soit t un tuple appartenant à $Sky(X)$. Alors t est dans le skyline *minimum* de $Sky(X)$ noté $min\_sky(X)$ si et seulement s'il n'existe

pas d'espace $Y \subset X$ tel que $t \in Sky(Y)$. La structure de données *Compressed SkyCube* consiste alors simplement à stocker pour chaque espace X l'ensemble $min\_sky(X)$. Les auteurs montrent que cette structure est sans perte dans ce sens que $Sky(T, X)$ peut être retrouvé à partir du contenu de $min\_sky(Y)$ sachant que $Y \subseteq X$. plus précisemment, $t \in Sky(T, X)$ si et seulement si $\exists Y \subseteq X$ tel que t appartient à $Sky(min\_sky(Y), X)$. Donc, la dimensionalité d peut devenir un goulot d'étranglement pour l'exécution des requêtes. En effet, exécuter $Sky(X)$ requiert le parcours de tous les sous-espaces de X (en nombre exponentiel du nombre d'attributs de X), de collecter tous les tuples s'y trouvant, et d'exécuter un algorithme standard de calcul de skyline afin de supprimer ceux qui sont dominés au regard de X.

Il est important de préciser que dans le cas où les valeurs sont distinctes au sein de chaque dimension, CSC et *skyline group* sont équivalents. En effet, sous cette hypothèse, le skyline est monotone : $X \subseteq Y \Rightarrow Sky(X) \subseteq Sky(Y)$. Donc, pour chaque espace X nous avons $Sky(X) \subseteq Sky(\mathcal{D})$. De là, à chaque tuple t est associée une seule paire de la forme $\langle \mathcal{D}, \{Y_1, \ldots, Y_p\} \rangle$. Où l'ensemble des espaces de la borne inférieure sont exactement ceux auxquels sont associés le tuple t dans la structure CSC.

Récemment, Bøgh *et al.* [2014] a proposé la structure Hashcube pour stocker l'ensemble du skycube. Intuitivement, cette méthode consiste à utiliser des vecteurs de bits pour encoder les sous-espaces pour lesquels les tuples appartiennent au skyline. Tandis que le temps de réponse de cette stucture de données est impressionnant, car en général il est seulement de 10% plus lent que le temps de réponse idéal. Il requiert par contre beaucoup de mémoire parce que sa consommation de mémoire est de 10% la taille du skycube. Puisque la taille du skycube augmente de façon exponentielle avec le nombre de dimensions, la structure Hashcube sature rapidement la mémoire disponible. Plus important encore, les auteurs ne fournissent pas de procédure pour directement construire cette structure à partir des données. À la place, ils prennent en entrée le skycube, supposé construit, et encodent celui-ci en la structure de données Hashcube ; alors que la construction du skycube est une tâche requérant beaucoup de temps mais aussi de mémoire.

3.2 Matérialisation Partielle des Skycubes

Le principal objectif de ce travail est de déterminer une solution à la matérialisation partielle des skycubes. L'exigence la plus importante étant de construire le skycube partiel le *plus petit possible* afin de minimiser à la fois l'espace nécessaire à son stockage et son temps de calcul. Avant de décrire plus formellement le problème que nous abordons, nous présentons quelques propriétés satisfaites par les skycuboids.

3.2.1 Monotonie des Motifs Skyline

Même si la requête skyline n'est pas monotone, nous pouvons mettre en exergue une propriété de monotonie entre $Sky(T, X)$ et $Sky(T, Y)$ tant que $X \subseteq Y$. Intuitivement, cette propriété est basée sur l'observation suivante : pour qu'un tuple t appartienne à $Sky(T, X)$, il est nécessaire et suffisant qu'il existe un tuple $t' \in Sky(Sky(T, Y), X)$, c'est-à-dire, le skyline au regard de X de l'ensemble des tuples appartenant à $Sky(T, Y)$, tel que $t'[X] = t[X]$. Cette observation fait l'objet du lemme suivant.

Lemme 3.1. *Soit $X \subseteq Y$. Alors $\forall t \in T$ nous avons $t \in Sky(T, X) \Leftrightarrow \exists t' \in Sky(Sky(T, Y), X)$ tel que $t'[X] = t[X]$.*

Démonstration. Nous montrons les deux implications.

❶ $\Rightarrow$: Soit $t \in Sky(T, X)$. Si $t \in Sky(Sky(T, Y), X)$ alors l'assertion est clairement satisfaite. Supposons à présent que $t \notin Sky(T, Y)$. Alors il existe $t' \in Sky(T, Y)$ tel que $t' \prec_Y t$ mais $t' \not\prec_X t$ car t appartient à $Sky(T, X)$. Ceci implique que $t'[X] = t[X]$ donc t' appartient également à $Sky(T, X)$. De ce fait, $t' \in Sky(Sky(T, Y), X)$.

❷ $\Leftarrow$: $t' \in Sky(Sky(T, Y), X) \Leftrightarrow \forall u \in Sky(T, Y), u \not\prec_X t'$. D'autre part nous avons $\forall v \in T \setminus Sky(T, Y), \exists u \in Sky(T, Y), u \prec_Y v$. Sachant que $u \not\prec_X t'$ et $u \preceq_X v$ car $X \subseteq Y$, nous avons $v \not\prec_X t' \Leftrightarrow t' \in Sky(T, X)$. Puisque $t[X] = t'[X]$ alors $t \in Sky(T, X)$.

□

Exemple 7. Considérons les sous-espaces B et BC. Comme présenté dans la table 3.2, $Sky(T, BC) = \{t_2, t_4\}$. De cet ensemble de tuples, nous déduisons que $Sky(Sky(T, BC), B) = \{t_2\}$ (car $t_2[B] = 3 < t_4[B] = 4$) et ceci ne correspond pas à $Sky(T, B)$ puisque ce dernier a un tuple supplémentaire qui est t_1. Notons cependant que (i) ce tuple manquant t_1 présente la même valeur que t_2 en B et (ii) c'est le seul satisfaisant cette propriété.

Comme conséquence, nous obtenons une forme de relation d'inclusion entre les tuples appartenant à $Sky(X)$ et ceux appartenant à $Sky(Y)$ lorsque $X \subseteq Y$. Plus précisément,

Théorème 3.2. *Soit $X \subseteq Y$ et soit $\pi_X(Sky(X))$ la projection sur X des tuples appartenant à $Sky(X)$. Alors, $\pi_X(Sky(X)) \subseteq \pi_X(Sky(Y))$.*

Exemple 8. $Sky(A) = \{t_1, t_2\}$ et $Sky(AC) = \{t_2, t_4\}$. Bien que $Sky(A) \not\subseteq Sky(AC)$, nous avons $\pi_A(Sky(A)) \subseteq \pi_A(Sky(AC))$. En effet, $\pi_A(Sky(A))$=$\{\langle 1\rangle\}$ et $\pi_A(Sky(AC))$=$\{\langle 1\rangle; \langle 4\rangle\}$.

Le théorème 3.2 montre que les points skyline de chaque sous-espace forment un treillis : il suffit de projeter chaque skyline sur les dimensions définissant le sous-espace et compléter les dimensions manquantes par le symbole spécial $*$.

Ces tuples généralisés définissent les *motifs* skyline.

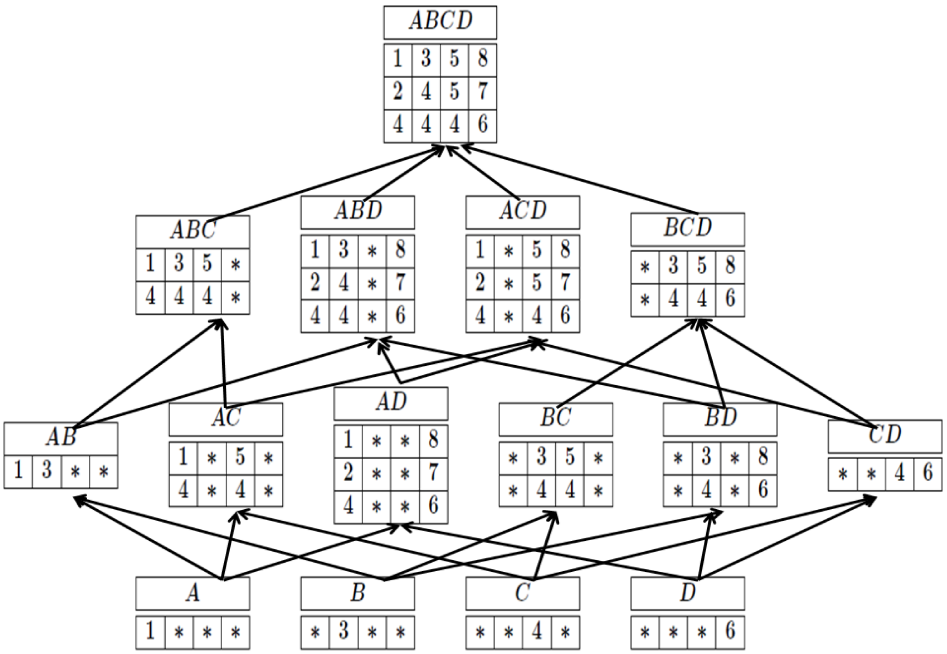


FIGURE 3.1 – Treillis des motifs skycube

Exemple 9. Considérons le sous-espace B. $Sky(B)$ contient deux tuples t_1 et t_2. Le seul motif définissant les tuples de $Sky(B)$ est le tuple généralisé $\langle *, 3, *, * \rangle$. La figure 3.1 présente les motifs du skycube à partir de notre exemple de données d'illustration.

Le lemme 3.1 montre que lorsque $X \subseteq Y$, le calcul de $Sky(X)$ pourrait bénéficier du fait que $Sky(Y)$ soit déjà calculé et matérialisé. L'algorithme 4 montre comment ceci peut être réalisé.

Algorithme 4 : Sky_X_from_Sky_Y

Input : Table T, $Sky(Y)$, X such that $X \subseteq Y$
Output : $Sky(X)$
1 Let $S_1 = \pi_X(Sky(Y))$;
2 Let $S_2 = Sky(S_1, X)$;
3 Return $T \bowtie S_2$;

Simplement, l'algorithme 4 commence par éliminier de $Sky(Y)$ les doublons relativement à l'espace X, pour obtenir S_1. La deuxième étape consiste à calculer le skyline de S_1 en considérant toutes ses dimensions, c'est-à-dire, X. On obtient alors l'ensemble S_2 pour lequel on fait une jointure avec la table T.

L'algorithme 4 est correct, en effet, puisque S_2 ne présente pas de doublons, nous avons la garantie qu'un tuple $t' \in T$ peut être retourné au plus une fois. De plus, le théorème 3.1 garantit que tout $t' \in Sky(X)$ appartiendra au résultat. Par conséquent, tous les tuples t' et uniquement ceux satisfaisants $t' \in Sky(X)$ seront retournés.

Exemple 10. Supposons que $Sky(ABC) = \{t_2, t_4\}$ soit déjà matérialisé et nous désirons calculer $Sky(AB)$. L'algorithme **Sky_X_from_Sky_Y** commence par projeter $Sky(ABC)$ sur AB et on obtient $S_1 = \{\langle 1,3\rangle; \langle 4,4\rangle\}$. Alors, la ligne 2 de l'algorithme retourne $S_2 = Sky(S_1, AB) = \{\langle 1,3\rangle\}$. Enfin (la ligne 3), on fait une jointure entre T et S_2, c'est-à-dire, seront retournés les tuples t' tels que $t'[AB] = \langle 1,3\rangle$. Il y a deux tuples satisfaisants ce critère : t_1 et t_2. Par conséquent, $Sky(AB) = \{t_1, t_2\}$.

Complexité temporelle de Sky_X_from_Sky_Y La projection de la ligne 1 peut être réalisée en $O(|Sky(Y)|)$ utilisant du hashage. La ligne 2 (calcul du skyline) peut être exécutée en $O(|S_1|.|S_2|)$, c'est-à-dire, chaque tuple de S_1 a été comparé à chacun des tuples du résultat S_2. L'opération de jointure entre S_2 et T peut être réalisée en $O(|T|+|S_2|)$ avec, par exemple, un algorithme de jointure par hashage : parcourir $S_2 = Sky(S_1, X)$ et insérer chaque t dans la table de hashage H, ensuite parcourir T, et vérifier en $O(1)$ que le hash de chaque $t'[X]$ appartient à H. Si c'est le case, alors retourner t'. En résumé, le temps de calcul global est borné par $O(|Sky(Y)|) + O(|S_1|.|S_2|) + O(|T| + |S_2|)$. Ainsi, si $|S_1|$ est proche de $|T|$, on calcule mieux $Sky(T, X)$ dont la complexité est $O(|T|.|Sky(X)|)$.

3.2.2 Matérialisation basée sur le topmost Skyline

Le lemme 3.1 nous permet de dériver une solution ad-hoc à la matérialisation partielle des skycubes. En effet, il suffit de matérialiser $Sky(T, \mathcal{D})$, c'est-à-dire, le skyline l'ensemble des dimensions et de l'utiliser afin de répondre à toute requête skyline $Sky(X)$ soumise en appelant **Sky_X_from_Sky_Y**$(T, Sky(\mathcal{D}), X)$. Cette solution est particulièrement efficace lorsque $Sky(\mathcal{D})$ est petit, par exemple $|Sky(\mathcal{D})| = O(\sqrt{n})$. Ceci pourrait survenir lorsque les dimensions sont corrélées ou/et lorsqu'il existe très peu de valeurs distinctes par dimension. De plus, une fois que $Sky(\mathcal{D})$ est calculé, nous savons que tout tuple $t \in T$ qui ne présente aucun motif de valeurs similaire à celui d'un tuple $t' \in Sky(\mathcal{D})$ en aucune dimension, peut être retiré de T parce que nous avons la certitude qu'il n'appartiendra à aucun skycuboid. L'exemple suivant illustre ce détail.

Exemple 11. Supposons que nous avons quatre dimensions indépendantes et chacune d'elles présente trois valeurs distinctes $\{0, 1, 2\}$ et équiprobables. Le nombre de tuples possibles est égal à $3^4 = 81$. Si le nombre de tuples de T est n avec $n \gg 81$, alors, il y a de grandes chances que le tuple $\langle 0,0,0\rangle$ soit présent. Si c'est le cas, alors tout tuple du topmost sera égal à $\langle 0,0,0\rangle$. Dans cette situation extrême,

le nombre de tuples distincts de $Sky(\mathcal{D})$ est égal à 1 quel que soit le nombre de ses réplicas dans les données. De plus, pour tout tuple $t \in T$ et $X \subseteq \mathcal{D}$, $t \in Sky(X)$ si et seulement si $t[X] = \langle 0, \ldots, 0 \rangle$. En d'autres termes, si t présente uniquement des valeurs différentes de 0, alors nous pouvons le retirer de T : sachant qu'il n'appartient à aucun skyline. Par conséquent, une fois que $Sky(\mathcal{D})$ est calculé, la table T peut être *nettoyée* pour obtenir $T_{clean} \subseteq T$ où T_{clean} contient uniquement des tuples ayant une chance d'apparaître dans le résultat d'une requête skyline.

Une solution semi-naïve à la matérialisation partielle est décrite à travers l'algorithme 5. Le nettoyage de T pourrait être réalisé en $O(n)$: pour chaque dimension D_i, on crée une table de hashage H_i correspondant à $\pi_{D_i}(Sky(\mathcal{D}))$. On obtient alors d tables de hashage. Ensuite, on parcours T vérifiant que pour chaque $t \in T$ on a $t[D_i] \in H_i$. Si c'est le cas, alors t est ajouté à T_{clean}. La complexité de cette opération est de l'ordre de $O(dn)$ et si d est fixé l'opération s'effectue en $O(n)$.

À la ligne 7, on crée un index bitmap sur T_{clean} relativement à chaque D_i dans le but d'optimiser l'opération de jointure lorsque la requête skyline est soumise, c'est-à-dire, la ligne 3 de l'algorithme 4. De manière plus précise, chaque requête skyline $Sky(X)$ est évaluée en appelant **Sky_X_from_Sky_Y**($T_{clean}, Sky(\mathcal{D}), X$). La ligne 3 de l'algorithme 4, c'est-à-dire, $S_2 \ltimes T_{clean}$, est exécutée comme suit : pour tout tuple $t \in S_2$, on sélectionne de T_{clean} les tuples t' satisfaisant $t[X] = t'[X]$. Grâce aux index bitmap $\mathcal{I}$, ces tuples sont retrouvés efficacement.

Algorithme 5 : SemiNaïvePartialSkyCube

Input : Table T
Output : $Sky(\mathcal{D}), T_{clean}$, index $\mathcal{I}$

```
1 Let S_D = Sky(T, D);
2 for every t ∈ T do
3     for every t' ∈ S_D do
4         if ∃D_i such that t[D_i] = t'[D_i] then
5             insert t into T_clean;
6             break;
7 create a bitmap index I on every D_i ∈ T_clean;
8 Return S_D, T_clean, index I;
```

Lorsque le skyline sur l'espace $\mathcal{D}$ est grand, matérialiser juste le topmost et l'index est clairement inefficace. En effet, le coût de la première étape de l'exécution des requêtes, c'est-à-dire, $Sky(Sky(T, \mathcal{D}), X)$, est presque le même que celui de l'exécution de $Sky(T, X)$. De ce fait, et dans une perspective d'optimisation, nous avons besoin de pré-calculer plus de skycuboids que juste $Sky(\mathcal{D})$.

La section suivante aborde le problème de la matérialisation partielle du skycube lorsque $Sky(\mathcal{D})$ est grand. Notre objectif sera de matérialiser une portion du

skycube suffisante pour répondre à toute requête skyline *sans avoir à accéder aux* données initiales.

3.2.3 Le sous-skycube Minimal Complet en Information (MICS)

Avant de présenter de manière formelle le problème abordé dans cette partie, nous commençons par définir quelques concepts. En commençant par la propriété de *complétude en information* d'un skycube partiel.

Définition 3.1 (Sous-skycube Complet en Information)**.** Soit S un sous-skycube de $\mathcal{S}$. S est un *Sous-skycube Complet en Information* (ICS) si et seulement si pour tout sous-espace X, il existe a sous-espace Y dont le skyline appartient S, $Sky(Y) \in S$, tel que $X \subseteq Y$ et $Sky(X) \subseteq Sky(Y)$.

Intuitivement, S est un ICS si et seulement s'il contient un ensemble suffisant de skycuboids permettant de répondre à n'importe quelle requête. Rappelons que $Sky(T, X) \subseteq Sky(T, Y) \Rightarrow Sky(T, X) = Sky(Sky(T, Y), X)$. En d'autres termes, $Sky(X)$ peut être évalué à partir de $Sky(Y)$ au lieu de T.

Exemple 12. On peut facilement vérifier que les sous-skycubes $S_1 = \{Sky(ABCD), Sky(ABD)\}$ et $S_2 = \{Sky(ABCD), Sky(ABD), Sky(AC)\}$ sont tous les deux des ICS. Si par exemple, S_1 est matérialisé, alors la requête $Sky(T, A)$ peut être exécutée comme suit $Sky(Sky(ABD), A)$ à partir de 4 tuples au lieu de l'ensemble de la table T qui contient 6 tuples. Remarquons que $S_3 = \{Sky(ABCD)\}$ n'est pas un ICS car, par exemple, il ne contient pas le skyline d'un sur-espace Y de ABD tel que $Sky(ABD) \subseteq Sky(Y)$.

Dans une perspective d'optimisation de l'espace de stockage, il est naturel d'essayer d'identifier les plus petits ICS.

Définition 3.2 (Complétude Minimale en Information)**.** S est un ICS *minimal* (MICS) si et seulement s'il n'existe pas un autre ICS S' tel que $S' \subset S$.

Exemple 13. $S_1 = \{Sky(ABCD), Sky(ABD)\}$ est plus petit que S_2 où $S_2 = \{Sky(ABCD), Sky(ABD), Sky(AC)\}$. On peut facilement vérifier que S_1 est l'unique MICS de T.

Nous pouvons à présent formaliser le problème de la matérialisation partielle du skycube.

Énoncé du problème : Étant donnée une table T et son ensemble de dimensions $\mathcal{D}$, trouver un MICS de T qui sera matérialisé dans le but de répondre à l'ensemble des requêtes sur les sous-espaces de $\mathcal{D}$.

La proposition suivante fait état de l'*unicité* du MICS pour toute table de données T.

Proposition 4. Étant donnée une table T, il existe un unique MICS de T.

Démonstration. Considérons le graphe orienté $G = (V, E)$ où $V = 2^{\mathcal{D}}$ et $E \subseteq V \times V$ tel que $(X, Y) \in E$ si et seulement si $X \subset Y$ et $Sky(X) \subseteq Sky(Y)$. Soit Σ l'ensemble des nœuds ne présentant aucune arrête sortante. Alors clairement $S = \{Sky(X) | \, X \in \Sigma\}$ est l'unique MICS. □

Identifier l'unique MICS est facile lorsque le skycube est calculé, cependant, ceci n'est pas efficace d'un point de vue pratique.

Dans la partie suivante de cette section, nous présentons une méthode, basée sur le concept de dépendances fonctionnelles, ayant pour but d'éviter la matérialisation complète du skycube. Pour cela, nous montrons que la présence de dépendances fonctionnelles implique l'inclusion entre skylines. Par conséquent, nous pouvons éviter de calculer certains d'entre eux.

3.2.4 Dépendances Fonctionnelles et Inclusion entre skylines

Rappelons que la dépendance fonctionnelle $X \rightarrow Y$ est satisfaite par T si et seulement si pour toute paire de tuples $t_1, t_2 \in T$, si $t_1[X] = t_2[X]$ alors $t_1[Y] = t_2[Y]$. Le théorème suivant représente notre principal résultat. Il montre comment l'existence d'une dépendance fonctionnelle peut être utilisée pour identifier une relation d'inclusion monotone entre deux skylines.

Théorème 3.3. *Soit $X \rightarrow Y$ une dépendance fonctionnelle satisfaite par T. Alors $Sky(T, X) \subseteq Sky(T, XY)$.*

Démonstration. Supposons que la déclaration du théorème soit fausse et soit $t \in Sky(T, X) \backslash Sky(T, XY)$. Puisque $t \notin Sky(T, XY)$, il devrait exister t' qui domine t sur l'espace XY, c'est-à-dire, $t' \prec_{XY} t$. t' ne peut pas dominer t au regard de X car autrement t ne pourrait pas appartenir à $Sky(T, X)$. Nous concluons que $t[X] = t'[X]$. D'autre part, $X \rightarrow Y$ est satisfaite (par hypothèse) et de $t[X] = t'[X]$ nous déduisons que $t[Y] = t'[Y]$ et donc $t[XY] = t'[XY]$ ce qui contredit le fait que t' domine t regard de XY. □

Exemple 14. Revenant à notre exemple illustratif, l'ensemble des dépendances fonctionnelles satisfaites par T est

$$\begin{array}{cccc} A \rightarrow B & A \rightarrow D & BD \rightarrow A & CD \rightarrow B \\ BC \rightarrow A & BC \rightarrow D & CD \rightarrow A & \end{array}$$

Les inclusions entre skylines identifiées par les dépendances fonctionnelles satisfaites par T sont représentées dans la figure 3.2. Chaque nœud X représente le skyline $Sky(T, X)$. Une arrête partant de X à Y représente une inclusion. Les chemins représentent également des inclusions entre skylines. Par exemple, nous pouvons observer que $Sky(T, A) \subseteq Sky(T, AB) \subseteq Sky(ABD)$. Il est nécessaire de

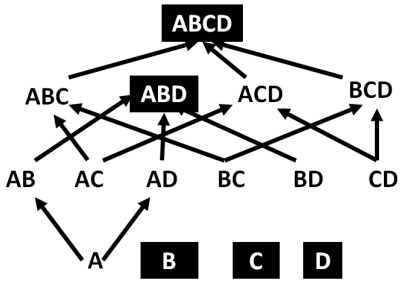


FIGURE 3.2 – Inclusions entre skycuboids revelées par les dépendances fonctionnelles

préciser que seules les inclusions décelables par la présence de dépendances fonctionnelles satisfaites par la table T sont représentées ici. Il pourrait exister d'autres relations d'inclusion qui ne sont pas liées à la présence de dépendances fonctionnelles. Par exemple, $Sky(C) = \{t_4\} \subseteq Sky(BC) = \{t_2, t_4\}$, et cette inclusion ne s'explique pas par les dépendances fonctionnelles satisfaites par la table T.

L'hypothèse de *valeurs distinctes* a déjà été considérée dans de précédents travaux Xia *et al.* [2012]; Pei *et al.* [2006]; Lee et won Hwang [2014] pour optimiser le calcul du skycube. Plus précisément, T satisfait la propriété de valeurs distinctes si et seulement si $|\pi_{D_i}(T)| = n$. En d'autres termes, toutes les valeurs apparaissant dans chaque dimension sont distinctes. Le résultat suivant montre que lorsque T satisfait cette propriété, il est garanti que la requête skyline est monotone.

Théorème 3.4. *Pei* et al. *[2006] Si T satisfait la propriété de valeurs distinctes alors, pour tous sous-espaces X et Y, l'implication suivante est vraie : $X \subseteq Y \Rightarrow Sky(X) \subseteq Sky(Y)$.*

Le précédent résultat peut être vu comme une conséquence du théorème 3.3. En effet, sous l'hypothèse de valeurs distinctes, chaque attribut pris individuellement détermine tous les autres attributs : il est une *clé* dans la terminologie des dépendances fonctionnelles. En particulier, $\forall X, Y$ nous avons $X \rightarrow Y$. Du théorème 3.3 nous déduisons que $Sky(X) \subseteq Sky(XY)$.

La proposition suivante montre comment les dépendances fonctionnelles peuvent être utilisées pour dériver un Skycube Complet en Information (ICS).

Proposition 5. Soit $\mathcal{I}$ l'ensemble des inclusions entre skylines dérivées à partir des dépendances fonctionnelles satisfaites par la table T et soit $\mathcal{G}_{\mathcal{I}} = (V, \mathcal{E})$ un graphe orienté où $V = 2^{\mathcal{D}}$ et $\mathcal{E} \subseteq V \times V$ tels que $(X, Y) \in \mathcal{E}$ si et seulement si $Sky(X) \subseteq Sky(Y) \in \mathcal{I}$. Soit $\Gamma = \{X \in V \mid X$ ne possède pas d'arrête sortante $\}$. Alors Γ est un ICS.

Démonstration. Il suffit de montrer que Γ inclut l'unique ICS minimal (MICS). Puisque l'ensemble des inclusions $\mathcal{I}$ est un sous-ensemble de l'ensemble des inclusions entre skylines possibles, un nœud du graphe G, comme défini dans la preuve de la proposition 4, ne possède pas d'arrête sortante seulement s'il n'en possède pas dans $\mathcal{G}_{\mathcal{I}}$. Ce qui prouve que Γ contient le MICS. □

Comme conséquence de la proposition précédente, nous pouvons déduire que disposer de Γ est suffisant pour déduire le MICS. Par exemple, dans la figure 3.2, les nœuds dans un fond noir forment Γ. B et ABD appartiennent à Γ. De la table 3.2, on peut observer que $Sky(B) \subseteq Sky(ABD)$. De ce fait, $Sky(B)$ sera retiré de Γ.

À présent, nous fournissons quelques propriétés concernant les éléments de Γ, propriétés qui nous permettrons de les identifier de manière efficace. En appliquant le théorème 3.3, nous concluons qu'un sous-espace X appartient à Γ si et seulement s'il n'existe pas de sous-espace Y tel que $X \cap Y = \emptyset$ et la dépendance fonctionnelle $X \rightarrow Y$ est satisfaite par T. Dans la littérature relative aux dépendances fonctionnelles, ces sous-espaces sont appelés des ensembles *clos* d'attributs. Nous rappelons brièvement la définition et suggérons un exemple de références Maier [1983]; Mannila et Räihä [1992] pour de plus amples détails à ce sujet.

Définition 3.3 (Sous-espace Clos)**.** Soit F l'ensemble couverture des dépendances fonctionnelles satisfaites par T et soit X un sous-espace de T. La *fermeture* de X relativement à F, notée X_F^+ ou simplement X^+, est le plus grand sous-espace Y tel que $F \vdash X \rightarrow Y$ où $\vdash$ représente l'*implication* entre les dépendances fonctionnelles. X est *clos* si et seulement si $X^+ = X$.

Exemple 15. Considerons notre exemple illustratif. A n'est pas clos car il existe un autre attribut déterminé par A. Par exemple, T satisfait $A \rightarrow D$. ABD est un clos car le seul attribut restant est C et T ne satisfait pas la dépendance $ABD \rightarrow C$.

Les éléments de Γ sont des sous-espaces clos. Par conséquent, disposer de l'ensemble des dépendances fonctionnelles satisfaites par T facilite la recherche du MICS. Il est important de préciser que les dépendances fonctionnelles dont il est fait mention ici sont celles qui sont satisfaites par l'instance T. Elles ne sont pas supposées connues à l'avance. Nous distinguons alors les dépendances fonctionnelles qui agissent comme contraintes qui sont satisfaites par toutes les instances de T et celles qui sont justes satisfaites par l'instance considérée. De ce fait, nous avons besoin d'un algorithme efficace qui permettra d'*extraire* les dépendances fonctionnelles satisfaites par l'instance T, dépendances fonctionnelles à partir desquelles nous dériverons les sous-espaces clos. Au meilleur de notre connaissance, aucun algorithme n'a été proposé jusque là dans la littérature pour résoudre le problème de la recherche des espaces clos. Des précédents travaux proposent plutôt des algorithmes efficaces pour le calcul de la fermeture d'un ensemble $\mathcal{F}$ de dépendances fonctionnelles, c'est-à-dire, toutes les dépendances fonctionnelles qui sont logiquement déduites de $\mathcal{F}$. Il est montré que cette fermeture peut être calculée en temps

linéaire de la taille de $\mathcal{F}$ Mannila et Räihä [1994]. La description de l'algorithme que nous proposons est donnée dans la section suivante.

3.2.5 Recherche des Sous-espaces Clos

L'algorithme naïf de recherche des espaces clos consisterait simplement à calculer la taille de chaque $\pi_X(T)$. S'il existe un espace Y, $Y \supset X$ tel que $|\pi_X(T)| = |\pi_Y(T)|$ alors X n'est pas clos parce que ceci signifie que la dépendance fonctionnelle $X \rightarrow Y \setminus X$ est satisfaite par T. Sinon, X est un espace clos. Cet algorithme n'est pas efficace parce qu'il requiert 2^d projections. Dans cette section nous développons un algorithme plus efficace dont le procédé de recherche élague des espaces pour lesquels il n'est pas nécessaire de faire une projection.

Notre algorithme peut se résumer comme suit : supposons que pour un attribut A, il nous est donné l'espace maximal X qui ne détermine pas A. Grâce à l'anti-monotonie des dépendances fonctionnelles, c'est-à-dire, si $X \not\rightarrow A$ alors $Y \not\rightarrow A$ pour tout $Y \subseteq X$, nous pouvons conclure que tous les sous-espaces de ces espaces maximaux sont potentiellement clos et tous les autres ne sont certainement pas clos. Dans le but de trouver efficacement ces espaces maximaux, notre algorithme essaye d'exploiter cette propriété d'anti-monotonie en évitant de tester les espaces qui peuvent être qualifiés à l'avance de non clos aussi bien que ceux qui sont potentiellement clos. Dans la suite de cette section, nous formalisons cette piste.

Nous commençons par présenter quelques lemmes permettant de caractériser les espaces clos. Pour cela, nous utilisons les notations suivantes :

— Det_{A_i} est l'ensemble des espaces minimaux X tels que $T \models X \rightarrow A_i$.
— $\mathcal{D}_i = \mathcal{D} \setminus A_i$.

Lemme 3.5. *Pour tout $X \in 2^{\mathcal{D}_i}$, s'il existe $X' \in Det_{A_i}$ tel que $X' \subseteq X$ alors X n'est pas un espace clos.*

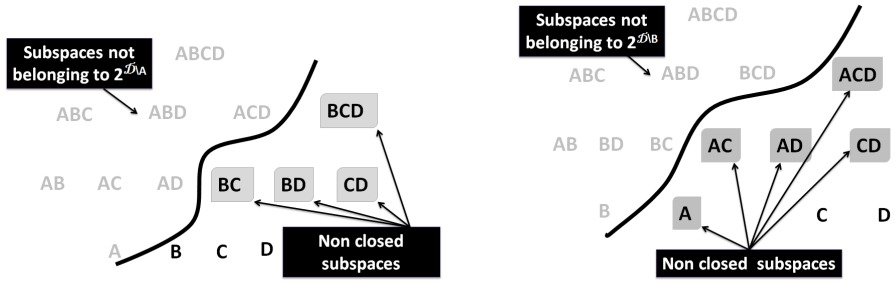
Démonstration. $X' \in Det_{A_i}$ signifie que $T \models X' \rightarrow A_i$. Par conséquent, $T \models X \rightarrow A_i$ et de ce fait $X^+ \ni A_i$ ce qui montre que X n'est pas un espace clos. □

Exemple 16. De l'exemple illustratif nous avons $Det_A = \{BD, CD, BC\}$. $BCD \in 2^{\mathcal{D}_A}$ n'est pas un espace clos car, par exemple, $BCD \supset BD$.

L'inverse du lemme précédent n'est pas vrai. En effet, même si $X \in 2^{\mathcal{D}_i}$ n'inclut aucun élément de Det_{A_i}, on a $X \not\rightarrow A_i$, ce qui n'implique pas nécessairement que X est clos car il est possible qu'il existe $A_j \neq A_i$ tel que $X \in 2^{\mathcal{D}_j}$ et $T \models X \rightarrow A_j$ faisant de X un espace non clos. En fait, il faut que la condition nécessaire et suffisante suivante soit satisfaite pour que X soit un espace clos.

Proposition 6. Soit $\mathcal{C}_i^-$ l'ensemble des espaces non clos dérivés par le lemme 3.5 et soit $\mathcal{C}^- = \bigcup_i \mathcal{C}_i^-$. Alors X est un espace clos si et seulement si

— $X = \mathcal{D}$ (l'espace complet) ou



(a) Espaces non clos relativement à A : B, C and D sont les espaces maximaux qui ne déterminent pas A. Par conséquent, tous leurs sur-espaces ne contenant pas A ne sont pas clos.

(b) Espaces nons clos relativement à B : C and D sont les espaces maximaux ne déterminant pas B. Tous les sur-espaces ne contenant pas B pris avec A ne sont pas clos.

FIGURE 3.3 – Espaces élagués au regard des attributs A et B : BCD est élagué par A et ne fait partie d'aucun des espaces cherchés associés à B, C et D. Par conséquent, $\pi_{BCD}(T)$ ne sera pas calculé. ACD est élagué par B. De ce fait, $\pi_{ACD}(T)$ ne sera pas calculé. $\pi_{AC}(T)$ est élagué par B car $A \rightarrow B$ est satisfaite et AC ne fait pas partie de $2^{D\setminus A}$ mais peut être testé avec D. De même que AD qui peut être testé avec C.

— $X \in 2^{\mathcal{D}} \setminus \mathcal{C}^-$.

Démonstration. Le premier item est évident donc nous prouvons le second. Nous commençons par prouver l'implication $X \in 2^{\mathcal{D}} \setminus \mathcal{C}^- \Rightarrow X$ est un espace clos : Par contradiction, soit $X \in 2^{\mathcal{D}} \setminus \mathcal{C}^-$, $X \neq \mathcal{D}$ et supposons que X n'est pas un espace clos. Dans ce cas, il devrait exister $A_i \notin X$ tel que $A_i \in X^+$. Par conséquent, il devrait exister $X' \subseteq X$ tel que $X' \in Det_{A_i}$ ce qui implique que $X \in \mathcal{C}_i^-$. Nous obtenons une contradiction. Montrons à présent que X clos $\Rightarrow X \in 2^{\mathcal{D}} \setminus \mathcal{C}^-$. Supposons le contraire, c'est-à-dire, X est un espace clos et $X \notin 2^{\mathcal{D}} \setminus \mathcal{C}^-$. Donc, il devrait exister A_i tel que $X \in \mathcal{C}_i^-$. Du lemme 3.5 nous concluons que X n'est pas un espace clos ce qui contredit l'hypothèse de départ. □

Exemple 17. À partir de l'exemple illustratif nous avons, $Det_A = \{BD, BC, CD\}$, $Det_B = \{A, CD\}$, $Det_C = \emptyset$ et $Det_D = \{A, BC\}$. De ces ensembles, nous déduisons $\mathcal{C}_A^- = Det_A \cup \{BCD\}$, $\mathcal{C}_B^- = Det_B \cup \{AC, AD, ACD\}$, $\mathcal{C}_C^- = Det_C \cup \emptyset$ et $\mathcal{C}_D^- = Det_D \cup \{AC, ABC\}$. Précisons par exemple que $ABD \notin \mathcal{C}_A^-$ même s'il est un sur-espace de BD. Rappelons également que pour $\mathcal{C}_A^-$ nous considérons uniquement les élements de $2^{\mathcal{D}_A}$ et ABD n'appartient pas à cet ensemble. Les figures 3.3(a) et 3.3(b) présentent une partie des espaces non clos que nous avons inférés à partir des espaces déterminants respectivement A et B.

La procédure **ClosedSubspaces** (c.f. algorithme 7) prend en entrée la table T et retourne les espaces clos. Comme on peut l'observer, la partie la plus critique de cet algorithme est l'instruction de la ligne 2 qui calcule l'ensemble des dépendances fonctionnelles non satisfaites.

Extraction des Dépendances Fonctionnelles Maximales non Satisfaites

Comme nous l'avons vu précédemment, tous les sous-espaces de la partie gauche d'une dépendance fonctionnelle non satisfaite sont potentiellement des espaces clos. Donc, étant donné un attribut A_i, la première étape de la procédure consisterait à déterminer l'espace maximal X tel que la dépendance fonctionnelle $X \rightarrow A_i$ n'est pas satisfaite par T. Il existe dans la littérature divers algorithmes pour le calcul de l'ensemble des dépendances fonctionnelles *minimales*, par exemple Yao et Hamilton [2008]; Lopes *et al.* [2000]; Huhtala *et al.* [1999]; Novelli et Cicchetti [2001]. Déterminer à partir de ceci, l'ensemble des dépendances fonctionnelles *maximales* non satisfaites peut être réalisé en calculant les hypergraphes transversaux minimaux Eiter et Gottlob [1995]. Puisque le calcul des minimaux transversaux est en général difficile, déterminer sa complexité précise est également un problème ouvert et aucun algorithme polynomial n'a été proposé jusque là pour résoudre ce problème dans le cas général, nous utilisons alors l'algorithme **MaxNFD** (mis pour partie gauche de Dépendances Fonctionnelles Maximales Non satisfaites). Il s'agit d'une adaptation de l'algorithme proposé dans Hanusse et Maabout [2011] recherchant les éléments fréquents maximaux.

MaxNFD est présenté dans l'algorithme 6. Il peut être décrit comme suit : soit X un candidat pour lequel nous voulons vérifier que $T \not\models X \rightarrow A_i$. Si (i) X vérifie cette assertion, alors il est possible qu'il soit un des espaces maximaux recherchés. Par conséquent, il est ajouté à **Max** et son *parent* est généré comme candidat à la prochaine itération. Le *parent* de X est simplement le sur-ensemble successeur de X dans l'ordre lexicographique. Par example, le parent de BDF est $BDFG$. Si (ii) au contraire X ne vérifie pas l'assertion , c'est-à-dire, $T \models X \rightarrow A_i$, alors (a) ses *fils* (*children*) seront candidats à la prochaine itération et (b) son *frère* (*sibling*) est candidat pour l'itération suivante. Par exemple, les *fils* de BDF sont BF et DF, c'est-à-dire, tous les sous-espaces de BDF contenant un attribut en moins mais qui n'en sont pas des préfixes (BD n'est pas un fils de BDF). Le *frère* de BDF est BDG, c'est-à-dire, F est remplacé par son successeur G. Si $|\mathcal{D}| = d$, il est prouvé dans Hanusse et Maabout [2011] qu'au plus $2d - 1$ itérations sont nécessaires pour chaque attribut A_i pour déterminer les espaces maximaux X qui ne déterminent pas A_i. Ce qui explique la boucle **While** de la ligne 3 de l'algorithme 6. La preuve de l'algorithme a déjà été établie dans Hanusse et Maabout [2011]. Une propriété importante de **MaxNFD** est sa souplesse pour être exécuté en parallèle. En effet, la boucle **Foreach** de la ligne 4 de l'algorithme montre que tous les sous-espaces qui appartiennent à l'ensemble $Candidates[k]$ peuvent être testés en parallèle.

Déduction des Sous-espaces Clos

Les sous-espaces retournés par la procédure précédente sont *potentiellement* clos. En effet, X est clos si et seulement s'il ne détermine aucun $A_i \in X$. Il ne suffit pas de faire une intersection entre ces ensembles car, par exemple, aucun de ces espaces X relatifs à A_i ne contient A_i, mais il pourrait encore exister un en-

Algorithme 6 : MaxNFD

Input : Table T, Target attribute A_i
Output : Maximal X s.t $T \not\models X \rightarrow A_i$

```
1  Candidates[1] ← {A_1};
2  k ← 1;
3  while k ≤ (2d − 1) do
4      foreach X ∈ Candidates[k] do
5          // Loop executed in parallel
6          if ∄Y ∈ Max st Y ⊇ X then
7              if T ⊭ X → A_i then
8                  Add X to Max;
9                  Remove the subsets of X from Max;
10                 Add the parent of X to Candidates[k + 1];
11             else
12                 Candidates[k + 1] ⊎ RightChildren(X);
13                 Candidates[k + 2] ⊎ RightSibling(X);
14     k ← k + 1;
15 Return Max;
```

semble clos contenant A_i. L'algorithme **ClosedSubspaces** exploite le résultat précédent pour déduire les espaces clos.

3.2.6 Calcul du Sous-skycube Minimal à Information Complète (MICS)

À présent, nous pouvons décrire la procédure qui, étant donnée la table T, retourne le MICS correspondant. Cette procédure est présentée dans l'algorithme 8. Cet algorithme commence par déterminer les sous-espaces clos, ensuite il calcule leurs skylines respectifs. La troisième étape consiste à retirer chaque sous-espace X dont le skyline est inclus dans celui d'un autre Y qui l'inclut.

Exemple 18. Rappelons que les espaces clos de notre exemple illustratif sont $ABCD$, ABD, B, C et D. L'algorithme 8 commence par calculer leurs skylines respectifs. Ensuite, il découvre les relations d'inclusions suivantes $Sky(B) \subseteq Sky(ABD)$, $Sky(C) \subseteq Sky(ABCD)$, $Sky(D) \subseteq Sky(ABD)$ et $Sky(D) \subseteq Sky(ABCD)$. Par conséquent, seuls $Sky(ABCD)$ et $Sky(ABD)$ appartiennent au MICS. C'est alors l'ensemble minimal de skycuboids qui sera matérialisé.

Algorithme 7 : ClosedSubspaces

Input : Table T
Output : Closed subspaces

```
for i = 1 to d do
    L⁻ = MaxNFD(T, A_i);
    L⁻ = SubsetsOf(L⁻, A_i);
    if i = 1 then
        Closed = L⁻;
    else
        Closed_i = {X ∈ Closed | X ∋ A_i};
        Closed = (Closed ∩ L⁻) ⋃ Closed_i;
Return Closed;
```

Algorithme 8 : MICS

Input : Table T
Output : MICS

```
Let ClosedSub = ClosedSubspaces(T);
for every X ∈ ClosedSub do
    // Loop executed in parallel
    compute Sky(T, X);
for every X ∈ ClosedSub do
    for every Y ∈ ClosedSub s.t X ⊂ Y do
        if Sky(T, X) ⊆ Sky(T, Y) then
            ClosedSub = ClosedSub \ {X};
            Break;
Return ClosedSub and their respective skylines;
```

3.2.7 Analyse du nombre d'Espaces Clos

Évidemment, plus il y a de dépendances satisfaites par la table T, moins il y a d'espaces clos. Le nombre de valeurs distinctes par dimension est un paramètre important qui a été identifié dans la littérature à propos de la recherche de dépendances fonctionnelles, dans ce sens qu'il impacte le nombre de dépendances fonctionnelles valides [1]. Intuitivement, lorsque ce paramètre est grand, il devient plus difficile d'un point de vue statistique, de trouver deux tuples partageant la même valeur en X et des valeurs différentes en Y afin de rendre non satisfaite la dépendance fonctionnelle $X \rightarrow Y$. Dans le but d'illustrer ce propos, supposons que chaque dimension possède n valeurs distinctes, c'est-à-dire, le nombre de lignes de T. Dans ce cas extrême, il est impossible de trouver des dépendances fonctionnelles non satisfaites (toutes les dépendances fonctionnelles sont satisfaites). En guise d'autre exemple extrême, supposons à présent que chaque dimensions possède uniquement deux valeurs distinctes. Si n est assez grand, c'est-à-dire, $n \gg 2^d$ alors, pour tous X, Y il y a une grande probabilité qu'il existe deux tuples partagent la même valeur en X et différents en Y rendant non satisfaite par T la dépendance fonctionnelle $X \rightarrow Y$. Ce résultat intuitif peut être formalisé sous certaines hypothèses concernant la distribution des données.

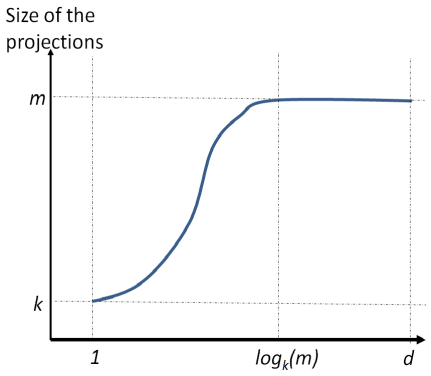
Soit $||X||$ le nombre d'attributs dans X, par exemple $||ABC|| = 3$, et soit k le nombre de valeurs distinctes par dimension et m le nombre de tuples distincts dans la table T. X est une clé de T si et seulement si $|X| = m$. Clairement, si X est une clé alors, tout espace Y tel que $Y \supseteq X$, Y n'est pas un espace clos (mis à part le cas spécial où Y est l'espace complet).

Supposons une distribution aléatoire uniforme des valeurs au sein des dimensions, si $||X|| \geq \log_k(m)$ alors il y a de fortes chances que X soit une clé Harinarayan *et al.* [1996]. De plus, puisque $X \supseteq Y \Rightarrow |X| \geq |Y|$, la probabilité pour X d'être une clé augmente lorsque $||X||$ augmente. Nous concluons que :

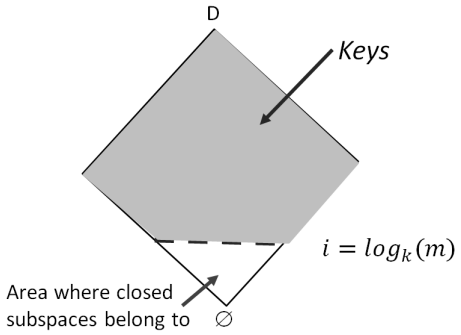
lorsque k augmente, un moindre nombre d'attributs $||X||$ suffit pour faire d'un espace X une clé, plus grand est le nombre de sur-espaces de X et moins il y a d'espaces clos.

La figure 3.4(a) montre l'évolution de la taille des projections par rapport au nombre de dimensions : lorsque ce nombre atteint $\log_k(m)$ les projections atteignent le nombre de tuples distincts de T ce qui signifie qu'elles sont des clés. Plus nous avons de clés, moins il y a de clos comme illustré dans la figure 3.4(b). Notons que la zone où les espaces clos apparaissent pourrait également contenir des espaces non clos.

1. Il est souvent appelé *facteur de corrélation* dans la littérature relative à la recherche des dépendances fonctionnelles.



(a) Taille des projections au regard de la dimensionalité $||X||$



(b) Les espaces clos dans le treillis des espaces

FIGURE 3.4 – Comportements des clés et des espaces clos

3.2.8 Analyse de la Taille du Skyline

Lorsque la cardinalité k des dimensions décroît, le nombre d'espaces clos augmente et pourrait atteindre $2^d - 1$. D'un point de vue matérialisation partielle, ceci est une mauvaise situation : tous les skylines seront à matérialiser. En fait, contrairement au nombre d'espaces clos, la taille des skylines est positivement liée à k. Ceci indique que lorsque k est petit, nous n'avons pas besoin de matérialiser d'autres skycuboids que le topmost skyline, c'est-à-dire, $Sky(D)$. Celui-ci est suffisant pour répondre à l'ensemble des requêtes skyline de manière efficace en faisant usage de l'algorithme 4 que nous décrivons dans la section 3.2.2. Le théorème suivant formalise ce résultat.

Définition 3.4. Soit $\mathscr{T}_k$ l'ensemble de toutes les tables T possédant d dimensions independantes, n tuples et au plus k valeurs distinctes par dimension (les valeurs de chaque dimension appartiennent à l'ensemble $\{1, 2, \ldots, k\}$). Les tuples de T ne sont pas nécessairement distincts.

Définition 3.5. Soit $\Pi(T)$ une table présentant les mêmes tuples que T mais dans laquelle chaque valeur de tuple apparaît une seule fois ; il s'agit de la projection de la table T sur l'espace complet.

Soit $Sky(T)$ le skyline de T sur l'ensemble de ses dimensions, c'est-à-dire, $Sky(T) = Sky(T, \mathcal{D})$, et soit $S(T)$ la taille de $Sky(T)$, c'est-à-dire, $S(T) = |Sky(T)|$.

Définition 3.6. Soit $\overline{S(T)}_{T \in \mathscr{T}_k}$ la moyenne (l'espérance) de $S(T)$ où T appartient à $\mathscr{T}_k$.

Théorème 3.6. *Nous avons le résultat suivant*

$$k \leq k' \Rightarrow \overline{S(\Pi(T))}_{T \in \mathscr{T}_k} \leq \overline{S(\Pi(T))}_{T \in \mathscr{T}_{k'}}$$

En d'autres termes, le théorème précédent déclare que pour n et d fixés, le nombre de tuples distincts apparaissant dans le skyline tend à augmenter lorsque le nombre de valeurs distinctes par dimension augmente. Dans le but de prouver cette assertion, les définitions et lemmes suivants sont énoncés.

Définition 3.7. Soit f une relation qui associe à chaque entier a la valeur aléatoire $2a - X$ où X est une variable aléatoire suivant la loi de Bernoulli de paramètre $\frac{1}{2}$. En d'autres termes, $f(a)$ peut être égal à $2a - 1$ ou à $2a$ avec la même probabilité. Soit $F(T)$ l'ensemble de toutes les tables qui peuvent être obtenues par application de f à toutes les valeurs de T. Soit $T_k \in \mathscr{T}_k$. Alors
$F(T_k) = \{T_{2k} \in \mathscr{T}_{2k} : T_{2k}[i,j] = f(T_k[i,j]), \forall_{i,j} 1 \leq i \leq n \text{ et } 1 \leq j \leq d\}$

Lemme 3.7. *L'ensemble* $\{F(T_k), T_k \in \mathscr{T}_k\}$ *est une partition de* $\mathscr{T}_{2k}$.

Démonstration. À chaque table $T_k \in \mathscr{T}_k$ est associée un ensemble $F(T_k)$ de tables $T_{2k} \in \mathscr{T}_{2k}$ tel que T_{2k} résulte de T_k par appliation de f à chacune de ses valeurs. Réciproquement, toute table $T_{2k} \in \mathscr{T}_{2k}$ est associée à une seule table $T_k \in \mathscr{T}_k$ (on retrouve T_k en considérant pour chaque valeur la partie entière du résultat de la division de la valeur correspondant dans T_{2k} par 2, $f^{-1}(a) = \lceil \frac{a}{2} \rceil$). La taille de $\mathscr{T}_k$ est donnée par $|\mathscr{T}_k| = k^{n \cdot d}$. En effet, chaque élément (i,j) de T_k peut prendre k valeurs et puisqu'il y a $n \times d$ éléments, on obtient $k^{n \cdot d}$. Par conséquent, $|\mathscr{T}_{2k}| = (2k)^{n \cdot d}$. D'autre part, pour tout T_k, nous avons $|F(T_k)| = 2^{n \cdot d}$ car chaque valeur de T_k peut être mappé par deux valeurs possibles dans T_{2k}. Clairement, $\bigcup_{T_k} F(T_k) \subseteq \mathscr{T}_{2k}$. Soit $T_{2k} \in \mathscr{T}_{2k}$. Montrons à présent qu'il existe une table T_k telle que $T_{2k} \in F(T_k)$. Soit T la table telle que chaque élément (i,j) de T_{2k} est remplacé par $\lceil T_{2k}[i,j]/2 \rceil$. T est dans $\mathscr{T}_k$ et clairement $T_{2k} \in F(T)$. Soit $T' \in \mathscr{T}_k$ et $T' \neq T$. Montrons que $F(T) \cap F(T') = \emptyset$. Puisque $T \neq T'$ alors il existe un élément (i,j) tel que $T[i,j] \neq T'[i,j]$. Soit $a = T[i,j]$ et $a' = T'[i,j]$. Par l'absurde, soit T" $\in F(T) \cap F(T')$. Ceci signifie que deux valeurs a et a' peuvent être associées à la même valeur a" de T". Donc, de T nous avons a" $\in \{2a - 1, 2a\}$ et de T' nous avons a" $\in \{2a' - 1, 2a'\}$. L'intersection de ces deux ensembles est non vide seulement si $a = a'$, ce qui contredit le fait que $a \neq a'$. □

Lemme 3.8. $\forall\, T_k \in \mathscr{T}_k, \forall\, T_{2k} \in F(T_k)$*, la taille du skyline* $S(\Pi(T_k))$ *est inférieure ou égale à* $S(\Pi(T_{2k}))$.

$$S(\Pi(T_k)) \leq S(\Pi(T_{2k}))$$

Démonstration. Supposons le cas général où chaque dimension j de la table T contient k_j valeurs distinctes. Si $S = S(\Pi(T))$ croît lorsque la cardinalité d'une seule dimension augmente alors par repetition de l'opération pour chacune des autres dimensions, la taille du skyline croît. Sans nuire à la généralité, supposons que f est appliqué à D_1. Soit $T^{(1)}$ la table obtenue. La première dimension de $T^{(1)}$ contient $2k_1$ valeurs distinctes. Soit $S^{(1)} = S(\Pi(T^{(1)}))$ le nombre de tuples distincts du skyline de $T^{(1)}$. Soit t un tuple appartenant à $\Pi(T)$, et soient t' et $\widehat{t}$ les deux tuples possibles images de t par f tels que $t'[D_1] = 2t[D_1]$ et $\widehat{t}[D_1] =$

$2t[D_1] - 1$, en d'autres termes $f(t) \in \{t', \widehat{t}\}$. Pour la suite, $f(t)$ signifiera indépendament t' ou $\widehat{t}$ apparaissant en tant qu'image de t par f. Il est évident que si $t \in Sky(\Pi(T)) \Rightarrow f(t) \in Sky(\Pi(T^{(1)}))$ alors $S^{(1)} \geq S$. De ce fait, il suffit de montrer que $t \in Sky(\Pi(T)) \Rightarrow f(t) \in Sky(\Pi(T^{(1)}))$.

Par l'absurde

$f(t) \notin Sky(\Pi(T^{(1)})) \Rightarrow \exists u \in \Pi(T)$ tel que $f(u) \prec f(t)$. Cependant $t \in Sky(\Pi(T))$ donc $u \not\prec t$. Nous devons considérer les trois cas de figure suivants :

- $u[1] = t[1]$. Nous savons que $f(u)[2 \dots d] = u[2 \dots d]$, $f(t)[2 \dots d] = t[2 \dots d]$ et $u[2 \dots d] \not\prec t[2 \dots d]$ (autrement u domine t). On conclut que $f(u)[2 \dots d] \not\prec f(t)[2 \dots d]$. De ce fait, $f(u)[2 \dots d] \not\prec f(t)[2 \dots d]$ et $f(u) \prec f(t) \Rightarrow f(u)[2 \dots d] = f(t)[2 \dots d] \Rightarrow (u = t)$. Ce qui constitue une contradiction car $u \neq t$ puisque tous les tuples sont distincts dans $\Pi(T_k)$.
- $u[1] < t[1] \Rightarrow f(u)[1] < f(t)[1]$. Cependant, $f(u)[2 \dots d] = u[2 \dots d]$ et $f(t)[2 \dots d] = t[2 \dots d]$. Alors $f(u) \prec f(t) \Rightarrow u \prec t$ ce qui crée une contradiction car $u \not\prec t$.
- $u[1] > t[1] \Rightarrow f(u)[1] > f(t)[1]$. Alors $f(u) \not\prec f(t)$ ce qui est contradictoire car $f(u) \prec f(t)$.

Nous pouvons conclure que $S \leq S^{(1)}$. Soit $S^{(j)}$ la taille du skyline de la table $\Pi(T^{(j)})$ qui constitue la projection sur D de la table T dans laquelle f a été appliquée à toutes les j premières dimensions ($S^{(0)} = S(\Pi(T)), S^{(1)} = S(\Pi(T^{(1)})), \dots, S^{(d)} = S(\Pi(T^{(d)}))$). Considérons $T = T_k$, alors nous obtenons la relation suivante

$$S(\Pi(T_k)) = S^{(0)} \leq S^{(1)} \leq S^{(2)} \leq \dots \leq S^{(d)} = S(\Pi(T_{2k}))$$

□

Démonstration. (du **Théorème** 3.6) On montre que l'espérance $\overline{S(\Pi(T))}_{T \in \mathscr{T}_k}$ de la variable aléatoire $S(\Pi(T_k))$ est inférieure à $\overline{S(\Pi(T))}_{T \in \mathscr{T}_{k'}}$ pour tout $k' = k(1 + \alpha)$ où α est un entier positif. Pour plus de commodité, on montrera le résultat pour $\alpha = 1$. La même preuve pouvant être établie pour α choisi arbitrairement.

$$\overline{S(\Pi(T))}_{T \in \mathscr{T}_k} = \frac{1}{k^{n \cdot d}} \sum_{T_k \in \mathscr{T}_k} S(\Pi(T_k))$$

$$\overline{S(\Pi(T))}_{T \in \mathscr{T}_{2k}} = \frac{1}{(2k)^{n \cdot d}} \sum_{T_{2k} \in \mathscr{T}_{2k}} S(\Pi(T_{2k}))$$

$$S(\Pi(T_k)) \leq S(\Pi(T_{2k})) \; \forall \, T_{2k} \in f(T_k) \Rightarrow S(\Pi(T_k)) \leq \frac{1}{2^{n \cdot d}} \sum_{T_{2k} \in f(T_k)} S(\Pi(T_{2k}))$$

En appliquant E (Opérateur espérance) de part et d'autre, on obtient

$$\begin{aligned}\frac{1}{k^{n\cdot d}}\sum_{T_k\in\mathscr{T}_k} S(\Pi(T_k)) &\leq \frac{1}{k^{n\cdot d}}\sum_{T_k\in\mathscr{T}_k}\frac{1}{2^{n\cdot d}}\sum_{T_{2k}\in f(T_k)} S(\Pi(T_{2k}))\\ &\leq \frac{1}{(2k)^{n\cdot d}}\sum_{T_k\in\mathscr{T}_k}\sum_{T_{2k}\in f(T_k)} S(\Pi(T_{2k}))\\ &\leq \frac{1}{(2k)^{n\cdot d}}\sum_{T_{2k}\in\mathscr{T}_{2k}} S(\Pi(T_{2k}))\end{aligned}$$

$$\overline{S(\Pi(T))}_{T\in\mathscr{T}_k} = E(S(\Pi(T_k))) \leq E(S(\Pi(T_{2k}))) = \overline{S(\Pi(T))}_{T\in\mathscr{T}_{2k}}$$

$$\overline{S(\Pi(T))}_{T\in\mathscr{T}_k} \leq \overline{S(\Pi(T))}_{T\in\mathscr{T}_{k'}}$$

Ce qui conclut la preuve du théorème. □

En d'autres termes, le théorème précédent établit que pour n et d fixés, la taille du skyline (en termes de nombre de tuples distincts) tend à augmenter lorsque le nombre de valeurs distinctes par dimension augmente. L'exemple suivant donne une illustration plus intuitive.

Exemple 19. Pour k et d donnés, le nombre de tuples possibles est k^d. Soit $n = 5$ et $d = 2$. Lorsque $k = 2$, nous somme sûr qu'il existe des doublons parmi les 5 tuples. De plus, il y a de fortes chances que le tuple $\langle 0, 0\rangle$ soit présent et dans ce cas, c'est l'unique tuple distinct du skyline. Considérons à présent le cas où $k = 4$. Le nombre de tuples distincts possibles passe alors de 4 à 16. Choisir 5 tuples parmi ces 16 tuples laisse moins de chance au tuple $\langle 0, 0\rangle$ de faire partie de la table. Par conséquent, le nombre de tuples dans le skyline tend à être supérieur à 1.

La figure 3.5 illustre comment évoluent à la fois le nombre d'espaces clos et la taille du skyline, lorsque varie le nombre de valeurs distinctes par dimension k avec $n < k^d$. La leçon à en tirer est celle selon laquelle, lorsque le nombre d'espaces clos augmente, la taille du skyline tend à diminuer. C'est pourquoi dans le cas où les données ne satisfont aucune dépendance fonctionnelle, nous sommes presque sûrs que le topmost skyline est de petite taille.

3.3 Evaluation des requêtes

Dans cette partie, nous montrons comment les requêtes skyline sont efficacement exécutées lorsque le MICS est matérialisé. Dans le but de simplifier la suite, on supposera que les skycuboids matérialisés sont uniquement ceux associés aux espaces clos. Celui-ci constitue un sur-ensemble du MICS, et est noté **SkycubeC**(T).

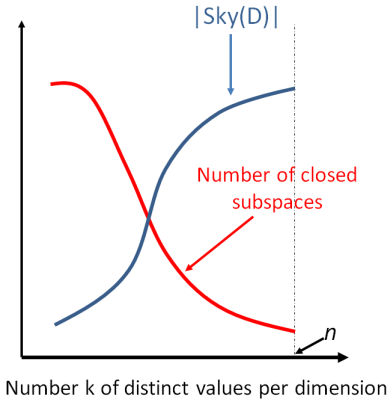


FIGURE 3.5 – L'évolution de la taille du skyline et du nombre de clos par rapport à la cardinalité k

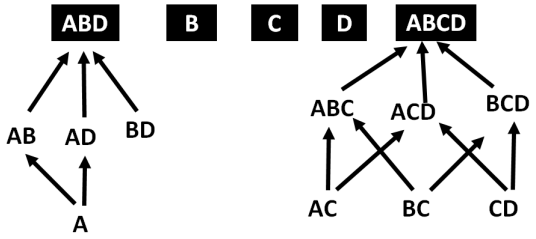


FIGURE 3.6 – Le Skycube Partiel

3.3.1 Calcul de $Sky(Sky(X^+), X)$

Lorsqu'une requête $Sky(T, X)$ est soumise, on commence par calculer la fermeture X^+ afin de déterminer le skyline ancêtre matérialisé à partir duquel la requête pourra être calculée. Par exemple, la requête $Sky(T, A)$ sera calculée à partir de $Sky(T, A^+) = Sky(T, ABD)$. En réalité, $Sky(T, ABD) = \{t_1, t_2, t_3, t_4\}$. Par conséquent, au lieu de calculer $Sky(T, A)$ à partir de T (nécessitant de comparer 6 tuples entre eux), on se base sur $Sky(T, ABD)$ contenant *uniquement* 4 tuples. La fermeture de chaque espace X peut également être codée en dur ou peut être calculée à la volée en utilisant l'ensemble des dépendances fonctionnelles qui ont déjà été extraites. Pour l'exemple illustratif, la figure 3.6 représente la partie matérialisée du skycube (les nœuds dans les boites noires) aussi bien que les relations de fermeture.

Nous faisons à présent un point sur l'analyse plus profonde des complexités. On rappelle tout d'abord que les algorithmes proposés jusque là, pour l'exécution d'une requête skyline à partir d'une table de données de taille n tuples ont une complexité dans le pire des cas de l'ordre de $O(n^2)$ qui reflète le nombre de comparaisons effectuées. Alors, nous pouvons espérer qu'en utilisant la matérialisation partielle, le coût de l'exécution des requêtes skyline soit d'une complexité inférieur à $O(n^2)$.

Supposons que la requête soit $Sky(X)$ et que $Sky(X^+)$ soit matérialisé. De ce fait, exécuter $Sky(X)$ est réalisé avec une complexité en temps de $O(|Sky(X^+)|^2)$. A moins que $X^+ = \mathcal{D}$ le coût de l'exécution de la requête à partir de $Sky(X^+)$ sera toujours strictement inférieur à celui de la requête revenant à la table de données initiale ; c'est l'objet de la section suivante.

3.3.2 Utilisation des Partitions

Dans le but d'optimiser l'exécution des requêtes skyline, dans cette partie, nous mettons en exergue quelques propriétés pouvant être déduites des dépendances fonctionnelles satisfaites par T. Ces propriétés sont basées sur le partitionnement du skyline.

Définition 3.8 (Partition)**.** Soit Y un espace tel que $Y^+ = X$. Alors $\mathcal{P}_Y(X) = \{p_1, \ldots, p_m\}$ dénote le partitionnement des tuples de $Sky(X)$ relativement à leurs valeurs dans le sous-espace Y, c'est-à-dire, $t_i \equiv t_j$ si et seulement si $t_i[Y] = t_j[Y]$. Soit $\tau_i \in p_i$, alors $[\mathcal{P}_Y(X)] = \cup_{i=1\ldots m}\{\tau_i\}$ est un *représentant* de $\mathcal{P}_Y(X)$.

Exemple 20. $A^+ = ABD$ et $Sky(ABD) = \{t_1, t_2, t_3, t_4\}$. $\mathcal{P}_A(ABD) = \{\{t_1, t_2\}; \{t_3\}; \{t_4\}\}$. $[\mathcal{P}_A(ABD)] = \{t_1, t_3, t_4\}$ et $[\mathcal{P}_A(ABD)] = \{t_2, t_3, t_4\}$ sont deux représentants de $\mathcal{P}_A(ABD)$.

Le lemme suivant montre que calculer $Sky(Y)$ peut être effectué à partir de $[\mathcal{P}_Y(X)]$ au lieu de $Sky(X)$.

Lemme 3.9. *Soit Y un espace tel que $Y^+ = X$ et soit $[\mathcal{P}_Y(X)]$ un représentant. Soit $t_i \in [\mathcal{P}_Y(X)]$ et soit t'_i un tuple de $Sky(X)$ tel que $t_i[Y] = t'_i[Y]$. Alors $t'_i \in Sky(Sky(X), Y)$ si et seulement si $t_i \in Sky([\mathcal{P}_Y(X)], Y)$.*

En d'autres termes, il suffit de calculer $Sky(\pi_Y(Sky(X)), Y)$ et si t_i est dans le résultat alors tous les tuples qui lui sont équivalents dans $Sky(X)$ appartiennent également à $Sky(Y)$.

Exemple 21. Supposons que pour calculer $Sky(A)$, le représentant $[\mathcal{P}_A(ABD)] = \{t_2, t_3, t_4\}$ soit utilisé. Ceci signifie que la requête $Sky([\mathcal{P}_A(ABD)], A)$ est exécutée et retourne $\{t_2\}$. Puisque t_1 est équivalent à t_2 alors t_1 fait également partie de $Sky(A)$.

Le lemme précédent suggère que pour chaque requête $Sky(Y)$, que l'on partitionne $Sky(X)$ relativement à Y. En fait, il serait plus simple et pas moins correct de partitionner une seule fois les tuples de $Sky(X)$ relativement à X et cette partition est exactement la même que celles que l'on obtiendrait pour tout Y tel que $Y^+ = X$.

Proposition 7. Soit $Y^+ = X$. Alors $\mathcal{P}_Y(X) = \mathcal{P}_X(X)$.

Démonstration. Car $T \models X \rightarrow Y$ si et seulement si $\mathcal{P}_X(T) = \mathcal{P}_{XY}(T)$. □

Par exemple, puisque $A^+ = ABD$ les partitions $\mathcal{P}_A(ABD)$ et $\mathcal{P}_{ABD}(ABD)$ sont identiques.

Le résultat précédent montre que la complexité du calcul d'un skycuboid non matérialisé ne dépend pas du nombre de tuples contenus dans ses ancêtres matérialisés mais plutôt de la taille de leur partition. Notons cependant que la propositions 7 ne répond pas complètement à notre précécdente interrogation puisque à priori, le nombre de parties pourrait être égal à $|Sky(X)|$ et puisque $|Sky(X)|$ lui même pourrait être égal à $n = |T|$, alors on ne gagnerait pas à évaluer le skyline à partir de $Sky(X)$ plutôt qu'à partir de T. La proposition suivante montre qu'en fait ceci ne pourrait arriver que dans un seul cas, lorsque $X = \mathcal{D}$. Plus précisement,

Proposition 8. Soit X un espace clos et soit $\mathcal{P}_X(X)$ la partition $\{p_1, \ldots, p_m\}$. Si $X \neq \mathcal{D}$ alors $m < |T|$.

Démonstration. Supposons que le nombre de parties soit égal à $|T| = n$. Ceci signifie qu'il y a n valeurs distinctes lorsque les tuples de $Sky(X)$ sont projetés sur X. De ce fait, X est une clé de T. Par conséquent, $X = \mathcal{D}$, ce qui contredit l'hypothèse de départ. □

Dans Raïssi *et al.* [2010], une classe spéciale de skycuboid a été identifiée, appelée la classe des skycuboids de *Type I*. Le auteurs utilisent cette classe pour déterminer le contenu des autres skycuboids. À travers le lemme 3.9 nous déduisons certains skylines sans aucun calcul lorsque nous sommes en présence de skycuboids de Type I. Rappelons tout d'abord la définition de ce concept.

Définition 3.9. $Sky(X)$ est de Type I si et seulement si $\forall t_1, t_2 \in Sky(X), t_1[A_i] = t_2[A_i]$ pour tout $A_i \in X$.

En d'autres termes, $Sky(X)$ est de Type I si et seulement si la projection de $Sky(X)$ sur X, c'est-à-dire, $\pi_X(Sky(X))$ possède un seul élément.

Exemple 22. De notre exemple courant, $Sky(AD) = \{t_1, t_2\}$ et c'est un skycuboid de Type I car $t_1[AD] = t_2[AD]$.

La proposition suivante montre que sous certaines conditions relatives aux dépendances fonctionnelles, certains skylines peuvent être dérivés très simplement.

Proposition 9. Si $Sky(X)$ est de Type I alors pour tout $Y \subseteq X$ tel que $Y \rightarrow X$, nous avons $Sky(Y) = Sky(X)$.

Démonstration. Ceci est la conséquence directe du lemme 3.9. En effet, si $Sky(X)$ est de Type I, alors la partition de ses éléments au regard de X ne présente qu'une seule partie. Par conséquent, tout tuple de cette partie est nécessairement dans $Sky(Y)$. Puisque $Y \subseteq X$, et $Y \rightarrow X$ alors $Sky(Y) \subseteq Sky(X)$ et nous concluons que $Sky(X) = Sky(Y)$. □

Exemple 23. Puisque $Sky(AD)$ est de Type I et puisque $A \rightarrow D$ est une dépendance fonctionnelle satisfaite par T, on conclut que $Sky(A) = Sky(AD)$.

3.3.3 Exécution de l'opération $Sky(Sky(Y), X)$

Lorsque seuls les skycuboids appartenant au MICS sont matérialisés, il pourrait arriver que $Sky(X^+)$ ne soit pas matérialisé pour certains X car nous avions observé qu'il pourrait exister un sur-espace clos Y de X tel que $Sky(X^+) \subseteq Sky(Y)$. Dans ces cas, les propriétés que nous avons développées dans la section précédente ne sont plus valables. Alors, nous avons besoin de faire usage d'algorithmes standards de skyline pour déterminer $Sky(X)$ à partir de $Sky(Y)$ (matérialisé) sans pouvoir exploiter toutes les optimisations que nous avons présentées jusque là.

Exemple 24. La table 3.2 montre que pour les espaces clos C et $ABCD$, nous avons par exemple $Sky(C) \subseteq Sky(ABCD)$. De ce fait, $Sky(C)$ n'appartient pas au MICS(T). Par conséquent, si un utilisateur demande à calculer $Sky(C)$, nous auront alors besoin de procéder ainsi : $Sky(C) = Sky(Sky(ABCD), C)$. Notons que tous les tuples de $Sky(ABCD)$ sont distincts entre eux. Donc la partition de $Sky(ABCD)$ au regard de $ABCD$ contiendrait 3 parties tandis que celle du même skyline au regard de C ne contiendrait que deux tuples $\langle 3\rangle$ et $\langle 4\rangle$.

L'exemple précédent montre qu'essayer de réduire l'espace de stockage en ne matérialisant pas certains skycuboids d'espaces clos entraine un certain coût lors de l'exécution des requêtes. Toutefois, ceci reste toujours meilleur que d'exécuter la requête à partir de la table de données en entrée.

3.3.4 Calcul du Skycube Complet

Un cas particulier d'exécution de requêtes survient par exemple lorsqu'on désire calculer l'ensemble des requêtes skyline. Ceci revient à construire le skycube complet. Pour accomplir efficacement cette tâche, c'est-à-dire éviter la solution naïve consistant à exécuter indépendamment chaque requête à partir de la table T, les précédents travaux dérivent des propriétés et des cas dans lesquels il est possible que des résultats de calculs soient réutilisés entre skycuboids, améliorant de ce fait le temps global.

Puisque cette matérialisation nécessite l'exécution de chaque skyline, les précédentes propriétés que nous avons exhibées peuvent facilement être exploitées dans ce contexte. De ce fait, nous proposons FMC (mis pour Full Materialization with Closed subspaces) comme solution pour résoudre le problème du calcul du skycube complet. Cette procédure est présentée à travers l'algorithme 9.
La procédure FMC commence par calculer le topmost skyline $Sky(\mathcal{D})$. Si celui-ci est de *petite* taille, alors comme présenté précédemment, chaque $Sky(X)$ peut être efficacement obtenu à partir $Sky(\mathcal{D})$ et T_{clean}. Si $Sky(\mathcal{D})$ n'est pas petit, alors FMC procède en trois principales étapes : (i) la recherche des espaces clos, ensuite (ii) le calcul de leurs skylines respectifs, et enfin (iii) pour tout espace X non clos, le calcul de son skyline $Sky(Sky(X^+), X)$.

Le principal avantage de FMC est que ses différentes étapes peuvent bénéficier d'une exécution en parallèle. Le second avantage est la faible quantité de mémoire

Algorithme 9 : FMC algorithm

Input : Table T
Output : Skycube of T

Let $S_{\mathcal{D}} = Sky(T, \mathcal{D})$;
if $|S_{\mathcal{D}}|$ *is small* **then**
    $T_{clean} = cleanup(T)$;
    create index $\mathcal{I}$;
    for *every* $X \subset \mathcal{D}$ **do**
        `// Loop executed in parallel`
        Let $S =$ **Sky_X_from_Sky_Y**$(T_{clean}, S_{\mathcal{D}}, X)$;
        $Skycube\_T = Skycube\_T \cup \{S\}$;
    Return $Skycube\_T$;
else
    $Closed$ = **ClosedSubspaces**(T);
    foreach $X \in Closed$ **do**
        `// Loop executed in parallel`
        Compute $Sky(T, X)$;
    foreach *subspace* X **do**
        `// Loop executed in parallel`
        Compute $Sky(Sky(X^+), X)$;
    Return $\bigcup_{X \in 2^{\mathcal{D}}} Sky(X)$;

requise pour le calcul : dans le cas où le topmost skyline est de petite taille, tout ce dont on a besoin c'est ce skyline et la structure d'index bitmap. Dans l'autre cas, on conserve le skyline des espaces clos, et ceux-ci ne sont pas nombreux comme nous l'avons montré dans la section 3.2.7.

En dépit de sa simplicité, FMC fait preuve d'une réelle efficacité en pratique et surpasse les algorithmes de l'état de l'art comme cela est présenté dans la section 3.5.

3.4 Mise en commun des composants

Dans cette section, nous résumons notre proposition en assemblant les différentes briques que nous avons introduites jusque là.

Dans un premier temps, nous présentons la manière de calculer le skycube et d'autre part, nous décrivons comment l'exécution des requêtes est réalisée. L'algorithme 10 décrit la matérialisation du skycube. Celui-ci prend en entrée la table T et une information supplémentaire indiquant si on désire la matérialisation complète ou partielle du skycube. Dans le second cas, le processus ne stocke que le topmost skyline si celui-ci est de *petite* taille. Ce caractère (de petite taille), peut être défini par l'utilisateur, par exemple, comme un pourcentage de la table en entrée T ou codé en dur dans le programme.

Algorithme 10 : Skycube Materialization

```
Input : Table T, TypeMat ∈ {full, partial}
Output : Partial or full Skycube of T
if TypeMat=full then
    Return FMC(T);
else
    // The user asks for a partial materialization
    Let S_D = Sky(T, D);
    if |S_D| is small then
        // The small condition can be input by the
           user. E.g., as a percentage of size(T)
        T_clean = cleanup(T);
        create index I;
        Return S_D;
    else
        // We need to minimize the memory storage
           space
        Return MICS(T);
```

D'autre part, l'exécution des requêtes dépend de la manière dans laquelle est

matérialisé le skycube (partiellement ou complètement). L'algorithme 11 montre comment les requêtes skyline sont gérées. Dans le cas où le skycube est matérialisé partiellement, nous savons que pour calculer $Sky(X)$, nous avons besoin de connaître le clos X^+ associé à X (Line 10). Cette information peut être stockée durant la recherche des dépendances fonctionnelles, ce serait une table qui associe à chaque espace X, le clos correspondant X^+. Une seconde option serait de stocker une fermeture minimale des dépendances fonctionnelles et de l'utiliser pour calculer X^+ ; sachant que le calcul de la fermeture de X peut être réalisé en temps linéaire de la taille de l'ensemble des dépendances fonctionnelles Mannila et Räihä [1992]. Enfin, lorsqu'on calcule le MICS, le skyline $Sky(X)$ de certains clos X est retiré lorsque celui-ci est inclus dans le skyline d'un autre clos Y *ancêtre* de X ; dans ce cas, on conserve cette information dans le but de l'utiliser lorsqu'on aura besoin de calculer $Sky(X)$ aussi bien que le skyline de tous les espaces Z dont X est le clos associé ($Z^+ = X$).

Algorithme 11 : Skyline Query Evaluation

Input : Table T, TypeMat $\in$ {full, partial}, subspace X
Output : $Sky(X)$

```
1  if TypaMat=full then
2      Return Sky(X);
3      // No required computation
4  else
5      if |S_D| is small then
6          // S_D is always materialized
7          Return Sky_X_From_Sky_Y(T, S_D, X) ;
8      else
9          if X+ is materialized then
10             Return Sky(Sky(X+), X) ;
11         else
12             // i.e., only skylines in MICS are stored
13             Let Y be the remaining ancestor of X+ in MICS;
14             Return Sky(Sky(Y), X);
```

3.5 Expérimentations

Nous avons mené des expérimentations dans le but d'illustrer les points forts et les limites de notre approche. Pour cela, nous avons considéré 3 directions principales : (i) nous avons comparé notre solution aux précédents travaux ayant pour objectif la construction du skycube complet. Dans cette partie, nous analysons la

Matérialisation	Notre algorithme	Autres algorithmes
Complète	FMC	OrionTail, BUS, TDS, QSkyCubeGS, QSkyCubeGL
Partielle	MICS	Orion, CSC, Hashcube

TABLE 3.3 – Comparaison des algorithmes relativement aux tâches

scalabilité au regard à la fois de la dimensionalité (d) et du nombre de tuples (n). Il en ressort que notre proposition surpasse les algorithmes de l'état de l'art construisant le skycube complet lorsque varient la taille de la table ou le nombre de dimensions ; (ii) concernant l'aspect matérialisation partielle, nous avons comparé notre proposition aux techniques de l'état de l'art en considérant trois paramètres : (a) le temps nécessaire pour construire la structure de données (skycube partiel), (b) la taille de l'espace mémoire requis pour conserver ladite structure, et (c) le temps d'exécution des requêtes. Enfin, (iii) nous analysons le gain imputable à la matérialisation (pré-calcul) dans l'exécution des requêtes. La table 3.3 présente une classification de l'ensemble des algorithmes comparés ici que soit pour la matérialisation partielle ou complète du skycube. Toutes les expérimentations ont été réalisées sur une machine possédant 24Go de RAM, deux processeurs hexacores de 3.3Ghz et disque de 1To sous le système d'exploitation Linux Redhat Entreprise.

3.5.1 Les données

Nous avons utilisé à la fois des données réelles et synthétiques. La table 3.4 décrit les jeux de données réels (NBA, IPUMS et MBL), très connus dans la littérature skyline. Compte tenu de la taille plutôt faible de ces données, nous avons augmenté leurs tailles de manière artificielle en les répliquant. Par exemple, le jeu de données NBA50 a été obtenu à partir de 50 réplications du jeu de données NBA. Nous appelons ces trois jeux de données et leurs variantes le *benchmark*. En plus, en raison de leurs tailles plus grandes, nous avons utilisés deux jeux de données appelés USCensus et HouseHolders. Ces deux jeux de données ont attirés notre attention parce qu'ils présentent des caractéristiques très différentes en matière de dépendances fonctionnelles. En effet, USCensus présente très peu de dépendances fonctionnelles tandis que HouseHolder en possède une multitude. Des données synthétiques ont été générées suivant un processus disponible à l'adresse `pubzone.org` et fourni par les auteurs de Börzsönyi *et al.* [2001]. Il prend en entrée les valeurs de n et d aussi bien que le type de la distribution désirée (corrélé, indépendant ou anti-corrélé) et retourne n tuples de d dimensions respectant la distribution souhaitée. Les valeurs des attributs sont des nombres réels compris dans l'intervalle $[0, 1]$.

Jeu de données	n	d
NBA	20493	17
IPUMS	75836	10
MBL	92797	18
NBA50	1024650	17
IPUMS10	75836	10
MBL10	927970	18
USCensus	2458285	20
Householder	2628433	20

TABLE 3.4 – Les données réelles

3.5.2 Calcul en Parallèle

Nous avons implémenté nos solutions en langage `C++` couplé à l'outil `OpenMP` afin de bénéficier du parallélisme. Cette API facilite grandement le lancement de plusieurs threads en parallèle. Par exemple, la tâche codée par la boucle suivante :

```
for(int i=0; i<1000; i++){f(i); }
```

peut être exécutée en 4 threads juste en ajoutant une directive de compilation (pragma) au code source comme suit :

```
♯pragma omp parallel for num_threads(4)
for(int i=0; i<1000; i++){f(i); }
```

En théorie, le temps d'exécution global est divisé par 4 si nous disposons de 4 unité de calcul. En pratique, c'est rarement le cas parce que (i) l'exécution de $f(i_1)$ peut demander plus ou moins de temps que celle de $f(i_2)$ donc ceci n'entraine pas forcement un équilibrage parfait et (ii) si à la fois $f(i_1)$ et $f(i_2)$ ont besoin d'accéder à une ressource partagée alors, le contrôle des accès concurrents (en utilisant les verrous par exemple) pénalise le parallélisme. En plus, suivant la loi de Amdhal, les algorithmes possèdent des parties séquentielles qui ne peuvent pas bénéficier de l'exécution en parallèle. Donc quel que soit le nombre d'unités de calcul disponibles, ces parties seront forcement exécutées par un seul thread.

3.5.3 Construction du Skycube complet

Dans cette partie, nous comparons le temps d'exécution de notre procédure FMC aux implémentations d'algorithmes de l'état de l'art appelés BUS et TDS Pei *et al.* [2006], OrionTail Raïssi *et al.* [2010], aussi bien que les propositions plus récentes que sont QSkycubeGS et QSkyCubeGL présentées dans Lee et won Hwang [2014]. Nous avons utilisé les implémentations fournies par les auteurs [2]. Ces procédures (toutes implémentées en `C++`) prennent en entrée la table de données et retournent le skycube correspondant. Pour FMC, nous avons utilisé

2. Nous remercions les auteurs de Lee et won Hwang [2014] qui nous ont fourni une implémentation de tous ces algorithmes.

l'implémentation de l'algorithme BSkyTree Lee et won Hwang [2010] calculant le skyline. Sauf indication contraire, les exécutions de FMC ont été réalisées en fixant le nombre de threads à 12 (nombre de cœurs disponibles). Puisque tous les algorithmes précédents sont séquentiels, et afin de mener des comparaisons équitables, nous reportons souvent le rapport de temps d'exécution (*speedup*) de FMC par rapport à celui de ses concurrents. Plus précisement,

$$\text{Speedup}=\frac{\text{execution time of algorithm i}}{\text{execution time of FMC}}$$

Si le *speedup* est plus grand que 12, nous pouvons sans risque conclure que FMC surpasse ses compétiteurs puisque son exécution séquentielles ne peut pas être 12 fois plus lente que son exécution en parallèle.

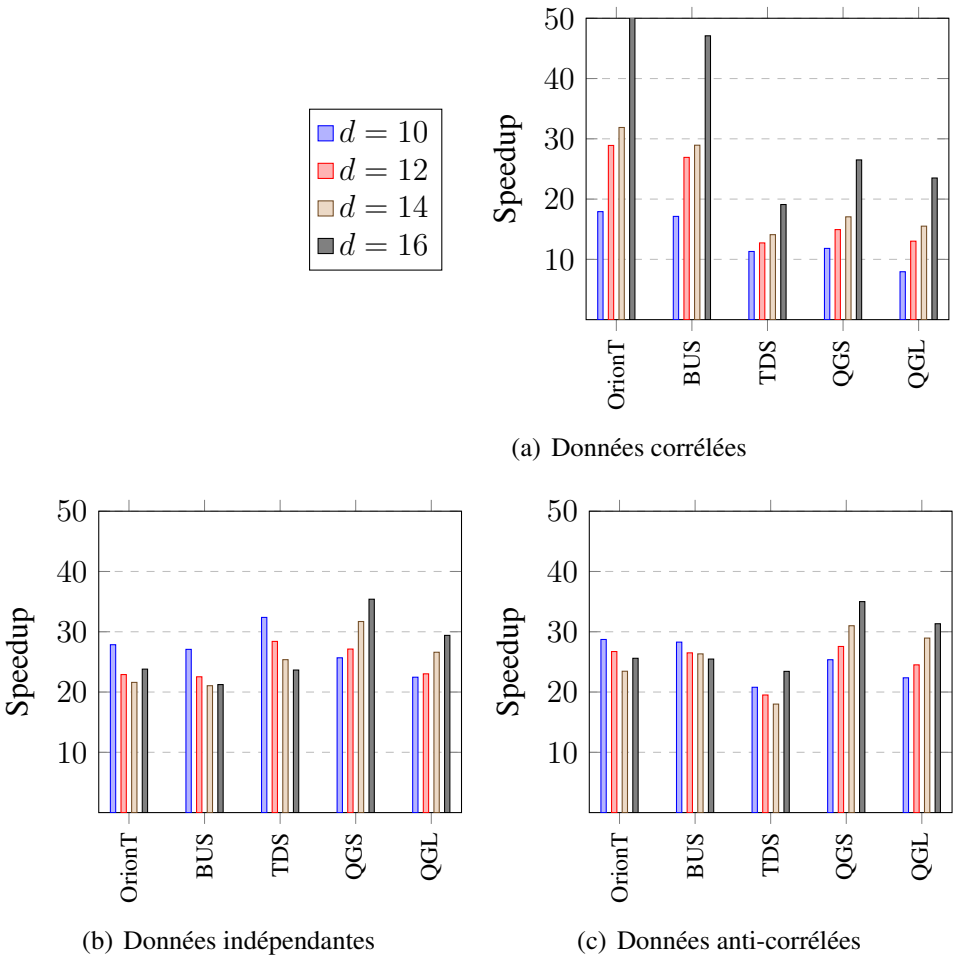
Pour qu'un topmost skyline soit considéré comme *petit*, nous avons fixé la condition que sa taille doit être inférieure à $\sqrt{n}$. L'idée derrière ce choix est de se rassurer que la complexité de l'exécution de la requête skyline soit linéaire de la taille de la table, c'est-à-dire de l'ordre de $O(n)$ ($\sqrt{n} \times \sqrt{n}$).

Données synthétiques : scalabilité au regard de la dimensionalité d

Pour analyser l'effet de la croissance du nombre de dimensions, nous avons commencé par des données synthétiques. Nous avons fixé le nombre de tuples n à 100K tuples (une relative petite valeur de n) et nous avons fait varier le nombre de dimensions d de 10 à 16. Nous avons éprouvé les trois catégories de distributions (corrélée, indépendante, anti-corrélée). Les résultats sont reportés dans les figures 3.7(a)-3.7(c). La figure 3.7(a) montre que le speedup obtenu par FMC augmente avec d. Ceci tend à démontrer que, pour ce type de distribution de données, FMC est plus approprié lorsque le nombre de dimensions augmente. Une analyse plus profonde des performances de QSkycubeGL laisse apparaître que lorsque $d = 10$, le speedup de FMC est seulement de 7. On pourrait avoir tendance à conclure qu'exécuter FMC séquentiellement est plus lente que QSkycubeGL. En réalité, il apparaît que les deux algorithmes ont pratiquement les mêmes performances dans ce cas. Nous devons également noter que ce temps d'exécution est d'environ 2 secondes. Pour les deux autres types de distribution, les figures 3.7(b) et 3.7(c) montrent que lorsque $d = 10$, nous avons un speedup supérieur à 12. Même si tous les algorithmes n'ont pas le même comportement relativement à d, FMC est considérablement plus efficace dans tous les cas.

Données synthétiques : scalabilité relativement à la taille des données n

Ici, nous fixons le nombre de dimensions d à 16 et faisons varier le nombre de tuples n qui prend successivement les valeurs $200K$, $500K$ et $1M$. Les résultats sont reportés dans la figure 3.8. Ceux-ci montrent que FMC surpasse tous les algorithmes dans tous les cas. De plus, le speedup augmente uniformément lorsque n augmente même s'il est important de préciser que QSkyCubeGL et QskyCubeGS sont les plus scalables (parmi les méthodes concurrentes).

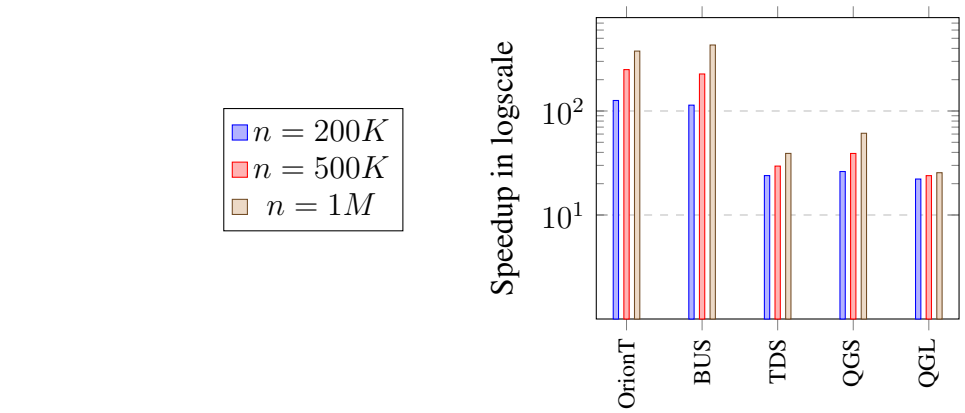


(a) Données corrélées

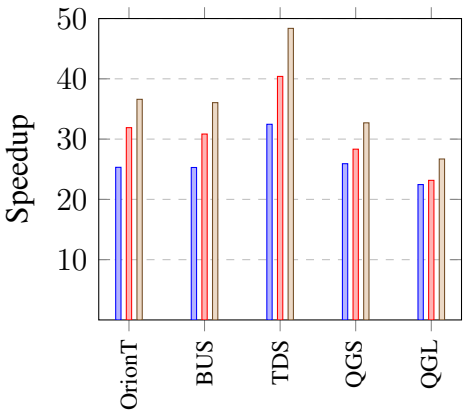
(b) Données indépendantes

(c) Données anti-corrélées

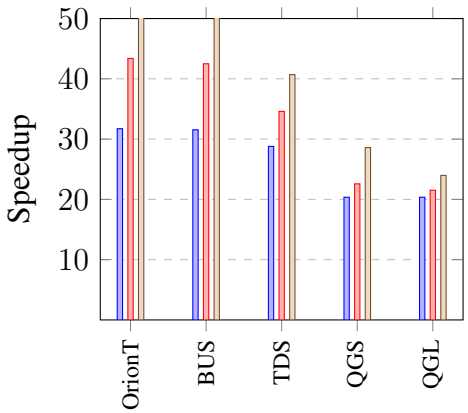
FIGURE 3.7 – Speedup par rapport à la dimensionalité d ($n = 100K$)



(a) Données corrélées



(b) Données indépendantes



(c) Données anti-corrélées

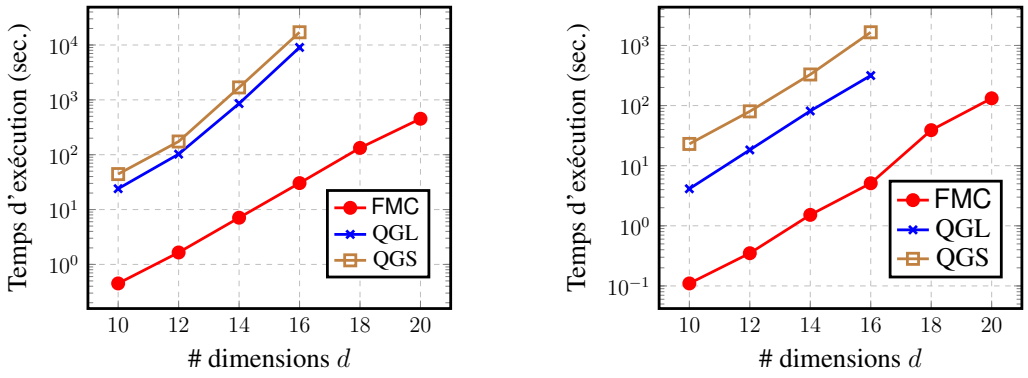
FIGURE 3.8 – Speedup par rapport à la taille des données n ($d = 16$)

Remarque 1 (La génération de données synthétiques et l'extension de FMC)**.** Au cours de nos expérimentations, nous avons constaté que les données synthétiques générées satisfaisaient la propriété de valeurs distinctes par dimension (chaque dimension présentant presque n valeurs distinctes. Ceci est dû au fait que les valeurs générées sont des réels tirés aléatoirement et uniformmément dans l'intervalle $[0, 1]$. Donc, la probabilité que la même valeur soit tirée deux fois dans cet intervalle, contenant théoriquement un nombre infini de valeurs, est presque nulle. Cette situation tend à réduire l'ensemble des espaces clos à $\mathcal{D}$ et quelques dimensions d'un attribut. Par conséquent, au cours de l'exécution de FMC beaucoup de skycuboids sont calculés à partir de $Sky(\mathcal{D})$. Dans le cas de données corrélées, cette démarche s'avère bénéfique en raison du fait que $Sky(\mathcal{D})$ est de petite taille comparé à celle de T. Ce n'est pas le cas des données anti-corrélées et des données indépendantes. Dans le premier cas (données anti-corrélées), FMC se rapproche de la procédure naïve puisque la taille de $Sky(\mathcal{D})$ est proche de celle de la table T et beaucoup de skycuboids sont calculés à partir de $Sky(\mathcal{D})$. Néanmoins, comme les tests précédents l'ont montré, FMC est aussi compétitive dans ces cas pour deux raisons essentiellement : (i) son exécution en parallèle et (ii) le fait que nous ne faisons aucun calcul supplémentaire pour rechercher de potentiels tuples manquant de $Sky(T, X)$ lorsqu'on exécute $Sky(Sky(\mathcal{D}), X)$. Ce second point représente le goulet d'étranglement des précédents travaux.

Remarque 2. FMC peut facilement être optimisé en exploitant l'inclusion entre skylines qui sont induites par la présence de dépendances fonctionnelles. Par exemple, supposons que $\mathcal{D} = \{A, B, C, D\}$ et que les valeurs sont toutes distinctes au sein des dimensions de T. Alors pour tout espace $X \neq ABCD$, FMC évalue la requête $Sky(Sky(ABCD), X)$. Une meilleure manière pourrait être d'évaluer le skyline de X à partir de celui de l'un de ses parents *immédiats*, par exemple, $Sky(A)$ à partir de $Sky(AB)$, $Sky(AC)$ ou $Sky(AD)$. Ceci peut être réalisé pour tout le skycube en effectuant soit un parcours en largeur, soit un parcours en profondeur du treillis des espaces.

Analyse des données réelles

Dans le but d'éviter des analyses et conclusions basées uniquement sur des données synthétiques, introduisant de ce fait un biais, nous avons mené les mêmes expérimentations que précédemment sur trois jeux de données réels. `US Census` est un jeu de données bien connu en machine learning disponible sur le dépôt UCI. Tous les attributs présentent des valeurs entières positives et même s'ils ne se prètent pas tous à la problématique de classement/comparaison, ce jeu de données s'avère intéressant parce que ses valeurs suivent une distribution réelle. Nous avons choisi 20 colonnes/dimensions de manière aléatoire pour nos expérimentations. Le second jeu de données est également disponible au public à partir du site de l'Institut français de statistique et d'économie, l'INSEE. Ce jeu de données décrit les ménages (`householders`) du sud-ouest de la France. Il présente plus de 2.5 millions de



(a) Données USCensus (b) Données Householders

FIGURE 3.9 – Calcul du skycube complet sur des données réelles

tuples décrits par 67 variables. Ici également, nous avons tiré aléatoirement 20 attributs mais contrairement à USCensus, ses dimensions présentent une sémantique se prêtant à la problématique de classement/comparaison (par exemple le nombre de personnes vivant dans le ménage, le nombre de pièce de l'habitation, ...). Bien que les deux jeux de données possèdent presque le même nombre de tuples et de dimensions, leurs distributions des valeurs sont radicalement différentes : tous les espaces sont des clos dans le jeu de données `USCensus` (ce qui signifie qu'il ne présente aucune dépendance fonctionnelle) tandis que `householders` ne présente que très peu de clos (moins de 2%) lorsque $d \geq 10$.

données US Census La figure 3.9(a) présente les temps d'exécution requis par FMC, QSkyCubeGS et QSkyCubeGL pour construire le skycube complet en variant d de 10 à 20. Nous considérons seulement ces deux méthodes concurrentes parce que les autres requièrent bien plus de temps. Néanmoins, notons qu'à partir de $d = 16$ ces deux méthodes (QSkyCubeGS et QSkyCubeGL) ont saturé la mémoire disponible (24 Go) et ont commencé à effectué des swap sur disque au cours de l'exécution. C'est pourquoi nous avons arreté leur exécution, autrement elles auraient pris trop de temps. Il est important de mentionner que nous avons pris le soin de modifier le code source original de ces méthodes afin que les skycuboids soient supprimés de la mémoire aussitôt qu'ils sont calculés ; donc la saturation de la mémoire n'est pas dû à la taille du skycube mais plutôt à `STreeCube`, la structure de données partagée utilisée par ces algorithmes dans le but d'accélérer l'exécution. Soulignons également la croissance rapide du speedup lorsque d augmente, atteignant 3 ordres de magnitude lorsque $d = 16$ pour QSkyCubeGL et presque 6 pour QSkyCubeGS. Une spécificité de ce jeu de données est le fait que ses dimensions possèdent un très petit nombre de valeurs distinctes : environ 10 valeurs distinctes pour chacunes d'elles. Ceci rend les dépendances fonctionnelles difficiles à satisfaire lorsque le nombre de tuples est aussi grand ; en fait, ce jeu de données n'en

NBA			
FMC(1)	**FMC(12)**	**QGL**	**QGS**
1.01	0.22	8.15	4.57
IPUMS			
FMC(1)	**FMC(12)**	**QGL**	**QGS**
1.36	0.45	2.27	10.24
MBL			
FMC(1)	**FMC(12)**	**QGL**	**QGS**
12.3	4.5	63.85	267.1

TABLE 3.5 – Temps d'exécution (sec.) sur des données réelles de petite taille. FMC est exécuté avec 1 ou 12 threads

satisfait aucune ; par conséquent, tous les espaces sont des clos. Cependant, le topmost skyline est assez *petit*. Par exemple, pour $d = 10$, $Sky(\mathcal{D})$ contient *seulement* 3873 tuples. De plus, ces tuples ont exactement les mêmes valeurs dans chaque dimension ; $Sky(\mathcal{D})$ est de Type I, ce qui facilite le calcul de n'importe quel skyline.

C'est l'algorithme *semi-naïf* de calcul du skycube qui est chosi par FMC (lignes 2-6 dans l'algorithme 9). Par conséquent, le calcul de $Sky(X)$ est réalisé par une opération de jointure qui consiste simplement à retrouver les tuples t de T tels que $t[X] = t'[X]$ où t' est l'*unique* représentative de $Sky(\mathcal{D})$. Pour optimiser cette opération de jointure, dans notre implémentation, nous avons fait usage d'index bitmap [3]Lemire *et al.* [2012]. Cette expérience empirique confirme l'existence de la relation entre le nombre de valeurs distinctes par dimension, le nombre de dépendances fonctionnelles et la taille du topmost skyline.

Données Householders Les résultats sont présentés dans la figure 3.9(b). Ici également, les deux méthodes QSkyCubeGS et QSkyCubeGL ont été incapables de gérer le cas où $d \geq 16$. Une différence perceptible entre ce jeu de données et le précédent est que plusieurs dépendances fonctionnelles y sont satisfaites. Le nombre d'espace clos est d'environ 29000 contre $2^{20} - 1$ espaces possibles. Le nombre de valeurs distinctes par dimension varie de 2 à 4248. Le topmost skyline possède 154752 tuples. Ceci montre l'efficacité de l'utilisation des skylines des espaces clos pour calculer ceux des autres espaces. En effet, même si les speedup obtenus sont moins impressionnants que ceux obtenus pour le jeu de données précédent, FMC est toujours au moins 50 fois plus rapide que QSkyCubeGL et bien plus comparé à QSkyCubeGS.

Benchmark NBA, IPUMS et MBL sont relativement de petits en taille. Pour ces jeux de données, nous avons exécuté FMC deux fois : Dans la première exécu-

3. Nous avons utilisé `EWAH` l'implémentation d'index bitmap pour D, disponible à l'adresse `https://github.com/lemire/EWAHBoolArray`

NBA50			
FMC(1)	**FMC(12)**	**QGL**	**QGS**
23	7	59	336
IPUMS10			
FMC(1)	**FMC(12)**	**QGL**	**QGS**
22	6	16	336
MBL10			
FMC(1)	**FMC(12)**	**QGL**	**QGS**
311	47	780	5963

TABLE 3.6 – Temps d'exécution (sec.) pour les données artificiellement augmentées FMC est exécuté avec 1 ou 12 threads

tion, dénotée FMC(1), nous avons mis en œuvre un seul thread (exécution séquentielle) et dans la seconde exécution dénotée FMC(12), nous avons mis en œuvre 12 threads. Nous n'avons effectué aucune modification du programme. La table 3.5 montre les temps d'exécution. Notons cependant que lorsque le nombre de dimensions est *petit*, ce qui est le cas pour IPUMS, le speedup de FMC(12) par rapport à QSkycubeGL est plutôt mauvais (environ 5.04). Il est plus grand en ce qui concerne NBA et MBL qui possèdent respectivement 17 et 18 dimensions. Ceci tend à montrer que l'exécution de FMC en parallèle est appropriée lorsque le nombre de dimensions devient grand. Il est important de souligner que, nous avons exécuté l'algorithme naïf pour IPUMS : pour chaque espace X, nous avons calculé $Sky(T, X)$. Cette boucle a également été exécutée en utilisant 12 threads. La procédure s'est achevée au bout de 0.68 secondes fournissant un speedup de 3.8 par rapport à QskycubeGL.

Nous avons artificiellement augmenté la taille de ces jeux de données en repliquant plusieurs fois leur tuples. La table 3.6 présente les temps d'exécution. FMC passe à l'échelle de manière presque linéaire par rapport à la croissance de la taille du jeu de données alors que QSkyCubeGS ne passe pas correctement à l'échelle et QSkyCubeGL semble être l'algorithme présentant le meilleur comportement au regard de la croissance de la taille du jeu de données. Notons cependant que c'est l'algorithme requérant le plus de mémoire. Par exemple, nous avons essayé d'exécuter QSkyCubeGL pour NBA100 et ceci a conduit à une saturation de la mémoire tandis que FMC a réussi cette tâche au bout de 13 secondes.

Enfin, nous avons analysé le comportement des trois algorithmes au regard de la dimensionalité. Nous avons utilisé MBL10 et varié d de 4 à 18. La figure 3.10 compare FMC(12), QSkycubeGS et QSkycubeGL d'un point de vue temps d'exécution. Le speedup de notre méthode est presque constant.

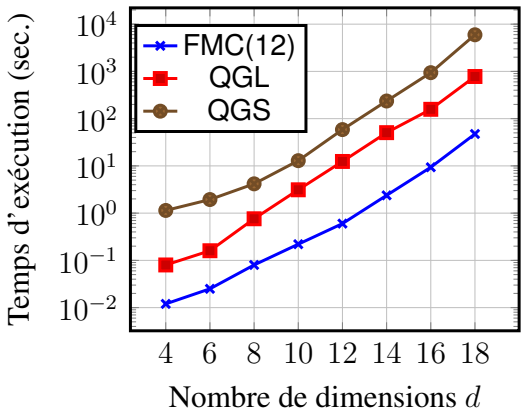


FIGURE 3.10 – Matérialisation complète pour MBL10

NBA		
Algorithme	temps de construction (sec.)	skycuboids stockés
MICS	12.8	5304
Orion	9	5304
IPUMS		
Algorithme	temps de construction (sec.)	skycuboids stockés
MICS	2.1	11
Orion	322	738
MBL		
Algorithme	temps de construction (sec.)	skycuboids stockés
MICS	172	29155
Orion	11069	43075

TABLE 3.7 – MICS vs Orion

3.5.4 La Matérialisation Partielle du Skycube

MICS vs Closed Skycubes :

Nous avons comparé notre solution à la méthode OrionClos proposé dans Raïssi *et al.* [2010] qui calcule le *Closed Skycubes*. Pour ce faire, les critères de comparaison sont (i) l'utilisation de l'espace mémoire et (ii) la vitesse de matérialisation du sous-skycube. Nous avons considéré les trois jeux de données réels utilisés par les auteurs : NBA, IPUMS et MBL. Nous avons comparé le nombre de classes d'équivalence, c'est-à-dire, le nombre de skylines effectivement stockés par *Closed Skycubes* et le nombre de skylines stockés dans MICS. Pour les deux techniques, nous avons également reporté le temps global de calcul. Les résultats sont présentés dans la table 3.7.

En général, nous observons que MICS requiert moins de skycuboids que closed

skycube. Pour **NBA** les deux solutions sont identiques. Pour **IPUMS**, nous avons stocké 73 382 tuples, correspondant à 11 skycuboids tandis que closed skycube requiert 530 080 tuples. En ce qui concerne **MBL**, nous avons stocké 118 640 340 tuples contre 133 759 420. Le rapport d'espace de stockage n'est pas très grand. Par contre, notre proposition est souvent plus rapide que sa concurrente. Plus précisement, le speedup semble augmenter lorsque le nombre de tuples n augmente. En plus de ces données, nous avons testé OrionClos sur de plus grands jeux de données mais il s'est avéré incapable d'accomplir la tâche dans un temps raisonnable. Par exemple, 36 heures n'ont pas été suffisantes pour exécuter les calculs sur un jeu de données corrélé présentant $d = 20$ dimensions et $n = 100K$ tuples ; tandis qu'en 20 secondes notre approche a construit le MICS du même jeu de données. Ceci est dû au fait que les générateurs de données tendent à produire des dimensions respectant la propriété de valeurs distinctes, ce qui entraine pratiquement autant de classes d'équivalence skyline qu'il y a d'espaces et complique la tâche de l'approche OrionClos.

Compressed Skycubes (CSC)

Dans cette section, nous comparons notre proposition à la méthode CSC (Compressed SkyCube) de Xia *et al.* [2012]. Notre comparaison s'articulera autour de trois critères : (i) le temps de construction, (ii) l'espace de stockage requis et (iii) le temps d'exécution des requêtes. Dans ce but, nous avons utilisé les trois jeux de données réels NBA, MBL et IPUMS aussi bien que trois jeux de données synthétiques de 100K tuples et 16 dimensions tour à tour corrélées, indépendantes ou anti-corrélées. En ce qui concerne l'évaluation des performances dans l'exécution des requêtes, nous avons considéré pour toutes les méthodes l'ensemble des $2^d - 1$ requêtes skyline possibles, dans le but de se faire une idée de la moyenne de temps d'exécution des requêtes. Afin de réaliser des comparaisons équitables, nos programmes ont été exécutés de manière séquentielle. Rappelons que notre proposition, soit stocke seulement le topmost skyline (si sa taille est petite), soit repose sur le MICS. Les résultats de cette expérience sont résumés dans la table 3.8. Comme cela peut être observé, notre proposition surpasse CSC dans tous les critères. Pour les données réelles NBA et MBL aussi bien que pour les données synthétique corrélées, notre solution requiert le stockage du topmost skyline de tailles respectives 3, 78 et 2 tuples.

Skycube Partiel vs Hashcube

Nous avons utilisé deux jeux de données, à savoir `NBA` et `Householder`, pour illustrer les avantages aussi bien que les limites de la structure de données Hashcube Bøgh *et al.* [2014]. Alors que le premier jeu de données est petit et corrélé, le second est plus grand et non corrélé. En termes de nombre de tuples stockés, nous avons reporté la taille de Hashcube et celle du skycube partiel que nous avons obtenu.

NBA			
Algorithme	construction (sec.)	stockage	requêtes (sec.)
CSC	631	57571	150
Skycube Partiel	0.0008	3	3.5
MBL			
Algorithme	construction (sec.)	stockage	requêtes (sec.)
CSC	47654	921911	7212
Skycube Partiel	10	78	58
IPUMS			
Algorithme	construction (sec.)	stockage	requêtes (sec.)
CSC	3776	129812	142
Skycube Partiel	6	73382	2
Données corrélées			
Algorithme	construction (sec.)	stockage	requêtes (sec.)
CSC	4533	498	10
Skycube Partiel	0.015	2	0.03
Données indépendantes			
Algorithme	construction (sec.)	stockage	requêtes (sec.)
CSC	36123	921911	6212
Skycube Partiel	35	83650	78
Données anti-corrélées			
Algorithme	construction (sec.)	stockage	requêtes (sec.)
CSC	51263	4556789	11972
Skycube Partiel	62	121347	129

TABLE 3.8 – Skycube Partiel vs CSC

NBA		
Algorithme	stockage	requête (sec.)
Hashcube	541316	0.029
Skycube Partiel	3	3.5
Householder		
Algorithme	stockage	requête (sec.)
Hashcube	2431675126	_
Skycube Partiel	49022451	128

TABLE 3.9 – Skycube Partiel vs Hashcube

nous avons également reporté le temps d'exécution des $2^d - 1$ requêtes possibles. La table 3.9 résume nos résultats.

Sur le jeu de données NBA, nous observons que Hashcube requiert beaucoup plus d'espace de stockage comparé à notre solution. Rappelons que le skycube lui-même requiert de stocker environ $4.7 * 10^6$ tuples. Par conséquent, Hashcube réalise, comme reporté dans Bøgh *et al.* [2014], un ratio de compression d'environ 10%. Notons cependant que l'exécution des requêtes par Hashcube est $100\times$ plus rapide.

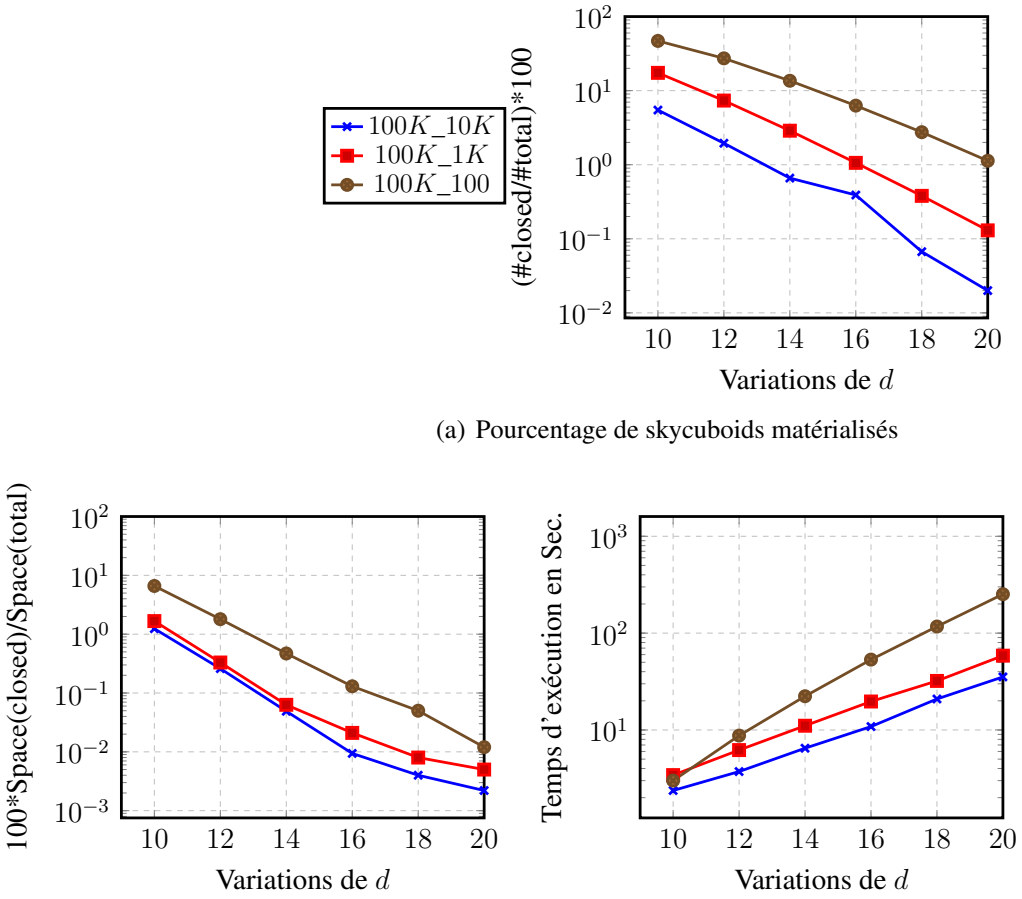
Pour le deuxième jeu de données, la taille de la structure de données Hashcube était trop élevée pour suffire dans la mémoire disponible. Le skycube contient environ $15 * 10^9$ tuples. Par conséquent, nous n'avons pas réalisé les tests d'exécution des requêtes. Précisons quand même que notre structure de données est $300\times$ plus petite que le skycube.

En somme, Hashcube peut s'avérer intéressant dans la situation suivante : (i) le skycube est disponible et sa taille est trop élevé pour suffire en mémoire et, (ii) la structure de données hashcube est capable de tenir en mémoire.

Analyse de l'Espace de Stockage

Dans la deuxième partie des expérimentations, nous avons généré un ensemble de données synthétiques indépendantes en fixant les paramètres suivants : d est le nombre d'attributs, n est le nombre de tuples, k est le nombre de moyen de valeurs distinctes par attribut. Par exemple, $100K\_10K$ designe un jeu de données pour lequel $n = 100K$ et le nombre k de valeurs distinctes par dimension est égal à $10K$. Puisque le générateur de données retourne des valeurs réelles dans l'intervalle $[0, 1]$, il suffit de conserver les f premières décimales pour obtenir un jeu de données dont les dimensions possèdent chacune en moyenne $k = 10^f$ valeurs distinctes. Par exemple, 0.0123 est remplacé par 12 si $k = 10^3$. Cette opération ne change pas sensiblement le niveau des corrélations entre les différents attributs. Même si nous reportons seulement les résultats obtenus avec les données indépendantes, nous tenons à préciser que nous avons réalisé les mêmes expérimentations avec des données corrélées et anti-corrélées ; la conclusion a été la même.

Nous avons tout d'abord investigué le nombre d'espaces clos par rapport au



(a) Pourcentage de skycuboids matérialisés

(b) Ratio d'espace utilisé

(c) Temps d'exécution pour la recherche d'espaces clos et la matérialisation de leurs skylines respectifs

FIGURE 3.11 – Analyse quantitative des espaces clos

nombre total de sous-espaces. Les résultats ont été reportés dans la figure 3.11(a). Notons que la proportion d'espaces clos décroît lorsque le nombre d'attributs augmente. Par exemple, considérons le jeu de données $100K\_10K$ lorsque $d = 10$. Il y a environ 7% d'espaces parmi les $2^{10} - 1$ qui sont clos. Cette proportion passe à 0.035% pour $d = 20$. En plus, le nombre d'espaces clos augmente lorsque le nombre de valeurs distinctes prises par chaque attribut décroît. Par exemple, lorsque $d = 10$, seulement 7% des espaces sont des clos pour $100K\_10K$ tandis il y a 84.5% d'espaces clos pour $100K\_100$. Ce second cas pourrait indiquer que le gain d'espace mémoire lorsqu'on ne stocke que les skycuboids associés aux espaces clos est marginal. La deuxième expérimentation (voir la figure 3.11(b)) montre que ceci n'est pas systématique. Ici, on calcule le rapport entre le nombre total de tuples devant être stockés lorsque seulement les skylines des espaces clos sont matérialisés et le nombre total de tuples du skycube. Nous observons que dans tous les cas, l'espace mémoire requis pour matérialiser les skylines relatifs aux espaces clos ne depasse jamais 10% de la taille du skycube. Par exemple, même si nous matérialisons 84.5% des skylines du skycube relatif au jeu de données $100K\_100$ lorsque $d = 10$, cet espace de stockage représente moins de 10% de la taille du skycube. Ceci indiquerait que soit notre proposition a tendance à éviter la matérialisation des skycuboids *lourds*, soit la taille des skylines a tendance à décroitre lorsque le nombre d'espaces clos croît. Nous montrons empiriquement ensuite que c'est plutôt parce que la taille des skylines décroît.

Enfin, la figure 3.11(c) présente les temps d'exécution des jeux de données que nous avons considérés jusque là. Nous insistons sur le fait que ces temps représentent à la fois le temps nécessaire à (i) l'extraction des espaces clos et (ii) le calcul de leurs skylines respectifs. Clairement, moins les dimensions présentent de valeurs distinctes, plus il y a d'espaces clos et plus il faut de temps pour les calculer.

Nous avons mené une deuxième série d'expérimentations dont le but est de montrer l'évolution de la taille du skyline lorsque varient à la fois la cardinalité k des dimensions et la dimensionalité d conservant fixe la taille des données n. Les résultats sont présentés dans la figure 3.12. Nous observons que la taille du skyline croît uniformement avec le nombre de valeurs distinctes par dimension k de même que lorsque le nombre de dimensions augmente. Ceci donne une explication claire au comportement souligné dans la figure 3.11(b), c'est-à-dire, lorsque k décroît, le nombre d'espaces clos croît tandis que la taille de leurs skylines respectifs décroît (voir théoreme 3.6).

Par conséquent, la leçon principale que nous retenons de l'expérience précédente est que lorsque le nombre d'espaces clos croît, la taille des skylines décroît. donc, même si notre solution stocke plus de skyline, ceci ne signifie pas nécessairement que la structure requiert plus d'espace mémoire. La deuxième leçon qui en découle est que lorsque k est petit, chaque skycuboid tend à être petit. De ce fait, d'un point de vue pragmatique, matérialiser juste le topmost skyline s'avère suffisant pour efficacement répondre à n'importe quelle requête skyline en utilisant l'algorithme 4. Ce résultat est en cohérence avec l'étude analytique développée dans

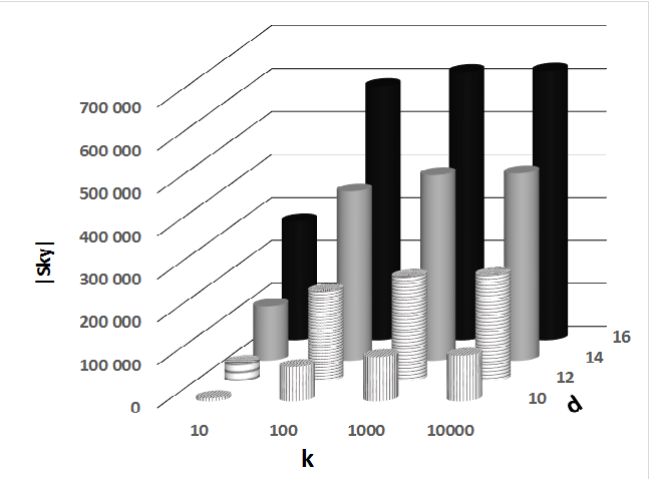


FIGURE 3.12 – Évolution de la taille du skyline relativement à celles de k et d avec $n = 10^6$

les sections 3.2.7 et 3.2.8.

3.5.5 Exécution des requêtes

Dans cette partie, nous analysons l'efficacité de notre proposition en termes de temps d'exécution des requêtes après une matérialisation partielle du skycube, c'est-à-dire, une fois les skylines des espaces clos calculés. Nous avons utilisé pour ce faire le jeu de données réel `Householders`, en faisant varier le nombre de dimensions d de 16 à 20, et faisant varier le nombre de tuples n de $500K$ à $2M$. Nous avons généré 1000 requêtes skyline distinctes parmi les $2^d - 1$ requêtes possibles comme suit : les $2^d - 1$ espaces ont été listés et triés suivant l'ordre lexicographique dans un vecteur Q. Nous avons tiré aléatoirement et uniformément un nombre entier i dans l'intervalle allant de 1 à $2^d - 1$. $Sky(Q[i])$ fera partie de la liste de 1000 requêtes si l'espace $Q[i]$ ne correspond pas à un espace clos (stocké). nous avons alors repété ce procédé jusqu'à obtenir 1000 requêtes skyline distinctes. Chaque fois que nous avons choisi une valeur i, la probabilité que celle-ci corresponde à une requête sur δ dimensions est $\frac{\binom{d}{\delta}}{2^d-1}$. Les résultats sont présenté dans la table 3.10. Pour chaque combinaison (n, d), nous avons reporté trois informations importantes : (i) le temps d'exécution total des requêtes lorsqu'on fait usage de notre structure de données, (ii) le temps d'exécution total des requêtes sans faire usage de notre structure de données (pour chaque reuête on revient à la table initiale T) et (iii) la proportion de skycuboids matérialisés. Par exemple, lorsque $n = 2M$ et $d = 20$, 0.049 secondes sont suffisantes pour exécuter les 1000 requêtes choisies et ce, en faisant usage de notre structure de données, tandis que la même tâche requiert 99.92 secondes lorsque s'il faut pour chaque requête, revenir à la table initiale. Donc, notre approche permet d'exécuter les requêtes plus de $2000\times$ plus rapidement. Cette per-

$n\backslash d$	16	18	20
$500K$	$0.024/18.9$ (1.19%)	$0.026/22.54$ (0.55%)	$0.027/25.78$ (0.13%)
$1M$	$0.034/36.78$ (2.197%)	$0.036/44.41$ (1.098%)	$0.047/49.68$ (0.274%)
$2M$	$0.041/73.74$ (2.22%)	$0.044/87.92$ (1.45%)	$0.049/99.92$ (0.31%)

TABLE 3.10 – Temps d'exécution des requêtes en seconde : optimisé/non optimisé et (la proportion de skycuboids matérialisés)

formance est obtenue en ne matérialisant que 0.31% (3250 skycuboids) parmi $2^{20}-1$ (plus d'un million) de skycuboids possibles. Une observation remarquable est celle selon laquelle, avec un petit effort de matérialisation, les requêtes sont exécutées de multiples ordres de magnitude plus rapidement à l'aide de MICS que de façon naïve. Enfin, nous tenons à mentionner que le calcul du MICS à chaque fois ne demandait que très peu de temps.

Conclusion

Dans ce travail, nous avons développé une manière d'exploiter les dépendances fonctionnelles pour identifier les inclusions entre les skylines de différents espaces. Grâce à cette information, nous avons investigué la matérialisation partielle des skycubes. Nous nous sommes également appuyé sur les dépendances fonctionnelles pour apporter une contribution à la problématique de la construction du skycube complet. Ceci s'avère surprenant car les dépendances fonctionnelles sont indépendantes de la relation d'ordre existant entre les valeurs d'un attribut alors que, les requêtes skyline reposent essentiellement sur cet ordre ; ce qui montre la robustesse des dépendances fonctionnelles. Notre solution à la matérialisation partielle du skycube peut être vue comme un compromis entre (i) la taille de l'espace requis (on essaye de stocker aussi peu que possible), (ii) le temps d'exécution des requêtes (on evite de lire entièrement les données) et (iii) le temps de construction de la structure à stocker. Nous avons comparé de manière expérimentale notre proposition aux implémentations des algorithmes de l'état de l'art. La conclusion étant que nos solutions passent mieux à l'échelle lorsque le volume (nombre de tuples et de dimensions) des données augmente. Sur des données réelles, en ne matérialisant qu'une petite fraction du skycube, nous engrangeons plusieurs ordres de magnitude dans le temps d'exécution des requêtes.

On pourrait imaginer des situations où on disposerait d'un budget mémoire alloué, à utiliser de manière optimale. Si le MICS est très petit par rapport à l'espace disponible, on envisagerait alors d'ajouter en mémoire des skycuboids ne faisant

pas partie du MICS, ce qui réduirait le temps moyen de réponse aux requêtes. La taille des skycuboids serait le critère sur lequel reposerait le choix des skycuboids à rajouter en mémoire. Étant donné que cette information (taille des skylines) n'est pas connue à priori, on pourrait se tourner vers des approximations (estimations), problème que nous avons abordé au chapitre précédent.

Chapitre 4

Calculer et Résumer le Skycube

Introduction

En raison du nombre exponentiel de requêtes skyline, la matérialisation du skycube est une opération chronophage et requiert beaucoup d'espace. Pour cette raison, deux directions de recherche ont été suivi depuis lors : d'une part, des algorithmes complexes pour accélerer le calcul du skycube ont été proposés, d'autre part des techniques de compression (réduction de la taille) du skycube ont été définies pour réduire l'espace requis tout en conservant le temps de calcul acceptable. Considérant la réduction de taille, certains algorithmes supposent le skycube comme faisant partie de l'entrée tandis que d'autres présentent des procédures de compression partant de l'ensemble des données brutes.

Le présent travail se base sur l'observation suivante : affirmer qu'un tuple p fait partie du skyline nécessite qu'il soit comparé à l'ensemble des autres points. Cependant, la comparaison de p à un autre tuple q nous permet de déduire un ensemble de sous-espaces pour lesquels p n'appartient pas au skyline. Partant de là, notre hypothèse de départ est alors que rechercher les sous-espaces au regard desquels p n'appartient pas au skyline peut se faire plus rapidement que le travail complémentaire. Une fois que cet ensemble de sous-espaces est déterminé, retrouver l'ensemble complémentaire est évident.

Sauvegarder pour chaque tuple p, l'ensemble des sous-espaces pour lesquels p n'appartient pas au skyline pourrait être impossible en pratique en raison de la grande quantité de mémoire qui pourrait être requise. C'est pourquoi, nous proposons une technique permettant de résumer cette information. L'exemple ci-dessous illustre notre approche.

Exemple 25. Considérons la liste d'hôtels présentée dans la table 4.1.

Nous avons une préférence pour les hôtels les moins coûteux, les plus près de la mer, présentant les chambres les plus grandes et disposant du Wifi. L'utilisateur a la possibilité de demander le skyline au regard de n'importe laquelle des $2^4 - 1$ requêtes correspondant aux sous-ensembles non vides de l'espace {**P**, **D**,

Id	(P)rix	(D)istance	(A)ire	(W)ifi
h_1	100	10	20	No
h_2	10	100	20	Yes
h_3	100	10	25	Yes

TABLE 4.1 – Les hôtels

A, **W**}. En comparant les hôtels h_2 et h_1, nous déduisons que h_2 domine h_1 en chacun des sous-espaces $subspaces_2$ = {**P**, **W**, **PW**, **AP**, **AW**, **APW**} [1]. Cependant, après cette comparaison, nous n'avons aucune information concernant les sous-espaces pour lesquels h_1 fait partie du skyline. Pour cela nous avons besoin de comparer h_1 et h_3 également. Une fois la comparaison effectuée, nous déterminons un nouvel ensemble de sous-espaces dans lesquels h_1 est dominé c'est-à-dire $subspaces_3$ ={**A**, **W**, **AW**, **AD**, **DW**, **AP**, **ADP**, **PW**, **ADW**, **APW**, **DPW**, **ADPW**}. $subspaces_2 \cup subspaces_3$ est l'ensemble des sous-espaces dans lesquels h_1 est dominé. Son complémentaire est l'ensemble des sous-espaces pour lesquels h_1 fait partie du skyline c'est-à-dire {**D**, **DP**}. En pratique, le nombre de sous-espaces associés à un tuple peut être très grand. Nous proposons donc une technique permettant de résumer cet ensemble tout en conservant l'efficacité des requêtes.

Organisation du chapitre et résumé des contributions Après la présentation de la problématique du résumé du skyline négatif, nous élaborons les algorithmes permettant de résoudre ce problème et montrons comment notre structure peut être utilisée à la fois (i) pour construire le skycube, en effet, notre structure de données requiert un temps de construction dans le pire des cas de l'ordre de $O\left([min(2^d, n)]^2 n\right)$ et occupe un espace dans le pire des cas de l'ordre de $O\left(min(2^d, n)n\right)$ (Section 4.1.1) ; ou (ii) pour accéléler le temps d'exécution des requêtes skyline sans avoir à construire le skycube ; bien que n'effectuant qu'un pré-calcul, l'exécution des requêtes (Section 4.1.2) ne nécessite pas des comparaisons supplémentaires entre les tuples. En effet, l'exécution des requêtes en faisant usage de la structure de données proposée consiste simplement en le parcours de celle-ci, sélectionnant les tuples satisfaisant un critère prédéfini. Nous allons plus loin en proposant également, une organisation de la structure de données, permettant de se passer de l'obligation de parcourir l'ensemble de la structure afin d'exécuter une requête. La dernière section du chapitre est consacrée aux résultats expérimentaux, afin de mettre en avant l'apport substantiel de notre approche comparée aux algorithmes de l'état de l'art. Nous terminons par une comparaison de nos deux approches proposées dans ce documents : le skycube partiel proposé au chapitre précédent et le skycube négatif que nous présentons dans la section suivante.

1. Nous verrons plus tard qu'il est facile d'obtenir cette information sans avoir à effectuer la comparaison au regard de chacun des sous-espaces.

Id	A	B	C	D
t_1	1	1	3	3
t_2	1	1	2	3
t_3	2	2	2	2
t_4	4	2	1	1
t_5	3	4	5	2
t_6	5	3	4	2

TABLE 4.2 – Une table de données T

4.1 Le Skycube Négatif

Dans ce chapitre, nous nous intéressons à l'optimisation des requêtes skyline. Notre contribution se résume en la proposition d'une structure de données qui, pour chaque tuple t appartenant à la table T, résume l'ensemble des sous-espaces X tels que t *n'appartient pas à* $Sky(X)$. Ceci est motivé par l'observation suivante : pour un tuple t, alors qu'il est nécessaire de le comparer à tous les autres tuples pour savoir s'il appartient à un skyline $Sky(X)$, le comparer à un seul autre tuple t' permet déjà de déterminer une partie de l'ensemble des sous-espaces dans lesquels t est dominé, c'est-à-dire, des sous-espaces pour lesquels il n'appartient pas aux skylines respectifs.

Exemple 26. La table 4.2 sert d'exemple tout le long de ce travail. Dans cet exemple, l'utilisateur pourrait demander les *meilleurs* tuples au regard de chaque combinaison de dimensions $\{A, B, C, D\}$. Par exemple, on a $Sky(AB) = \{t_1, t_2\}$ et $Sky(ABCD) = \{t_2, t_3, t_4\}$.

Nous commençons par présenter quelques définitions préliminaires.

Définition 4.1 (Sous-espaces de dominance)**.** Soit $t \in T$ et $X \subseteq \mathcal{D}$. X est un *espace de dominance* pour t si et seulement si $t \notin Sky(X)$.

Définition 4.2 (Skycube Négatif)**.** Soit $t \in T$ et soit $DomSubspaces(t)$ l'ensemble des sous-espaces de dominance de t. Le *skycube négatif* de T est l'ensemble $\{DomSubspaces(t) \mid t \in T\}$.

En d'autres termes, le skycube négatif maintient pour chaque tuple t, les sous-espaces dans lesquels t n'appartient pas à leurs skylines respectifs.

Exemple 27. À partir de l'exemple de la table 4.2, il est facile de vérifier que $DomSubspaces(t_1) = \{ABCD, ABC, ACD, BCD, AC, BC, CD, C, D\}$.

Clairement, si le skycube négatif de T est disponible, alors le calcul de tout skyline $Sky(X)$ est évident : pour chaque tuple t, $t \in Sky(X)$ si et seulement si $X \notin DomSubspaces(t)$.

Donc, le skycube négatif de T apparaît comme le complémentaire de son skycube (positif).

À présent, nous montrons que le skycube négatif peut être calculé plus efficacement que la solution naïve consistant à calculer d'abord le skycube *positif* et ensuite de dériver le skycube négatif. Nous commençons par montrer comment la comparaison de t et t' permet d'obtenir un ensemble de sous-espaces appartenant à $DomSubspaces(t)$.

La définition suivante part de l'idée selon laquelle une seule comparaison pourrait permettre de déterminer un certain nombre d'espaces dans lesquels un tuple est dominé. Par exemple, si, de la table 4.2, nous comparons t_2 à t_5, nous observons que t_2 domine t_5 pour chacune des dimensions A, B et C. De ce fait, nous pouvons conjecturer que t_5 n'appartient à aucun skyline $Sky(X)$ tel que $X \subseteq ABC$. par conséquent, cette unique comparaison nous a permis de déterminer le statut *skyline* de t_5 par rapport aux sept sous-espaces non vides de ABC. Dans la suite, nous formalisons cette intuition en commençant par définir ce que signifie *comparer* deux tuples.

Définition 4.3. Soit $t, t' \in T$. Nous définissons la fonction $\mathcal{COMPARE}$ entre deux tuples comme suit : $\mathcal{COMPARE}(t, t') = \langle X|Y\rangle$ tel que X est l'ensemble des dimensions D_j telles que $t'[D_j] < t[D_j]$ et Y est l'ensemble des dimensions D_k pour lesquelles t et t' sont égaux [2]. Alors, $t[D_\ell] < t'[D_\ell]$ pour tout $D_\ell \in D \setminus X \cup Y$.

Pour simplifier la notation, nous utilisons $\mathcal{COMPARE}(t)$ pour désigner l'ensemble $\cup_{t' \in T}\{\mathcal{COMPARE}(t, t')\}$.

Exemple 28. De la table 4.2, nous avons $\mathcal{COMPARE}(t_5, t_6) = \langle BC|D\rangle$ parce que $t_6[B] < t_5[B]$ (3 vs. 4), $t_6[C] < t_5[C]$ (4 vs. 5) et les deux tuples sont égaux sur la dimension D (2). Nous avons $\mathcal{COMPARE}(t_6, t_5) = \langle A|D\rangle$.

Évidemment, si $\mathcal{COMPARE}(t, t') = \langle X|Y\rangle$ alors $X \cap Y = \emptyset$.

Définition 4.4 (Couverture)**.** Soit $\langle X|Y\rangle$ une paire d'espaces disjoints et soit Z un espace. On dit que la paire $\langle X|Y\rangle$ *couvre* Z si et seulement si $Z \subseteq XY$ et $Z \cap X \neq \emptyset$.

Exemple 29. $p = \langle AC|B\rangle$ couvre les espaces A, AB, C, AC, BC and ABC. B n'est pas couvert par p car malgré le fait que $B \subseteq ACB$, on a $B \cap AC = \emptyset$.

La proposition suivante montre que les espaces couverts par la paire obtenue en comparant t à t' sont précisemment les espaces dans lesquels t' domine t.

Proposition 10. Soit $t, t' \in T$ et soit $\mathcal{COMPARE}(t, t') = \langle X|Y\rangle$. Alors t' domine t en chacun et uniquement les espaces Z couverts par $\langle X|Y\rangle$, c'est-à-dire, $t \notin Sky(Z)$.

2. des paires similaires sont définies dans Bøgh *et al.* [2014] mais ne sont pas utilisées de la même façon.

Démonstration. Soit Z un sous-espace couvert par la paire $\langle X|Y\rangle$. Alors, il existe deux sous-espaces disjoints Z_1 et Z_2 tels que $Z_1 \cup Z_2 = Z$, $Z_1 \subseteq X$, $Z_2 \subseteq Y$ et $Z_1 \neq \emptyset$. $Z_1 \subseteq X$ implique que $t' \prec_{Z_1} t$ et $Z_2 \subseteq Y$ implique que $t'[Z_2] = t[Z_2]$ donc $t' \prec_{Z=Z_1 \cup Z_2} t$. □

En fait, $\mathcal{COMPARE}(t, t')$ est un résumé de l'ensemble des espaces pour lesquels t est dominé par t'. Par conséquent, c'est un résumé d'une partie de l'ensemble des espaces pour lesquels t n'appartient pas aux skylines respectifs.

Définition 4.5. Soit E un ensemble de paires d'espaces. On note $\mathcal{COVER}(E)$ l'ensemble de tous les espaces couverts par au moins une des paires appartenant à E.

Exemple 30. $\mathcal{COVER}(\{\langle A|B\rangle, \langle AD|\emptyset\rangle\}) = \{A, AB, D, AD\}$

Définition 4.6. $\mathcal{DOM}(t)$ dénote l'ensemble des espaces Z tels qu'il existe t' qui domine t au regard de l'espace Z.

En d'autres termes, $\mathcal{DOM}(t)$ est l'ensemble de tous les sous-espaces de $\mathcal{D}$ pour lesquels t est dominé. Donc $2^{\mathcal{D}} \setminus \mathcal{DOM}(t, T)$ est l'ensemble des sous-espaces pour lesquels t appartient au skyline.

À ce niveau, il est possible de présenter un algorithme construisant le skycube négatif. Cette structure peut alors être utilisée pour répondre aux requêtes skyline.

L'algorithme 12 montre comment cette structure de données est construite. Son idée principale est simplement de comparer chaque paire de tuples (t, t') et d'ajouter $\mathcal{COMPARE}(t, t')$ à l'ensemble associé à t.

Dans le cas où $\mathcal{COMPARE}(t, t')$ est égal à la paire $\langle \mathcal{D}|\emptyset\rangle$ (voir ligne 5), cela signifie que t' domine t en toutes les dimensions. Par conséquent, t n'appartient à aucun skyline et l'ensemble qui lui est associé peut être réduit à une seule paire.

Algorithme 12 : BUILDNSC

Input : Table T
Output : A data structure NSC summarizing the skycube.

```
1 begin
2     foreach t ∈ T do
3         NSC[t] ← ∅
4         foreach t' ∈ T do
5             Add COMPARE(t, t') to NSC[t]
6     return NSC
```

Exemple 31. De la table 4.2, la structure de données retournée par BUILDNSC est présentée dans la table 4.3. Notons que les paires $\langle X|Y\rangle$ pour lesquelles $X = \emptyset$ ne sont pas considérées parce qu'elles ne couvrent aucun espace (voir Proposition 10).

Tuples	Paires
t_1	$\langle C\|ABD\rangle, \langle CD\|\emptyset\rangle, \langle D\|\emptyset\rangle$
t_2	$\langle D\|C\rangle, \langle CD\|\emptyset\rangle, \langle D\|\emptyset\rangle$
t_3	$\langle AB\|\emptyset\rangle, \langle AB\|C\rangle, \langle CD\|B\rangle$
t_4	$\langle AB\|\emptyset\rangle, \langle A\|B\rangle, \langle A\|\emptyset\rangle$
t_5	$\langle ABC\|\emptyset\rangle, \langle ABC\|D\rangle, \langle BCD\|\emptyset\rangle, \langle BC\|D\rangle$
t_6	$\langle ABC\|\emptyset\rangle, \langle ABC\|D\rangle, \langle ABCD\|\emptyset\rangle, \langle A\|D\rangle$

TABLE 4.3 – NSC relatif à T

L'algorithme 13 montre comment la structure de données précédente peut être utilisée pour exécuter une requête skyline $Sky(Z)$. Pour chaque tuple t, il scanne l'ensemble des paires qui lui sont associées. Si une paire couvrant Z est rencontrée, alors t n'appartient pas à $Sky(Z)$ sinon t est un point skyline de Z.

Algorithme 13 : EVALUATESKYLINE

```
Input : NSC structure, subspace Z
Output : Sky(Z)
1  begin
2      Sky(Z) ← ∅
3      foreach t ∈ T do
4          covered ← false
5          foreach p ∈ NSC[t] do
6              if p covers Z then
7                  covered ← true
8                  break
9          if covered=false then
10             Add t to Sky(Z)
11     return Sky(Z)
```

Propriétés de NSC

Puisque chaque tuple t est comparé à tous les autres, le nombre de paires qui lui sont associées ne peut pas excéder $n-1$. D'autre part, le nombre maximal de paires distinctes dépends du nombre de dimensions. En effet, ce nombre est $N = \sum_{i=1}^{d} \binom{d}{i} 2^i$ où d est le nombre de dimensions. Donc, une borne supérieure de la taille de NSC est $O(n * min(n, N))$. Il s'agit également de la complexité en temps de l'algorithme 13.

Notons que si la taille de NSC est $O(n * 2^d)$ alors cette taille est comparable à celle du skycube, ce qui signifie que non seulement que NSC ne présente pas d'avantage en terme de mémoire par rapport au skycube, mais en plus, elle n'est pas meilleure en terme de temps de requête. Dans la section suivante, nous abordons le problème de réduction de la taille de NSC.

4.1.1 Réduction de la taille de NSC

Réduire la taille de NSC permet non seulement de réduire la consommation de mémoire mais en plus d'améliorer le temps de réponse des requêtes skyline. Alors, le problème que nous abordons ici consiste à (i) supprimer les tuples fallacieux, c'est-à-dire ceux qui n'appartiennent à aucun skyline et (ii) pour chaque tuple restant, trouver un *ensemble minimal* de paires couvrant exactement les espaces couverts par $\mathcal{COMPARE}(t)$. Afin de donner une intuition à propos de ce procédé, considérons l'exemple suivant.

Exemple 32. Supposons $\langle \mathcal{D}|\emptyset\rangle \in \mathcal{COMPARE}(t)$. Puisque cette paire couvre tous les sous-espaces, nous concluons que t n'appartient à aucun skyline. Donc, t et l'ensemble de paires qui lui est associé peuvent être supprimés de NSC.

Exemple 33. Soit $p = \langle\emptyset|Y\rangle$ une paire associée à t. Clairement, $\mathcal{COVER}(p) = \emptyset$. De ce fait, la paire p peut être retirée sans que cela n'affecte le résultat des requêtes skyline.

Exemple 34. Soient $p_1 = \langle A|BC\rangle$ et $p_2 = \langle AB|C\rangle$ deux paires associées à t. On a $\mathcal{COVER}(\{p_1\}) = \{A, AB, ABC, AC\}$, de même, $\mathcal{COVER}(\{p_2\}) = \{A, AB, ABC, AC, B, BC\}$. En raison de l'inclusion, p_1 peut être supprimé sans que cela n'affecte le résultat des requêtes skyline.

Le test d'inclusion entre les paires peut être réalisé sans avoir besoin de générer l'ensemble des espaces que celles-ci couvrent. En effet, rappelons que le nombre de sous-espaces couverts est exponentiel de la taille de la paire, ce qui rend le test d'inclusion coûteux à réaliser.

Lemme 4.1. *Soient $p_1 = \langle X_1|Y_1\rangle$ et $p_2 = \langle X_2|Y_2\rangle$. Alors $\mathcal{COVER}(\{p_1\}) \subseteq \mathcal{COVER}(\{p_2\})$ si et seulement si (i) $X_1Y_1 \subseteq X_2Y_2$ et (ii) $X_1 \subseteq X_2$.*

Démonstration. Si $X_1Y_1 \not\subseteq X_2Y_2$ alors p_2 ne peut pas couvrir l'espace X_1Y_1 tandis que p_1 le couvre. Supposons à présent que la condition (i) est satisfaite mais pas la condition (ii). Alors, il existe une dimension $D_j \in X_1 \setminus X_2$ telle que la paire p_1 couvre le sous-espace $Z = D_j$ alors que ce n'est pas le cas pour p_2. Donc la condition (ii) est nécessaire. Ce qui conclut la preuve. □

L'égalité suivante est immédiate :
$\mathcal{DOM}(t, T) = \mathcal{COVER}(\bigcup_{t' \in T} \mathcal{COMPARE}(t, t'))$

Alors, matérialiser l'ensemble des paires $\bigcup_{t' \in T} \mathcal{COMPARE}(t, t')$ pour chaque tuple t nous permet de savoir si t appartient au skyline relatif à n'importe quel sous-espace $Z \in 2^{\mathcal{D}}$.

L'inconvénient de matérialiser l'ensemble $\bigcup_{t' \in T} \mathcal{COMPARE}(t, t')$ est le fait qu'il pourrait contenir de la redondance. Dans la section suivante, nous montrons comment réduire le nombre de paires tout en conservant les propriétés.

Approches pour la réduction de la taille de NSC

Le problème de réduction de la taille de NSC consiste à trouver, pour chaque tuple t, un ensemble minimal $\mathcal{P}$ de paires tel que l'ensemble $\mathcal{COVER}(\mathcal{P})$ soit égal à l'ensemble $\mathcal{COVER}(\mathcal{COMPARE}(t))$.

Nous abordons ce problème selon deux approches :

— Nous considérons en entrée l'ensemble des paires associées à t et essayons de trouver un sous-ensemble minimal de $\mathcal{COMPARE}(t)$.

— Nous considérons en entrée les espaces dans lesquels t est dominé et déterminons un ensemble minimal de paires couvrant ces espaces.

Du point de vue résumé, la solution de la seconde approche est naturellement préférable puisqu'elle est de plus petite taille. Notons cependant que l'entrée de la première approche est de taille plus petite que celle de la seconde approche : les paires contre les espaces couverts par ces paires.

Notre problème pourrait être formalisé comme suit :étant donné un tuple t et l'ensemble de paires qui lui est associé $\mathcal{COMPARE}(t)$. Comment déterminer un ensemble minimal de paires E^* tel que $\mathcal{COVER}(E^*) = \mathcal{COVER}(\mathcal{COMPARE}(t))$?

Ce problème peut également être reformulé en deux types de problèmes :

(i) le problème sous contrainte de ressource : $E^* \subseteq \mathcal{COMPARE}(t)$;

(ii) le problème sans contrainte : E^* n'est pas forcement un sous-ensemble de $\mathcal{COMPARE}(t)$

La section suivante (section 4.1.1) aborde le problème sous-contrainte inclusion (4.1.1) tandis que la section 6 aborde le problème sans cette contrainte (4.1.1).

Minimisation de $\mathcal{COMPARE}(t)$

Soit E un ensemble de paires, l'objectif de cette partie est de trouver un autre ensemble de paires $E^* \subset E$ dont la cardinalité (nombre de paires) est la plus petite possible et tel que $\mathcal{COVER}(E^*) = \mathcal{COVER}(E)$

Exemple 35. Soit $p_1 = \langle AB|C\rangle$, $p_2 = \langle BC|A\rangle$ et $p_3 = \langle AC|\emptyset\rangle$ trois paires associées au tuple t. Il est facile d'observer qu'il n'y a pas d'inclusion entre les ensembles d'espaces couverts par ces paires. Mais, la paire p_3 peut être retirée. En effet, $\mathcal{COVER}(p_3) = \{A, AC, C\}$. A et AC sont couverts par p_1 tandis que C est couvert par p_2. Donc, la paire p_3 est redondante.

Puisque $\mathcal{COVER}(E^*) = \mathcal{COVER}(E)$, matérialiser E^* est moins coûteux que de matérialiser E pour deux raisons :

— E^* requiert moins d'espace que E

— requêter E^* (vérifier qu'un espace Z est couvert) est plus rapide que requêter E

Le problème abordé dans cette section peut alors être formalisé comme suit :

> **Le problème RSP :** Étant donné un tuple t et l'ensemble de paires qui lui est associé $\mathcal{COMPARE}(t)$. Réduire la taille de l'ensemble de paires $\mathcal{COMPARE}(t)$ (**RSP**) revient à trouver un sous-ensemble de paires $E \subseteq \mathcal{COMPARE}(t)$ de taille minimale tel que $\mathcal{COVER}(E) = \mathcal{COVER}(\mathcal{COMPARE}(t))$.

Le théorème suivant montre que le problème **RSP** est NP-Difficile.

Théorème 4.2. RSP *est NP-Difficile.*

Démonstration. (esquisse) Évidemment, ce problème de minimisation est NP. La preuve qu'il est NP-Difficile est basée sur la réduction à partir du problème de l'ensemble minimal couvrant (minimal set cover, **MSC**). Étant donnée une instance de problème **MSC**, nous construisons une table T de tuples distincts t dans laquelle le nombre de dimensions d est égal au nombre d'éléments devant être couverts dans **MSC** et où le nombre de tuples n est égal au nombre initial d'ensembles dans **MSC**.

Nous établissons une correspondance entre E et les ensembles de **MSC** et montrons que toute solution de **MSC** est une solution de **RSP**.

Soit $s = \{s_1, s_2, \ldots, s_n\}$ l'ensemble d'ensembles d'éléments, entrée de l'instance du problème **MSC**. Soit $t = (1, 1, \ldots, 1)$ un tuple ayant d valeurs. Pour chaque ensemble $s_j \in$ **MSC**, nous ajoutons à T un tuple t' tel que $t'[i] = 0$ si et seulement si $i \in s_j$ sinon $t'[i] = 1$. Par conséquent, $\mathcal{COMPARE}(t, t') = \langle X|Y\rangle$ si et seulement si $\exists s \in$ **MSC** tel que $X = s$.

Par exemple, soit $s = \{s_1 = \{1, 2\}; s_2 = \{2, 3\}; s_3 = \{1, 3\}\}$ l'instance de **MSC**. Le nombre d'éléments à couvrir est $d = 3$ et le nombre d'ensembles $n = 3$. Alors, nous obtenons la table T ayant $n + 1 = 4$ tuples (incluant t) et 3 dimensions. Cette table est la suivante.

Id	**1**	**2**	**3**
t	1	1	1
t_1	0	0	1
t_2	1	0	0
t_3	0	1	0

Il y a une correspondance entre les $s_i \in s$ et les $p_i = Compare(t, t_i)$. Par exemple, $Compare(t, t_1) = \langle 12|3\rangle$ correspond à $s_1 = \{1, 2\}$. De ce fait, il peut être prouvé

qu'un ensemble s_i appartient à la solution de l'instance de **MSC** si et seulement si sa paire correspondante appartient à la solution du problème **RSP**. □

Chvatal [1979] a proposé un algorithme glouton approximatif en temps polynomial pour résoudre le problème ***MSC***. Cet algorithme choisit à chaque étape l'ensemble couvrant le plus grand nombre d'éléments non encore couverts par les ensembles précédemment choisis. L'adaptation de cet algorithme pour résoudre le problème **RSP** est évidente et présentée à travers l'algorithme 14.

Algorithme 14 : $ApproxMSP$

Input : Set of pairs : I
Output : Set of pairs : O

```
1 begin
2     O ← ∅
3     U ← COVER(I)
4     while U ≠ ∅ do
5         p ← arg max_{p'∈I} |COVER(p') \ COVER(O)|
6         O ← O ∪ {p}
7         U ← U \ COVER(p)
8     return O
```

Algorithme 15 : $listPairsTuple$, creates a reduced list of pairs associated to a tuple

Input : tuple : t, Set of tuples : $Topmost$
Output : Set of pairs : $listOut$

```
1 begin
2     listOut ← ∅
3     for each t' in Topmost do
4         add the pair COMPARE(t, t') to listOut
5     listOut ← ApproxMPC(listOut)
6     return listOut
```

L'algorithme 15 montre comment créer une liste réduite de paires associées à un tuple. Cet algorithme appelle la fonction *ApproxMSP* qui est une adaptation de la proposition de Chvatal [1979] au problème **RSP**.

Exemple 36. La minimisation de la table 4.3 utilisant l'algorithme 15 est présenté dans la table 4.4. Notons que le nombre de paires est passé de 20 à 8.

Tuples	Listes des paires associées
t_1	$\langle C\|ABD\rangle, \langle CD\|\emptyset\rangle$
t_2	$\langle CD\|\emptyset\rangle$
t_3	$\langle AB\|C\rangle, \langle CD\|B\rangle$
t_4	$\langle AB\|\emptyset\rangle$
t_5	$\langle ABC\|D\rangle$
t_6	$\langle ABCD\|\emptyset\rangle$

TABLE 4.4 – Liste des paires synthétisant les ensembles de dominance

Minimisation sans contrainte d'inclusion

Il est important de préciser que la solution exacte du problème **RSP** ne présente pas la garantie d'être le *plus petit* ensemble de paires codant l'ensemble des espaces dans lesquels t est dominé. Son idée maîtresse est juste de retirer les paires *redondantes*. Une autre manière d'aborder ce problème de minimisation est de considérer l'ensemble U des espaces couverts par l'ensemble des paires $\mathcal{P}$ associées au tuple t et d'essayer de dériver un ensemble minimal de paires $\mathcal{Q}$ qui couvre U. Notons que $\mathcal{Q}$ n'est pas forcement inclus dans $\mathcal{P}$. Nous illustrons la différence entre les deux approches à travers l'exemple suivant.

Exemple 37. Soit $\mathcal{P} = \{\langle AB|C\rangle, \langle BC|A\rangle\}$ l'ensemble des paires associées à t. $\mathcal{P}$ est minimal dans ce sens qu'il est imposible d'en retirer une paire quelconque tout en couvrant l'ensemble des espaces couverts par $\mathcal{P}$. Notons cependant que $\mathcal{Q} = \{\langle ABC|\emptyset\rangle\}$ couvre exactement les mêmes espaces que $\mathcal{P}$ et est de taille plus petite.

> **Le problème MSP :** Étant donnés un tuple t et l'ensemble qui lui est associé $\mathcal{DOM}(t)$ qui représente l'ensemble des espaces dans lesquels t est dominé. Le problème **MSP** consiste à trouver un ensemble minimal de paires E tel que $\mathcal{COVER}(E) = \mathcal{DOM}(t)$.

La difficulté que nous rencontrons à résoudre ce problème tient du fait que le skyline n'est pas monotone, c'est-à-dire $Y \subseteq X \not\Rightarrow Sky(Y) \subseteq Sky(X)$. La conséquence de cette propriété dans notre travail peut être présentée come suit : soit $X \in \mathcal{DOM}(t)$. On pourrait être tenté de coder cette information par la paire $\langle X|\emptyset\rangle$. Cependant, cette paire couvre tous les sous-espaces de X alors qu'il est possible qu'il existe un sous-espace $Y \subseteq X$ tel que $t \in Sky(Y)$. Dans la suite, nous mettons en exergue quelques propriétés nous permettant d'accomplir notre tâche consistant à résumer de cette information.

Lemme 4.3. *Soit $t \in T$ et soient X, Y deux espaces. Si $t \notin Sky_T(XY)$ et $t \in Sky_T(X)$ alors $t \notin Sky_T(Y)$.*

Preuve par l'absurde. Supposons que $\underbrace{t \in Sky_T(X)}_{(1)}, \underbrace{t \in Sky_T(Y)}_{(2)}$ et $\underbrace{t \notin Sky_T(XY)}_{(3)}$

De (1) et (3), il existe $u \in Sky_T(X)$ tel que $\underbrace{u[X] = t[X]}_{(4)}$ et $\underbrace{u \prec_Y t}_{(5)}$

De (2) et (3), il existe $v \in Sky_T(Y)$ tel que $\underbrace{v[Y] = t[Y]}_{(6)}$ et $\underbrace{v \prec_X t}_{(7)}$

Nous obtenons une contradiction entre (1) et (7) car t est dominé au regard de X et t appartient au skyline de X. Nous obtenons également une contradiction entre (2) et (5) car t est dominé au regard de Y et t appartient au skyline de Y. □

Ceci implique que, si $X \in \mathcal{DOM}(t)$ alors il existe au plus un fils X_1 de X [3] tel que $X_1 \notin \mathcal{DOM}(t)$.

Ce résultat peut être expliqué par le fait que pour tous X_1, X_2, deux sous-espaces distincts tels que $||X_1|| = ||X_2|| = ||X|| - 1$, nous avons $X_1 \cup X_2 = X$. En appliquant le proposition 4.3, si $t \notin Sky_T(X)$ et $t \in Sky_T(X_1)$ alors $t \notin Sky_T(X_2)$; alors il existe au plus un fils X_1 de X tel que $t \in Sky_T(X_1)$.

En allant plus loin, nous pouvons généraliser en établissant le résultat suivant. Intuitivement, il déclare qu'il existe au plus un plus grand sous-espace Z de X tel que $t \in Sky(Z)$ tandis que $t \notin Sky(X)$. plus formellement,

Proposition 11. Soit $X \in \mathcal{DOM}(t)$. Alors l'ensemble $\underset{Z \subset X, Z \notin \mathcal{DOM}(t)}{Argmax} ||Z||$ contient au plus un élément.

Ceci signifie que si $t \notin Sky(X)$ alors il existe au plus un sous-espace $Z \subset X$ de taille maximale (en nombre d'attributs) tel que t appartient au skyline de Z.

Démonstration. Supposons qu'il existe deux sous-espaces X_1 et X_2 de taille maximale tels que $X_1 \notin \mathcal{DOM}(t)$ et $X_2 \notin \mathcal{DOM}(t)$ alors $X_1 \cup X_2 \notin \mathcal{DOM}(t)$ (voir Proposition 4.3) ; D'autre part, puisque $X_1 \cup X_2 \subseteq X$ et $X_1 \neq X_2$ alors $||X_1 \cup X_2|| > max(||X_1||, ||X_2||)$. Il y a une contradiction car ceci reviendrait à dire qu'il existe un autre espace $(X_1 \cup X_2)$ plus grand que X_1 et X_2, inclus dans X et n'appartenant pas à $\mathcal{DOM}(t)$. □

De la proposition précédente, nous pouvons déduire le résultat suivant qui intuitivement montre que si Y est le plus grand sous-espace de X tel que $t \notin Sky(X)$ et $t \in Sky(Y)$ alors chaque $Z \subseteq X$ tel que $t \in Sky(Z)$ est nécessairement un sous-espace de Y. Formellement,

Proposition 12. Soit $X \in \mathcal{DOM}(t)$, Soit $Y = \underset{Z \subset X, Z \notin \mathcal{DOM}(t)}{Argmax} ||Z||$ alors $\forall Z \subset X$ tel que $Z \notin \mathcal{DOM}(t)$ nous obtenons $Z \subseteq Y$.

3. X_1 est un fils de X si et seulement si $X_1 \subset X$ et $||X_1|| = ||X|| - 1$.

Preuve par l'absurde. Soit $W \subset X$ tel que $W \notin \mathcal{DOM}(t)$ et $W \not\subseteq Y$ alors $W \cup Y \subseteq X$ et $W \cup Y \notin \mathcal{DOM}(t)$ (car W et Y n'appartiennent pas à $\mathcal{DOM}(t)$, voir Proposition 4.3), alors il y a une contradiction car $||W \cup Y|| > ||Y||$, $Y \neq \underset{Z \subset X, Z \notin \mathcal{DOM}(t)}{Argmax} ||Z||$. □

Intuitivement, ce résultat établit que chaque sous-espace de $X \in \mathcal{DOM}(t)$ qui n'appartient pas à $\mathcal{DOM}(t)$ est inclus dans un autre espace (noté Y) pour lequel t appartient au skyline. Alors, chaque sous-espace appartenant à $2^X \setminus 2^Y$ appartient à $\mathcal{DOM}(t)$, de ce fait, $\langle X \setminus Y | Y \rangle$ est la paire couvrant à la fois X et le maximum de sous-espaces de X inclus dans $\mathcal{DOM}(t)$.

La propriété précédente peut être exploitée pour résoudre le problème **MSP** comme suit : soit $X \in \mathcal{DOM}(t)$. Alors X correspond à la paire $\langle X \setminus Y | Y \rangle$ telle que Y est le plus grand sous-espace de X tel que $t \in Sky(Y)$. Si chaque $X \in \mathcal{DOM}(t)$ est remplacé comme décrit précédemment, alors nous obtenons un ensemble de paires dont la minimisation consiste simplement à supprimer les paires p_i telles qu'il existe une paire p_j qui la couvre. Cette procédure est décrite dans l'algorithme 16.

Algorithme 16 : $MinimalSP$, minimal set of pairs covering $\mathcal{DOM}(t)$

Input : $\mathcal{DOM}(t)$, d
Output : Set of pairs : $OutputSet$

```
1  begin
2      OutputSet ← ∅
3      Marqued ← ∅
4      Sort elements of DOM(t) w.r.t. their size in the descending order
5      for each X ∈ DOM(t) such that X ∉ Marqued do
6          Found ← false
7          Level ← ||X|| − 1
8          Y ← ∅
9          while not(Found) and Level ≥ 1 do
10             if ∃Z such that Z ⊂ X, ||Z|| = Level and Z ∉ DOM(t) then
11                 Found ← true
12                 Y ← Z
13             else
14                 Level ← Level − 1
15         Insert (X \ Y; Y) into OutputSet
16         Insert X into Marqued
17         for each W ∈ DOM(t) such that W is covered by (X \ Y; Y) do
18             Insert W into Marqued
19     return OutputSet
```

Proposition 13. La sortie de l'algorithme 16 couvre les espaces de $\mathcal{DOM}(t)$ et est calculée exactement en temps $O(|\mathcal{DOM}(t)|^2)$. En plus, cet ensemble est de taille minimale.

Démonstration. Soit $OutputSet$ la sortie de l'algorithme 16.

(I) nous montrons tout d'abord que $OutputSet$ couvre tous et seulement les espaces appartenant à $\mathcal{DOM}(t)$ (double inclusion).

(i) nous commençons par montrer que $\mathcal{COVER}(p) \subseteq \mathcal{DOM}(t)$, $\forall p \in OutputSet$. Soit $\langle W|T\rangle$ une paire appartenant à $OutputSet$, T est (par construction) le plus grand sous-espace de WT qui n'appartient pas à $\mathcal{DOM}(t)$. Alors T inclut tout sous-espace de WT n'appartenant pas à $\mathcal{DOM}(t)$ (voir Proposition 12), alors $2^{WT} \setminus 2^T = \mathcal{COVER}(\langle W|T\rangle) \subseteq \mathcal{DOM}(t)$

(ii) montrons que pour chaque espace Z appartenant à $\mathcal{DOM}(t)$ il existe une paire de $OutputSet$ qui couvre Z. Chaque itération (voir cinquième ligne) de l'algorithme ajoute à $OutputSet$ une paire qui couvre un espace de $\mathcal{DOM}(t)$, non encore couvert par les paires ajoutées précédemment. Et l'algorithme a dans le pire cas une complexité en temps de $O(m^2)$ où m est la taille de l'entrée. Alors, l'algorithme s'arrête après un temps fini, lorsque tous les espaces appartenant à $\mathcal{DOM}(t)$ sont couverts par au moins une paire de $OutputSet$

(II) Ensuite, montrons que $OutputSet$ est de taille minimale. Supposons que lp soit une liste de paires couvrant exactement les espaces de $\mathcal{DOM}(t)$. Nous allons montrer qu'il existe une fonction injective f allant de $OutputSet$ vers lp ce qui reviendrait à dire que la taille de $OutputSet$ est plus petite (ou égale à) la taille de tout lp. Pour f, nous choisissons la fonction qui associe à chaque paire $\langle X|Y\rangle \in OutputSet$ la paire $\langle x|y\rangle \in lp$ telle que $\langle x|y\rangle$ couvre XY (s'il y a plusieurs paires satisfaisant cette condition, alors nous choisissons la plus petite dans l'ordre lexicographique).

(i) Montrons que la fonction f est définie pour toute paire $\langle X|Y\rangle$ appartenant à $OutputSet$. $\forall\langle X|Y\rangle \in OutputSet$, l'espace XY appartient à $\mathcal{DOM}(t)$ donc il existe au moins une paire $p \in lp$ telle que p couvre XY.

(ii) Montrons enfin que la fonction f est injective ; en d'autres termes, ceci est équivalent à montrer que pour tous $\langle X_1|Y_1\rangle$ et $\langle X_2|Y_2\rangle$, deux paires appartenant à $OutputSet$, il n'existe pas de paire $\langle x|y\rangle \in lp$ couvrant à la fois X_1Y_1 et X_2Y_2. Par l'absurde, supposons qu'il existe une telle paire $\langle x|y\rangle \in lp$; alors $\underbrace{X_1Y_1 \subseteq xy}_{(1)}$ et $\underbrace{X_1Y_1 \cap x \neq \emptyset}_{(2)}$ aussi bien que $\underbrace{X_2Y_2 \subseteq xy}_{(3)}$ et $\underbrace{X_2Y_2 \cap x \neq \emptyset}_{(4)}$ (1) et (3) implique que $X_1Y_1 \cup X_2Y_2 \subseteq xy$ et (2) et (4) implique que $(X_1Y_1 \cup X_2Y_2) \cap xy \neq \emptyset$ alors $X_1Y_1 \cup X_2Y_2 \in$

$\mathcal{DOM}(t)$. Donc, il existe aussi une paire $\langle X|Y\rangle \in OutputSet$ couvrant $X_1Y_1 \cup X_2Y_2$ ce qui implique que $X \cap (X_1Y_1 \cup X_2Y_2) \neq \emptyset$, en d'autres termes $X \cap X_1Y_1 \neq \emptyset$ ou $X \cap X_2Y_2 \neq \emptyset$. Donc, $\langle X|Y\rangle$ couvre $\langle X_1|Y_1\rangle$ ou $\langle X_2|Y_2\rangle$ ce qui est absurde car, par construction, l'algorithme 16 ne peut pas ajouter à $OutputSet$ la paire $\langle X_1|Y_1\rangle$ (resp. $\langle X_1|Y_1\rangle$) si l'espace X_1Y_1 (resp. X_2Y_2) est déjà couvert par une autre paire ajoutée ($\langle X|Y\rangle$).

□

Comparer les tuples au Topmost skyline suffit

Dans cette partie, nous montrons que pour construire notre structure de données, il suffit de comparer les tuples à ceux appartenant au topmost skyline c'est-à-dire $Sky(\mathcal{D})$, au lieu de les comparer à l'ensemble des tuples de la table T. Ceci est particulièrement intéressant lorsque la taille du topmost skyline est petite par rapport à n.

Théorème 4.4. *Soit* $E = \bigcup_{t' \in Sky(\mathcal{D})} \mathcal{COMPARE}(t,t')$.
Alors $\mathcal{COVER}(E) = \mathcal{COVER}(\mathcal{COMPARE}(t))$.

Démonstration. On sait que

$$\mathcal{COMPARE}(t) = \bigcup_{t' \in Topmost} \mathcal{COMPARE}(t,t') \cup \bigcup_{t' \notin Topmost} \mathcal{COMPARE}(t,t')$$

or $\bigcup_{t' \in Topmost} \mathcal{COMPARE}(t,t') = E$,

En raison de la distributivité de l'opérateur $\mathcal{COVER}$ sur l'*union* nous pouvons écrire

$$\mathcal{COVER}(\mathcal{COMPARE}(t)) = \mathcal{COVER}(E) \cup \mathcal{COVER}\left(\bigcup_{t' \notin Topmost} \mathcal{COMPARE}(t,t')\right)$$

Il suffit alors de prouver que $\mathcal{COVER}\left(\bigcup_{t' \notin Topmost} \mathcal{COMPARE}(t,t')\right) \subset \mathcal{COVER}(E)$;
en d'autres termes, on pourrait juste montrer que pour tout $t' \notin Topmost$,

$$\mathcal{COVER}(\mathcal{COMPARE}(t,t')) \subseteq \mathcal{COVER}(E)$$

Soit t' un tuple de T tel que $t' \notin Topmost$. par définition du skyline, il existe un tuple $u \in Topmost$ tel que u domine t', c'est-à-dire, $u \prec_{\mathcal{D}} t'$. Soit $\langle X_1|Y_1\rangle = \mathcal{COMPARE}(t,u)$ et $\langle X_2|Y_2\rangle = \mathcal{COMPARE}(t,t')$. Pour chaque sous-espace Z couvert par $\langle X_2|Y_2\rangle$, nous avons (*i*) $t' \prec_Z t$. D'autre part, $u \prec_{\mathcal{D}} t'$ implique que (*ii*) $u \preceq_Z t'$ (car $Z \subseteq \mathcal{D}$).

De (*i*) et (*ii*) on a $u \prec_Z t$; donc Z est couvert par $\langle X_1|Y_1\rangle$. Tout espace couvert par $\langle X_2|Y_2\rangle$ est aussi couvert par $\langle X_1|Y_1\rangle$

Par conséquent, pour tout $t' \notin Topmost$, la paire $\mathcal{COMPARE}(t, t')$ est *redondante* dans la liste de paires associées au tuple t. □

Exemple 38. De notre exemple précédent, puisque le topmost skyline est constitué des tuples t_2, t_3 et t_4, les tuples seront uniquement comparés à ceux-ci. Par exemple, la liste de paires associée au tuple t_1 est $\{\langle C|ABD\rangle, \langle CD|\emptyset\rangle\}$ dont la taille est 2 en considérant le $Topmost$ au lieu d'un ensemble de 3 paires si on avait comparé t_1 à l'ensemble des tuples.

En plus du gain d'espace dû à la réduction du nombre de comparaisons pour chaque tuple, cette optimisation rend l'algorithme glouton utilisé pour résoudre le problème **RSP** plus rapide en raison de la réduction de la taille de l'entrée.

En termes de complexité, nos procédés requièrent dans le pire des cas en temps :

— n^2 comparaisons pour la recherche du topmost skyline

— $[min(2^d, n)]^2$ comparaisons par tuple pour la réduction des listes associées

Dans le pire des cas la structure de données occupe un espace de l'ordre de $O\left(min(2^d, n)n\right)$, ce chiffre correspond au cas où toutes les listes n'ont pas sensiblement été réduites.

4.1.2 Exécution des requêtes skyline

Dans cette section, nous décrivons comment la structure NSC peut être organisée et comment les requêtes skyline sont exécutées dans ces différents cas. Nous présentons alors deux procédures d'exécution des requêtes skyline à l'aide de la structure de données NSC. Chacune de ces procédures est basée sur une représentation spécifique du skycube négatif :

— une organisation orientée tuple dans laquelle à chaque tuple nous associons une liste de paires d'espaces ;

— une organisation orientée sous-espace dans laquelle à chaque espace nous associons une liste de paires de la forme $(tuple, subspace)$.

Dépendant de la nature des données et de la requête sousmise, l'une des représentations peut fournir de meilleures performances que l'autre.

Organisation orientée tuple

La manière la plus simple (et naïve) d'exécuter une requête relative à un espace donné est de vérifier que chaque tuple est dominé (par rapport à l'espace en question) en parcourant la liste des paires qui lui sont associées et en testant la couverture de l'espace en question par une des paires. S'il existe une paire couvrant l'espace en question alors le tuple correspondant n'appartient pas au skyline, sinon le tuple fait partie du skyline. Cette procédure est décrite plus formellement dans l'algorithme 17.

Algorithme 17 : NSC_SKYLINE from tuples

Input : Negative Skycube structure NSC, subspace Z, table T
Output : $Sky_T(Z)$

```
1 begin
2     NotSkylinePoints = ∅
3     foreach t ∈ T do
4         foreach pair ⟨X|Y⟩ in Compare(t) do
5             if Z is covered by ⟨X|Y⟩ then
6                 Add t to NotSkylinePoints
7                 break
8     return T \ NotSkylinePoints
```

L'inconvénient majeur de cette méthode d'exécution est le fait de devoir parcourir toute la structure de données pour chaque requête. Les paires pourraient être structurées de façon à ne parcourir et vérifier que les paires *pertinentes* ; ceci est l'objet de la section suivante.

Organisation orientée sous-espace

Dans cette vision, au lieu d'associer à chaque tuple une liste de paires, c'est-à-dire une représentation orientée tuple, nous utilisons une représentation orientée sous-espace. Plus précisement, nous utilisons une table qui associe à chaque sous-espace, une liste de paires de la forme $\langle tuple|subspace\rangle$ comme suit : soit $\langle X|Y\rangle$ une paire associée à t. Alors la paire $\langle t|Y\rangle$ est ajoutée à la liste correspondant à l'espace XY. L'algorithme 18 montre comment construire la structure de données NSC.

Exécuter une requête relative à un espace donné Z requiert de vérifier que Z est couvert par une paire de la liste associée au tuple. Pour toute paire $\langle X|Y\rangle$, cette vérification de la couverture consiste à réaliser deux tests :

— Z est-il un sous-espace de XY ?
— $Z \cap X$ est-il différent de l'ensemble vide ($\emptyset$) ?

En fait, les paires peuvent être organisées de telle sorte que seul le second test soit réalisé, ceci en ne parcourant que les espaces incluant l'espace Z sans plus avoir besoin de le vérifier. L'algorithme 18 montre comment cette structure de données est construite et l'algorithme 19 montre comment la reqête skyline $Sky_T(Z)$ est exécutée à partir de celle-ci.

Exemple 39. De notre exemple précédent, la structure de données associée est présentée dans la table 4.5. Remarquons que les espaces n'étant associés à aucun tuple ont été supprimés.

L'exemple suivant illustre le procédé des requêtes sur notre jeu de données 4.2.

Algorithme 18 : BUILD_NSC

Input : Table T (set of tuples)
Output : Negative Skycube Structure NSC

```
1 begin
2     Topmost = ComputeSkyline(T, D)
3     foreach tuple t in T do
4         list = listPairsTuple(t, Topmost)
5         foreach pair ⟨X|Y⟩ in list do
6             NSC[XY] = NSC[XY] ∪ {(t[id], Y)} ;
7     return NSC
```

Espaces	Liste de paires
AB	$\{\langle t_4 \vert \emptyset \rangle\}$
ABC	$\{\langle t_3 \vert C \rangle\}$
CD	$\{\langle t_1 \vert \emptyset \rangle, \langle t_2 \vert \emptyset \rangle\}$
BCD	$\{\langle t_3 \vert B \rangle\}$
ABCD	$\{\langle t_1 \vert ABD \rangle, \langle t_5 \vert D \rangle, \langle t_6 \vert \emptyset \rangle\}$

TABLE 4.5 – Organisation orientée sous-espace de NSC

Algorithme 19 : NSC_SKYLINE from subspaces

Input : Negative Skycube structure NSC, subspace Z, table T
Output : $Sky_T(Z)$

```
1 begin
2     NotSkylinePoints = ∅
3     foreach subspace W such that W ⊇ Z do
4         foreach pair (t, Y) in NSC[W] do
5             X = W \ Y
6             if Z is covered by ⟨X|Y⟩ then
7                 Add t to NotSkylinePoints
8     return T \ NotSkylinePoints
```

Données	d	n
NBA	17	20493
MBL	18	92797
IPUMS	10	75836

TABLE 4.6 – Les caractéristiques des données réelles

Exemple 40. Nous présentons comment la requête $Sky(AB)$ est exécutée en utilisant la table 4.5. Les listes associées aux sur-espaces de AB, c'est-à-dire, AB, ABC, et $ABCD$, sont parcourues. Pour AB, on trouve $\langle t_4|\emptyset\rangle$ ce qui signifie que nous avions associé à t_4 la paire $\langle AB|\emptyset\rangle$ qui nous informe que t_4 est dominé en AB. Pareil pour ABC et le tuple t_3. De la liste associée à $ABCD$, nous déduisons que pour t_1 nous avions la paire $\langle C|ABD\rangle$ qui ne couvre pas l'espace AB. De ce fait t_1 n'est pas dominé. Au regard des tuples t_5 et t_6, les paires $\langle ABC|D\rangle$ et $\langle ABCD|\emptyset\rangle$ leurs sont respectivement associées et couvrent AB signifiant que t_5 et t_6 sont dominés au regard de AB. Donc les seuls tuples qui ne sont pas dominés sont t_1 et t_2 qui constituent alors le skyline de AB, $Sky(AB)$.

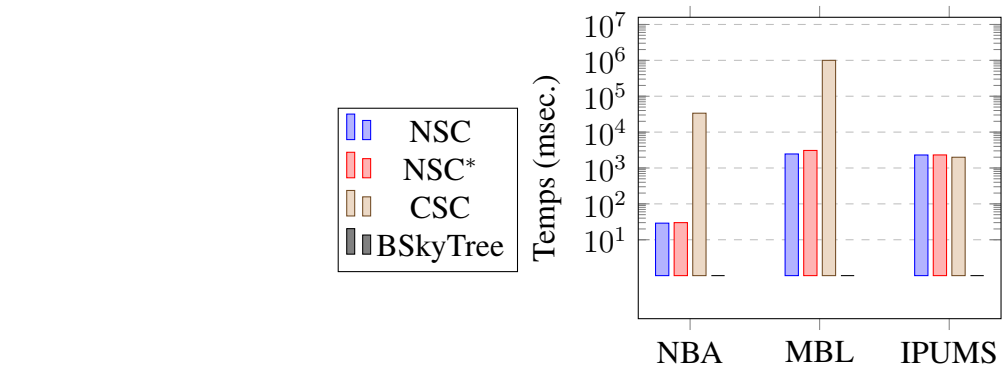
4.2 Évaluations expérimentales

Dans le but de comparer notre proposition aux algorithmes de l'état de l'art, nous avons implémenté nos solutions aussi bien que CSC Xia *et al.* [2012]. Nous avons utilisé les implémentations de BSkyTree Lee et won Hwang [2010], Hashcube Bøgh *et al.* [2014] et QSkyCube Lee et won Hwang [2014] fournies par leurs auteurs respectifs. Tous les programmes ont été écrits en C++. Nous avons utilisé une machine équipée de deux processeurs hexa-core et 24 Go sous Linux. Dans cette section, nous dénotons par NSC la structure skycube négatif réduite à l'aide de l'algorithme approximatif répondant au problème **RSP** et nous notons NSC* la structure utilisant la solution du problème **MSP**. Clairement, NSC* devrait être de plus petite taille que NSC.

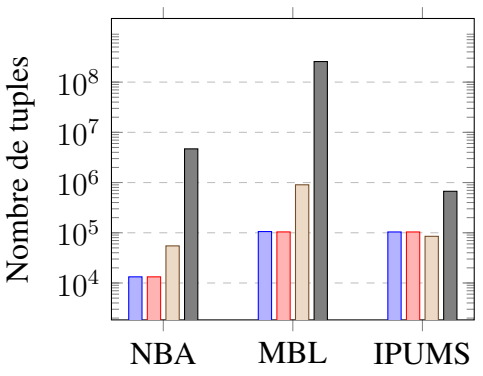
Pour les expérimentations, nous utilisons des données synthétiques obtenues suivant le procédé décrit par Börzsönyi *et al.* [2001]. Nous utilisons également un ensemble de données réelles régulièrement exploitées au sein de la communauté skyline. Nous considérons trois critères de comparaison : le temps de construction, la quantité de mémoire requise et le temps des requêtes. En ce qui concerne le dernier critère et pour éviter tout biais, nous considérons le temps nécessaire pour exécuter *l'ensemble* des $2^d - 1$ requêtes skyline.

4.2.1 Données réelles

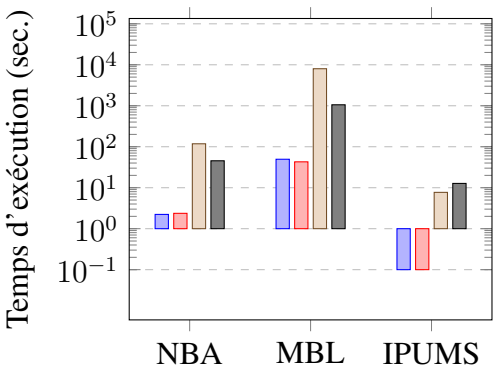
Ces données sont décrites dans la table 4.6.



(a) Temps de construction



(b) Utilisation de la mémoire



(c) Temps des requêtes

FIGURE 4.1 – Données réelles

Les résultats obtenus présentés dans la figure 4.1 montrent que NSC surpasse CSC en tous les critères : NSC est 100 fois plus rapide pour la construction de la structure. En ce qui concerne l'utilisation de la mémoire NSC ne requiert pas plus d'espace mémoire que CSC, par exemple, pour les données IPUMS, les deux algorithmes demandent presque la même quantité de mémoire, ce qui représente environ 10% de la mémoire requise par le skycube. Ce qui donne une première idée sur le *taux* de compression en fonction du nombre de dimensions : $d = 10$ pour IPUMS contre respectivement 17 et 18 pour NBA et MBL.

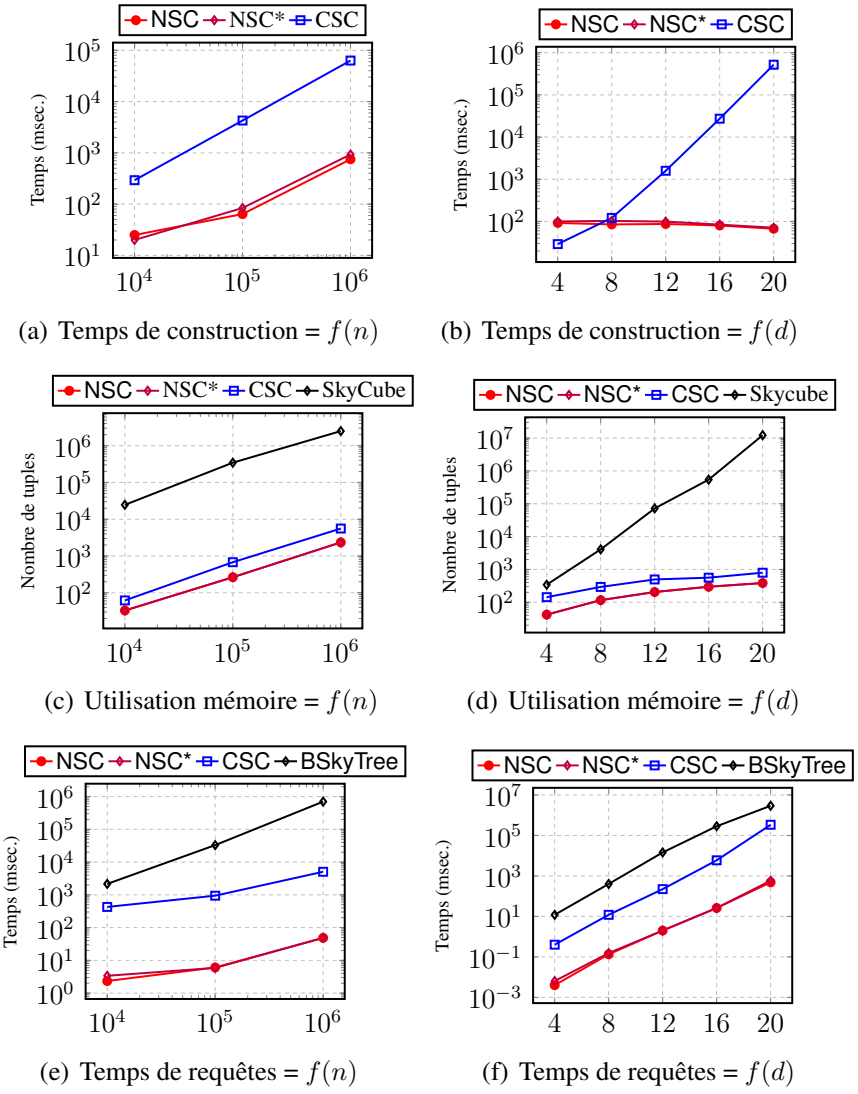
Nous constatons, au regard du temps de réponse que pour un grand nombre de dimensions (NBA et MBL), CSC a de pires performances que BSkyTree, une solution ne faisant aucun pré-calcul. Notons que la même observation avec des données synthétiques a été soulignée dans Bøgh *et al.* [2014]. Il est important de préciser que NSC et NSC* ont des comportement similaires quel que soit le critère retenu. Leurs courbes respectives sont presque toujours confondues.

4.2.2 Données synthétiques

Nous générons trois types de données synthétiques : corrélées, indépendantes et anti-corrélées suivant l'outil fournit par Börzsönyi *et al.* [2001]. Pour chacun d'eux, nous comparons NSC et NSC* à CSC en faisant varier le nombre de tuples n dans l'ensemble $\{10^4, 10^5, 10^6\}$ et en maintenant le nombre dimensions d égal à 14. Nous faisons ensuite varier le nombre de dimensions d dans l'ensemble $\{4, 8, 12, 16\}$, tout en conservant le nombre de tuples n égal à 10^5.

Pour tous les jeux de données, nous avons fixé le nombre de valeurs distinctes par dimension à 10^3. Ceci parce que le générateur original tend à retourner des données présentant des valeurs toutes distinctes par dimension, restreignant le champ des expérimentations à un cas très particulier de données. Pour obtenir nos données, nous avons juste multiplié chaque flottant, dont les valeurs sont comprises entre 0 (inclus) et 1 (exclus) par 10^3 et retenu la partie entière. Donc, nos données ont des valeurs entières allant de 0 à 999. Par exemple, les valeurs 0.0524469 et 0.0528678 sont toutes mappées à 52. Les résultats sont présentés dans les figures 4.2, 4.3 et 4.4.

Pour les données corrélées (figure 4.2), nous observons que lorsque d augmente, NSC surpasse largement CSC au regard des temps de construction et de requêtes (voir figures 4.2(b) et 4.2(f)). Au regard de n, les deux méthodes semblent atteindre des temps de construction et de requêtes presque linéaires (voir figures 4.2(a) et 4.2(e)). En termes d'utilisation de la mémoire, les deux algorithmes sont comparables même si NSC est légèrement plus petit. Les résultats obtenus avec les données indépendantes sont présentés dans la figure 4.3. Les mêmes remarques tiennent pour les données indépendantes (figure 4.3) et les données anti-corrélées (figure 4.4). Cependant, il est important de préciser que lorsque $n = 10^6$ et $d = 20$ (figure 4.4(b)) CSC n'avait pas terminé la construction de la structure de données après 36 heures tandis que NSC a été construite en environ de 3 heures. De la structure de

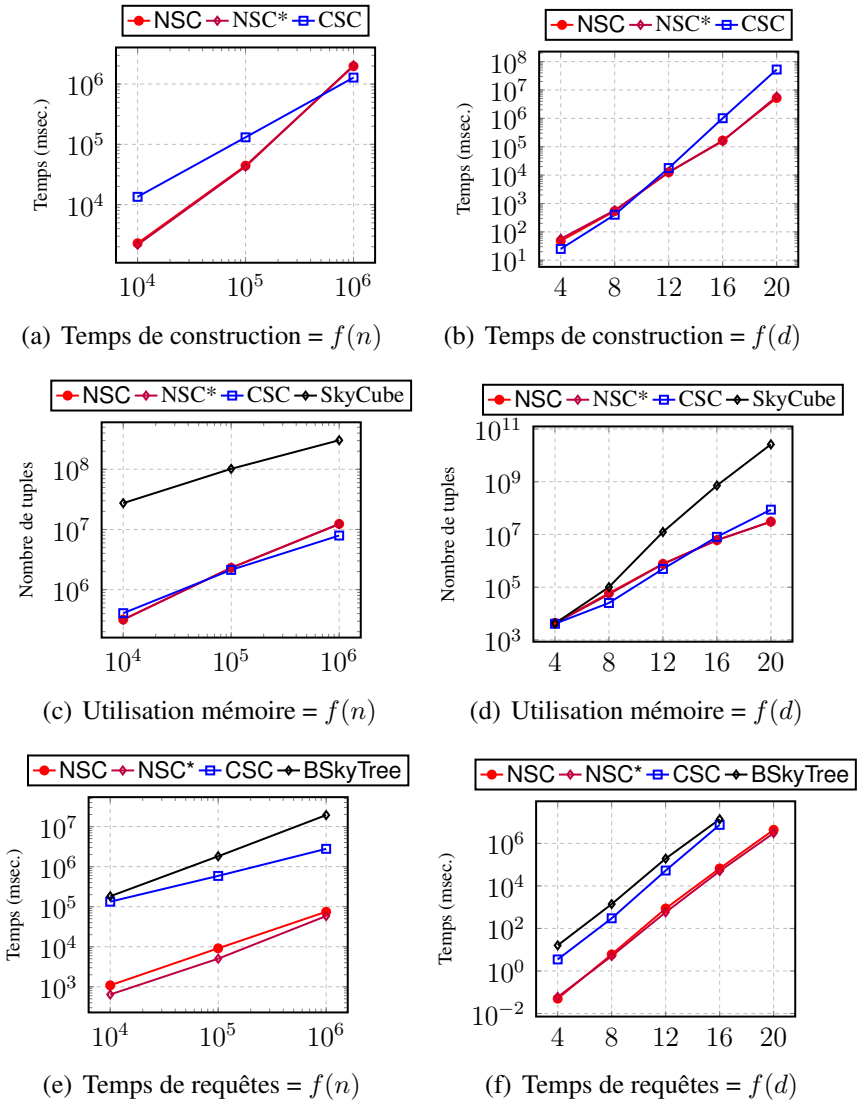


(a) Temps de construction = $f(n)$ (b) Temps de construction = $f(d)$

(c) Utilisation mémoire = $f(n)$ (d) Utilisation mémoire = $f(d)$

(e) Temps de requêtes = $f(n)$ (f) Temps de requêtes = $f(d)$

FIGURE 4.2 – Données synthétiques corrélées



(a) Temps de construction = $f(n)$

(b) Temps de construction = $f(d)$

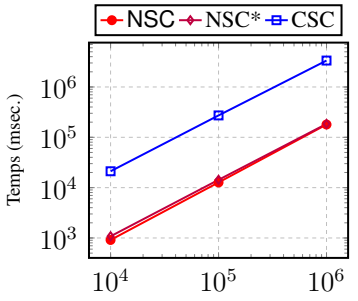
(c) Utilisation mémoire = $f(n)$

(d) Utilisation mémoire = $f(d)$

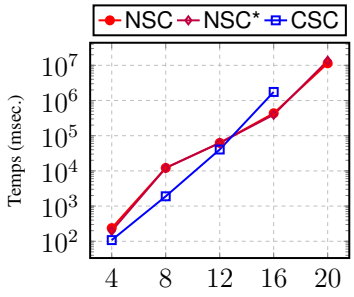
(e) Temps de requêtes = $f(n)$

(f) Temps de requêtes = $f(d)$

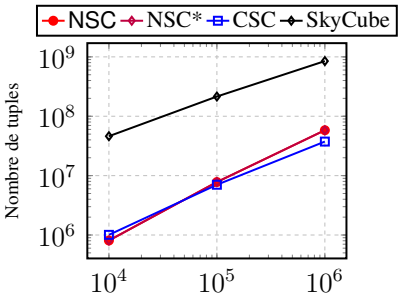
FIGURE 4.3 – Données synthétiques indépendantes



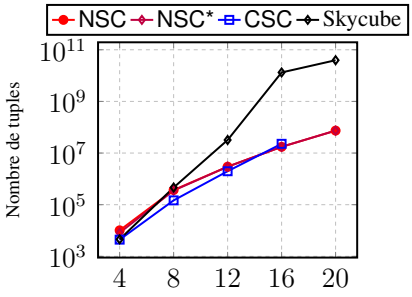
(a) Temps de construction = $f(n)$



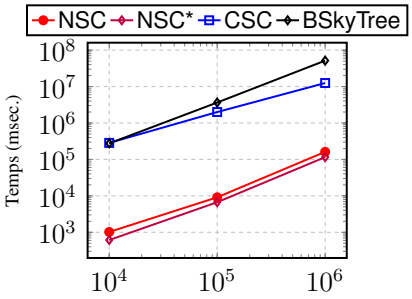
(b) Temps de construction = $f(d)$



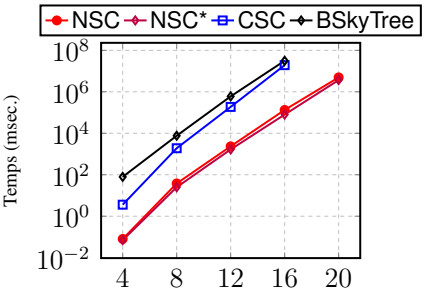
(c) Utilisation mémoire = $f(n)$



(d) Utilisation mémoire = $f(d)$



(e) Temps de requêtes = $f(n)$



(f) Temps de requêtes = $f(d)$

FIGURE 4.4 – Données synthétiques anticorrélées

données construite, les $2^{20} - 1$ requêtes ont été exécutée en environ une heure. CSC pourrait alors plus rapidement être construite à partir du skycube obtenu de cette façon qu'en utilisant la version originale de l'algorithme fournie dans Xia *et al.* [2012]. Ceci revient à dire que, utiliser un algorithme naïf pour construire CSC en construisant en premier le skycube et ensuite en déterminant pour chaque tuple les plus petits espaces pour lesquels il appartient au skyline est plus rapide que l'algorithme complexe et inefficace proposé.

Il est cependant intéressant de remarquer que, pour de petites valeurs de d (figure 4.3(d) et 4.4(d)), $d = 4$), non seulement CSC requiert moins d'espace mémoire que NSC mais c'est aussi le cas pour l'ensemble du skycube. En d'autres termes, le nombre de skylines auxquels les tuples n'appartiennent pas est plus élevé que le nombre de skylines auxquels les tuples appartiennent. D'autre part, nous remarquons que CSC a presque la même taille que le skycube. Ceci signifie que dans de telles situations (d petit), il n'est pas recommandé d'utiliser NSC et encore moins CSC : le skycube entier est la meilleure option.

4.2.3 Comparaison à Hashcube

Dans cette partie, nous comparons NSC à la structure de données Hashcube proposée dans Bøgh *et al.* [2014]. Nous avons utilisé l'implémentation de Hashcube fournie par les auteurs [4]. Il est important de préciser que l'implémentation de Hashcube a besoin que l'ensemble du skycube soit calculé puisque c'est ce dernier qu'elle prend en entrée. De ce fait, nous ne pouvons pas considérer le temps de construction. Avec les données NBA, NSC garde en mémoire 13259 tuples tandis que Hashcube stocke 541316. D'autre part, répondre à l'ensemble des $2^{17} - 1$ requêtes requiert 30 millisecondes à Hashcube contre 2.1 secondes pour NSC.

Dans le but d'éprouver la structure de données Hashcube, nous avons généré un jeu de données synthétique présentant des dimensions indépendantes. Ce jeu de données contient 10^7 tuples et 16 dimensions. Son skycube requiert de stocker $20 * 10^9$ tuples. Nous avons essayé de construire la structure de donnée Hashcube correspondante mais les 24Go de mémoire disponibles n'ont pas été suffisants pour la conserver. Pour le même jeu de données, NSC utilise moins de 12Go pour stocker environ $4.5 * 10^7$ tuples. Par conséquent, lorsque le skycube est *trop grand*, ce qui arrive lorsque le nombre de dimensions est élevé, Hashcube ne peut pas être entièrement chargé en mémoire. Ce que nous pouvons tirer de cette expérimentation est que : on pourrait choisir Hashcube lorsque le nombre de dimensions est relativement petit. Dans de telles situations, il est préférable d'entièrement matérialiser le skycube, sinon NSC doit alors être choisi.

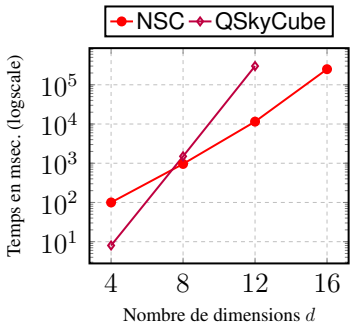


FIGURE 4.5 – Construction du Skycube : NSC vs QSkyCube

4.2.4 Construction du skycube

Dans cette section, nous comparons notre contribution à QSkyCube Lee et won Hwang [2014] récemment proposé pour le calcul du skycube complet. La figure 4.5 montre certains résultats que nous avons obtenus. Nous avons généré des données synthétiques indépendantes fixant le nombre de tuples n à 10^5 et faisant varier le nombre de dimensions d dans l'ensemble $\{4, 8, 12, 16\}$. Il est important de souligner le fait que pour NSC, nous avons reporté le temps d'exécution total, c'est-à-dire le temps de construction de la structure et celui de l'exécution des $2^d - 1$ requêtes. Comme cela peut être observé, lorsque d augmente, c'est-à-dire $d > 8$, NSC devient plus rapide. Il est intéressant de noter que lorsque d atteint 16, QSkyCube a été arrêté car ayant saturé toute la mémoire disponible. Cette expérimentation montre que (i) notre proposition est compétitive pour le calcul du skycube (particulièrement) lorsque d devient grand, et (ii) notre proposition peut être considérée comme la première étape pour construire Hashcube puisque ce dernier requiert que le skycube soit entièrement construit.

4.2.5 Comparaison avec le Skycube Partiel (MICS)

Dans l'objectif de comparer nos deux propositions (NSC et MICS), nous avons considéré des jeux de données présentant des dimensions indépendantes. Dans un premier temps nous avons fait varier le nombre de tuples n, prenant les valeurs successives 10^4, 10^5 et 10^6, tout en fixant à 14 le nombre de dimensions. Dans une seconde expérience, nous avons fait varier le nombre de dimensions, prenant les valeurs 8, 12 et 16, et nous avons fixé le nombre de tuples à 10^5. Pour les deux approches, nous avons effectué le pré-calcul de la structure de données et exécuté l'ensemble des 2^d requêtes possibles. Les temps de construction, taille des structures de données et temps des requêtes respectifs sont reportés graphiquement dans la figure 4.6.

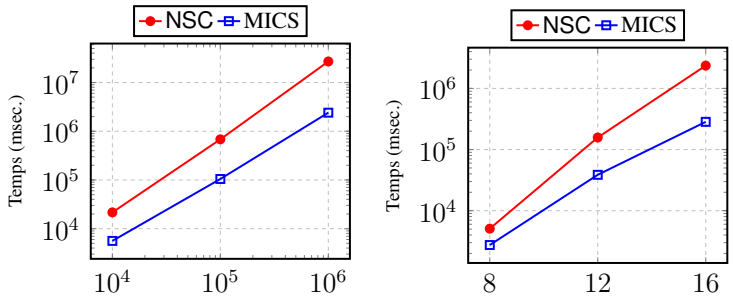
4. Pour les besoins de comparaison, les auteurs ont également implémenté la technique CSC. Nous ne l'avons pas utilisée parce qu'elle était plus lente que la notre.

L'observation est presque automatiquement toujours la même : MICS est un peu moins de 10 fois plus performant en temps et en espace que NSC (Figures 4.6(a), 4.6(b), 4.6(c) et 4.6(d)) tandis que ce dernier est environ 100 fois plus rapide que MICS lors de l'exécution des requêtes (Figures 4.6(e) et 4.6(f)). Ceci pourrait s'expliquer par le fait que MICS effectue les comparaisons entre les tuples au moment des requêtes (skyline d'un skycuboid) tandis que NSC les effectue uniquement dans la phase de pré-calcul.

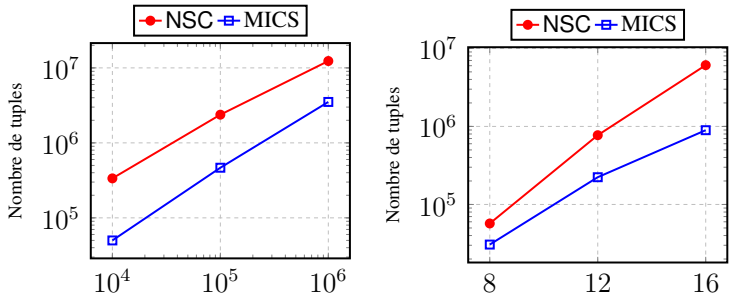
On voit bien, une nouvelle fois, à une échelle moins extrême, cet aspect *compromis pré-calcul/requête*. De ce fait, le choix de l'une ou l'autre des deux approches pourrait également dépendre du cas d'utilisation ou encore d'autres critères liés aux propriétés des structures de données ; parmi ces propriétés on pourrait par exemple citer : la facilité avec laquelle elle pourrait être implémentée en environnement distribué ou encore la simplicité d'adaptation aux données dites dynamiques (lorsqu'on envisagerait des insertions ou suppressions de tuples).

Conclusion

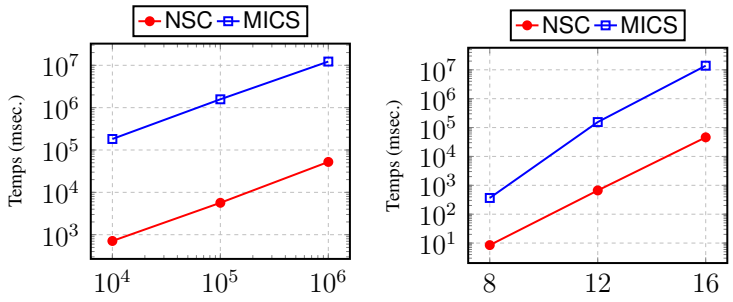
Nous avons présenté la structure de données Skycube Négatif et fourni les algorithmes permettant de la réduire. L'idée maitresse de cette structure est de stocker pour chaque tuple un résumé de l'ensemble des sous-espaces pour lesquels *il n'appartient pas au skyline*. Ceci est à contre-courant avec les anciens travaux dont l'objectif était de résumer l'ensemble des sous-espaces pour lesquels le tuple *appartient* au skyline. Nos expérimentations ont montré que NSC est particulièrement efficace en terme de temps de construction, d'utilisation de la mémoire et de temps de réponse. Dans les précédents travaux, soit les algorithmes demandaient beaucoup de temps pour la construction, soit la structure de données produite était trop grande pour tenir en mémoire. Nos expérimentations montrent également que cette structure est plus rapide pour construire le skycube que les algorithmes de l'état de l'art spécialement conçus dans cet objectif.



(a) Temps de construction = $f(n)$ (b) Temps de construction = $f(d)$



(c) Utilisation mémoire = $f(n)$ (d) Utilisation mémoire = $f(d)$



(e) Temps de requêtes = $f(n)$ (f) Temps de requêtes = $f(d)$

FIGURE 4.6 – Données synthétiques indépendantes

Chapitre 5

Le Skycube Négatif : Extensions

Introduction

Dans ce chapitre, nous présentons deux extensions de la structure *Skycube Négatif*. En pratique, nous abordons, à partir de la structure de données NSC, d'autres problématiques liées aux requêtes skyline. Nous commençons par étudier un autre type de requête, à savoir les requêtes k-dominant skyline (dont les requêtes skyline sont un cas particulier). Après un bref parcours des travaux relatifs à ce sujet (Section 5.1.1), nous décrivons une manière efficace de répondre à ces requête à partir du *Skycube Négatif* (Section 5.1.2). Efficacité qui s'avère confortée par les expérimentations comparatives qui suivent (Section 5.1.3). Dans la seconde partie du chapitre, nous abordons la question des requêtes skyline au sein de données dynamiques. Certes, cette problématique est abordée dans la littérature comme nous le présenterons brièvement (Section 5.2.1), mais les approches existantes formulent généralement des contraintes supplémentaires sur les données (par exemple l'ordre de suppression des tuples doit être le même que celui de l'insertion). Nous apportons alors une modification à notre structure de données *Skycube Négatif* (Section 5.2.2) afin qu'elle puisse supporter la mise à jour des données (insertions et suppressions de tuples).

5.1 Les requêtes k-dominant Skyline

Dans le but de réduire la taille du résultat des requêtes skyline, la requête k-dominant skyline a été définie. Un tuple est k-dominé ($k > 0$) s'il existe un autre tuple et un sous-espace dont la taille est au moins k, dans lequel le premier est dominé par le second. Alors, le k-dominant skyline est l'ensemble des tuples qui ne sont pas k-dominés. Cette section est dédiée à la présentation de l'apport de la structure de données NSC dans l'exécution de requêtes k-dominant skyline, dont les requêtes skyline sont un cas particulier.

5.1.1 Travaux relatifs

Il existe plusieurs travaux autour des requêtes k-dominant skyline, la plus part d'entre-eux Chan *et al.* [2006a]; Siddique et Morimoto [2009]; Md. Anisuzzaman et Yasuhiko [2010]; Dong *et al.* [2010]; Siddique et Morimoto [2012] présentent des algorithmes pour exécuter cette requête, mais seulement Dong *et al.* [2010] présente une méthode pour optimiser le calcul de plusieurs requêtes, ces requêtes sont obtenues en faisant varier le paramètre k au sein du même espace de référence $\mathcal{D}$. Ils proposent deux algorithmes calculant le k-dominant *skyline cube* qui est l'ensemble de tous les k-dominant skyline ($k = 1 \ldots d$). Le premier algorithme, appelé *Bottom-Up Algorithm* (en abrégé *BUA*), repose sur le résultat de $(k-1)$-dominant skyline pour calculer le k-dominant skyline ; le second algorithme, appelé *Up-Bottom Algorithm* (en abrégé UBA), utilise le résultat du $(k+1)$-dominant skyline afin de calculer le k-dominant skyline. Aucun des précédents travaux ne fournit un algorithme optimisant le calcul de toutes les requêtes k-dominant skyline obtenues en faisant varier non seulement k mais aussi l'espace de référence. Siddique et Morimoto [2012], le plus récent, présente une technique pour calculer le k-dominant skyline reposant sur l'indexation (Domination Power Index). Les auteurs montrent, tout comme Dong *et al.* [2010], que leur algorithme est plus efficace que l'algorithme *TSA* élaboré par Chan *et al.* [2006a].

5.1.2 NSC et le k-dominant skyline

Dans cette partie, nous rappelons la définition de la requête k-dominant skyline et nous montrons comment le Skycube Négatif peut être utilisé pour répondre à cette classe de requêtes.

Définition 5.1 (k-dominer)**.** Étant donné $k > 0$, on dit que le tuple t k-domine le tuple t' si et seulement si $\exists X \subseteq \mathcal{D}, ||X|| = k$ tel que $t \prec_X t'$

Définition 5.2 (k-dominant skyline)**.** Le k-dominant skyline est l'ensemble des tuples qui ne sont pas k-dominés.

Le k-dominant skyline a été défini pour réduire la taille du résultat des requêtes skyline. En effet, la borne supérieure de la taille du résultat d'une requête skyline n'est limitée que par la taille de l'entrée. Le k-dominant skyline peut être retrouvé à partir de l'ensemble skyline car un tuple t appartient au k-dominant skyline si et seulement si il n'existe pas de sous-espace X ($||X|| \geq k$) au regard duquel t est dominé. Ceci implique que si t n'est pas k-dominé alors t n'est pas l-dominé ($k \leq l \leq d$). Alors, t n'est pas d-dominé (t appartient au skyline).

Contrairement aux requêtes skyline, les requêtes k-dominant skyline peuvent retourner un résultat vide, dans le cas où chaque tuple est k-dominé par un autre. En guise d'exemple, supposons que T contient exactement deux tuples t_1 et t_2 définis en deux dimensions telles que $t_1 = (1, 2)$ et $t_2 = (2, 1)$. Le 1-dominant skyline est vide car $t_1 \prec_{D_1} t_2$ et $t_2 \prec_{D_2} t_1$. Néanmoins, il existe un seuil k_{min} au dessus duquel

toute requête k-dominant skyline ($k \geq k_{min}$) n'est pas vide parce qu'un tuple t qui n'est pas k_{min}-dominé ne peut pas être k-dominé ($k \geq k_{min}$), donc t appartient à tous les k-dominant skyline ($k \geq k_{min}$).

En construisant le skycube négatif NSC, le seuil k_{min} pourrait être déterminé comme suit.

Proposition 14. Soit E_i l'ensemble des paires associées au tuple t_i. Au regard de l'espace complet $\mathcal{D}$, le seuil k_{min} au-delà duquel chaque requête k-dominant n'est pas vide est

$$k_{min} = \min_{i=1...n} \left\{ \max_{\langle X|Y\rangle \in E_i} ||XY|| \right\} + 1$$

Démonstration. Soit t_i un tuple de la table T et soit E_i un ensemble de paires synthétisant les espaces dans lesquels t_i est dominé.

$$E_i = \{\langle X_{i,1}|Y_{i,1}\rangle, \langle X_{i,2}|Y_{i,2}\rangle, \ldots, \langle X_{i,m_i}|Y_{i,m_i}\rangle\}$$

Soit $c_{i,j}$ le nombre d'attributs composant l'espace $X_{i,j}Y_{i,j}$, notons $c_{i,j} = ||X_{i,j}Y_{i,j}||$. Le tuple t_i est $c_{i,j}$-dominé parce qu'il existe un espace dont la taille est au moins $c_{i,j}$ dans lequel il est dominé (cet espace est $X_{i,j}Y_{i,j}$). Mais t_i ne peut pas être $\left(\max_{j=1...m_i} c_{i,j} + 1\right)$-dominé car il n'existe pas de sous-espace dont la taille est supérieure à $\max_{j=1...m_i} c_{i,j}$ dans lequel t_i est dominé (autrement il existerait une paire appartenant à E_i contenant au moins $\max_{j=1...m_i} c_{i,j} + 1$ attributs). Alors, pour toute requête k-dominant skyline telle que $k \geq \max_{j=1...m_i} c_{i,j} + 1$, t_i appartient au résultat. Mais t_i ne peut appartenir à un k-dominant skyline avec $k \leq \max_{j=1...m_i} c_{i,j}$.

Plus généralement, toute requête k-dominant skyline avec $k \geq k_{min}$ avec $k_{min} = \min_{i=1...n} \left\{ \max_{j=1...m_i} c_{i,j} + 1 \right\}$ ne peut être vide car le résultat contient au moins un tuple t_a où

$$a = \arg\min_{i=1...n} \left\{ \max_{j=1...m_i} c_{i,j} \right\}$$

Puisque t_a n'est pas k_{min}-dominé alors t_a n'est pas k-dominé avec $k > k_{min}$.

Mais chaque tuple t_i est k-dominé avec $k < k_{min}$ car E_i contient au moins une paire d'au moins $k_{min} - 1$ attributs. □

Après avoir montré comment calculer k_{min} à partir de la structure de données NSC, nous passons à l'exécution des requêtes k-dominant skyline proprement dites.

L'algorithme 20 présente comment la structure de données NSC nous permet d'exécuter toute requête k-dominant skyline au sein d'un sous-espace Z ($1 \leq k \leq d$ and $Z \subseteq D$). Il apparait clairement que la requête k-dominant skyline est exécutée à partir de la structure NSC presque de la même manière que cela est fait pour les requêtes skyline (voir l'algorithme 19). ceci est dû au fait que la requête skyline est un cas particulier de requête k-dominant skyline (le cas où k est égal à la

taille de l'espace pour lequel la requête est effectuée). Puisque chaque paire $\langle X|Y\rangle$ associée au tuple t indique le plus grand espace dans lequel t est dominé par un tuple particulier, s'il n'existe pas de paire $p = \langle X|Y\rangle$ associée à t telle qu'il existe un sous-espace $V \subseteq Z$ couvert par p, $||V|| \geq k$ alors t n'est pas k-dominé dans le sous-espace Z. Cette explication intuitive de l'exécution de requêtes k-dominant skyline est formellement traduite sans l'algorithme 20.

Algorithme 20 : NSC_k-DOMINANT_SKYLINE from subspaces

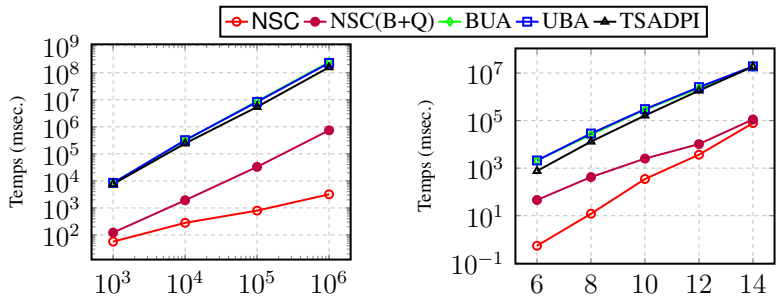
Input : Negative Skycube structure NSC, subspace Z, Threshold k, table T
Output : $kDomSky_T(Z)$

```
1 begin
2     kDominatedPoints = ∅
3     foreach subspace W such that ||W ∩ Z|| ≥ k do
4         foreach pair (t, Y) in NSC[W] do
5             X = W \ Y
6             if W ∩ Z is covered by ⟨X|Y⟩ then
7                 Add t to kDominatedPoints
8     return T \ kDominatedPoints
```

5.1.3 Expérimentations

Dans cette partie, nous analysons la performance des requêtes utilisant la structure NSC en la comparant à ceux des algorithmes de l'état de l'art. Pour les besoins de la comparaison, nous considérons les algorithmes UBA, BUA décrits dans Dong *et al.* [2010], aussi bien que $TSADPI$ défini dans Siddique et Morimoto [2012]. Nous définissons le k-dominant skycube comme l'ensemble de toutes les requêtes k-dominant skyline possibles relatives au jeu de données, c'est-à-dire pour chaque sous-espace $X \subseteq D$, nous exécutons toutes les requêtes k-dominant skyline où $1 \leq k \leq ||X||$. Comme première étape, nous considérons seulement le temps d'exécution lorsque le skycube négatif est déjà construit, nous l'appelons *NSC*. Ensuite, nous ajoutons au temps de requête le temps de construction de la structure de données, cette version sera notée *NSC(B+Q)*, où *B+Q* est mis pour le temps de construction + requête (Build+Query). Nous utilisons des données synthétiques indépendantes. Dans la première expérimentation, le nombre de dimensions d est fixé à 10 et nous varions le nombre de tuples $n \in \{10^3, 10^4, 10^5, 10^6\}$. Dans la seconde expérimentation, le nombre de tuples n est fixé à 10^5, le nombre de dimensions prend ses valeurs dans l'ensemble $\{6, 8, 10, 12, 14\}$ et les dimensions possèdent chacune 100 valeurs distinctes.

La figure 5.1 présente le résultat de ces expérimentations. Il apparait clairement que *NSC* et *NSC(B+Q)* sont au moins 100 fois plus rapides que leurs concurrents



(a) Variations de n (b) Variations de d

FIGURE 5.1 – Temps de calcul du k-dominant skycube

(figure 5.1(a)). Il est plus rapide de construire la structure de données NSC et calculer le k-dominant skycube à partir de celle-ci que de procéder suivant l'un des algorithmes de l'état de l'art. Si nous ne considérons que le temps des requêtes pour *NSC* (sans le temps de construction), le rapport de vitesses passe de 10^2 (lorsque $n = 10^3$) à 10^5 lorsque $n = 10^6$. En dépit du fait que *TSADPI* est légèrement plus rapide que *BUA* et *UBA*, toutes ces méthodes sont presque équivalentes. Au regard de la variation du nombre de dimensions (figure 5.1(b)), il apparait que lorsque d augmente, le temps pour construire NSC devient négligeable comparé à celui des requêtes, alors il est encore plus intéressant d'utiliser le skycube négatif NSC pour les requêtes k-dominant skyline.

5.2 Requêtes du skycube et données dynamiques

Les propositions pour l'optimisation des requêtes skyline qui ont été présentées jusque là considèrent les données à un instant précis. Ceci constitue un handicap dans la mesure où les données en réalité sont régulièrement sujettes à des mises à jour (ajouts/suppressions de tuples et moins fréquemment des ajouts et suppression de dimensions). D'autre part, la maintenance d'une structure de données relative au calcul des requêtes skyline présente un grand challenge dans ce sens que, la simple insertion d'un tuple pourrait faire passer la taille du résultat d'une requête de n à 1 (dans le cas où le tuple nouvellement inséré dominerait tous les autres). De même, la suppression d'un tuple pourrait faire passer la taille du résultat d'une requête skyline de 1 à n (dans le cas où le tuple supprimé dominait tous les autres qui sont incomparables entre eux).

Dans cette partie, après un bref parcours de la littérature relative à l'exécution de requêtes skyline dans un contexte de données dynamiques (insertion et suppression de tuples), nous allons présenter une approche qui, moyennant une extension de notre Skycube Négatif, permet de prendre en considération des données dites dynamiques. Nous nous restreignons également, dans ce travail, au dynamisme des tuples et non à celui des attributs.

5.2.1 Travaux Relatifs

Parmis les techniques dédiées à l'optimisation des requêtes du skycube, seul le travail Xia *et al.* [2012] propose une technique de mise à jour de sa structure de données Compressed Skycube (CSC) afin de prendre en considération le dynamisme des tuples (ajout, modification, suppression). Nous avons déjà comparé nos propositions à CSC dans l'hypothèse de données statiques et avons montré l'inefficacité de cette approche non seulement dans la rapidité de construction de la structure, la consommation de l'espace mais aussi le temps d'exécution des requêtes. En effet, au mieux de nos lectures, toutes les propositions (mis à part CSC) considérant les données dynamiques se focalisent sur le calcul du skyline de l'espace complet. Les premiers travaux dans ce sens, Tao et Papadias [2006] et Morse *et al.* [2007a] font l'hypothèse selon laquelle les tuples sont supprimés dans l'ordre d'arrivée (FIFO : First In First Out) ; ce qui réduit la problématique au calcul du skyline sur une fenêtre glissante des données ; leurs approches sont comparées expérimentalement dans le travail Alexander *et al.* [2016]. Wu *et al.* [2007] pour sa part, se focalise sur la maintenance de l'ensemble skyline après suppression d'un de ses éléménts ; il s'agit en réalité de la principale difficulté associée à la mise à jour du skyline de données dynamiques car, se passant de l'hypothèse FIFO formulée par les travaux précédents, il se peut que la suppression d'un tuple skyline entraîne l'entrée dans l'ensemble skyline de tuples qui en avaient précédemment été rétirés, or ces tuples auraient déjà été supprimés sous l'hypothèse FIFO. Afin de contourner cette difficulté, Hsueh *et al.* [2008, 2011] maintiennent, en plus du skyline courant, un skyline alternatif qui résulte du calcul du skyline de l'ensemble des tuples ne faisant pas partie du skyline. Il s'agit de la deuxième couche skyline ; et c'est dans cet ensemble qu'apparaissent les tuples susceptibles de passer dans l'ensemble skyline après suppression d'un point skyline. D'autres travaux plus récents, Matteis *et al.* [2016]; Montahaie *et al.* [2015]; Li et Yoo [2016], s'attaquent également à ce problème ; le premier, propose une technique de parallèlisation de l'algorithme *Eager* proposé par Tao et Papadias [2006], le second s'appuie sur la distance de manhattan pour élaguer les tuples (réduire les comparaisons) et propose également une technique de calcul en parallèle tandis que le dernier s'inspire du partitionnement des données pour limiter l'impact d'une mise à jour (en terme de nombre de comparaisons).

5.2.2 Maintenance du Skycube Négatif

La structure de données NSC dans sa version présentée précédemment, permettrait de procéder à des insertions de tuples sans avoir à reconstruire l'ensemble de la structure de données. La première étape consisterait à comparer le tuple t à insérer à l'ensemble topmost question de définir son statut. Deux cas de figure se présentent alors :

— le tuple à insérer est dominé, alors l'ensemble topmost ne sera pas modifié. Il suffira de calculer l'ensemble des paires relatives au tuple t, les réduire (voir

algorithme 15) et les insérer dans la structure de données NSC en s'inspirant des lignes $5-6$ de l'algorithme 18.

— dans le second cas, le tuple à insérer n'est pas dominé par ceux du topmost skyline. Il est même possible que celui-ci domine une partie des tuples du topmost skyline. Si tel est le cas, alors les tuples dominés sont retirés de cet ensemble, de même que toutes les paires leurs étant associées dans la structure de données NSC. Ensuite, Une paire supplémentaire sera ajoutée à la liste des paires associée à chaque tuple de la table. Cette paire résulte de la comparaison du tuple correspondant avec le nouveau tuple du topmost (tuple à insérer). Enfin, les paires relatives au tuple t sont calculées (à partir du nouveau topmost), réduites et insérées dans la structure de données.

Le problème se pose lorsque survient plutôt une suppression d'un tuple t, précisément lorsque t fait partie du topmost skyline. Les paires associées à tous les tuples et résultant de la comparaison de ceux-ci avec t devraient être supprimées de la structure de données. On se heurte alors à trois obstacles, (i) il est impossible d'identifier ces paires (la trace du tuple dominant n'est pas conservée lors d'une comparaison), (ii) il est également possible que ces paires ne nécessitent pas d'être supprimées, par exemple lorsque avant réduction, il existait plusieurs paires identiques ; et dans une moindre mesure (iii) ces paires pourraient ne pas avoir été conservées parce que *couvertes* par d'autres.

La résolution de ce problème nécessite de revenir en amont et redéfinir plusieurs concepts ; ceci est l'objet de la sous-section suivante.

Redéfinition des concepts

Nous adaptons dans cette section, des concepts, ayant précédemment été définis, pour les rendre compatibles avec la maintenance de notre structure de données dans les cas d'insertion/suppression de tuples.

Définition 5.3 ($\mathcal{COMPARE}3$)**.** Soit $t, t' \in T$. Nous définissons la fonction de comparaison $\mathcal{COMPARE}3$ comme suit : $\mathcal{COMPARE}3(t, t') = \langle X|Y|t'\rangle$ tel que X est l'ensemble des dimensions D_j telles que $t'[D_j] < t[D_j]$ et Y est l'ensemble des dimensions D_k pour lesquelles t et t' sont égaux.

Définition 5.4 (Couverture)**.** Soit $\langle X|Y|t\rangle$ un triplet dont $\langle X|Y\rangle$ est une paire d'espaces disjoints et soit Z un espace. On dit que le triplet $\langle X|Y|t\rangle$ *couvre* Z si et seulement si $Z \subseteq XY$ et $Z \cap X \neq \emptyset$.

Définition 5.5 ($\mathcal{COVER}3$)**.** Soit E un ensemble de triplets $\langle X|Y|t\rangle$ tels que $X, Y \subset \mathcal{D}$ et $t \in T$. On note $\mathcal{COVER}3(E)$ l'ensemble de tous les espaces couverts par au moins un triplet appartenant à E.

Après avoir redéfini les concepts de comparaison (fonction $\mathcal{COMPARE}3$) entre tuples, de couverture d'un espace par un triplet ou un ensemble de triplets, on se doute bien que le squelette des algorithmes contribuant à la construction de la structure NSC ne change pas radicalement.

Construction de la structure

Nous commençons par reécrire l'algorithme dédié à la réduction du nombre de paires (à présent des triplets). Nous ne considérons ici que celui fournissant un résultat approximatif (solution au problème de réduction sous contrainte d'inclusion). Donc, après avoir défini des triplets suite à la comparaison de chaque tuple avec le topmost Skyline, il s'agit de déterminer la plus petite partie couvrant les mêmes espaces que l'ensemble complet des triplets. La procédure donnant un résultat approximatif est alors décrite comme présenté par l'algorithme 21.

Algorithme 21 : $ApproxMST$

Input : Set of triplets : I
Output : Set of triplets : O

```
1 begin
2     O ← ∅
3     U ← COVER3(I)
4     while U ≠ ∅ do
5         tr ← arg max_{triplet∈I} |COVER3(triplet) \ COVER3(O)|
6         O ← O ∪ {tr}
7         U ← U \ COVER3(tr)
8     return O
```

Cet algorithme repose également sur la solution approximative (proposée par Chvatal [1979]) au problème de l'ensemble couvrant minimal. Mis à part le fait de conserver en plus des paires $\langle X|Y\rangle$, une trace du tuple ayant généré la paire en question, le résultat est identique à celui de l'algorithme 14.

L'algorithme 21 est intégré dans la procédure (algorithme 22) construisant la liste réduite des triplets pour un tuple donné et l'insérant dans la structure de données NSC. Nous choisissons de définir cette procédure indépendamment de la procédure construisant la structure de données complète car elle servira individuellement par la suite.

Nous pouvons ensuite écrire l'algorithme construisant la structure de données (voir algorithme 23). Ce dernier ne se différenciant de la version précédente (algorithme 18) que par le fait de conserver l'information (une trace) relative au tuple ayant produit la paire $\langle X|Y\rangle$.

La procédure 24 dont l'objectif est de calculer une requête skyline donnée reste pratiquement inchangée.

Exemple 41. Partant de l'exemple (table 4.2) du chapitre précédent, avant réduction, au tuple t_4 sera associée la liste de triplets suivante : $\{\langle AB|\emptyset|t_2\rangle, \langle A|B|t_3\rangle\}$, résultant de sa comparaison avec chacun des autres tuples du topmost skyline (t_2, t_3). Après réduction de la liste, on obtient le résultat suivant $\{\langle AB|\emptyset|t_2\rangle\}$.

Algorithme 22 : ListTriplets

Input : $Sky_T(\mathcal{D})$: $topmost$
Negative Skycube Structure NSC
a tuple t
Output : Negative Skycube Structure NSC

```
1 begin
2     list = listTripletsTuple(t, topmost)
3     foreach triplet ⟨X|Y|t'⟩ in list do
4         NSC[XY] = NSC[XY] ∪ {(t[id], Y, t'[id])} ;
5     return NSC
```

Algorithme 23 : BUILD_NSC

Input : Table T (set of tuples)
Output : Negative Skycube Structure NSC
$Sky_T(\mathcal{D})$: $topmost$

```
1 begin
2     topmost = ComputeSkyline(T, D)
3     Initialize NSC
4     foreach tuple t in T do
5         ListTriplets(topmost, NSC, t)
6     return NSC
```

Algorithme 24 : NSC_SKYLINE from subspaces

Input : Negative Skycube structure NSC, subspace Z, table T
Output : $Sky_T(Z)$

```
1 begin
2     NotSkylinePoints = ∅
3     foreach subspace W such that W ⊇ Z do
4         foreach triplet (t, Y, t') in NSC[W] do
5             X = W \ Y
6             if Z is covered by ⟨X|Y|t'⟩ then
7                 Add t to NotSkylinePoints
8     return T \ NotSkylinePoints
```

La maintenance de la structure de données repose sur la définition des procédures concourant à la prise à charge des insertion et suppression de tuples qui sont décrites dans les deux sections suivantes. Ces procédures sont dédiées au maintient de la cohérence de l'ensemble de la structure de données. Ladite cohérence peut se définir à travers quelques principes : (i) l'ensemble topmost skyline doit être à jour (ii) les informations de NSC doivent correspondre aux résultats des comparaisons des tuples de la table avec les tuples du topmost skyline et seulement ceux-ci.

Insertion d'un tuple

La manière dont est prise en charge l'insertion d'un nouveau tuple dans la structure de données dépend du statut qu'aura ce tuple : il peut, soit être dominé, soit faire partie du topmost skyline au quel cas il pourrait en plus, modifier le topmost skyline en dominant certains de ses tuples. Pour ce faire, la première procédure consiste à déterminer ce statut ; dans le cas où le tuple nouvellement arrivé (t) ferait partie du topmost skyline, il s'agit également de déterminer l'ensemble des tuples du topmost skyline qui en sortiront éventuellement car dominés par t, cette procédure est décrite à travers l'algorithme 25. Il suit le principe standard d'un algorithme naïf de calcul du skyline. À un instant donné, le résultat intermédiaire (skyline de la partie lue des données) est conservé (le topmost skyline), chaque fois qu'un nouveau tuple t est lu, on détermine de quelle manière celui-ci influencera le skyline courant : il ne modifie pas le skyline courant lorsqu'il est dominé, ou il fait partie du skyline courant auquel cas il pourrait éventuellement en supprimer des tuples.

Algorithme 25 : getStatusOfNewTuple

Input : $Sky_T(\mathcal{D})$: $topmost$, Tuple to be added : t
Output : Tuples to be removed from topmost : $toBeRemoved$,
Status of t : $tStatus$

```
1  begin
2      tStatus ← "add"
3      toBeRemoved ← ∅
4      foreach tuple u in topmost do
5          if u ≺_D t then
6              tStatus ← "discard"
7              return (toBeRemoved, tStatus)
8          else if t ≺_D u then
9              toBeRemoved ← toBeRemoved ∪ {u}
10     return (toBeRemoved, tStatus)
```

Cet algorithme retourne à son terme deux éléments, le premier $tStatus$ valant "add" si le tuple t fera partie du nouveau topmost skyline (il n'est pas dominé

dans la table) et "$discard$" s'il existe un tuple du topmost skyline le dominant. La seconde variable $toBeRemoved$ contient la liste des tuples du topmost skyline qui devront en être supprimés car dominés par t.

Une fois cette procédure établie, nous pouvons réaliser celle de l'insertion proprement dite du tuple t dans la structure de données. Comme décrit à travers l'algorithme 26. Cet algorithme repose sur la propriété suivante. Elle laisse sous-entendre que les listes des triplets associées aux tuples ne nécessitent pas d'être reconstruites après l'insertion d'un nouveau tuple.

Lemme 5.1. *Soient u et v deux tuples de la table T tels que $v \in Sky_T(\mathcal{D})$ et soit t un tuple à insérer dans T tel que $t \prec_{\mathcal{D}} v$. Soit X un espace dans lequel v domine u alors t domine également u en X.*

Démonstration. Il s'agit de la conséquence du théorème 4.4 présenté dans le chapitre précédent. En d'autres termes, u ne nécessiterait pas d'être comparé à v car v ne fait pas (plus) partie du topmost skyline. □

Ceci revient à dire qu'en supprimant de NSC les triplets dans lesquels apparaissent les tuples sortant du topmost skyline, on ne perd pas la cohérence de l'information car un nouveau triplet y sera inséré (résultant de la comparaison avec le nouveau tuple t) et couvrant au moins les mêmes espaces que ceux des triplets supprimés.

Donc, lorsque t n'est pas dominé, on commence tout d'abord par supprimer de NSC les triplets associés aux tuples de l'ancien topmost qui sont dominés par t car ceux-ci deviennent redondants comme cela a été montré à travers le théorème 4.4 du chapitre précédent. Afin de conserver la cohérence, et puisque t fait partie du skyline de l'espace complet, alors les triplets résultant de sa comparaison avec tous les autres tuples doivent être respectivement ajoutées aux listes associées à ceux-ci. Dans tous les cas (que t soit dominé ou pas), le calcul de la liste des triplets associés à t est effectué, cette liste est réduite et inserée dans la structure de données.

Suppression d'un tuple

En ce qui concerne la suppression d'un tuple t, il s'agit de l'opération inverse à celle de l'insertion. Deux cas de figure se présentent également dépendant du fait que t fait partie ou pas du topmost skyline actuel. Dans le cas affirmatif, la première étape consiste à déterminer les tuples ne faisant pas partie du skyline mais qui en feront partie après la suppression de t. Si t ne fait pas partie du skyline il suffit de supprimer la liste des triplets qui lui est associée dans la structure de données.

L'algorithme 27 définit la procédure de recherche des tuples qui entreront dans le topmost skyline une fois que t en sera retiré. Cette procédure repose sur l'assertion selon laquelle, les tuples présentant t dans l'un des triplets, $\langle X|Y|t\rangle$ qui leur est associé et tel que $X \cup Y = \mathcal{D}$, sont les seuls susceptibles de passer dans la topmost skyline. La preuve est établie ci-dessous.

Algorithme 26 : Insert new tuple

Input : Table : T, Negative Skycube NSC,
$Sky_T(\mathcal{D})$: $topmost$, Tuple to be added : t
Output : Table : T, Negative Skycube NSC, $Sky_T(\mathcal{D})$: $topmost$

begin
  ($toBeRemoved$, $tStatus$) $\leftarrow getStatusOfNewTuple(topmost, t)$
  foreach *tuple* u *in* $toBeRemoved$ **do**
    remove from NSC triplets like $\langle *| * |u\rangle$
  $topmost \leftarrow topmost \setminus toBeRemoved$
  $list \leftarrow listTripletsTuple(t, topmost)$
  foreach *triplet trip in* $list$ **do**
    Insert $trip$ into NSC w.r.t. t
  if $tStatus =$ "add" **then**
    foreach *tuple* u *in* T **do**
      Insert $\mathcal{COMPARE}3(u, t)$ into NSC w.r.t. u
    Insert t into $topmost$
  Insert t into T
  return *(*T*,* NSC*,* $topmost$*)*

Algorithme 27 : CheckDominanceDelete

Input : Negative Skycube NSC, $Sky_T(\mathcal{D})$: $topmost$,
Tuple to be deleted t
Output : Tuples to be inserted into $topmost$: $toBeInserted$

begin
  $toBeInserted \leftarrow \emptyset$
  foreach *tuple* u *s.t.* $\exists(X, Y, t) \in NSC$ *w.r.t.* u *and* $X \cup Y = \mathcal{D}$ **do**
    if u *is not dominated by any tuple from* $topmost \setminus t$ **then**
      $toBeInserted \leftarrow toBeInserted \cup \{u\}$
    $toBeInserted \leftarrow Sky(toBeInserted, \mathcal{D})$
  return $toBeInserted$

Lemme 5.2. *Soient t et u deux tuples de T tels que $t \in Sky(T, \mathcal{D})$ et $u \notin Sky(T, \mathcal{D})$. Soit $List_u$ la liste des triplets (réduite ou pas) associée à u. S'il n'existe pas de triplet $\langle X|Y|t\rangle \in List_u$ tel que $X \cup Y = \mathcal{D}$ alors, $u \notin Sky(T \setminus \{t\}, \mathcal{D})$.*

Par l'absurde. Supposons qu'il n'existe pas de triplet $\langle X|Y|t\rangle \in List_u$ tel que $X \cup Y = \mathcal{D}$ et que néanmoins $u \in Sky(T \setminus \{t\}, \mathcal{D})$. Etant donné que u n'appartient pas à $Sky(T, \mathcal{D})$, il existe forcément un triplet $\langle X_v|Y_v|v\rangle \in List_u$ tel que $X_v \cup Y_v = \mathcal{D}$ avec $v \neq t$. En d'autres termes $v \prec_{\mathcal{D}} u$, la suppression de t ne changera pas la véracité de cette assertion (u restera dominé sur $\mathcal{D}$). Donc même après la suppression de t, u ne pourra faire partie du topmost skyline (contradiction). □

Cette propriété permet de considérablement réduire la complexité de la mise à jour de la structure de données. En effet il ne sera pas nécessaire de recalculer l'ensemble topmost après chaque suppression de tuple.

Par contre, il sera nécessaire de recalculer entièrement les listes de triplets associés aux tuples u pour lesquels t apparaissait dans l'un des triplets. Ceci est dû au fait que la suppression de ces triplets contenant t peut être source d'incohérence. Nous expliquons ce phénomène à travers un exemple précis.

Exemple 42. Soit t, t_1, u et v des tuples de T. Supposons qu'avant réduction des listes, à u et v soient respectivement associées les listes $List_u$ et $List_v$ suivantes : $List_u = \{\langle AB|\emptyset|t_1\rangle, \langle AB|C|t\rangle\}$ et $List_v = \{\langle D|E|t_1\rangle, \langle AB|C|t\rangle\}$. Après réduction nous obtenons $List_u = \{\langle AB|C|t\rangle\}$ et $List_v = \{\langle D|E|t_1\rangle, \langle AB|C|t\rangle\}$. Si après suppression de t nous supprimons simplement les triplets lui correspondant dans les listes, alors :

- pour u, il se crée une incohérence car u n'était pas dominé que par t, mais également par t_1 et cette information est perdue (aucune trace du fait que u soit dominé sur AB par exemple car $t_1 \prec_{AB} u$).
- par contre la suppression du triplet correspondant de v ne crée par d'incohérence car il ne s'agissait pas d'un doublon dans la liste non réduite.

En raison de cette obligation de conserver la cohérence sur l'état de chaque tuple, nous recalculons, comme présenté dans l'algorithme 28, les listes de triplets associées aux tuples affectés par la suppression d'un autre (lignes 7 à 9).

Nous constatons que l'algorithme n'applique aucun traitement aux tuples pour lesquels l'élément supprimé t n'apparaît pas dans les listes de triplets associées. Nous n'insérons pas de nouveaux triplets résultant de leur comparasion avec les tuples entrant dans le topmost skyline. Le lemme suivant appuie cette démarche.

Lemme 5.3. *Soient t et u deux tuples de T et soit $List_u$ la liste (réduite ou pas) de triplets associée au tuple u. Si t n'apparaît pas dans $List_u$ alors la suppression de t de la table T garde cohérente $List_u$.*

Démonstration. Nous distinguons deux cas de figure. Dans un premier temps considérons qu'il existe un espace X dans lequel u était dominé par t. Le fait que t n'apparaisse pas dans $list_u$ signifie qu'il existe un autre tuple v dominant également u

Algorithme 28 : DeleteTupleFromNSC

Input : Table T, Negative Skycube NSC, $Sky_T(\mathcal{D}) : topmost$, Tuple to be deleted t
Output : Table T, Negative Skycube NSC, $Sky_T(\mathcal{D}) : topmost$

```
1  begin
2      toBeAdded ← toBeInserted(NSC, topmost, t)
3      toBeComputed ← {u ∈ T | ∃⟨u| ∗ |t⟩ ∈ NSC}
4      Remove any triplet like ⟨∗| ∗ |t⟩ from NSC
5      Remove t from topmost and from T
6      topmost ← topmost ∪ toBeAdded
7      foreach tuple u in toBeComputed do
8          Clear NSC[u]
9          ListTriplets(topmost, NSC, u)
10     return (T,NSC,topmost)
```

sur X et apparaissant dans $List_u$. Donc après suppression de t, u reste dominé en X ce qui n'entraîne aucune incohérence dans $List_u$. De plus les tuples susceptiples d'entrer dans le topmost skyline sont dominé par t, donc ne peuvent dominer u qu'en un espace dans lequel t dominait déjà u (mentionné par le triplet faisant apparaître v). Supposons à présent qu'il n'existe aucun espace X dans lequel t domine u alors il est normal que $List_u$ reste inchangée après la suppression de t (l'ensemble $\mathcal{DOM}(u)$ demeure inchangé). En effet, puisque les tuples entrant dans le topmost skyline sont dominés par t, ils ne peuvent pas dominer u dans un espace dans lequel t ne dominait déjà u. □

Taille croissante des listes de triplets

En observant de plus près les différentes méthodes proposées pour la maintenance de la structure en cas d'insertion ou de suppression d'un tuple, nous constatons qu'il est possible que pour chacune d'elles, il existe des listes de triplets dont la taille augmentera. En ce qui concerne la suppression il s'agit essentiellement des tuples qui présentaient le tuple supprimé dans l'un de leurs triplets. On a la garantie que la taille de liste obtenue après exécution de l'algorithme sera approximativement optimale car ces tuples correspondent à ceux pour lesquels la liste est entièrement recalculée. Par contre, dans le cas de l'insertion d'un tuple, si ce dernier fait partie du nouveau topmost skyline, alors un triplet sera inséré à toutes les listes de triplets de la table. Ce nouveau triplet, pour chacune des listes, pourrait (ou pas) apporter de la redondance d'information. Etant donné que les listes n'augmentent que d'un seul élément, nous ne jugeons pas nécessaire d'exécuter l'algorithme de réduction de la taille de la liste de triplets à chaque insertion. Néanmoins un dispositif, ou une condition (par exemple lorsque la taille de la structure double, ou après

$\sqrt{|topmost|}$ nouvelles insertions dans le topmost skyline) pourrait être établi pour *déclencher* la compression des listes. Cette procédure consiste juste à exécuter l'algorithme 21 pour chaque liste de triplets et de mettre à jour la structure de données NSC à partir du résultat. Les inconvénients de l'absence d'un tel procédé sont dans un premier temps une consommation inutile de l'espace de stockage (mémoire) et ensuite la perte sensible de performance lors de l'exécution des requêtes car, plus petite est notre structure de données et plus rapidement sont réalisées les requêtes.

Conclusion

Dans ce chapitre, nous avons abordé deux autres problématiques en apportant une extension à notre structure de données Skycube Négatif.

D'une part, nous avons conçu une méthode d'exécution des requêtes k-dominant skyline, quels que soient la valeur de k et l'espace de référence. Solution qui s'avère très efficace comparée aux algorithmes de l'état de l'art qui n'avaient pas envisagé un pré-calcul partiel. Notre méthode, reposant sur la structure de données NSC, est également plus efficace que ceux de la littérature pour la construction de l'ensemble du k-dominant skycube lorsque tel que l'objectif (Dong *et al.* [2010]).

Nous abordons enfin la problématique du calcul des requêtes skyline en présence de données susceptibles de subir des mises à jour (insertion et suppression de tuples). Cette problématique reste très peu traitée surtout dans l'optique de l'optimisation de l'ensemble des requêtes du skycube (Xia *et al.* [2012]). Nous présentons alors une extension de NSC dans ce sens. En effet, nous avons dû redéfinir quelques concepts afin de conserver une trace de la relation de dominance entre les tuples. Cette trace, nous permettant de réduire la tâche de la mise à jour de l'ensemble de la structure.

Chapitre 6

Conclusion et Perspectives

Cette thèse s'est consacrée à la contribution dans la résolution de quelques problèmes relatifs aux requêtes de préférence, plus précisement les requêtes skyline dans un contexte multidimensionnel. L'opérateur skyline présentant un grand intérêt dans le cadre de la sélection multi-critère des meilleurs éléments d'une base de données. Son principal atout étant de ne pas nécessiter la définition d'une fonction de score (pas toujours adéquate) pour ordonner les enregistrements. Malheureusement, cet opérateur présente également des inconvénients, parmi lesquels on peut citer tout d'abord la complexité quadratique (en taille de la base de données) du calcul, surtout dans un contexte où les utilisateurs ne choisissent pas les mêmes ensembles de critères pour les requêtes. L'autre inconvénient est le fait de n'avoir que le nombre de tuples dans les données comme limite dans la taille du résultat ; ce dernier point rend difficile la connaissance à priori de la taille du résultat d'une requête.

Dans le chapitre 2, nous abordons la problématique de l'estimation de la taille du résultat d'une requête skyline. Nous proposons deux principales approches reposant sur la connaissance de la distribution des valeurs pour chacune des dimensions. La première, se basant sur l'expression analytique de la probabilité pour un tuple donné d'être dominé, parcours un échantillon (éventuellement l'ensemble) de la table afin de déterminer la moyenne de cette probabilité et déterminer une estimation de la taille du skyline. La seconde approche, sans avoir à parcourir les données, estime l'espérance de la probabilité d'être dominé, pour un tuple dont les valeurs suivent les distributions connues, et déduit une estimation de la taille du skyline. Nous avons comparé de manière expérimentale ces approches aux algorithmes de l'état de l'art. Il en ressort que plus les données satisfont notre hypothèse d'indépendance des dimensions, meilleures sont nos estimations. Pour les données présentant des corrélations, nos résultats sont intéressants dans le cas de skylines sur peu de dimensions.

Dans les chapitres 3 et 4, nous étudions l'optimisation des requêtes du skycube (ensemble des requêtes skyline d'un jeu de données). La différence majeure entre les deux méthodes proposées tient de l'approche utilisée pour aborder le problème.

En effet, il existe deux grandes catégories de solution à ce problème. La première (plus fréquente) raisonne à l'echelle des espaces (ensembles de critères) et recherche des moyens de mutualiser les calculs des skylines entre eux. Tandis que la seconde, se ramène à l'echelle des tuples pris individuellement et essaye de résumer l'information d'appartenance aux skylines.

Le chapitre 3 est consacré à la matérialisation partielle du skycube pour optimiser l'ensemble des requêtes. Il s'agit, comme dans le cas des cubes de données, de sélectionner un ensemble minimum de cuboids (ici les skycuboids) permettant de répondre à toutes les autres requêtes. En raison du fait que la fonction skyline ne soit pas monotone, cette tâche est plus ardue que dans le cas des cubes de données. À partir d'une propriété établie reposant sur les dépendances fonctionnelles, nous avons pu définir une forme de monotonie (conditionnelle) à l'opérateur. Cette propriété, combinée à la monotonie du skyline sur les motifs des tuples, nous a permis de proposer une solution complète. Les expérimentations effectuées (en comparaison aux algorithmes existants) montrent la pertinence de cette approche, non seulement pour le pré-calcul mais également en ce qui concerne l'exécution des requêtes. En effet, il s'agit d'un compromis intéressant entre le fait de calculer un skyline à la demande et le pré-calcul complet du skycube.

Le chapitre 4 quant à lui, aborde la même problématique avec une approche différente. En effet, la méthode proposée ici se rapproche des techniques de compression du skycube. Les algorithmes de l'état de l'art à ce sujet se proposent de résumer l'ensemble des espaces pour lesquels un tuple donné fait partie du skyline ; dans ce travail, nous raisonnons sur l'information complémentaire : résumer l'ensemble des espaces pour lesquels le tuple donné est dominé. Nous estimons, et cela a été conforté par les expérimentations, que cette formulation est plus bénéfique en termes de temps de calcul mais aussi d'espace de stockage. Notre problématique s'est traduite en une question bien connue sous l'expression d'ensemble couvrant minimal, pour laquelle nous avons pu mettre à disposition deux solutions distinctes. Nous avons également comparé de manière expérimentale nos deux solutions en mettant en exergue les points forts de chacune d'elles mais aussi les limites de l'une par rapport à l'autre.

Dans le dernier chapitre (5), nous proposons des extensions à la structure de données Skycube Négatif. Dans un premier temps, nous montrons comment cette structure, sans aucune adaptation peut être exploitée pour répondre efficacement aux requêtes k-dominant skyline, une forme plus générale de requête skyline dédiée à réduire la taille du résultat. Ensuite, nous proposons une restructuration du Skycube Négatif afin qu'il puisse supporter à moindre coût le dynamisme dans les données (insertion et suppression de tuples). Enfin, nous présentons quelques idées pertinentes de pistes susceptibles d'être étudiées afin d'étendre les travaux dans le domaine.

Considérant ce travail comme une base, plusieurs autres orientations peuvent être poursuivies.

Estimation de la taille du skyline et dimensions corrélées Dans le cadre de l'estimation de la taille du skyline, afin de surmonter les limites de nos approches, par exemple lorsque les données s'éloignent de notre hypothèse d'indépendance entre les dimensions (données corrélées), nous envisageons de prendre en considération les distributions jointes des dimensions ; par exemple la probabilité pour un tuple de présenter à la fois les valeurs a_i et a_j respectivement pour les dimensions D_i et D_j. On pourrait également définir une notion de corrélation multidimensionnelle plus simple (par rapport à la corrélation linéaire) à intégrer dans les calculs analytiques.

Calcul en Environnement Distribué On pourrait imaginer des situations dans lesquelles les données disponibles sont très volumineuses pour être stockées sur un seul nœud de calcul, elles seraient alors répartie dans un cluster. Il serait question de créer une structure distribuée ou centralisée permettant de répondre aux requêtes skyline.

À partir de notre travail sur la matérialisation partielle (structure de données MICS), il n'est pas encore clair qu'il faille considérer les dépendances fonctionnelles satisfaites localement ou uniquement celles qui sont vérifiées globalement. Tout comme l'opérateur de jointure et la théorie de la décomposition sans perte, il est tentant de refléchir à la théorie de la décomposition sous contrainte de recomposition du skyline à l'aide des dépendances fonctionnelles ou peut être s'appuyant sur d'autres types de dépendance comme les *Dépendances d'Ordre*.

En ce qui concerne le skycube négatif, sa construction peut être décomposée en quatre principales étapes :

(i) le calcul du topmost skyline
(ii) le calcul de la liste des paires (ou triplets) pour chaque tuple
(iii) la réduction de la liste pour chaque tuple
(iv) la mise en commun de l'ensemble pour les requêtes

Les deux étapes coûteuses en termes d'échange entre les nœuds de calcul seraient (i) et (ii) car à ce niveau chaque tuple a besoin d'être comparés avec pratiquement tous les autres. Leur mise en œuvre pourrait s'inspirer des algorithmes de calcul de skyline en environnement distribué qui existent (par exemple Mullesgaard *et al.* [2014]). L'étape (iii) s'exécuterait parfaitement en environnement distribué comme en parallèle car les listes associées aux tuples sont indépendantes ; donc chaque liste pourrait être réduite par un nœud de calcul et même plusieurs listes pourraient être réduites en parallèle par un nœud afin de bénéficier éventuellement de l'architecture multi-cœur du nœud. Dépendant de la volonté ou de la capacité de construire une structure NSC centralisée ou distribuée l'étape (iv) sera exécutée sur un seul nœud (nécessitant alors le transfert de toutes les listes réduites vers celui-ci) ou sur plusieurs nœuds réparties selon notre volonté. Sachant que le choix fait à ce niveau influencera la façon dont seront exécutées les requêtes (centralisées ou distribuées).

Préférences non prédéfinies ou données géolocalisées Il existe des critères susceptibles d'intéresser les utilisateurs mais qui ne respectent pas nos hypothèses travail : lorsqu'une relation d'ordre n'est pas établie au sein des valeurs ou cette relation d'ordre n'est pas *unanime*. Il s'agit notamment des attributs pour lesquels l'ordre de préférence des valeurs peut varier d'un utilisateur à un autre. Plusieurs cas courant peuvent être sujets à cette problématique :

— lors du choix d'un billet d'avion dans un comparateur de vols, l'ordre de préférence des compagnies n'est pas unanime
— l'ordre de préférence des couleurs si on rajoute cette variable (couleur par exemple) parmi les critères de choix d'un produit

Il est alors question de pouvoir intégrer une telle variable dans une structure de données pré-calculée pour répondre aux requêtes du skycube.

Cette problématique peut également être assimilée à celle de l'intégration du critère de proximité. Par exemple : chercher les meilleurs restaurants en intégrant la proximité du lieu où on se trouve. En fonction du lieu (donnée spatiale) où on se trouve le résultat de la requête est différent. Il s'agit donc, d'une variable dont la préférence n'est pas prédéfinie car, dépendant du lieu où on est, on a un ordre de préférence différent pour cette variable, ordre de préférence qui influence à son tour le résultat de la requête.

La structure de données MICS reposant sur les dépendances fonctionnelles pourrait parfaitement convenir dans la résolution de cette nouvelle problématique. En effet, les dépendances fonctionnelles sont invariantes aux transformations isomorphiques des données, les inclusions décelées par les dépendances fonctionnelles sont invariantes à l'ordre de choix des valeurs au sein des dimensions. Donc, moyennant un moindre effort, cette structure semble naturellement se proposer dans ce sens.

Skyline au sein de graphes On pourrait imaginer des situations où les données ne se présenteraient pas sous une forme nous permettant d'exécuter de manière classique les requêtes skyline. Prenons l'exemple d'une table, décrivant les portions de trajets (les rues par exemple) d'une zone géographique donnée. Pour chacune de ces portions nous avons un ensemble d'informations renseignées, par exemple, le coût du trajet (en consommation de carburant, péage d'autoroute, ...), une note sur le trajet en terme d'intérêt pour le tourisme (nombre de places à visiter sur ce trajet), des notes sur les commodités (nombre d'hôtels, d'aires d'autoroute, etc). La problématique serait, non pas de sélectionner les meilleures portions de trajets mais les meilleurs chemins (séquences de trajets adjacents) allant d'un point A à un point B et optimisant les critères ci-dessus mentionnés. Cette problématique présente une extension intéressante des requêtes skyline appliquée aux données de graphes.

Bibliographie

AFRATI, Foto N., KOUTRIS, Paraschos, SUCIU, Dan et ULLMAN, Jeffrey D., 2015. Parallel skyline queries. *Theory Comput. Syst.*, 57(4) :1008–1037.

AGARWAL, Sameet, AGRAWAL, Rakesh, DESHPANDE, Prasad, GUPTA, Ashish, NAUGHTON, Jeffrey F., RAMAKRISHNAN, Raghu et SARAWAGI, Sunita, 1996. On the computation of multidimensional aggregates. Dans *proc. of VLDB conf.*

AGRAWAL, Sanjay, CHAUDHURI, Surajit et NARASAYYA, Vivek R., 2000. Automated selection of materialized views and indexes in SQL databases. Dans *proc. of VLDB conf.*

ALEXANDER, Tzanakas, ELEFTHERIOS, Tiakas et YANNIS, Manolopoulos, 2016. *Skyline Algorithms on Streams of Multidimensional Data*, pages 63–71. Springer International Publishing, Cham. ISBN 978-3-319-44066-8. doi :10.1007/978-3-319-44066-8_7.

BARTOLINI, Ilaria, CIACCIA, Paolo et PATELLA, Marco, 2008. Efficient sort-based skyline evaluation. *ACM Trans. Database Syst.*, 33(4).

BENTLEY, J. L., KUNG, H. T., SCHKOLNICK, M. et THOMPSON, C. D., 1978. On the average number of maxima in a set of vectors and applications. *J. ACM*, 25(4) :536–543. doi :10.1145/322092.322095.

BØGH, Kenneth S., CHESTER, Sean, SIDLAUSKAS, Darius et ASSENT, Ira, 2014. Hashcube : A data structure for space- and query-efficient skycube compression. Dans *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM*.

BÖRZSÖNYI, Stephan, KOSSMANN, Donald et STOCKER, Konrad, 2001. The skyline operator. Dans *Proceedings of the 17th International Conference on Data Engineering*, pages 421–430. IEEE Computer Society, Washington, DC, USA. ISBN 0-7695-1001-9.

BUCHTA, C., 1989. On the average number of maxima in a set of vectors. *Inf. Process. Lett.*, 33(2) :63–65. doi :10.1016/0020-0190(89)90156-7.

CHAN, Chee Yong, JAGADISH, H. V., TAN, Kian-Lee, TUNG, Anthony K. H. et ZHANG, Zhenjie, 2006a. Finding k-dominant skylines in high dimensional space. Dans *Proceedings of SIGMOD International Conference*.

CHAN, Chee Yong, JAGADISH, H. V., TAN, Kian-Lee, TUNG, Anthony K. H. et ZHANG, Zhenjie, 2006b. On high dimensional skylines. Dans *Proceedings of EDBT conference*.

CHAUDHURI, Surajit, 1998. An overview of query optimization in relational systems. Dans *Proceedings of PODS conference*, pages 34–43. ACM.

CHAUDHURI, Surajit, DALVI, Nilesh et KAUSHIK, Raghav, 2006. Robust cardinality and cost estimation for skyline operator. Dans *Proceedings of the 22Nd International Conference on Data Engineering*, ICDE '06, pages 64–. IEEE Computer Society, Washington, DC, USA. ISBN 0-7695-2570-9. doi : 10.1109/ICDE.2006.131.

CHESTER, Sean, SIDLAUSKAS, Darius, ASSENT, Ira et BØGH, Kenneth S., 2015. Scalable parallelization of skyline computation for multi-core processors. Dans *Proceedings of ICDE conference*.

CHOMICKI, Jan, GODFREY, Park, GRYZ, Jarek et LIANG, D., 2003. Skyline with presorting. Dans *Proc. of ICDE conference*.

CHVATAL, V., 1979. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3) :233–235. doi :10.2307/3689577.

DONG, Lei-gang, CUI, Xiao-wei, WANG, Zhen-fu et CHENG, Shu-wei, 2010. Finding k-dominant skyline cube based on sharing-strategy. Dans *Proceedings of FSKD conference*.

EFRON, B., 1979. Bootstrap methods : Another look at the jackknife. *Ann. Statist.*, 7(1) :1–26. doi :10.1214/aos/1176344552.

EITER, Thomas et GOTTLOB, Georg, 1995. Identifying the minimal transversals of a hypergraph and related problems. *SIAM J. Comput.*, 24(6) :1278–1304.

GODFREY, P., SEIPEL, D. et TURULL-TORRES, J.M., 2004. Skyline cardinality for relational processing : how many vectors are maximal ? Rapport technique, York Univ., Toronto, Ont., Canada.

GODFREY, Parke, SHIPLEY, Ryan et GRYZ, Jarek, 2005. Maximal vector computation in large data sets. Dans *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 229–240. VLDB Endowment. ISBN 1-59593-154-6.

GODFREY, Parke, SHIPLEY, Ryan et GRYZ, Jarek, 2007. Algorithms and analyses for maximal vector computation. *VLDB Journal*, 16(1).

GRAY, Jim, CHAUDHURI, Surajit, BOSWORTH, Adam, LAYMAN, Andrew, REICHART, Don, VENKATRAO, Murali, PELLOW, Frank et PIRAHESH, Hamid, 1997. Data cube : A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Mining and Knowledge Discovery*, 1(1) :29–53.

GROZ, Benoît et MILO, Tova, 2015. Skyline queries with noisy comparisons. Dans *Proceedings of PODS conference*.

HANUSSE, Nicolas, KAMNANG WANKO, Patrick et MAABOUT, Sofian, 2016a. Computing and summarizing the negative skycube. Dans *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management, CIKM 2016, Indianapolis, IN, USA, October 24-28, 2016*, pages 1733–1742. doi : 10.1145/2983323.2983759.

HANUSSE, Nicolas, KAMNANG WANKO, Patrick et MAABOUT, Sofian, 2016b. Using histograms for skyline size estimation. Dans *Proceedings of the 20th International Database Engineering & Applications Symposium, IDEAS 2016, Montreal, QC, Canada, July 11-13, 2016*, pages 125–134. doi :10.1145/2938503.2938531.

HANUSSE, Nicolas, KAMNANG WANKO, Patrick, MAABOUT, Sofian et ORDONEZ, Carlos, 2016c. Skycube materialization using the topmost skyline or functional dependencies. *ACM Trans. Database Syst.*, 41(4) :25 :1–25 :40. doi : 10.1145/2955092.

HANUSSE, Nicolas et MAABOUT, Sofian, 2011. A parallel algorithm for computing borders. Dans *Proc. of CIKM conf.*

HANUSSE, Nicolas, MAABOUT, Sofian et TOFAN, Radu, 2009. A view selection algorithm with performance guarantee. Dans *Proceedings of EDBT conference*, tome 360, pages 946–957. ACM.

HARINARAYAN, Venky, RAJARAMAN, Anand et ULLMAN, Jeffrey D., 1996. Implementing data cubes efficiently. Dans *SIGMOD conf.*

HORVITZ, Daniel G. et THOMPSON, Donovan J., 1952. A generalization of sampling without replacement from a finite universe. *Journal of the American Statistical Association*, 47(260) :663–685.

HSUEH, Yu-Ling, ZIMMERMANN, Roger et KU, Wei-Shinn, 2008. Efficient updates for continuous skyline computations. Dans *Database and Expert Systems Applications, 19th International Conference, DEXA 2008, Turin, Italy, September 1-5, 2008. Proceedings*, pages 419–433. doi :10.1007/978-3-540-85654-2_38.

HSUEH, Yu-Ling, ZIMMERMANN, Roger, KU, Wei-Shinn et JIN, Yifan, 2011. Skyengine : Efficient skyline search engine for continuous skyline computations. Dans *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 1316–1319. doi : 10.1109/ICDE.2011.5767944.

HUHTALA, Ykä, KÄRKKÄINEN, Juha, PORKKA, Pasi et TOIVONEN, Hannu, 1999. Tane : An efficient algorithm for discovering functional and approximate dependencies. *Computer Journal*, 42(2) :100–111.

IM, Hyeonseung, PARK, Jonghyun et PARK, Sungwoo, 2011. Parallel skyline computation on multicore architectures. *Information Systems*, 36(4) :808–823.

LEE, Jongwuk et WON HWANG, Seung, 2010. BSkyTree : scalable skyline computation using a balanced pivot selection. Dans *Proc. of EDBT conf.*

LEE, Jongwuk et WON HWANG, Seung, 2014. Toward efficient multidimensional subspace skyline computation. *VLDB Journal*, 23(1) :129–145.

LEMIRE, Daniel, KASER, Owen et GUTARRA, Eduardo, 2012. Reordering rows for better compression : Beyond the lexicographic order. *ACM Trans. Database Syst.*, 37(3).

LI, He et YOO, Jaesoo, 2016. An efficient continuous skyline query processing scheme over large dynamic data sets. doi :http://dx.doi.org/10.4218/etrij.16.0116.0010.

LI, Jingni, TALEBI, Zohreh A., CHIRKOVA, Rada et FATHI, Yahya, 2005. A formal model for the problem of view selection for aggregate queries. Dans *Proc. of ADBIS conf.*

LIN, Xuemin, YUAN, Yidong, ZHANG, Qing et ZHANG, Ying, 2007. Selecting stars : The k most representative skyline operator. Dans *Proceedings of ICDE conference*.

LIU, Jinfei, XIONG, Li, PEI, Jian, LUO, Jun et ZHANG, Haoyu, 2015. Finding pareto optimal groups : Group-based skyline. *PVLDB*, 8(13) :2086–2097.

LOPES, Stéphane, PETIT, Jean-Marc et LAKHAL, Lotfi, 2000. Efficient discovery of functional dependencies and armstrong relations. Dans *proc. of EDBT conf.*

LUO, Cheng, JIANG, Zhewei, HOU, Wen-Chi, HE, Shan et ZHU, Qiang, 2012. A sampling approach for skyline query cardinality estimation. *Knowledge and Information Systems*, 32(2) :281–301.

MAIER, David, 1983. *The Theory of Relational Databases*. Computer Science Press.

MANNILA, Heikki et RÄIHÄ, Kari-Jouko, 1992. *Design of Relational Databases*. Addison-Wesley.

MANNILA, Heikki et RÄIHÄ, Kari-Jouko, 1994. Algorithms for inferring functional dependencies from relations. *Data and Knowledge Engineering*, 12(1) :83–99.

MATTEIS, Tiziano De, GIROLAMO, Salvatore Di et MENCAGLI, Gabriele, 2016. Continuous skyline queries on multicore architectures. *Concurrency and Computation : Practice and Experience*, 28(12) :3503–3522. doi :10.1002/cpe.3866.

MD. ANISUZZAMAN, Siddique et YASUHIKO, Morimoto, 2010. k-dominant and extended k-dominant skyline computation by using statistics. *International Journal on Computer Science and Engineering*, (5) :1934.

MONTAHAIE, Ehsan, GHAFOURI, Milad, RAHMANI, Saied, GHASEMI, Hanie, BAKHTIAR, Farzad Sharif, ZAMANSHOAR, Rashid, JAFARI, Kianoush, GAVAHI, Mohsen, MIRZAEI, Reza, AHMADZADEH, Armin et GORGIN, Saeid, 2015. Efficient continuous skyline computation on multi-core processors based on manhattan distance. Dans *13. ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2015, Austin, TX, USA, September 21-23, 2015*, pages 56–59. doi :10.1109/MEMCOD.2015.7340469.

MORSE, Michael D., PATEL, Jignesh M. et GROSKY, William I., 2007a. Efficient continuous skyline computation. *Inf. Sci.*, 177(17) :3411–3437. doi :10.1016/j.ins.2007.02.033.

MORSE, Michael D., PATEL, Jignesh M. et JAGADISH, H. V., 2007b. Efficient skyline computation over low-cardinality domains. Dans *Proceedings of VLDB conf.*

MULLESGAARD, Kasper, PEDERSENY, Jens Laurits, LU, Hua et ZHOU, Yongluan, 2014. Efficient skyline computation in mapreduce. Dans *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014.*, pages 37–48. doi :10.5441/002/edbt.2014.05.

NOVELLI, Noël et CICCHETTI, Rosine, 2001. FUN : An efficient algorithm for mining functional and embedded dependencies. Dans *Proc. of ICDT conf.*

PAPADIAS, Dimitris, TAO, Yufei, FU, Greg et SEEGER, Bernhard, 2005. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, 30(1).

PEI, Jian, JIANG, Bin, LIN, Xuemin et YUAN, Yidong, 2007. Probabilistic skylines on uncertain data. Dans *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 15–26.

PEI, Jian, JIN, Wen, ESTER, Martin et TAO, Yufei, 2005. Catching the best views of skyline : A semantic approach based on decisive subspaces. Dans *Proc. of VLDB conf.*

PEI, Jian, YUAN, Yidong, LIN, Xuemin, JIN, Wen, ESTER, Martin, LIU, Qing, WANG, Wei, TAO, Yufei, YU, Jeffrey Xu et ZHANG, Qing, 2006. Towards multidimensional subspace skyline analysis. *ACM TODS*, 31(4) :1335–1381.

RAÏSSI, Chedy, PEI, Jian et KISTER, Thomas, 2010. Computing closed skycubes. *Proc. of VLDB conf.*

SARMA, Atish Das, LALL, Ashwin, NANONGKAI, Danupon et XU, Jun, 2009. Randomized multi-pass streaming skyline algorithms. Dans *Proceeding of VLDB*.

SHANG, Haichuan et KITSUREGAWA, Masaru, 2013. Skyline operator on anti-correlated distributions. *PVLDB*, 6(9).

SHENG, Cheng et TAO, Yufei, 2012. Worst-case i/o-efficient skyline algorithms. *ACM Trans. Database Syst.*, 37(4).

SIDDIQUE, M.A. et MORIMOTO, Y., 2012. Efficient k-dominant skyline computation for high dimensional space with domination power index. *Journal of Computers*, 7(3) :608 – 615.

SIDDIQUE, Md. Anisuzzaman et MORIMOTO, Yasuhiko, 2009. K-dominant skyline computation by using sort-filtering method. Dans *Proceedings of PAKDD conference*.

TAO, Yufei et PAPADIAS, Dimitris, 2006. Maintaining sliding window skylines on data streams. *IEEE Trans. Knowl. Data Eng.*, 18(2) :377–391. doi :10.1109/TKDE.2006.48.

WU, Ping, AGRAWAL, Divyakant, EGECIOGLU, Ömer et EL ABBADI, Amr, 2007. Deltasky : Optimal maintenance of skyline deletions without exclusive dominance region generation. Dans *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 486–495. doi :10.1109/ICDE.2007.367894.

XIA, Tian, ZHANG, Donghui, FANG, Zheng, CHEN, Cindy X. et WANG, Jie, 2012. Online subspace skyline query processing using the compressed skycube. *ACM TODS*, 37(2).

YAO, Hong et HAMILTON, Howard J., 2008. Mining functional dependencies from data. *Data Mining and Knowledge Discovery*, 16(2) :197–219.

YUAN, Yidong, LIN, Xuemin, LIU, Qing, WANG, Wei, YU, Jeffrey Xu et ZHANG, Qing, 2005. Efficient computation of the skyline cube. Dans *Proc. of VLDB conf.*

ZHANG, Zhenjie, YANG, Yin, CAI, Ruichu, PAPADIAS, Dimitris et TUNG, Anthony, 2009. Kernel-based skyline cardinality estimation. Dans *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 509–522. ACM, New York, NY, USA. ISBN 978-1-60558-551-2. doi :10.1145/1559845.1559899.

ZHAO, Yihong, DESHPANDE, Prasad et NAUGHTON, Jeffrey F., 1997. An array-based algorithm for simultaneous multidimensional aggregates. Dans *Proc. of SIGMOD conf.*