Contributions à l'Optimisation de Requêtes Multidimensionnelles

Sofian Maabout

Habilitation à Diriger des Recherches

UNIVERSITÉ DE BORDEAUX

École doctorale de Mathématiques et Informatique de Bordeaux

DOMAINE DE RECHERCHE : Informatique

Présentée par

Sofian Maabout

Contributions à l'Optimisation de Requêtes Multidimensionnelles

Soutenue le 12 Décembre 2014

Devant le jury composé de

Mme. Christine Collet	Professeur à l'INP de Grenoble	Rapporteur
M. Nicolas Hanusse	Directeur de Recherche au CNRS	Examinateur
M. Yves Métivier	Professeur à Bordeaux INP	Examinateur
M. Jean-Marc Petit	Professeur à l'INSA de Lyon	Rapporteur
M. Pascal Poncelet	Professeur à l'Université Montpellier 2	Rapporteur
M. Philippe Pucheral	Professeur à l'Université de Versailles Saint-Quentin	Examinateur
M. David J. Sherman	Directeur de Recherche à INRIA Bordeaux Sud-Ouest	Examinateur
M. Jef Wijsen	Professeur à l'Université de Mons	Examinateur

Table des matières

Chapitre 1

Introduction Générale

L'analyse mutidimensionnelle des données est un sujet vase qui va des statistiques avec des techniques telles que l'analyse en composantes principales au calcul haute performance en passant par l'extraction de connaissances via du clustering ou la classification, les bases de données, l'algorithmique et les techniques d'approximation.

Nos contributions à l'analyse multidimensionnelle suivent la même démarche, à savoir : Nous considérons un type particulier d'analyse, dans la terminologie bases de données nous parlons de requête multidimensionnelle, et nous proposons des solutions d'optimisation. En général, il s'agit de minimiser le temps d'exécution de l'analyse. Mais pas seulement. En effet, certaines requêtes sont gourmandes en mémoire. Ainsi, un deuxième critère que nous tentons d'optimiser est la consommation mémoire.

Notre démarche pour ce faire consiste à exploiter autant que faire se peut l'infrastructure matérielle à disposition notamment par la disponibilité de plusieurs unités de calcul. En effet, même les ordinateurs portables sont actuellement équipés de plusieurs processeurs. Ainsi, l'algorithmique parallèle n'est plus un luxe réservé à quelques centres de calcul de grands organismes de recherche. A titre d'exemple, en 2013 une machine avec 12 processeurs et une mémoire de 64 Go peut être achetée à moins de 5KE.

Certains problèmes sont théoriquement prouvés qu'ils sont *difficiles*. N'empêche qu'en pratique, la pire des situations est rarement atteinte et avec de bonnes heuristiques et des *astuces* de programmation, l'on arrive a résoudre des instances de grandes tailles. L'exemple notoire est le problème SAT qui est en quelque sorte l'étalon des problèmes NP-Complets et qui malgré tout continue à susciter des avancées, théoriques et pratiques, qui permettent actuellement d'envisager de résoudre des instances à plusieurs millions de variables. Nos travaux tiennent compte de ce fait d'où le soucis d'accompagner à chaque fois l'analyse théorique d'expérimentations permettant si ce n'est de *valider* l'approche, du moins d'en donner un aperçu sur son potentiel.

Un troisième axe que nous considérons est celui ayant trait à l'algorithmique approché. En effet, dans plusieurs applications pratiques, un résultat approché (avec cependant une marge d'erreur maîtrisée) est largement suffisant du moment que ce dernier peut être obtenu avec une complexité acceptable. Ceci est d'autant plus vrai quand il s'agit de traiter de grandes quantités de données et où même un algorithme exact de complexité quadratique n'est pas viable.

Comme dit plus haut, nos derniers travaux ont porté sur quelques requêtes multidimensionnelles bien précises auxquelles nous avons tenté de répondre en utilisant autant que possible les trois axes d'optimisation.

Les requêtes que nous avons choisies de décrire dans ce manuscrit sont (i) le calcul des bordures, (ii) les requêtes d'agrégation telles qu'on les rencontre dans les cubes de données et (iii) les requêtes multidimensionnelles de préférence type Skyline.

– le concept de *bordure* a été introduit par Mannilla et Toivonen dans leur article fondateur de

1997 [70]. Ils ont montré la généricité de ce concept en le déclinant selon différentes applications. Pour en donner une intuition sans une définition précise, considérons le cas d'un parent qui s'est fixé un certain budget pour acheter les cadeaux de Noël à ses enfants et qui dispose d'un catalogue du prix des jouets proposés dans un magasin. Comme le parent veut faire plaisir à ses enfants, il est intéressé par trouver les ensembles qui contiennent le maximum de jouets et dont la somme des prix ne dépasse pas son budget. Ces ensembles de jouets forme en quelque sorte une une frontière entre les ensembles qu'il peut acheter (le prix ne dépasse pas le budget) et ceux qui sont trop chers. Bien que le problème d'extraction de bordures fût montré NP-difficile, plusieurs algorithmes ont été proposés afin de le résoudre d'une manière efficace. Celle-ci étant le plus souvent étayé par des expérimentations. Notre contribution dans ce domaine est la proposition d'un algorithme parallèle imitant le meilleur algorithme séquentiel, dans le sens où il effectue exactement les mêmes calculs tout en exploitant au maximum la puissance de calcul disponible. Nous avons implémenté notre algorithme dans plusieurs contextes et avons comparé ses performances avec les approches de l'état de l'art. Il s'avère que plus le nombre de dimensions croît, plus notre approche devient performante vis à vis de la concurrence.

– Dans les applications OLAP (On Line Analytical Processing), on utilise souvent un modèle multidimensionnel pour appréhender les données. En effet, les décideurs fixe un sujet d'analyse. Par exemple les ventes dans une chaîne de magasins. L'analyse du sujet choisi se fait à travers des combinaisons de dimensions pouvant avoir un impact sur le sujet d'étude. Par exemple, le lieu de la vente, le produit vendu et le client qui a acheté représentent trois dimensions. L'organisation des données sous forme multidimensionnel et notamment le concept de cube de données, a été essentiellement introduite et formalisée dans l'article de Jim Gray *et al.* [40]. L'idée est d'offrir à l'utilisateur une interface lui permettant de naviguer à travers les différentes dimensions en sélectionnant à chaque fois celles qui l'intéressent. Chaque choix correspond en réalité à une requête d'agrégation. Nous sommes donc en présence de 2^d requêtes possibles si d représente le nombre de dimensions. Les premiers travaux dans l'implémentation des cubes ont essentiellement porté sur l'optimisation de sa matérialisation totale. Plus précisément, il s'agit de précalculer le plus rapidement possible toutes les 2^d requêtes et les stocker. Ainsi, lors de l'exploration du cube par les décideurs, les réponses aux requêtes sont immédiates. Cependant, on a rapidement pris conscience que le calcul d'un cube entier n'était pas faisable en pratique pour deux raisons : le temps d'exécution et l'espace de stockage qui sont tous deux prohibitifs. S'est alors posé le problème de la matérialisation partielle des cubes. Typiquement, les travaux de la littérature considèrent des contraintes matérielles (par exemple un espace de stockage fixé) et essayent de trouver la *meilleure* partie du cube à précalculer. Par "meilleure", on entend généralement celle qui réduit le temps d'exécution des requêtes. Notre contribution sur ce sujet consiste à poser le problème d'une manière différente : l'utilisateur fixe la performance avec laquelle il veut que les requêtes soient évaluées et tout le problème consiste à minimiser les ressources (exemple, espace mémoire) pour y parvenir. Nous pensons que voir l'optimisation des requêtes sous cet angle est plus proche de la réalité actuelle notamment avec la vision *cloud computing* : l'utilisateur est prêt à payer le prix quand les performances sont garanties. Nous proposons plusieurs algorithmes (exactes et approchés) pour résoudre ce problème. Il est remarquable néanmoins que la notion de *bordure* trouve aussi une application dans ce contexte.
– Les requêtes de préférences sont étudiées depuis plusieurs années, que ce soit par la communauté bases de données ou bien les chercheurs en intelligence artificielle et recherche d'information. Globalement, il s'agit d'offrir à l'utilisateur de définir un *ordre* permettant de trier le résultat d'une requête standard. Cet ordre peut être obtenu par la combinaison des va-

leurs de plusieurs attributs. Ceci est d'autant plus important lorsque le résultat de la requête contient un grand nombre de tuples. Il s'agit alors de faire en sorte que les tuples les plus *importants* apparaissent en premier. C'est ce qui est par exemple étudié dans le cadre des requêtes Top-K où il s'agit de ne retourner que les K meilleurs tuples. Cette façon de procéder consiste essentiellement à associer à chaque tuple un *poids*, donc une valeur atomique, fruit de la combinaison de plusieurs valeurs, qui permet de classer les tuples selon un ordre total. Dans certains cas, il est difficile, voire impossible, d'avoir une composition d'attributs, qui ait un sens pour l'utilisateur. On se retrouve donc à devoir comparer les tuples selon plusieurs critères *pris séparément*. Ce problème a depuis longtemps été étudié en mathématiques et informatique sous le nom de *vecteurs maximaux* (voir par exemple [56]. En 2001, Kossmann *et al* dans [14] a introduit ce concept à la communauté "bases de données" sous le nom de skyline. L'exemple le plus souvent utilisé pour expliquer le concept est celui d'une table relationnelle qui contient une description de chambres d'hôtel par un attribut qui représente son prix par nuitée et la distance de l'hôtel où se trouve la chambre vis à vis de la plage. Les utilisateurs sont intéressés par réduire *simultanément* et le prix et la distance. Or, ces deux caractéristiques sont généralement antagonistes. Ce que l'on peut faire dans ce cas, c'est de retourner à l'utilisateur les chambres qui ne sont pas *dominées* par d'autres chambres, i.e., moins chères et plus proches de la plage. Notre contribution dans ce domaine consiste à considérer la situation où l'utilisateur dispose d'un ensemble de dimensions parmi lesquelles, il peut choisir un sous-ensemble qui sera utilisé pour le calcul du skyline. Pour rester sur l'exemple des chambres d'hôtel, un Emir ne va peut être pas chercher à minimiser le prix de la chambre mais tiendra à maximiser sa superficie. A contrario, un étudiant tiendra plus particulièrement à minimiser le prix. On se retrouve donc avec une structure de cube de données particulière, appelée dans la littérature *skycube*. Nous proposons des solutions pour précalculer totalement et/ou partiellement les skycubes en se basant sur la notion de dépendances fonctionnelles elles même extraites via l'exploitation des bordures.

Organisation du manuscrit

Le présent manuscrit est composé, en plus de l'introduction et de la conclusion, de trois chapitres décrivant nos travaux sur les trois thématiques décrites ci-haut. Il est à noter que l'ordre de présentation ne respecte pas l'ordre chronologique avec lequel nous les avons abordés. En fait, c'est le problème de l'optimisation des requêtes dans les cubes qui nous a amené à traiter les bordures. A travers ces dernières, nous avons été amené à étudier entre autres, leur application pour l'extraction des dépendances fonctionnelles. Ces dernières se sont avérées utiles pour l'optimisation dans les skycubes. Les preuves des résultats sont omises afin d'alléger la lecture du présent document. Le lecteur intéressé peut les trouver dans les articles scientifiques s'y rapportant.

Chapitre 2

Calcul Parallèle de Bordures et Applications

The *border* concept has been introduced by Mannila and Toivonen in their seminal paper [70]. This concept finds many applications, e.g maximal frequent itemsets, minimal functional dependencies, emerging patterns between consecutive database instances and materialized view selection. For large transactions and relational databases defined on n items or attributes, the running time of any border computation is mainly dominated by the time T (for standard sequential algorithms) required to test the *interestingness* of candidates sets.

In this chapter we propose a general parallel algorithm for computing borders whatever the application is. We prove the efficiency of our algorithm by showing that : (i) it generates exactly the same number of candidates as the standard sequential algorithm and, (ii) if the interestingness test time of a candidate is bounded by Δ then for a multi-processor shared memory machine with p processing units, we prove that the total interestingness checking time T_p satisfies $T_p < T/p + 2\Delta n$ where n designates the dimensionality of the treated problem (e.g., n is the number of items, the number of attributes, ...). We implemented our algorithm in various settings and our experiments confirm our theoretical performance guarantee.

2.1 Introduction

Let us first recall the definition of *borders* as it has been introduced in [70]. Let $\mathcal{O}$ be a set of objects, $\mathbf{r}$ be a database and q be a predicate evaluating the *interestingness* of $O \subseteq \mathcal{O}$ over $\mathbf{r}$. In other words, $q(O,\mathbf{r}) = True$ iff O is interesting. On the other hand, let $\sqsubseteq$ be a partial order between the subsets of $\mathcal{O}$. The border of $\mathcal{O}$ with respect to $\mathbf{r}$, q and $\sqsubseteq$ is the set of subsets $O \subseteq \mathcal{O}$ such that (i) $q(O,\mathbf{r}) = True$ and (ii) there is no $O' \neq O$ such that $O \sqsubseteq O'$ and $q(O',\mathbf{r}) = True$. We illustrate this notion by the following examples :

- Let $\mathcal{O}$ be a set of items and $\mathbf{r} = \{O_1, \ldots, O_m\}$ be a transactions database defined as a multi-set with $O_j \subseteq \mathcal{O}$ for $1 \leq j \leq m$. The support of $O \subseteq \mathcal{O}$ is the number of transactions $O_j \in \mathbf{r}$ such that $O \subseteq O_j$. Let σ be a support threshold. We define $q(O,\mathbf{r}) = True$ iff $support(O,\mathbf{r}) \geq \sigma$. By considering $\sqsubseteq$ as the set inclusion relationship, the border of $\mathcal{O}$ in this context is actually the set of maximal frequent itemsets. The problem of extracting maximal frequent itemsets (MFI's) has been studied for a long time (see e.g. [11, 16, 38, 39, 101]). The most efficient implementations follow a depth first strategy which we will explain later in Section 2.4.1.
- Let $\mathbf{r}$ be relational table and let $\mathcal{A}$ be its set of attributes. Let $A \in \mathcal{A}$ and $\mathcal{O} = \mathcal{A} \backslash \{A\}$. A set of attributes $O \subseteq \mathcal{O}$ is interesting (i.e $q(O,\mathbf{r}) = True$)) iff $\mathbf{r}$ satisfies the functional dependency $O \rightarrow A$. Finding the minimal subsets O of $\mathcal{O}$ such that $\mathbf{r} \models O \rightarrow A$ aims at finding the border of $\mathcal{O}$ by considering $O \sqsubseteq O'$ iff $O \supseteq O'$. Hence, finding the minimal non trivial functional dependencies (FD's) satisfied by a relation turns to be a border computation. Again, extracting the exact or approximate FD's has attracted a real interest either for query optimization or

database reorganization (see e.g [52, 66, 75, 94]. [94] is among the most efficient algorithms designed for this purpose this why it has been adapted in [28] for mining conditional FD's. One should notice that this algorithm also follows a depth first strategy.

– Let $\mathcal{O}$ be a set of items. If $I \in \mathcal{O}$ then $I.price$ denotes the price of I. Let $\mathbf{r}$ be a table containing the price of each item I. $O \subseteq \mathcal{O}$ is interesting (i.e $q(O,\mathbf{r}) = True$)) iff $\sum_{I \in O} I.price \leq \sigma$ where σ is a budget threshold. If $\sqsubseteq$ denotes the set inclusion relationship, then the border with respect to $q, \mathbf{r}$ and $\sqsubseteq$ is the maximal subsets of $\mathcal{O}$ whose prices (the sum of their elements prices) do not exceed threshold. This problem has been studied in [83] in an e-commerce application where a customer searches for a product by providing his/her budget. The system returns not only the searched product but also suggests other related items while not exceeding the customer's budget.

Other recent applications of borders can be found e.g. in [74] where it is used to characterize the emerging parts of a datacube between two consecutive states, in [46] for datacube partial materialization and in [72] for extracting the most interesting attributes w.r.t. a query log. Without trying to be exhaustive, we hope the reader is convinced that borders are found in several applications.

However, computing borders is a very time consuming task. Indeed, [41] shows that it is NP-Hard with respect to the number of dimensions. Therefore, either we rely on heuristics to optimize the computation (for example, the way we traverse the search space), try to approximate the result (for example, by using probability arguments) or exploit parallelism (data and/or task parallelism). [70] proposed a general level-wise algorithm for computing the borders when q is *anti-monotone*. Recall that q is anti-monotone iff whenever $q(O,\mathbf{r}) = False$ then $q(O',\mathbf{r}) = False$ for each $O' \sqsupseteq O$. In the three examples above, one may easily check that q is anti-monotone. The main parts of any border computations algorithms are the candidates management and the interestingness tests. Whatever is the underlying data structure (tables, FP-trees, ...), it turns out that for large datasets, the interestingness test is the most consuming task. The algorithm of [70], akin to A priori [6], turns to be inefficient in practice in that it tests much more candidates than the algorithms following a depth first strategy (DFS). The reason is that DFS exploits the anti-monotone property upward and downward to prune the candidates. In order to parallelize DFS algorithms, the immediate solution would be to partition the data and each time we have to check the interestingness of a candidate, we test it in parallel in every part then we combine the results, i.e., data parallelism. The problem here is that we have no guarantee that doing so, the time required is not more than that of a sequential algorithm, e.g. for MFI mining, prefix trees [44] are often used to summarize the data and it may happen that each tree corresponding to a part has the same size as the global one. Therefore, the performance of the parallel version is much worse than the sequential one. To sum up and to the best of our knowledge, no existing parallel algorithm for computing borders does provide a theoretical guarantee to run faster than a sequential execution.

In this chapter we propose a parallel algorithm, **MineWithRounds**, which guarantees a speed up over the standard sequential depth first algorithm. To simplify its description we consider two applications : mining maximal frequent itemsets and mining functional dependencies. This shows that with very little adaptation, the same algorithm can be used in various settings.

This chapter contains two main parts : the first one is devoted to MFI's mining and the second one to dependencies extraction. We first start with MFI's.

2.2 Basic Concepts Related to MFI's

Let us first recall the basic concepts related to maximal frequent itemset (MFI) mining. Let $\mathcal{I} = \{I_1, \ldots, I_n\}$ be a set of items and $\mathbf{T} = \{T_1, \ldots, T_m\}$ be a transaction database where $T_j \subseteq \mathcal{I}$ for

$1 \leq j \leq m$. The support of $I \subseteq \mathcal{I}$, noted $support(I)$, is the number of transactions $T_j \in \mathbf{T}$ such that $I \subseteq T_j$. Let σ be a support threshold. Then I is frequent iff $support(I) \geq \sigma$ and I is an MFI iff it is frequent and there is no $J \supset I$ such that J is frequent.

Example 1. *Let $\mathcal{I} = \{A, B, C, D, E\}$, $\sigma = 2$ and the transaction database $\mathbf{T}$ described below :*

TId	$Transactions$
1	ABC
2	AD
3	$ABCD$
4	$ACDE$
5	ABC

The itemset AB is frequent since $support(AB) = 3 \geq \sigma$. However, it is not an MFI since, ABC is also frequent and contains AB. The MFI's are $\{ABC, ACD\}$.

The problem of extracting maximal frequent itemsets has been studied for a long time (see e.g. [11, 16, 38, 39, 101]). The most efficient implementations follow a depth first strategy which we will explain later in Section 2.4.1. Among the reasons of this interest, one can notice that the MFI's is a good summary of frequent itemsets. Indeed, due to the monotonicity of the support, all frequent itemsets (FI's) can be recovered from the maximal ones. Of course, this summary is lossy in that one cannot recover the actual support of the FI's. This is a problem when we are interested in mining association rules. However, in many applications we just need the MFI's. As an example, suppose that items represent the relational attributes used in queries conditions. Hence, each transaction is actually the set of attributes used in a query. Finding the maximal frequent attributes set is sufficient to obtain a set of index candidates that may help queries optimization, see e.g., [15].

2.3 Related Work w.r.t MFI's Mining

MFI mining is an NP-Hard problem [41, 97]. Hence, to speed up its computation, one should either use heuristics, parallelism, approximation and/or clever traversal of the exponential search space. For this last item, there are essentially two strategies that have been followed so far : (i) the depth first strategy (DFS), e.g., Mafia [16], GenMax [38], DepthProject [4], FPmax* [39] and PADS [101], and (ii) the level wise or breadth first strategy (BFS), e.g., Pincer search [64] and MaxMiner [11]. DFS strategy provides better performance than BFS. This is essentially because the former does not exploit downward pruning : if an itemset is frequent then its subsets cannot be maximal so they can be pruned.

DFS algorithms organize the search space as a tree ; the so called lexicographic tree. Let us recall this structure by considering $\mathcal{I} = \{A, B, C, D, E\}$ as the set of items. DFS algorithms traverse the tree depicted in Figure 2.1.

In order to compare **DFS** and **BFS**, suppose that the unique MFI is $\{ABCDE\}$. **DFS** *climbs* the left most branch of the tree until $ABCDE$ and stops. Hence, it computes *only* n supports while **BFS** computes the support of all subsets of $ABCDE$ thus 2^n supports.

Suppose now the MFI's are $\{ABCD, E\}$. **DFS** will compute the support of A, AB, ABC, $ABCD$ and $ABCDE$ (in this order) then all the supersets of E. Hence, for *just* 2 itemsets, **DFS** may compute $O(2^n)$ supports where n is the number of items. For the same set of MFI's, **BFS** does not test all supersets of E but does test all subsets of $ABCD$. Finally, for this example, both algorithms make the same number of tests. This case is the worst case for both **DFS** and **BFS**.

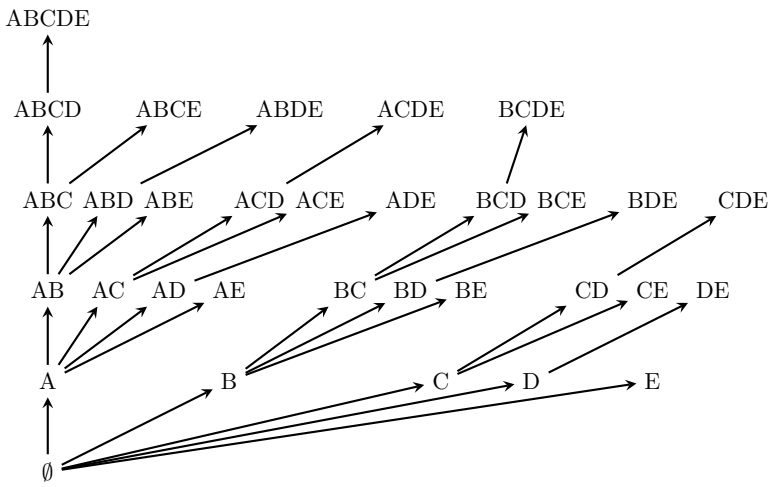


Figure 2.1 – The lexicographic tree of $\{A, B, C, D, E\}$

These examples show (i) how the worst case complexity ($O(2^n)$) can be attained by both **DFS** and **BFS** and why **DFS** can be much more efficient than **BFS**.

Most algorithms make use of heuristics in order to maximize the pruning opportunities. For example, the lexicographic order corresponds actually to items ordering with respect to their ascending support ordering. The rational behind this is that the maximal frequent itemsets with less support should should the the roots of the largest subtrees. Since the MFI's containing them should rather be short (in terms of the number of items they contain), hence one may hope to maximize upward pruning.

The most efficient implementations for MFI computation (FPMAX and PADS) use projection (or conditional transactions) in order to speed up support computation. The principle is as follows : when the support of A is computed, we keep track of those transactions that contain A. This set, denoted $\mathcal{T}_A$, is the projection of the original data onto the itemset A. Then, if we need to compute the support of AB, we just need to compute the support of B in $\mathcal{T}_A$. Doing so, we also obtain $\mathcal{T}_{AB}$. During tree traversal, each $\mathcal{T}_I$ is kept in memory until the subtree rooted at I is completely mined. Even its effectiveness, this technique could be quite memory consuming when the projections are large. Moreover, in a parallel algorithm, this could be unpractical because we could have to keep too many projections.

Some algorithms use even dynamic ordering, e.g., in Figure 2.1, when A is found frequent, instead of considering AB as the next candidate, the subtree rooted at A is reordered with regard to the individual supports of $B, \ldots, E$ in the projected transaction data base on A, i.e., the restriction of transactions to those containing A. Furthermore, the FP-tree data structure [43] is an efficient way to summarize the underlying data making the support computation fast.

Look ahead is another heuristics used by the algorithms. It consists simply to test each candidate together with the maximal itemset belonging to its subtree, e.g., B is tested processed together with $BCDE$. If the latter is frequent, then the whole subtree rooted at B can be pruned.

[72, 83] make use of an approximation technique based on randomization. The principle is as follows : first choose randomly an item then traverse the search space bottom-up until reaching an MFI. At each stage, the next candidate is also randomly selected. The process stops when all previously mined MFI's have been found twice. It is proven that the result does contain *all* MFI's with high probability. We note however that the worst case complexity of this technique is also

exponential. Therefore, even if the authors found it efficient from a practical point of view, no argument is provided for supporting this claim nor do they compare it with other solutions.

To the best of our knowledge, two parallel algorithms [21, 27] have been proposed so far. Both of them use data parallelism. [21] partitions the data and parallelizes the support computation of each candidate by using MaxMiner [11]. Due to its BFS strategy, MaxMiner does compute the support of more candidates than DFS algorithms. [27] guesses a set of MFI's candidates by considering all branches of the prefix tree summarizing the data. If a branch is frequent then it is potentially an MFI otherwise it is intersected with other non frequent candidates to generate new ones. Each node in of the network executes this procedure in its local data then sends the result to the master node. This later combines the partial results then it generates the next potential candidates which are to be tested by the slaves nodes.

[61] proposed FP-arrays, a data structure for representing FP-trees in an array fashion so that logically close nodes of the tree (parent-child) are effectively close to each other in the physical memory. This data structure reduces drastically cache misses. The authors proposed a lock free parallel algorithm for building this structure. However, the parallel mining process is again data oriented as it has been proposed in [19]. We should note that this proposal is for computing all frequent itemsets not only the maximal ones.

2.4 Basic Definitions

In this section we introduce the main definitions. $\mathcal{I}$ will denote the set of items present in the transaction database. Capital letters $I, J, \ldots$ denote itemsets and lowercase letters $i, j, \ldots$ denote items. We start by defining a total order among the items.

Definition 1 (Item Rank)**.** *Let $\lhd$ be a total order over the items of $\mathcal{I}$. Then $Rank(i) = r$ iff $|\{j \mid j \lhd i\}| = r - 1$.*

Example 2. *Let $\mathcal{I} = \{A, B, C, D\}$ be the set of items. Suppose that $\lhd$ is the alphabetical order. Then $Rank(A) = 1, Rank(B) = 2, \ldots, Rank(D) = 4$.*

Without loss of generality, the order $\lhd$ is equivalent to the alphabetical order. We also consider only *ordered itemsets* i.e., if $I = i_1 i_2 \ldots i_k$ is a k-itemset then $j < l \Leftrightarrow i_j \lhd i_l$ for $1 \leq j, l \leq k$. For example, ACD is an ordered itemset while DAC is not. As usual, from the order $\lhd$ we can define the lexicographic order between itemsets as follows : Let $I = i_1 i_2 \ldots i_m$ and $J = j_1 j_2 \ldots j_\ell$ two ordered itemsets. Then $I \prec J$ iff there exists p such that $i_k = j_k$ for $k < p$ and $i_p \lhd j_p$. For example $ABD \prec B$. Moreover, we say that I is an ancestor of J iff $I \supseteq J$ and I is a parent of J iff I is a $J's$ ancestor and $|I| = |J| + 1$. For example, ABC is an ancestor of B and it is a parent of AB. With respect to an itemset I and to the order $\prec$ we distinguish two kinds of ancestors (resp. parents) : those that precede I, called *left ancestors*, and those that follow it, called *right ancestors*. For example, AB and BC both are parents of B. Since $AB \prec B$ and $B \prec BC$, then AB is a *left parent* of B while BC is a *right parent* of B. A set of itemsets $\mathcal{S}$ *covers* I iff $\mathcal{S}$ contains an ancestor of I. We define the *position* of an itemset I, denoted $Pos(I)$, as the rank of I with respect to $\prec$ order. More precisely,

Definition 2. *Let $(2^{\mathcal{I}}, \prec)$ be the ordered set of itemsets. $Pos(I) = |\{J \mid J \prec I\}|$.*

For example, if $\mathcal{I} = \{A, B, C, D\}$ then $Pos(A) = 1$ (A follows $\emptyset$), $Pos(D) = 2^4 - 1$ and $Pos(AB) = 2$. From these definitions, the structure of the lexicographic tree should be clear : $\mathcal{T}$ is the lexicographic tree of $\mathcal{I}$ iff for each itemset $I \subseteq \mathcal{I}$ all the right ancestors of I belong to the subtree of $\mathcal{T}$ rooted at I. The tree in Figure 2.1 is the lexicographic tree of $\mathcal{I} = \{A, B, C, D, E\}$.

Finally, let $I = i_1 \dots i_m$ be an ordered itemset. The successor of I, denoted $Successor(I)$, is the itemset $J = i_1 \dots i_m i_k$ such that $rank(i_k) = rank(i_m) + 1$. Not all itemsets have a successor. For example, if $\mathcal{I} = \{A, B, C, D, E\}$ then $Successor(AC) = ACD$ while AE has no successor. Clearly, the successor of I is its first parent with respect to $\prec$.

2.4.1 Pure Depth Traversal

We start with a simple algorithm for extracting MFI's. This DFS algorithm traverses the lexicographic tree. We assume that the itemsets are coded by a binary vector $V(1 \dots n)$ where $V[j] = 1$ means that item i_j is present and $V[j] = 0$ otherwise. We note the set of MFI's as **MFI** where **MFI** stands for Border. Parameters n and σ are respectively the total number of items and the support threshold. Integer i is a position index in the vector V. The algorithm is called by initializing i to 1, **MFI** as empty and all positions of V set to 0.

Procedure DFS(integer i)

```
   Input: integer i, s : MinSupport
   Output: MFI
1  if i ≤ n then
2      V[i] ← 1;
3      if MFI covers V then
4          // V is not maximal
5          DFS(i+1);
6      else
7          if support(V) ≥ s then
8              // V is potentially maximal
9              Add V to MFI;
10             Remove from MFI the subsets of V;
11             DFS(i+1);
12     V[i] ← 0;
13     DFS(i+1);
```

To illustrate this algorithm, let us assume that $\mathcal{I} = \{A, B, C, D, E\}$ and the maximal frequent itemsets are $\{ABDE, BC\}$. Figure 2.2 shows the trace of DFS execution. Itemsets in a red area are upward pruned and those in green are are downward pruned. The support computation is performed for the rest of itemsets.

Despite the simplicity of this algorithm, most of the efficient solutions proposed so far use a variation of it in order to optimize the two critical operations it performs namely, the covering test (line 3) and the support test (line 7). The covering test may require the comparison of V to all elements of **MFI**. Several techniques have been proposed in order to reduce or optimize this test. For example, [38] maintains a *local* subset of **MFI** where the test is performed. Another variation consists in using a *tail* structure associated to each candidate [16, 38]. These structures are used when backtracking by avoiding to visit hopeless candidates, i.e., those that are surely covered. For example, when considering candidate AB in the example above, its tail is first set to $\{C, D, E\}$. If subsequently ABD is found frequent then D is removed from $tail(AB)$ and when $ABDE$ is tested, E is also removed from $tail(AB)$. Hence, when the execution backtracks to AB it will not visit ABE since E is no more in $tail(AB)$. The process backtracks immediately to AC. One should notice that these techniques do not reduce the number of support computations.

From the discussion above, we may conclude that when mining MFI's with **DFS**, we are facing two bottlenecks : support and covering tests. When the number of **MFI**$'s$ is much smaller than the data, which is the case when data is very large, the support tests dominate the overall computation

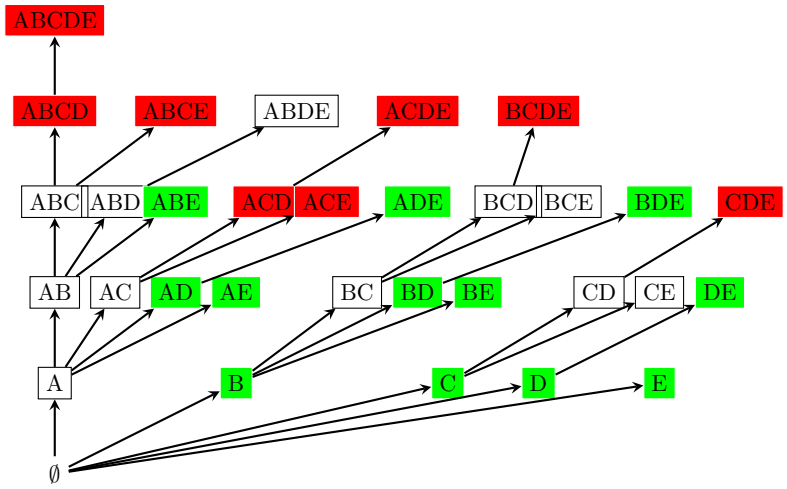


Figure 2.2 – Execution trace of DFS

time. This is the reason why in this chapter we rather focus on parallelizing the supports computation task.

2.5 MineWithRounds Algorithm

As we have seen in the previous section, **DFS** processes *sequentially* the set of itemsets following the lexicographic order $\prec$. From time to time, it makes *jumps* that correspond to upward pruning. In this section we propose a new algorithm which *mimics* **DFS** and whose rationale consists in triggering the processing of an itemset *as soon as possible* while still respecting the $\prec$ order, i.e., we have no chance to obtain new information about the status of this itemset if we postpone its treatment. More precisely, suppose that I has just been processed (its support has been checked) and let J be an itemset such that $I \prec J$. We want to start the processing of J whenever none of the itemsets K lying between I and J, *i.e.,* $I \prec K \prec J$, is able to provide any insight about J.

For example, consider the itemsets AB and B from Figure 2.2. Once AB is processed, we can start the processing of B. Indeed, either AB is frequent and in this case B is covered [1] or AB is not frequent and in this case none of the itemsets whose position is between $Pos(AB)$ and $Pos(B)$ will provide an information telling that B is frequent nor B is not frequent. Figure 2.3 illustrates this. The shaded region contains the itemsets whose processing is useless for gaining any information about itemset B once AB is processed.

The second principle of the proposed algorithm consists in processing several itemsets in parallel. Roughly speaking, we partition the set of itemsets into rounds $\mathcal{R}_1, \ldots, \mathcal{R}_d$ such that an itemset is processed during the i^{th} iteration if it belongs to $\mathcal{R}_i$. Let us first formalize our partitioning procedure.

Definition 3 (Depth First Partitions)**.** *Let $(\mathcal{R})_{i\geq 1}$ be a partition of $2^{\mathcal{I}}$. $\mathcal{R}$ is a Depth First Partition (DFP) of $2^{\mathcal{I}}$ w.r.t. $\prec$ if and only if for each itemset I,*

1. *$I \in \mathcal{R}_k$ and I has a successor $\Leftrightarrow$ $Successor(I) \in \mathcal{R}_\ell$ where $\ell > k$ and*
2. *$I \in \mathcal{R}_k \Leftrightarrow \forall$ left parent I' of I, $I' \in \mathcal{R}_l$ where $l < k$.*

1. Processing B in this situation means just discarding it as a candidate.

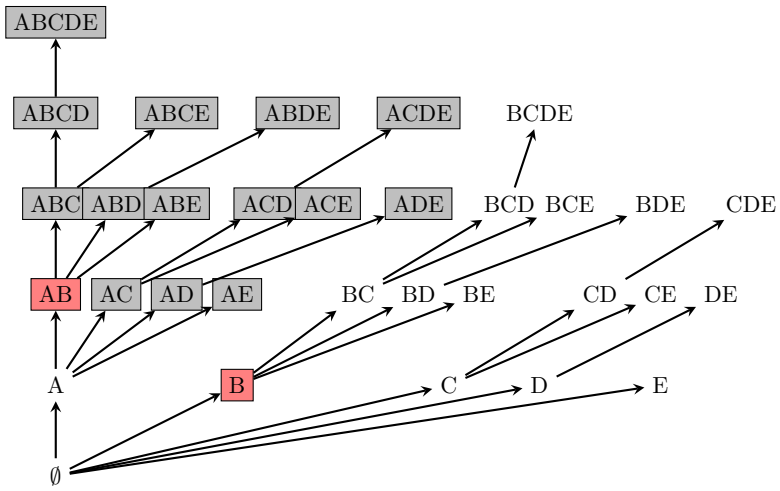


Figure 2.3 – Useless itemsets for B once AB is processed.

Informally, the above conditions impose that (1) each itemset should be processed before its successor and (2) each itemset should not be processed until all its left parents have been processed. These two conditions together insure that the processing respect the order $\prec$ followed by DFS. Clearly, there may be many DFP's for the same $\mathcal{I}$. As an example, the partition $\mathcal{R} = \{\mathcal{R}_1, \ldots, \mathcal{R}_{2^n-1}\}$ where $I \in \mathcal{R}_j \Leftrightarrow Pos(I) = j$ is a DFP partition.

Now we give a very naïve algorithm which, given a DFP, analyses its parts iteratively for finding the MFI's. This algorithm is very inefficient because it assumes that the DFP is statically given which means that one should provide the 2^n possible candidates beforehand. Still, it is interesting because it shows the fundamental role of DFP's regarding a depth first search strategy. Indeed, we show that this algorithm performs exactly the same support tests as **DFS**.

Procedure Naïve($\mathcal{R}$)

Input: $\mathcal{R} = \{\mathcal{R}_1, \ldots, \mathcal{R}_m\}$: a DFP, s : MinSupport
Output: **MFI**

```
1  MFI = ∅;
2  for i = 1...m do
3      foreach I ∈ R_i do
4          if I is not covered by MFI then
5              if Support(I) ≥ s then
6                  Remove from MFI the subsets of I;
7                  Add I to MFI;
8              else
9                  for j = i...m do
10                     Remove from R_j the supersets of I;
11 Return MFI
```

Theorem 1. *Let $\mathcal{R}$ be a DFP. Then* **Naïve** *and* **DFS** *perform exactly the same support tests.*

From task parallelism point of view, not all DFP's are interesting, e.g., the partition $\mathcal{R} = \{\mathcal{R}_1, \ldots, \mathcal{R}_{2^n-1}\}$ where we have as many parts as there are itemsets is not interesting since each part contains only one candidate : thus we always have just one candidate to test.

Therefore, we seek the *smallest* partitions in terms of the number of parts, i.e., the ones that

merge as many itemsets as possible in each $\mathcal{R}_i$ while respecting DFP conditions. We actually show that there exists a unique minimal partition.

Definition 4. *Let $\mathcal{R}$ and $\mathcal{R}'$ be two DFP's. Then, $\mathcal{R}$ is smaller than $\mathcal{R}'$ iff $|\mathcal{R}| < |\mathcal{R}'|$ where $|\mathcal{R}|$ denotes the number of parts in $\mathcal{R}$.*

Example 3. *Let $\mathcal{I} = \{A, B, C\}$. One can easily check that $\mathcal{R} = \{\{A\}, \{AB\}, \{B\}, \{ABC\}, \{AC, BC\}, \{C\}\}$ and $\mathcal{R}' = \{\{A\}, \{AB\}, \{B\}, \{ABC\}, \{AC\}, \{BC\}, \{C\}\}$ are both DFP's. $\mathcal{R}$ is smaller than $\mathcal{R}'$.*

Definition 5 (Round of Itemsets)**.** *Let $I = I_1 \dots I_k$ be an itemset such that $rank(I_k) = r$. Then $Round(I) = 2r - k = i$*

Example 4. *If $\mathcal{I} = \{A, B, C, D, E\}$ then $Round(ABD) = rank(D) - 3 = (2 \times 4) - 3 = 5$.*

Theorem 2. *Let $\widehat{\mathcal{R}} = \{\mathcal{R}_1, \dots, \mathcal{R}_m\}$ be a partition of $2^{\mathcal{I}}$ such that $I \in \mathcal{R}_j \Leftrightarrow Round(I) = j$. Then $\widehat{\mathcal{R}}$ is the smallest DFP partition.*

From here on, $Round(I)$ denotes the round where I belongs to. As a consequence of the above theorem, we have the following property : the number of rounds (parts) $|\widehat{\mathcal{R}}|$ is $2n-1$. The following example illustrates the different notions we introduced.

Example 5. *Let $\mathcal{I} = \{A, B, C, D, E\}$ be the set of items. Then, the round of each itemset is depicted in Figure 2.4.*

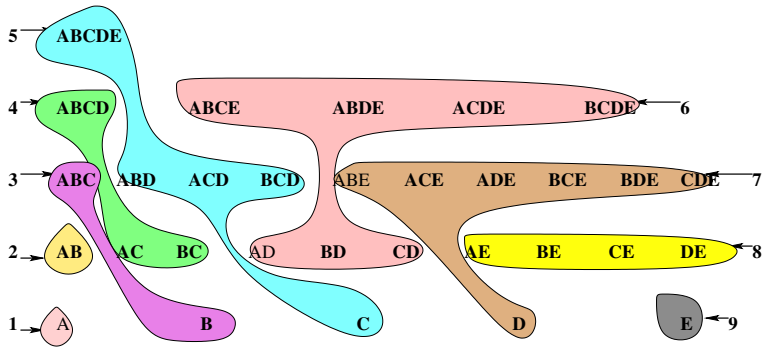


Figure 2.4 – Rounds assignment

We now describe **MineWithRounds** which uses the partitioning defined before. Intuitively, it uses a loop which considers at each iteration the itemsets of the corresponding round. We use the following data structures : $\mathcal{C}$ is an array of sets of itemsets sets. $\mathcal{C}[i]$ is the set of candidates to be processed during round i and **MFI** is the set of MFI's found so far. The right children of ABC are AC and BC (just do not consider its prefix). The right sibling of ABC is $\overline{ABD}$ (just replace the last item, here C, by its successor, here D). Note that if $Round(I) = i$ and J is the right sibling of I then $Round(J) = i + 2$. Intuitively, the algorithm proceeds as follows : If a candidate $I \in \mathcal{C}[k]$ is found frequent, then its successor is a candidate for the next iteration $k+1$. If I is infrequent then (i) its right children are *potential* candidates for the next iteration $k+1$ and (ii) its sibling is a *potential* candidate for iteration $k+2$. To explain the covering tests, suppose AC and BC are candidates tested during the same round. Suppose AC is frequent but not BC. Thus AC generates its parent ACD as a future candidate and BC generates both C (its right child) and BD (its sibling). Note that since C is covered by ACD, there is no need to consider it as a future candidate.

Algorithm 1: MineWithRounds

```
1  C[1] ← {I_1};
2  k ← 1;
3  while k ≤ 2n − 1 and C[k] ≠ ∅ do
4      foreach I ∈ C[k] do
5          // Loop executed in parallel
6          if support(I) ≥ σ then
7              Add I to MFI;
8              Remove the left child I' of I from MFI;
9              Add the right parent I'' of I to C[k + 1];
10         else
11             Add the right children of I to Children;
12             Add the right sibling of I to C[k + 2];
13     foreach I ∈ C[k + 1] do
14         // Loop executed in parallel
15         if I is covered by MFI then
16             Remove I from C[k + 1];
17     foreach I ∈ Children do
18         // Loop executed in parallel
19         if I is not covered by MFI then
20             Add I to C[k + 1];
21     k ← k + 1;
```

The reader should notice from the above algorithm description that the candidates are generated dynamically. Actually, at each stage, $C[k] \subseteq \mathcal{R}_k$. $\mathcal{R}_k$ is upward and downward pruned with respect to the previous computations.

In our implementation, the three **Foreach** loops in lines 4, 13 and 17 are executed in a parallel fashion. The first one essentially tests the support, the second checks the coverage of the sibling candidates that have been generated during iteration $k-1$ and the last one tests the coverage of the children that have been generated during iteration k. It is worthwhile to note that there are barriers between these loops, i.e. the second loop cannot start before the first has finished.

The following theorem, which is our main result, compares the performance of the above algorithm with DFS.

Theorem 3. DFS *and* **MineWithRounds** *perform exactly the same number of support computations.*

As a consequence, we can state the theoretical speed up with respect to the number of available processing units.

Corollary 1. *Let T and T_p be respectively the computations times of* **DFS** *and* **MinWithRounds** *when p processors are available. Then*

$$\frac{T}{p} \leq T_p \leq \frac{T}{p} + (2n-1)\Delta$$

Where n is the total number of items and Δ is the maximal time needed for one support computation.

Note that, since in general the number of candidates r is much larger than n, $(2n-1)*\Delta$ can be neglected with regard to T/p. Hence, the speed up of **MineWithRounds** is almost perfect. The worst case is when $r_i \mod p = 1$, e.g. suppose $p = 16$ and $r_i = 33$. Suppose that the 33^{rd} test takes $2sec$ and the others take $1sec$. Then, the sequential algorithm requires $34sec$. With p cores, the first 32 candidates are processed within $2sec$ while the last candidate requires $2sec$ by itself so a total of $4sec$. Hence, the speed up is $34/4 = 8.5$. Note that if we had 47 candidates and it is the 47^{th} one which requires $2sec$ then the parallel total time is again $4sec$ and the speed up is this time equal to $48/4 = 12$.

The following example illustrates the MFI's computation with our algorithm.

Example 6. *Let $\mathcal{I} = \{A, B, C, D, E\}$ and suppose that the maximal frequent itemsets are $\{ABDE, BC\}$. Figure 2.5 shows candidates generation. The arrows show candidates generation. For example, A is found frequent in round 1, therefore its parent AB is generated as a candidate. ABC is found infrequent during round 3, so its sibling ABD is generated together with its right children AC and BC. Candidates colored in red are those that are first generated then removed because they turn to be covered. For example, in round 4, AC is found infrequent so its right children C is generated as a candidate to be tested during round 5. But meanwhile, BC is found frequent. Hence, BC is added to the provisional set of MFI's and its parent BCD is scheduled for round 5. Hence, when the execution of round 5 starts, we find C covered by BC. The same remark holds for the sibling AD of AC which is first generated for round 6 then removed because during round 5, ABD is found frequent. Note that in this small example just two iterations contain more than one candidate, namely iterations 4 and 5. Thus, with 2 processors just these rounds will benefit from parallelism. However, realistic data sets with much more items exhibit rounds with thousands of candidates.*

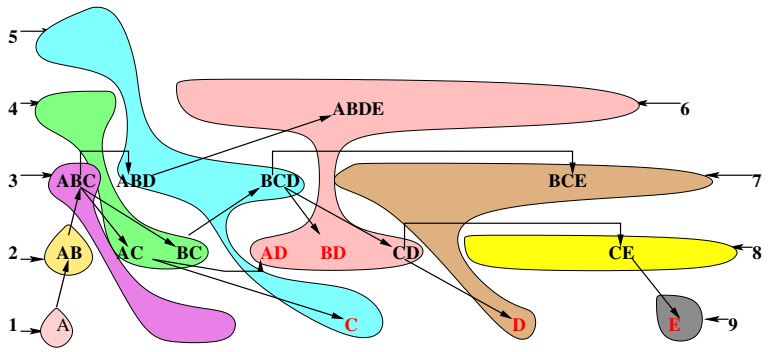


Figure 2.5 – Rounds assignment and **MineWithRounds** computation

2.6 Data Distribution

In this section we show how to adapt our algorithm to the case when data is distributed. An obvious way to do so is to consider one distinguished node as the *master* and the others are *slaves*. The master is responsible of the candidates and the border management while the slaves compute the interestingness on their respective local data. At each iteration, (1) the master sends all candidates to all slaves, (2) each slave computes the local support of all candidates, (3) each slave sends its results to the master, (4) the master combines the partial results to find the global *interesting* candidates, (5) it updates the border and (6) it generates new candidates

In the MFI setting, the communication complexity is the total size of messages. Let r_i be the the number of candidates processed during iteration i and p be the number of slaves. At each iteration i

the master sends a message of size r_i to the slaves and each slave notifies the master with a message of length r_i. Thus the total communication cost is $2p\sum_{i=1}^{2n-1} r_i$. Of course, other implementations are possible, e.g., instead of computing the support of each candidate by every slave, we let the later to make this computation only if the already computed support is not sufficient to make the candidate frequent. This will reduce the overall computation time. As one can see, our proposal of candidates partitioning is orthogonal to the way data is distributed. In fact, it leads itself easily to Map-Reduce setting : it is essentially the same problem as computing the number of words occurrences.

2.7 MFI's Mining Experiments

2.7.1 OpenMP

We implemented our algorithm using C++ and the STL library together with OpenMP [76] : a multi-shared programming API. OpenMP makes it very easy to exploit new multi-core architecture by just adding compilation directives. For example, the loop :

```
For (i=0; i<1000; i++)
     a[i]=f(i);
```

can be executed in parallel by adding just one line of code as follows

```
#pragma omp parallel for num_threads(4) schedule(dynamic)
For (i=0; i<1000; i++)
     a[i]=f(i);
```

The number of parallel threads that are launched in this case is fixed to `4`. Moreover, OpenMP enables the user to choose how threads should be scheduled. In the example above, we have chosen a `dynamic` scheduling meaning that the fours parallel threads will execute respectively the instruction `a[i]=f[i]`, for `i=1..4` then the first among them which terminates, will execute the same instruction by considering `i=5`, and so on. This type of scheduling is particularly interesting when the execution time of function `f` is variable depending on the `i` value. Our purpose here is not to explain the many facilities offered by OpenMP but rather to show how it is easy to parallelize source codes when using it.

We have checked the correctness of our implementation by comparing its results with other previous implementations (Mafia and PADS) as well as by considering the results reported in [31].

2.7.2 Machine

Our tests were conducted on a machine equipped with two quad-core Intel Xeon X5570 2.93GHz processors running under Redhat Linux enterprise release version 5.4. Thanks to their multi-threading ability, these processors can execute concurrently two threads per core. Therefore, we were able to launch up to 16 threads. Figure 2.6 shows this internal structure. It shows for example that in node 1, `core 0` contains two *logical* processing units `p#0` and `p#8` meaning that multi-threading is enabled.

2.7.3 Data sets.

We present the results we obtained with six well known datasets : Chess, Mushroom, T10I4D100K, T40I10D100K, Kosarak and Webdocs. Their respective characteristics are described in Table 2.1.

Depending on the number of transactions, we categorized the samples into small, medium and large data sets. With each dataset, we varied the minimal support threshold σ and with every such value, we varied the number of parallel threads executed for every **Foreach** loop in the algorithm

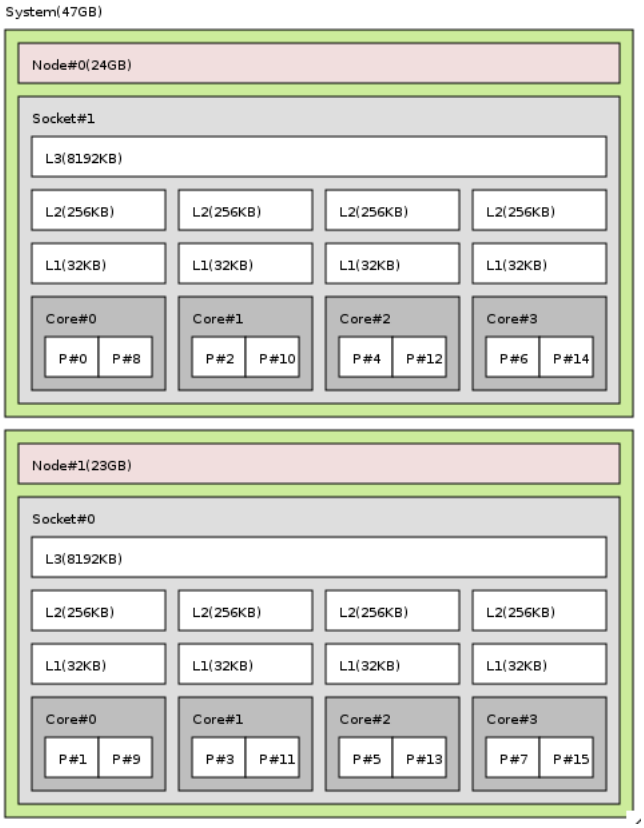


Figure 2.6 – The internal structure of the machine used for our experiments.

Dataset	# trans.	# Items	Avg. transact. length	Size
Chess	3196	75	37	Small
Mushroom	8124	119	23	Small
T10I4D100K	10^5	10^3	10	Medium
T40I10D100K	10^5	10^3	40	Medium
Kosarak	$\sim 10^6$	40348	8	Large
Webdocs	$\sim 1.7 * 10^6$	$\sim 5.3 * 10^6$	178	Large

Table 2.1 – Data description.

(the same number of threads for the three loops). This number varies from 1 to 16 as a power of 2. In order to gain load balancing, we used the *dynamic scheduling* of OpenMP.

2.7.4 Results Analysis

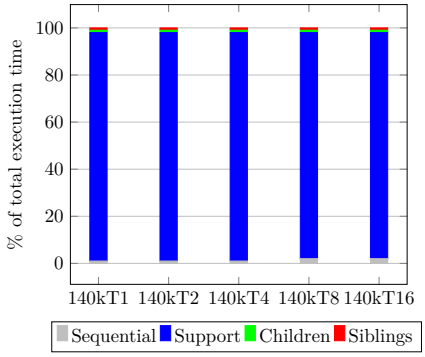
For each data set, we measured the time devoted to each of the three loops present in our algorithm (cf. Algorithm 1). The first loop essentially computes the support of a set of candidates while the two remaining loops make covering tests : One of them (line 17) tests the coverage of the candidates that have been generated as children of non frequent itemsets and the other (line 13) tests the candidates that have been generated as siblings. In our main theoretical result (Corollary 1), we claimed that when the time devoted to support computation is the dominating time, the speedup of our algorithm is almost perfect. The experiments not only confirm this result but also show that when this test is not very time consuming we still get interesting speed ups.

Figures 2.7, 2.8 and 2.9 show some of the obtained results. For each dataset, we consider two support thresholds and for each, there is one subfigure showing the proportion of time devoted to each of the three loops as well as the execution time of the sequential part of the algorithm. The second subfigure shows the speedup of each loop as well as the total speedup. The X_axis of each figure represents the minimal support value concatenated to the number of threads. For example, in subfigure 2.7(a), 140K_T16 means that the minimal threshold is 140K and the number of threads is fixed to 16.

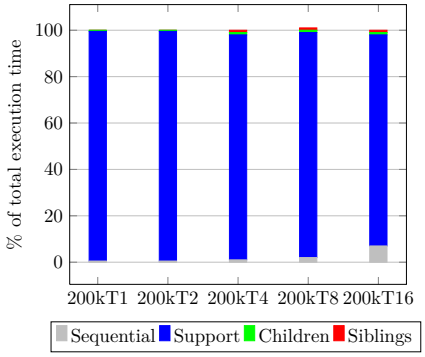
Large data

With Webdocs dataset, more than 90% of the time is consumed by the support computation (see Figures 2.7(a) and 2.7(b)). Hence, the speed up of the overall execution is almost equal to the support computation speed up. Note that Figure 2.7(c) shows a total speedup of 30 while the number of threads is *just* 16. Moreover, the number of physical processors is 8. This super-linear speed-up is explained by data locality. Indeed, when two threads access the same data, the system will first check the different levels of the cache before making an access to the memory. Recall from our partitioning technique that all the candidates with the same last item and the same total number of items are processed in parallel. For example, ACD, BCD and ABD. We use a prefix tree with a header to summarize the underlying data in the very same way as FP-trees do. Hence, for computing the support of ACD, we traverse the list of nodes associated to item D (the last one of the itemset). Each time we check whether the path from the root of the tree to the node contains ACD. It if is the case, the counter value of the node is added to the support of ACD. Note that the same list is traversed for BCD and ABD. Hence, if we have three different threads processing each of these three candidates, then we have good chances that one of them will bring the data (nodes of the prefix tree) from main memory to the different levels of the cache then to the register but the other threads could find it without accessing main memory making their computation much faster. To favor this optimization, we sorted the candidates wrt their last item. Moreover, we used an array structure to code the branches of the tree because dynamic node allocation does not guarantee space locality between successive nodes. We have been inspired by the data structures first proposed in [35] and further optimized in [61].

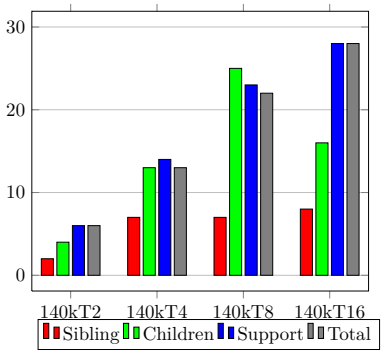
Even if the Kosarak data set is relatively large, we note that the proportion of time devoted to support computation is comparable to the time spent in covering tests when the minimal support σ is set to $1K$ (Figure 2.7(e)). This is not the case when $\sigma = 2k$. We note however that the speedup for support computation is in both cases almost perfect(Figures 2.7(g) and 2.7(h)). Finally, we should notice that the average size of transactions is quite short (8 items). Thus, the maximal frequent



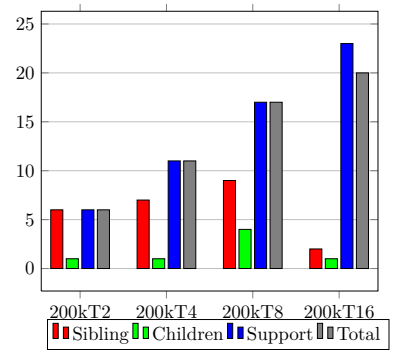
(a) Webdocs : 140K-Execution time distribution



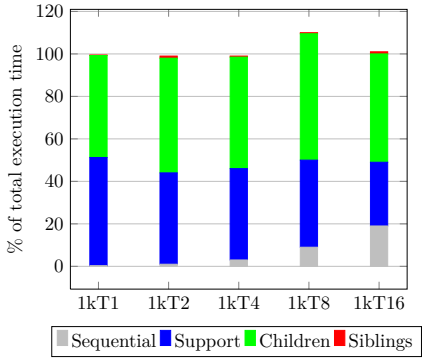
(b) Webdocs : 200K-Execution time distribution



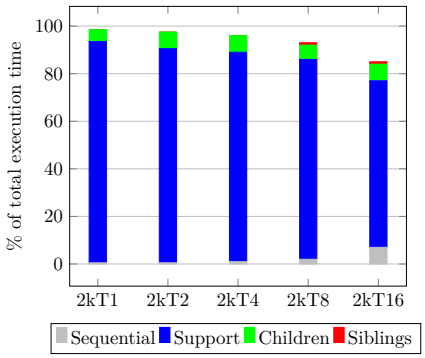
(c) Webdocs : 140K-Speed up



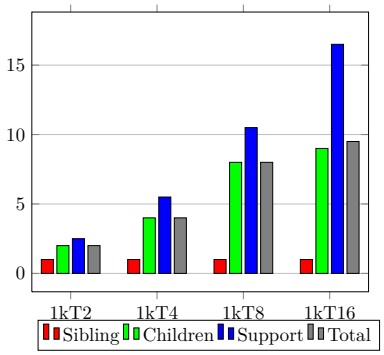
(d) Webdocs : 200K-Speed up



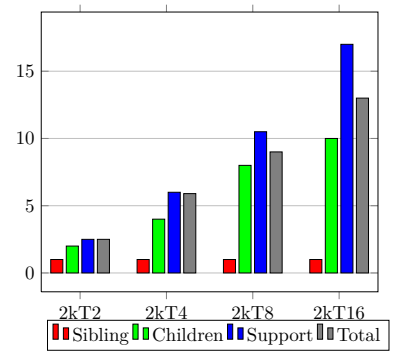
(e) Kosarak : 1K-Execution time distribution



(f) Kosarak : 2K-Execution time distribution



(g) Kosarak : 1K-Speed up



(h) Kosarak : 2K-Speed up

Figure 2.7 – Large Data : Webdocs and Kosarak

itemsets tend to be short as well. This reduces the impact of downward pruning exploited by DFS algorithms.

Medium Size Data

The dataset 40I10D100K of medium size is denser than T10I4D100K. Figures 2.8(a) and 2.8(b) show that support computation time is important. Nevertheless, we do not reach the same speed ups as those with Webdocs. We note however that the total execution time is always divided by almost the number of threads.

The average length of the transactions in T10I4D100K data set is 10. Hence, maximal frequent itemsets are rapidly reached when DFS is used (note that in Kosarak, this average length is even lower). This makes support computation not very time consuming, even when the minimal support is set to 20 thanks to the summarization capacity of the FP-trees.

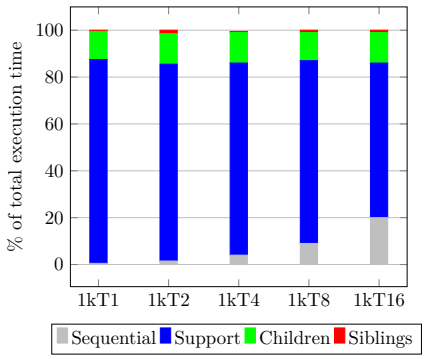
Small Data

Both these Chess and Mushroom data sets are small. Thus, it is not surprising to find that support computation is negligible w.r.t. candidates management (see Figure 2.9). With Chess, we note a bad speed up when $\sigma = 800$. These last two experiments tend to confirm that our proposed algorithm is rather tailored towards situations when support computation is the bottleneck of the execution time. Moreover, we note for Chess data set that the number of MFI's grows rapidly even when the support threshold id high. For example, when $\sigma = 800$ (a minimal frequency of 25%) the number of MFI's is more than 26000 (more than six times the number of original transactions). This makes coverage test harder than support computation. For Mushroom data set, we have to lower the support threshold to around 100 (frequency $\sim 1\%$) to make the number of MFI's larger than the number of transactions. Nevertheless, it turns that coverage test takes much larger time than support test. The explanation for this is the "non uniform" behavior of the MFI's length. For example, when σ is set to 10%, the longest MFI has 16 items while there is no MFI of length 13 or 14. This phenomenon has been noticed in [30].

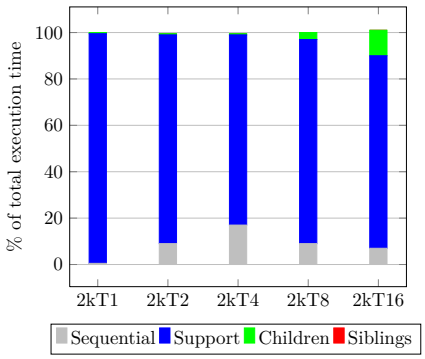
2.7.5 MineWithRounds vs PADS

As a final experiment, we present the execution time of **PADS** [101] which is, to our best knowledge, the most efficient implementation of a sequential MFI mining algorithm and compare it with that of **MineWithRounds**. **PADS** is not a *pure* **DFS** implementation in that it uses several heuristics to reduce computation time, e.g., each time a candidate is found frequent, all its right parents are evaluated and reordered w.r.t their increasing support. Hence, the $\prec$ order is dynamically and continuously modified. Figure 2.10 shows the execution times w.r.t. support thresholds. Both implementations have been executed on the same machine as before. The time needed to load the data and to construct the FP-trees is not taken into account. One should however notice that our execution time is measured when 16 threads are executed in parallel. It took about 2 minutes for both implementations to start the mining procedure. When the minimal support is less than or equal to 140000 PADS had a segmentation fault. To be complete, we should mention that our implementation is naive in that we used the C++ STL library which makes it very easy to program prototypes but if we want to fine tune the optimizations, as in PADS, one should program his/her own data structures [2].

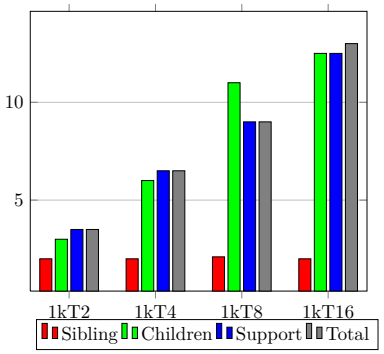
2. Our source code contains about 600 lines.



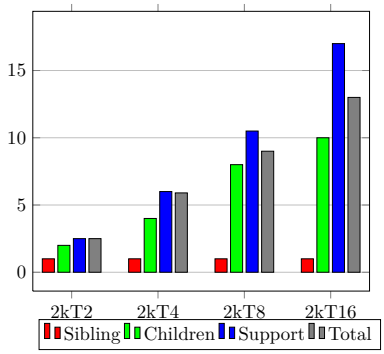
(a) T40 : 1K-Execution time distribution



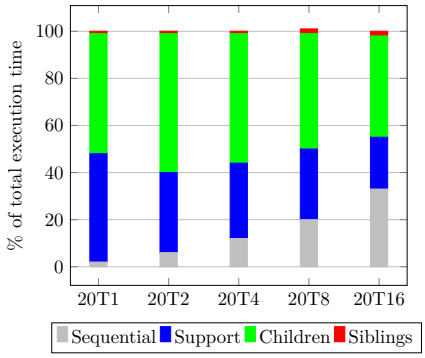
(b) T40 : 2K-Execution time distribution



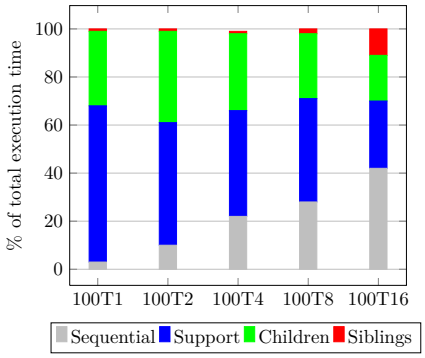
(c) T40 : 1K-Speed up



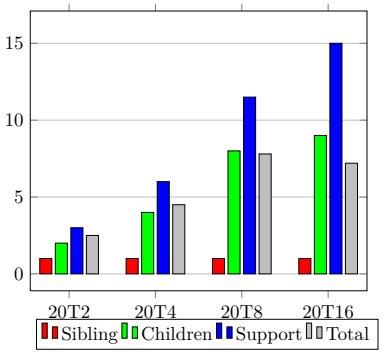
(d) T40 : 2K-Speed up



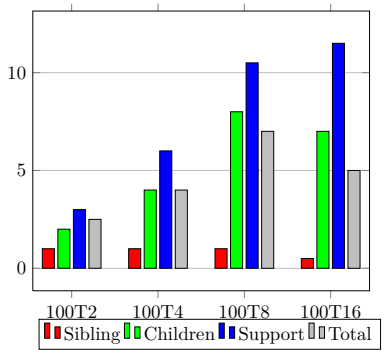
(e) T10 : 20-Execution time distribution



(f) T10 : 100-Execution time distribution

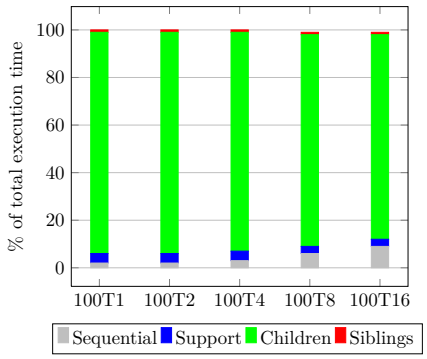


(g) T10 : 20-Speed up

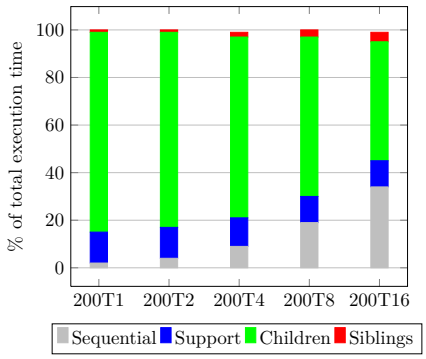


(h) T10 : 100-Speed up

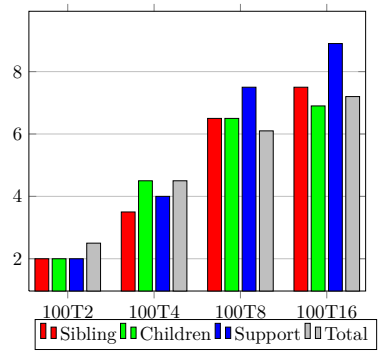
Figure 2.8 – Medium data sets : T40 and T10



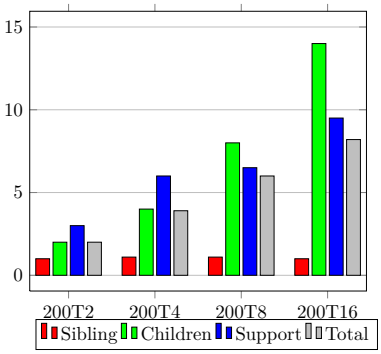
(a) Mushroom : 100-Execution time distribution



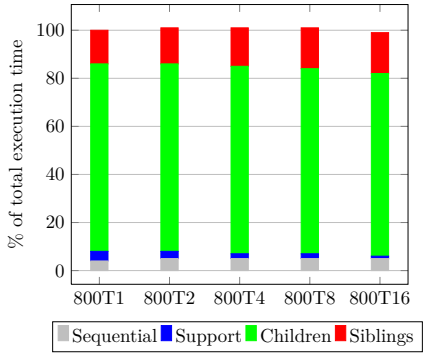
(b) Mushroom : 200-Execution time distribution



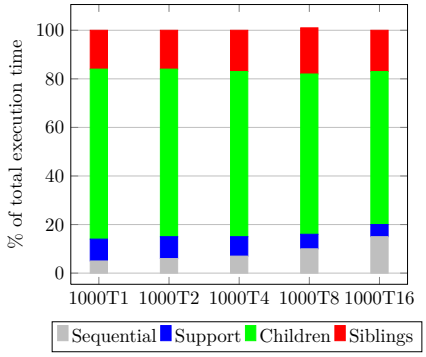
(c) Mushroom : 100-Speed up



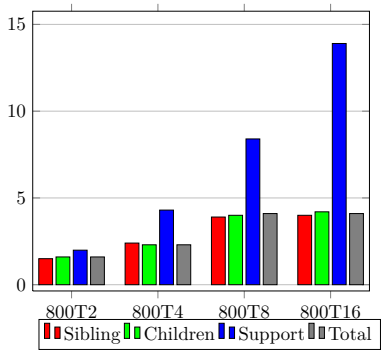
(d) Mushroom : 200-Speed up



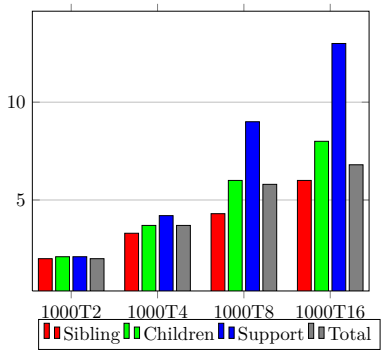
(e) Chess : 800-Execution time distribution



(f) Chess : 1000-Execution time distribution



(g) Chess : 800-Speed up



(h) Chess : 1000-Speed up

Figure 2.9 – Small Data : Mushroom et Chess

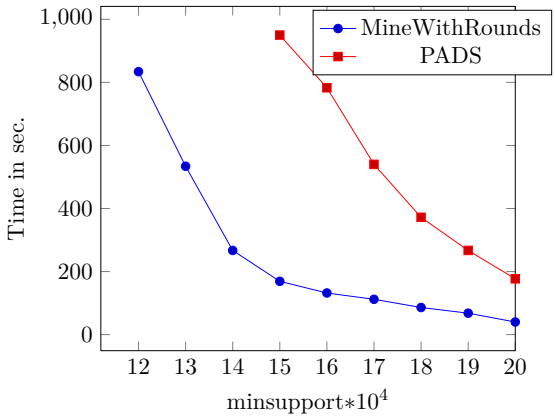


Figure 2.10 – Execution times for PADS and **MineWithRounds** with Webdocs

2.8 Concluding Remarks on MFI's Computation

We presented **MineWithRounds**, a parallel algorithm for computing maximal frequent itemsets. It mimics **DFS** in that it tests the support of exactly the same candidates. The theoretically proved speed up of our algorithm regarding the execution time devoted to supports computation is confirmed by the extensive experiments we conducted. With large datasets, this speed up is even better than what was expected thanks to cache management in modern machines.

With the generalization of multi-core CPU's and their cache management, some proposals fine tuned the previous techniques so that data caching and pre-fetching become effective [35]. An efficient parallel algorithm for building the FP-trees has already been proposed by [61] but the mining process has been made data parallel not task parallel. Hence, no guarantee that the parallel process will be faster than the sequential version. With our present work, we make a step further in exploiting modern multi-core architectures. Our experiments show that without necessarily using very sophisticated optimization techniques, we are able to have execution times much better than state of the art implementations.

Parallelizing algorithms that extract other kinds of patterns e.g. sequences, trees or graphs is a natural extension of the present work, e.g. [96] use a *canonical* order for mining frequent graphs. By defining the parent/children/sibling relationships as analogously as we did for itemsets, then for extracting maximal frequent graphs, it *suffices* to use the fundamental idea behind our algorithm, i.e., start the evaluation of a subgraph as soon as all its left parents have been evaluated. We leave the development of this issue to future work.

Finally, we should mention that even if our implementation assumes a shared memory architecture, we showed in Section 2.6 how it could be adapted to distributed data setting.

In the remaining part of this chapter, we shall turn our focus towards dependency mining.

2.9 Parallel Mining of Dependencies

The problem of extracting functional dependencies (FDs) from databases has a long history dating back to the 90's. Still, efficient solutions taking into account both material evolution and the amount of data that are to be mined, are still needed. We propose a parallel algorithm which, upon small modifications, extracts (i) the minimal keys, (ii) the minimal exact FDs, (iii) the minimal approximate FDs and (iv) the Conditional functional dependencies (CFDs) holding in a relational table. Under some natural conditions, we prove a theoretical speed up of our solution with respect

to a baseline algorithm which follows a depth first search strategy. Since mining most of these dependencies require a procedure for computing the *number of distinct values* (NDV) which is a space consuming operation, we show how sketching techniques for estimating the exact value of NDV can be used for reducing both memory consumption as well as communications overhead when considering distributed data while guaranteeing a certain quality of the result. Our solution is implemented and some experimental results are reported here showing the efficiency and scalability of our proposal.

2.10 Related Work w.r.t Mining Functional Dependencies

Several algorithms have been proposed to find the minimal set of FDs from a relation. These algorithms can be classified along three criteria : (i) the way they traverse the search space, *i.e.,* breadth or depth first, (ii) pre-computation and (iii) incremental computation. TANE [52], FUN [75] and FD_Mine [99] use a levelwise strategy to explore the candidates lattice. They start by constructing a partition of the tuples for each attribute (two tuples belong to the same part with respect to some attribute A iff they share the same value A's value) then they build new partitions from already constructed ones, *i.e.,* they perform the partition product. If $\mathcal{P}_X(T)$ denotes the partition of relation T w.r.t. X then T satisfies $X \rightarrow A$ iff $|\mathcal{P}_X(T)| = |\mathcal{P}_{XA}(T)|$ where $|\mathcal{P}_X(T)|$, resp. $|\mathcal{P}_{XA}(T)|$, denotes the number of parts there are in $\mathcal{P}_X(T)$, resp. $\mathcal{P}_{XA}(T)$. For example, suppose that $\mathcal{P}_A(T), \mathcal{P}_B(T)$ and $\mathcal{P}_C(T)$ are computed. Then verifying whether $AB \rightarrow C$ consists in combining [3] $\mathcal{P}_A(T)$ and $\mathcal{P}_B(T)$ to get $\mathcal{P}_{AB}(T)$ and then this result is *combined* with $\mathcal{P}_C(T)$ to get $\mathcal{P}_{ABC}(T)$. The main difference between these algorithms resides in the pruning strategies they use. The three algorithms are incremental in the sense that the computation of the partitions at some level is made from the computed partitions of the previous one. Clearly, when data and/or the left hand side of the FDs are large, *i.e.,* large number of candidates by level, the memory consumption becomes a sever bottleneck. In our experiments, this actually happened, i.e., the memory was saturated, even with moderate size data and not so excessive number of attributes (40). Dep-Miner [67] uses others concepts. It first computes the *agree sets* for each pair of tuples, that is the set of attributes for which they share the same values. Clearly, if $ag(t_1, t_2) = X$ then $T \not\models X \rightarrow A$ for each $A \in \mathcal{A} \setminus X$. After that, Dep-Miner computes maximal difference sets (*i.e.,* complements of agree sets) to build an hypergraph for each fixed target attribute and seeks their minimal transversals. Intuitively, this turns to compute minimal exact FDs from maximal incorrect FDs. The authors show that this method outperforms TANE. FastFD [94] uses the same technique as Dep-Miner but follows a depth first strategy when traversing the search space. The principal drawback of Dep-Miner and FastFDs is their pre-computation phase whose complexity is $O(|T|^2)$ which is prohibitive when T is large.

On another side, traversing the search space in breadth first reduces the pruning possibilities. Indeed, when a functional $X \rightarrow A$ is discovered, there is no need to test the supersets of X, we know a priori that they determine A, so they can be pruned. In the same time, if $X \rightarrow A$ doesn't hold, then no need to test the subsets of X since they cannot determine A, so they are pruned. Breadth first traversal can utilize only one way pruning : upward or downward. One advantage however of BFS is that it can be naturally parallelized : all candidates of the same level are processed in parallel.

Our aim is to combine the advantages of each approach, *i.e.,* use DFS in order to prune candidates both upward and downward, do not do any pre-computation and use parallelism. For this purpose, we adapt **MineWithRounds** algorithm. In fact, that algorithm can directly be used to mine the set of maximal FDs that "are not satisfied". The set of minimal FDs that are satisfied is actually the dual of the former set. It can be obtained by applying algorithms devoted to the

3. Technically, we talk about partition product.

minimal transversals (or hitting sets) of a hypergraph computation. Note that the exact computation complexity of that problem is still open [26, 42]. The adaptation we make in the present work leads however to a direct computation of the minimal FDs without relying to transversals computation.

While the set of minimal keys of a table can be derived from the functional dependencies that are satisfied by a relation, thus one may use the result of the previous algorithms described above, there have been specific algorithms for directly mining these keys because the problem is actually easier than mining all FDs. For example, [1, 88] adopted levelwise traversal of the search space and thus cannot completely benefit from the pruning power of DFS. A more recent reference [51] proposes random walk strategy over the search space. The main idea consists in launching in parallel several threads. Each of which will discover a key. The discovered keys are not necessarily distinct nor minimal. From these discovered keys, minimal transversals are computed and are considered as the starting point of the next parallel iteration. The minimal transversals represent the potential maximal non keys that can be derived for the knowledge obtained so far. The process stops when no new possible candidate can be generated, *i.e.,* the potential maximal non keys are the exact ones. This procedure is very similar to the one proposed in [41]. For accelerating the computation, the authors make use of the partitions product technique. We should note that even if the implementation they propose is built on Hadoop framework, they do not consider distributed data since they consider that data are totally replicated in every node. As we will see, this simplifies very much the parallel computation but cannot handle large data sets.

Conditional FDs (or CFDs) [13] are a generalization FDs in that they do hold only in a horizontal portion of the underlying data. Actually, they are a solution for handling approximate FDs, *e.g.,* if $AB \rightarrow C$ does not hold in T, it may happen that $\sigma_{A=a}(T) \models AB \rightarrow C$. All algorithms for mining CFDs proposed so far are essentially adaptations of those targeting FDs. In [28], the authors present CFDMiner which finds only *constant* CFDs, *i.e.,* all the attributes in the left hand side of the FDs are constrained to be equal to some constants. They also propose CTANE and FastCFD for general CFDs. The three algorithms are extensions of FD_Mine, TANE and FastFDs respectively. In [24], CFUN, an extension of FUN is proposed for mining constant CFDs. All these extensions suffer form the same drawbacks as their antecedents, i.e., BFS strategy, memory consumption and pre-computation costs.

For a more detailed state of the art on dependency mining, we refer the reader to the recent survey [65].

2.11 Basic Definition w.r.t Dependencies

2.11.1 Functional dependencies

Exact functional dependencies : let $T(A_1, \ldots, A_n)$ be a relation. $\mathcal{A}(T)$ denotes its set of attributes, i.e., $\mathcal{A}(T) = \{A_1, \ldots, A_n\}$. We shall simply use $\mathcal{A}$ when T is understood from the context. Let $X, Y \subseteq \mathcal{A}$, then T satisfies the FD $X \rightarrow Y$ (noted $T \models X \rightarrow Y$) iff for each $t_1, t_2 \in T$, $t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]$. Equivalently, let π denote the projection operation of the relational algebra, then $T \models X \rightarrow Y$ iff $|X| = |XY|$ where $|X|$ denotes the cardinality of $\pi_X(T)$ [4].

Approximate functional dependencies. Functional dependencies can be extended to different definitions of approximations as follows. We define a directed bipartite *dependency graph* $G_{X,Y} = (V, E, w)$. This graph maps $\pi_X(T)$ to $\pi_Y(T)$ and the edges are weighted by a function $w : E \rightarrow \mathbb{N}$. More precisely :

– $V = \pi_X(T) \cup \pi_Y(T)$ is the set the vertices ;

4. Note that set semantics is considered here, hence duplicates are automatically eliminated.

- $E = \{(x,y) | \exists t$ such that $t[X] = x$ and $t[Y] = y\}$ is set of edges ;
- $w(x,y) = |\{t | t[X] = x$ and $t[Y] = y\}|$, that is the number of occurences of value (x,y) in T. By extension, $w(x) = \sum_y w(x,y)$.

Several known measures $\mathcal{M}$ of approximate functional dependencies validity can be rewritten using the dependency graph. $\mathcal{M}$ associates a real value between 0 and 1 to the dependency graph. Informally, 1 stands for exact functional dependencies. The choice of $\mathcal{M}$ definition can be driven by the context. For instance, one definition that has been used is that of *strength* $S(G_{X,Y}) = \frac{|\pi_X(T)|}{|E|}$, the *confidence* $C(G_{X,Y}) = \sum_{x \in \pi_X(T)} \max_y \frac{w(x,y)}{w(x)}$. Likewise, the g_1, g_2 and g_3 measures of [54] can also be expressed wrt to $G_{X,Y}$.

$X \rightarrow Y$ is a $(\mathcal{M}, \alpha)$-FD if and only if $\mathcal{M}(G_{X,Y}) \geq \alpha$. In the following, we mainly focus on monotonic measures. $\mathcal{M}$ is *monotonic* if and only for every X, X' and Y subsets of $\mathcal{A}$, we have $\mathcal{M}(G_{X,Y}) \leq \mathcal{M}(G_{X \cup X',Y})$. For example, the confidence is monotonic while the strength is not.

The monotony of exact FDs is a classical result of relational database theory (see [71]), i.e., $T \models X \rightarrow Y$ and $X \subseteq Z$ then $T \models Z \rightarrow Y$. Conversely, if $T \not\models Z \rightarrow Y$ and $X \subseteq Z$ then $T \not\models X \rightarrow Y$. Moreover, $\forall X, Y, Z \subseteq \mathcal{A}$, T satisfies $X \rightarrow Y$ and $X \rightarrow Z$ iff T satisfies $X \rightarrow YZ$ where YZ denotes $Y \cup Z$. This second property shows that it is sufficient to consider the FDs with just one attribute in their right hand side to recover all the FDs satisfied by T. So, from here on, we consider only FDs with one attribute in the right hand side. Let $F = X \rightarrow A$ be satisfied by T, then F is *minimal* iff $\forall X' \subset X$, $T \not\models X' \rightarrow A$. F is trivially satisfied if $X \ni A$. $LHS(F) = X$ denotes the left hand side of F while $RHS(F) = A$ denotes the attribute in the right hand side.

2.11.2 Keys

X is *a key* of T iff $\forall A_i \in \mathcal{A}$, $T \models X \rightarrow A_i$. X is a *minimal key* if it does not contain a key. Clearly, X is a key of T iff $|X| = |\mathcal{A}|$. Therefore, if T is a relation (without duplicates) then X is a key iff $|X|$ is equal to the number of tuples in T [5]

2.11.3 Conditional Functional Dependencies

A CFD is a functional dependency that is satisfied by an horizontal part of T resulting from the application of a selection operation whose condition is just a conjunction of equalities. More formally, a CFD is a pair $(F|P)$ where F is a functional dependency and P is a tuple $\langle a_{j_1}, \ldots, a_{j_m} \rangle$ where a_{j_k} is either (i) a constant in the domain $dom(A_j)$ of A_j provided that $A_j \in LHS(F)$ or (ii) an unnamed constant $'\_'$. For example, $(ACE \rightarrow D | a_1, \_, e_2)$ is a CFD. Let $\phi = (F|P)$ be a CFD, then $Cond(\phi) = \bigwedge_{A_{j_k} = a_{j_k}}$ where $a_{j_k} \neq \_$. For the example above, $Cond(\phi)$ is the condition $A = a_1 \wedge E = e_2$. A CFD ϕ is satisfied by T, noted $T \models \phi$ iff F is satisfied by $\sigma_{Cond(\phi)}(T)$ where σ is the selection operation of the relational algebra. Just like FDs, we define the minimality of CFDs. Let $\phi = (F|P)$ and $\phi' = (F'|P')$ be two CFDs where $RHS(\phi) = RHS(\phi')$. Then ϕ is *more general* than ϕ' iff $LHS(F) \subseteq LHS(F')$ and $Cond(\phi) \subseteq Cond(\phi')$ [6]. Let $T \models \phi$ then ϕ is *minimal* iff there exists no ϕ' such that $T \models \phi'$ and ϕ' is more general than ϕ. In this case, we note $\phi' \sqsubset \phi$.

The following example illustrates the different concepts introduced so far.

Example 7. *Table T in Figure 2.11 will be used as our running example throughout the next sections. $\phi_1 = (D \rightarrow C)$ is satisfied by T because $|D| = |CD| = 3$. $\phi_2 = (AD \rightarrow C)$ is also satisfied but is not minimal. A is a minimal key of T since $|A| = 4 = |ABCD|$. $ABCD$ is also a key but it*

5. Some recent works use the term of *uniques* to designate sets of attributes that form a key, e.g., [50].
6. For notation convenience, we consider here that $Cond(\phi)$ is a set of atomic equalities.

A	B	C	D
a_1	b_1	c_1	d_1
a_2	b_1	c_2	d_2
a_3	b_2	c_2	d_2
a_4	b_2	c_2	d_3

Figure 2.11 – Table T.

is not minimal. $\theta_1 = (D \rightarrow C|\_)$ *is a CFD satisfied by* T*. It is actually equivalent to* ϕ_1*.* $T \models \theta_2 = (AD \rightarrow C|a_1, \_)$ *and* θ_1 *is more general than* θ_2*.*

2.11.4 Problems statement

We address the following problems. Given a table T

1. find all the minimal keys of T,
2. find all the minimal FDs satisfied by T (the canonical set of FD's) and
3. find all the minimal CFDs satisfied by T.

In the rest of the chapter, we will show that with almost the same parallel algorithm, we can solve each of the above problems. We start with the simplest one, i.e., minimal keys extraction.

2.12 Mining Minimal Keys

As we have seen before, X is a key iff $|X| = |T|$. This property is monotone in that $|X| \neq |T| \Rightarrow |X'| \neq |T| \forall X' \subset X$. Hence, one can use a levelwise algorithm *à la* Apriori [6] in order to compute all the minimal keys. The drawback with this method is that before discovering that X is a key, we have to test all the subsets of X which may become a large number of candidates when $||X||$ is large ($||X||$ denotes the number of attributes of X). Hence, we shall consider a depth first search algorithm (DFS) in order to avoid to check systematically all the subsets of X. Recall that the search space is the lattice $(2^{\mathcal{A}}, \subset)$ where $2^{\mathcal{A}}$ is the set of subsets of $\mathcal{A}$. By considering some total order $\lhd$ on the attributes, we may define a lexicographic order on sets of attributes. These ones are assumed to be sorted following $\lhd$. Hence, we define the search tree that is traversed by DFS.

Example 8. *The search tree of our running example is given in Figure 2.12.*

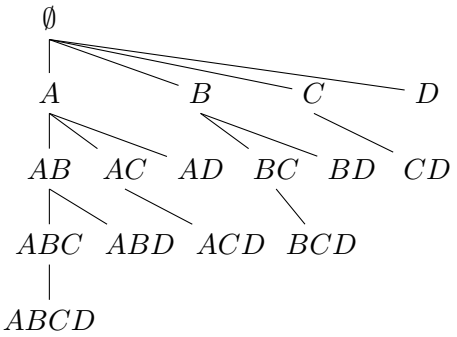


Figure 2.12 – Search tree $\mathcal{T}$ of our running example.

SeqKeys (cf. Procedure SeqKeys) is a sequential algorithm for mining keys respecting a depth first strategy. Let's assume that the candidates are encoded by a bit vector $V(1\ldots n)$ where $V[j]=1$ means that attribute A_j is present and $V[j]=0$ otherwise. $\mathbf{\Gamma}^+$ and $\mathbf{\Gamma}^-$ denote respectively the set of smallest keys and the largest non keys found during DFS execution. Integer i is a position index in the vector V. The algorithm is called by initializing i to 1, $\mathbf{\Gamma}^+$ and $\mathbf{\Gamma}^-$ as empty and all positions of V are set to 0. At the end of the execution, $\mathbf{\Gamma}^+$ contains all the minimal keys. The coverage test in line 3 consists simply in verifying whether $\mathbf{\Gamma}^-$ contains an element which is a superset of V. If it is the case, the current candidate cannot be a key because it is included in a non key set of attributes. The function $SizeOf$ mentioned in line 7 consists in projecting T onto the attributes set encoded by V and it returns the size (number of tuples) of this projection. If the candidate is a key (line 7), it is added to $\mathbf{\Gamma}^+$ and all its supersets are removed from $\mathbf{\Gamma}^+$. Otherwise, the candidate is added to $\mathbf{\Gamma}^-$ and all its subsets are removed from $\mathbf{\Gamma}^-$. At the end (lines 16 & 17), the procedure will consider the sibling of the current candidate. As one may notice, this procedure is almost the same as the **DFS** procedure we used for MFI's.

Procedure SeqKeys(integer i)

```
   Input: integer i, |T|
   Output: Γ+
1  if i ≤ n then
2      V[i] ← 1;
3      if Γ− covers V then
4          // V is not a key
5          SeqKeys(i+1);
6      else
7          if sizeOf(V) = |T| then
8              // V is a key (maybe not minimal)
9              Add V to Γ+;
10             Remove from Γ+ the supersets of V;
11         else
12             // V is not a key
13             Add V to Γ−;
14             Remove from Γ− the subsets of V;
15             SeqKeys(i+1);
16     V[i] ← 0;
17     SeqKeys(i+1);
```

Example 9. *Applying algorithm* **SeqKeys** *to our running example will start with A which is found to be a key. Then the algorithm continues with B (not a key), then BC not a key then BCD which is a key. At this stage,* $\mathbf{\Gamma}^+=\{A,BCD\}$ *and* $\mathbf{\Gamma}^-=\{BC\}$*. The algorithm will then test BD which is a key, hence it is added to* $\mathbf{\Gamma}^+$ *and BCD is removed from there in order to guarantee the minimality of the returned keys. The next candidate is C and is found covered by is* $\mathbf{\Gamma}^-$*) hence it is not tested and CD is generated. This later is found not a key so added to* $\mathbf{\Gamma}^-$ *. The final candidate is D which is covered by* $\mathbf{\Gamma}^-$ *thus no need to test it : it is not a key. The returned result is* $\mathbf{\Gamma}^+=\{A,BD\}$*.*

The following proposition states the correctness of **SeqKeys**.

Proposition 1. *At the end of the execution of* **SeqKeys**, $\mathbf{\Gamma}^+$ *contains all and only the minimal keys of* T.

Of course, it is possible to rewrite this algorithm in order to reduce the coverage tests. This however won't reduce the $sizeOf$ calculations. We consider that this last operation is more expensive than the former.

Inspired by **MineWithRounds** algorithm and the concept of depth first partitions, we provide a parallel version of **SeqKeys** which is described in Algorithm 2. The only difference with **MineWithRounds** is that the dimensions we consider represent attributes instead of items and the interestingness measure is the fact that the candidate is a key instead of the fact that the itemset is frequent.

Algorithm 2: ParaKeys

```
   Input: Table T
1  P_1 ← A_1;
2  r ← 1 ;
3  while r ≤ 2n − 1 do
4      foreach X ∈ P_r do
5          \* Loop in parallel *\;
6          if X is not covered by Γ⁻ then
7              if X is a key then
8                  Γ⁺ ← Γ⁺ ∪ {X};
9                  Γ⁺ ← Γ⁺ \ {Y | Y ⊃ X};
10                 P_{r+1} ← P_{r+1} ∪ RightP(X);
11                 P_{r+2} ← P_{r+2} ∪ Sibling(X);
12             else
13                 Γ⁻ ← Γ⁻ ∪ {X)};
14                 Γ⁻ ← Γ⁻ \ {Y | Y ⊂ X};
15                 P_{r+1} ← R_{r+1} ∪ Child(X);
16     r ← r + 1;
17 Return Γ⁺;
```

The following proposition shows that **ParaKeys** and **SeqKeys** check exactly the same candidates. In other words, they perform the same number of costly operations.

Proposition 2. *Given a table* T. **SeqKeys** *and* **ParaKeys** *perform the same number of keys checks.*

The above proposition not only shows the correctness of **ParaKeys** but also the speed-up w.r.t **SeqKeys** we could expect from it. The following theorem formalizes this result.

Theorem 4. *Let* $T_s = T_s^c + T_s^o$ *be the execution time of* **SeqKeys** *where* T^c *denotes the time devoted for keys checking and* T^o *is the remaining execution time. Likewise, let* $T_p = T_p^c + T_p^o$ *be the execution of* **ParaKeys**. *Let* p *be the number of available processors,* k *be the number of checked candidates by either algorithms,* t *be the time for checking whether a single candidate is a key and* n *be the number of attributes. Then*

$$\left\lfloor \frac{k}{p} \right\rfloor * t \leq \underline{T_p^c} \leq \left\lceil \frac{k}{p} \right\rceil * t + (2n-1) * t$$

Before going further, one should note that in many cases when the size of the mined table is large, T_s^o and T_p^o are respectively negligible compared to T_s and T_p. Indeed, checking whether X is a key becomes less expensive than checking the coverage of the candidate means that the number of non dependencies is larger than the table size. Even if this case is possible in theory, we argue that in practice, this situation is encountered quite rarely. Therefore, a guarantee on the speed up about keys checking operation is in many practical cases a guarantee about the total execution time. We will see that in the experiments section.

2.13 Mining Functional Dependencies

The parallel algorithm **ParaDe** we propose for this purpose follows the same lines as **ParaKeys**. Indeed, it suffices to iterate over each attribute A_i by considering it as the current target attribute, i.e., seek the minimal dependencies of the form $X \rightarrow A_i$. The search space for A_i is $2^{\mathcal{A}\setminus\{A_i\}}$. Without detailing further the process, we just give the algorithm and stress the fact that a small modification of **ParaKeys** leads to an algorithm for mining all the minimal FDs. We also note that a small reformulation of Theorem4 carries over to the present context. The following algorithm discovers the minimal exact FDs as well as the approximate ones.

Algorithm 3: ParaDe

Input: Table T, target attribute A_i, Approximation measure $\mathcal{M}$, threshold α
Output: $\mathbf{\Gamma}^+$= set of the minimal FDs $X \rightarrow A_i$ s.t $\mathcal{M}(X \rightarrow A_i) \geq \alpha$

```
1  P_1 ← A_1;
2  r ← 1 ;
3  while r ≤ 2(n−1)−1 do
4      foreach X ∈ P_r do
5          \* Loop in parallel *\;
6          if X is not covered by Γ⁻ then
7              if T ⊨ (X → A_i) then
8                  Γ⁺ ← Γ⁺ ∪ {X};
9                  Γ⁺ ← Γ⁺ \ {Y | Y ⊃ X};
10                 P_{r+1} ← P_{r+1} ∪ RightP(X);
11                 P_{r+2} ← P_{r+2} ∪ Sibling(X);
12             else
13                 Γ⁻ ← Γ⁻ ∪ {X)};
14                 Γ⁻ ← Γ⁻ \ {Y | Y ⊂ X};
15                 P_{r+1} ← P_{r+1} ∪ Child(X);
16     r ← r+1;
17 output Γ⁺;
```

As one may see, the only difference between **ParaKeys** and **ParaDe** resides in the interestingness of the patterns (Lines 7 and 6 respectively) as well as the search space under consideration ($2^{\mathcal{A}}$ and $2^{\mathcal{A}\setminus A_i}$ respectively).

Finally, the above algorithm can be used to mine all the FDs by iterating over the target attributes A_i.

2.13.1 Distinct Values Approximation

One way to proceed for mining exact keys as well as exact minimal FDs consists in counting the number of distinct elements belonging to $\pi_X(T)$ (is X a key ?) and $\pi_{XA_i}(T)$ (does $X \rightarrow A_i$ hold ?). Counting distinct values can be done either by sorting or hashing. Unless using multiple copies of table T which is very space consuming, sorting is incompatible with parallelism because we cannot sort T with respect to X and in the same time with X' when X and X' are both candidates to be examined in parallel. On the other hand, hashing uses $O(|T|)$ and $\Theta(|X|)$ of memory space. When T is large and the number of available processors is also large, the memory usage may become a bottleneck. Hence, sketch techniques could be used in order to reduce this memory usage by scarifying the correctness of the computation. One such technique is to use an (ϵ, δ)-approximation estimator, which given any positive $\epsilon < 1$ and $\delta < 1$, returns an estimate X^* of relative error less than ϵ with probability at least $1-\delta$. For instance Hyperloglog [29] (HLL) needs only $O(\log|T| + \frac{\log\log|T|}{\epsilon^2})$ bits of memory to compute an $(\epsilon, \epsilon^{-2})$-estimator. Another technique based on Bloom filters has been proposed in [79].

Let us recall HLL main principles : take as input values of ϵ (the required error margin). HLL first compute a value b w.r.t. ϵ (e.g., $b = 10$ when $\epsilon = 5\%$). From b, HLL constructs a vector of integers M of size 2^b. Intuitively, for estimating the value of $|X|$ HLL works as follows :

1. every tuple $t[X]$ is hashed to a number $h(t[X])$. We thus obtain a pseudo-random sequence of 0s and 1s.
2. The first b bits serve to find which $M[i]$ concerns $h(t[X])$. $M[i]$ contains the position of the *largest left most 1* in the binary code of all $t[X]$'s that when hashed, they have the same first b bits.
3. Once all the tuples are read and hashed, a function f is applied to M and returns a cardinality estimate.

The error guarantee of HLL is relative in that the returned value is in the interval $[|X|*(1-\epsilon), |X|*(1+\epsilon)]$ with a high probability (typically, more than 95%). To make things more concrete, for $\epsilon = 3\%$, the value $b = 11$. Hence, HLL needs a memory space of 2^{11} for estimating quite accurately a cardinality up to 10^9 within a 3% error margin and with a probability greater than 95% [7].

Let $X \rightarrow Y$ be a candidate FD, ϵ be the accuracy of HLL, $\widehat{|X|}$ and $\widehat{|XY|}$ be the cardinality estimations of respectively $|X|$ and $|XY|$. Then it is easy to see that $\left(\frac{1-\epsilon}{1+\epsilon}\right) * \frac{|X|}{|XY|} \leq \frac{\widehat{|X|}}{\widehat{|XY|}} \leq \left(\frac{1+\epsilon}{1-\epsilon}\right) * \frac{|X|}{|XY|}$. If we consider the lower bound of the interval as the condition under which $X \rightarrow Y$ is considered as valid, i.e., $\frac{\widehat{|X|}}{\widehat{|XY|}} \geq \frac{1-\epsilon}{1+\epsilon}$ then at worst, $\frac{|X|}{|XY|} = \frac{1-\epsilon}{1+\epsilon}$. Recall that $\frac{|X|}{|XY|}$ is an approximation measure of FDs which is called the *strength*. It has been proven (e.g., [77]) that $strength(X \rightarrow Y) \leq Confidence(X \rightarrow Y) \leq 1$. As long as $\frac{1-\epsilon}{1+\epsilon}$ is close to 1, we would accept, at worst, an FD which is *almost* satisfied, i.e. its confidence is even closer to 1. For example, for $\epsilon = 3\%$, $X \rightarrow Y$ is accepted if $confidence(X \rightarrow Y) \geq 94\%$.

One interesting property with HLL is that it is *associative*. More precisely, suppose that T is partitioned into $T_1, \ldots, T_k$ parts. Let M_k be the vector used by HLL to estimate the distinct values in T_k. Then, to estimate the distinct values in T, it suffices to consider M where $M[i] = \max_{j=1}^{k} M_j[i]$ and apply $f(M)$ to get the estimate. This is very appealing when data is distributed. In fact, this property can be used in a MapReduce implementation. Indeed, suppose that $\mathcal{C}$ is a set of candidates X for which we want to estimate $|X|$ and let T be partitioned into $T_1, \ldots, T_k$ horizontal chunks each of which is stored in a separate machine. The first step of the algorithm consists in mapping every

7. A very interesting and intuitive explanation of this method can be found at `http://research.neustar.biz/2012/10/25/`

candidate X to a vector M containing only 0's. The (X, M) pairs are then broadcast to the k workers machines. Every machine i will locally update the pairs and return (M_i, X). The reducer groups the pairs sharing the same X and combine the M_i's by computing the Max value for each position j, i.e., $M[j] = \max_i M_i[j]$. Once M is obtained, the estimate of $|X|$ is obtained by computing $f(M)$. The returned result is exactly the same as the one we would obtain of the data were centralized. Note that doing so the total amount of data transfers for estimating $|X|$ is the cost of broadcasting (M_i, X) to every worker and every worker returning its local result (M_i, X). Comparatively to the exact computation of $|X|$ which needs to transfer all the $T_i[X]$'s, we may conclude that we have a huge gain in data transfer cost when T is large.

2.14 Mining Conditional Functional Dependencies

This problem of extracting CFDs is harder than that of FDs essentially because the search space is much larger. Indeed, if $q = |dom(A_i)|$ then each $X \in 2^{\mathcal{A}}$ gives rise to 2^q elements in the new search space. Therefore, parallelism is even more important for this case. We follow the same principles for organizing this new search space as we did with the previous cases in that we shall use a partitioned search tree. Given a target attribute A_i, the patterns we consider are of the form $(X|P)$ where X is an element of $2^{\mathcal{A}\setminus\{A_i\}}$ and P is a conjunction of equality conditions over the attributes of X. These patterns represent the CFDs $(X \to A_i|P)$. Let us denote the search space relative to A_i by $\mathcal{U}_i$. Instead of giving a long list of formal definitions, we rely on an intuitive explanation of how we proceed.

For constructing the search tree, we proceed as follows. We first consider the search tree for mining FDs relative to the target A_i. We replace each element X by $(X|\_\ldots\_)$. The number of _'s is equal to $||X||$.

Example 10. *For our running example and D as being the target attribute, the first version of the search tree is depicted in Figure 2.13.*

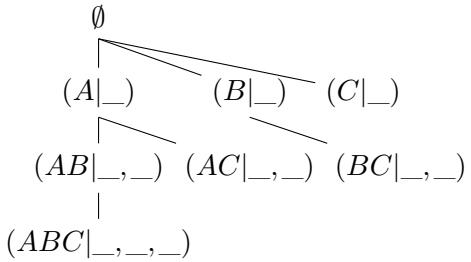


Figure 2.13 – First step.

The second step is described as follows : each node of the form $(X|P)$ gives rise to elements of the form $(X|P')$ where P' corresponds to the conditions in P *plus* one condition. All these elements are added to the first child of node $(X|P)$. For example, $(AB|\_,\_)$ gives rise to $(AB|a,\_)$ and $(AB|\_,b)$. The pattern $(AB|a,\_)$ is a representative of all CFDs of this form by taking every constant value $a \in dom(A)$. If $(X|P)$ is a leaf in the previous tree, then a new node is added and becomes the child of $(X|P)$. For example, $(ABC|\_,\_,\_)$ is a leaf node giving rise to $(ABC|a,\_,\_)$, $(ABC|\_,b,\_)$ and $(ABC|\_,\_,c)$. All these elements belong to the child of $(ABC|\_,\_,\_)$. Note here that this child node will actually contain several elements.

Example 11. *Continuing the previous example, the final tree is depicted in Figure 2.14.*

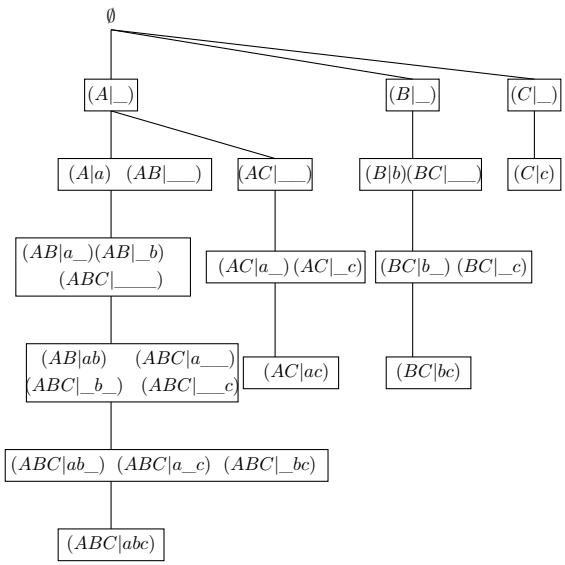


Figure 2.14 – DFS tree $\mathcal{T}_D$

The intuitive idea behind this organization is that when we traverse this tree in a depth first manner, either a dependency is valid then the subtree (a part of) is pruned or we have two ways to be less general than the current candidate : (i) add an attribute or (ii) add a condition.

The notions of right sibling, first child and right parents introduced previously are extended to the present context. We won't give further details in order to avoid a cumbersome list of definitions.

The universe $\mathcal{U}_i$ is also divided into parts respecting the depth first partitions. For the running example with D as being the target attribute, the universe is partitioned as shown in Figure 2.15. The description of **ParaCoDe** is given in Algorithm 4.

Example 12. *Suppose that D is the target attribute and we seek conditional dependencies. Figure 2.16(a) shows considered candidates with in subscript the iteration in which they are dealt and in superscript $+$ denotes the fact that the CFD is valid and $-$ denotes a not valid one. Table 2.16(b) shows the evolution of $\mathbf{\Gamma}^+$ and $\mathbf{\Gamma}^-$ by iteration.*

2.15 Dependencies Mining Experiments

We implemented our solution in C++ together with OpenMP, a multi-shared memory programming API. All tests were conducted on a machine equipped with two hexa-cores Intel Xeon X5680 3.33GHz processors running under Debian Linux version 3.2.32-1 and 96GB of RAM while the caches are respectively L1 = 32KB, L2 = 256KB and L3 = 12MB.We used synthetic data generated following three parameters : the number of tuples (NT), the number of attributes (NA) and a real number CF (correlation factor) lying between 0 and 1, e.g. if NT=1000, NA=10 and CF=0.1 then for each attribute, the number of distinct values is on average $NT * CF = 100$ values. Due to lack of space, we do not report the obtained results for minimal keys nor for conditional FDs.

2.15.1 Exact FDs

In this section we analyze the performance of our algorithm for mining classical FDs. First we show its efficiency w.r.t to pruning. Figure 2.17 shows the ratio between the number of tested FDs and that of returned FDs. One can see that this ratio does not exceed 2 giving evidence of

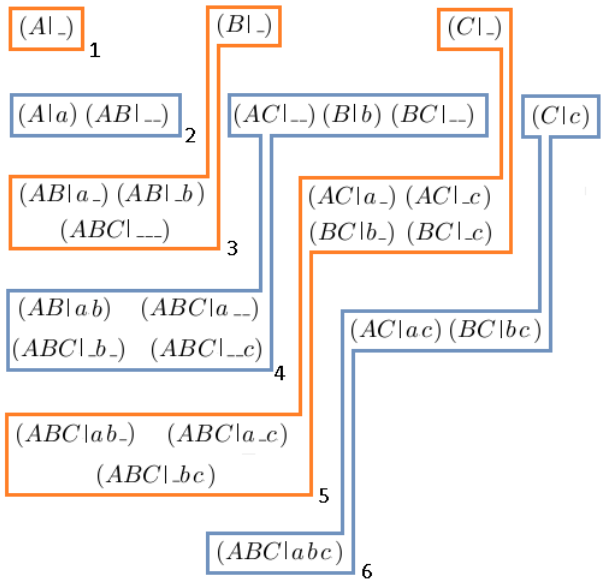
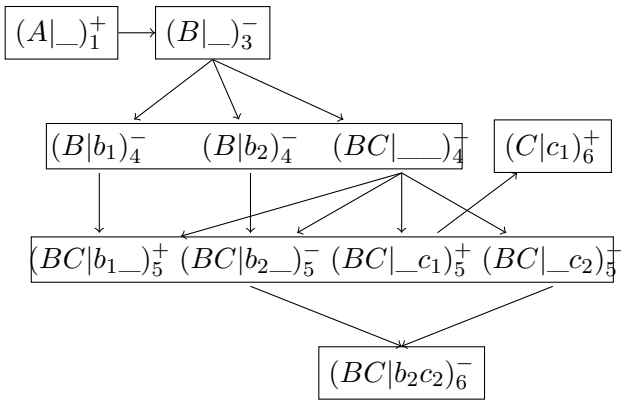


Figure 2.15 – Depth first partition of $\mathcal{T}_D$

Algorithm 4: ParaCoDe

Input: Table T, target attribute A_i
Output: $\mathbf{\Gamma}^+$

$\mathcal{P}_1 \leftarrow (A_1|\_)$;
$r \leftarrow 1$;
while $\mathcal{P}_r \neq \emptyset$ *or* $\mathcal{P}_{r+1} \neq \emptyset$ **do**
  $Children \leftarrow \emptyset$;
  foreach $(X|P) \in \mathcal{R}_r$ **do**
    * Loop in parallel *\;
    if $(X|P)$ *is not covered by* $\mathbf{\Gamma}^-$ **then**
      if $T \models (X \rightarrow A_i|P)$ **then**
        $\mathbf{\Gamma}^+ \leftarrow \mathbf{\Gamma}^+ \cup \{(X|P)\}$;
        $\mathbf{\Gamma}^+ \leftarrow \mathbf{\Gamma}^+ \setminus \{(Y|P') \mid (Y|P') \sqsupset (X|P)\}$;
        $\mathcal{P}_{r+1} \leftarrow \mathcal{P}_{r+1} \cup RightP(X|P)$;
        $\mathcal{P}_{r+2} \leftarrow \mathcal{P}_{r+2} \cup Sibling(X|P)$;
      else
        $\mathbf{\Gamma}^- \leftarrow \mathbf{\Gamma}^- \cup \{(X|P)\}$;
        $\mathbf{\Gamma}^- \leftarrow \mathbf{\Gamma}^- \setminus \{(Y|P') \mid (Y|P') \sqsubset (X|P)\}$;
        $Children \leftarrow Children \cup Child(X|P)$;
  foreach $(X|P) \in Children$ **do**
    if $(X|P)$ *is not covered by* $\mathbf{\Gamma}^+$ **then**
      $\mathcal{P}_{r+1} \leftarrow \mathcal{P}_{r+1} \cup \{(X|P)\}$;
  $r \leftarrow r+1$;
Return $\mathbf{\Gamma}^+$;



(a) Running example execution

1	$\mathbf{\Gamma}^+ = \{(A\|\_)\}$ $\mathbf{\Gamma}^- = \{\}$
2	$\mathbf{\Gamma}^+ = \{(A\|\_)\}$ $\mathbf{\Gamma}^- = \{(B\|\_)\}$
3	$\mathbf{\Gamma}^+ = \{(A\|\_)\}$ $\mathbf{\Gamma}^- = \{(B\|b_1), (B\|b_2), (BC\|\_\_\_)\}$
4	$\mathbf{\Gamma}^+ = \{(A\|\_), (BC\|b_1\_), (BC\|\_c_1\}$ $\mathbf{\Gamma}^- = \{(B\|b_1), (BC\|b_2\_), (BC\|\_c_2)\}$
5	$\mathbf{\Gamma}^+ = \{(A\|\_), (BC\|b_1\_), (C\|c_1)\}$ $\mathbf{\Gamma}^- = \{(B\|b_1), (BC\|b_2c_2)\}$

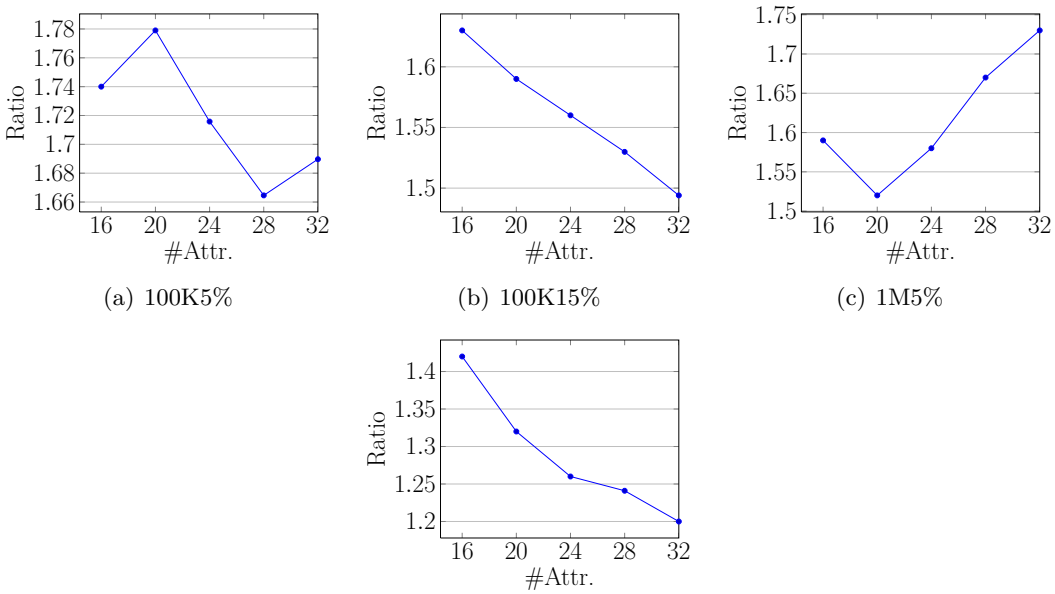
(b) Evolution of $\mathbf{\Gamma}^+$ and $\mathbf{\Gamma}^-$

Figure 2.16 – Execution example

the pruning power of DFS. Actually, the depth first strategy is very sensitive to the order by which attributes are sorted. We used the following heuristics to sort the attributes which in practice turned to provide an efficient pruning : Let A_{n+1} be the target attribute. $\{A_1, \ldots, A_n\}$ are sorted in a way such that $A_i \triangleleft A_j$ iff $confidence(A_i \rightarrow A_{n+1}) \geq confidence(A_j \rightarrow A_{n+1})$. Recall that the confidence of an FD is the maximum fraction of tuples that can be kept in the table without violating this FD. The overhead of this pre-computation is negligible with regard to execution time saving we noticed in the experiments. The intuitive idea behind this ordering comes from the fact that the confidence measure is monotone. Hence, when combined, attributes providing greater confidences are more likely to functionally determine the target attribute and thus a large part of the search tree is pruned. One may notice that this is the same strategy used for mining maximal frequent itemsets where items are ordered in ascending order of their support, see *e.g.*, [101].

We performed the same kind of experiments with FUN [75] and we got the same conclusion, i.e., levelwise algorithms test in general more candidates. It is noticeable that FUN, which is to our best of knowledge the most efficient implementations for discovering FDs was not able to mine a table with 64 attributes and 10^6 tuples due to its excessive memory usage.

Figure 2.18 shows how NT, NA and CF parameters influence the execution time of our implementation as well as the speed up w.r.t. the available processing units. We repeat the same execution by varying the number of threads launched in parallel. The overall speed up is almost linear. It lies between 6.5 and 12 depending on the number of candidates processed during each iteration. From Figure 2.19 we see that when CF increases, the average size of each round decreases which impacts the parallel execution, i.e. the threads do not have enough work to make them busy. This is not a real drawback because having a small number of candidates means that most FDs have a small left hand side (number of attributes). This case represents the easiest situation to deal with, i.e., even the sequential algorithm will find the FDs efficiently. As an extreme example, suppose that all attributes are keys. In this situation we will have the worst speed up since at each iteration we have at most one FD to test. Besides the fact that the speed up increases when CF decreases, the second lesson we learn form these experiments is that our solution tend to be more interesting when



(a) 100K5% (b) 100K15% (c) 1M5%

(d) 1M15%

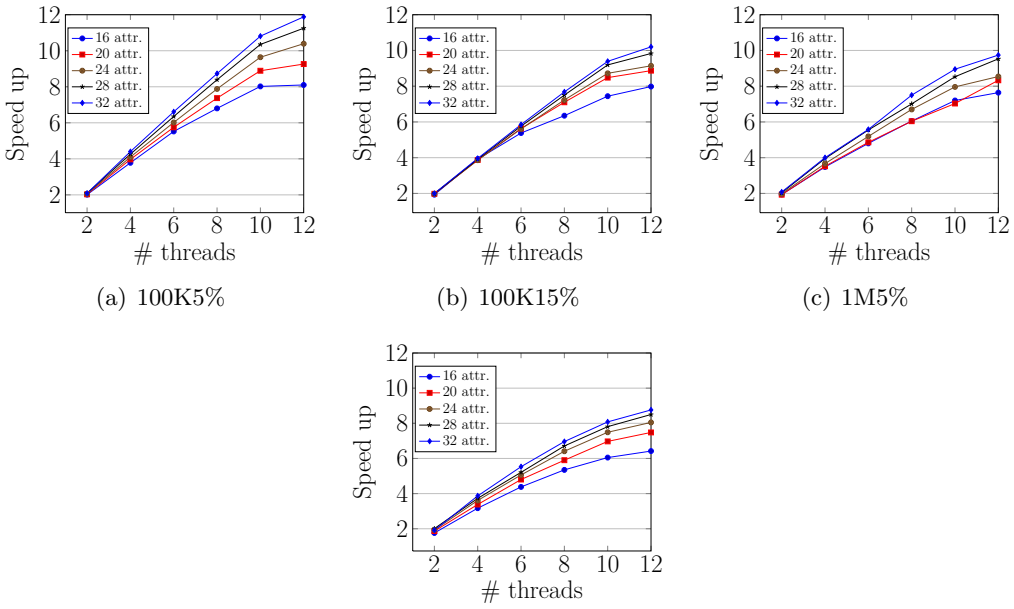
Figure 2.17 – Number of candidates/Number of minimal FDs

the number of attributes increases. Indeed, this has the same impact than when CF decreases, i.e., the number of candidates examined in parallel tends to increase when the number of attributes gets larger. A third lesson we derive is that when the data size (the number of tuples) gets larger, the speed up decreases a little bit. This is explained by the fact that at each parallel iteration, the number of candidates is not an exact multiple of the processors number. Hence, at the end of the iteration if for example only one candidate remains to be tested then one processor will be busy and the remaining are just waiting. The impact of this situation gets more importance when one test takes a long time and this happens when the data set gets larger.

We tested two implementations of our algorithm depending on whether all computations are memorized for potential future reuse or not. For example, when testing $AB \rightarrow C$ we compute $|AB|$ and this value could be reused if when considering the target D, we test $AB \rightarrow D$. While memorization has a real impact in total execution time saving, in terms of speed-up the memory less strategy is in general better. The results reported here are those obtained when previous computations are reused.

2.15.2 Approximating FDs

The HyperLogLog [29] approximation method allows to estimate cardinalities by using less memory than with the exact method. So, since the validity of an FD is given by the equality test of cardinalities, we must measure its rate of error. We have two kinds of errors : (i) non minimal valid FDs and (ii) non valid FDs. For both cases we measure, respectively, a *distance* between non minimal FDs and the minimal ones that they cover and the confidence of non valid FDs. Intuitively, (i) by the distance, we aim to show that for non minimal FDs we need to remove a small number of attributes to make them minimal and by the confidence, we want to show that inexact FDs need a small number of tuples to be removed in order to make them valid, hence they are actually *almost* valid. More precisely, let $X' \supseteq X$ be two sets of attributes and let $d(X', X) = \frac{||X' \backslash X||}{||X'||}$ where $||X||$



(a) 100K5% (b) 100K15% (c) 1M5%

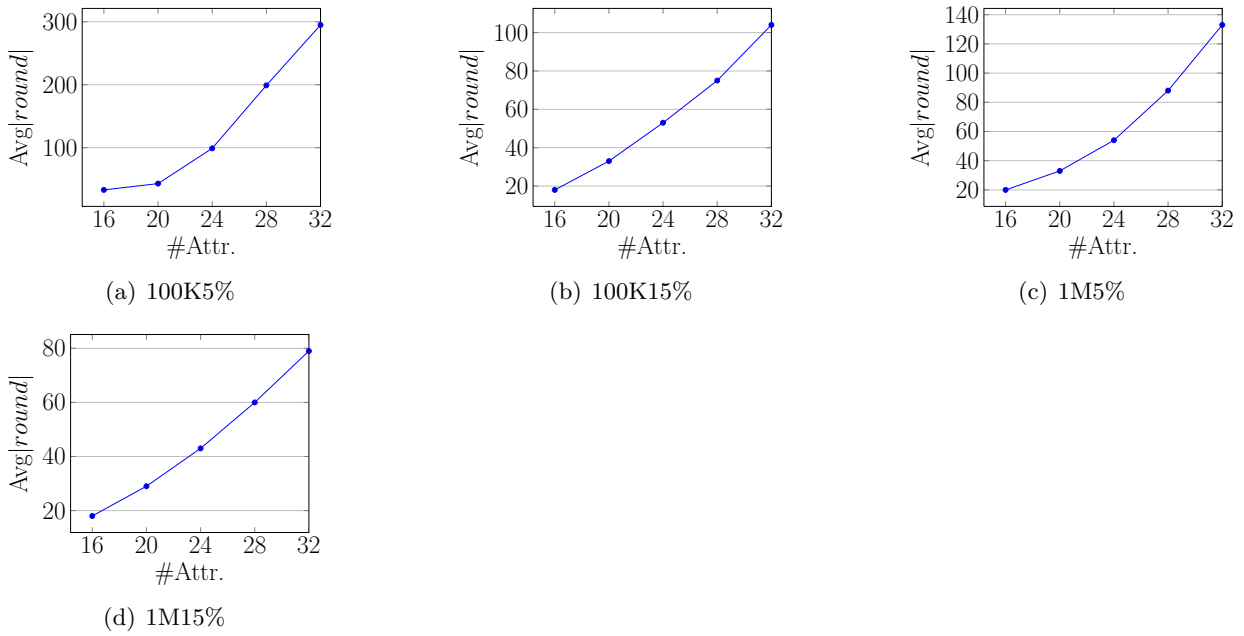
(d) 1M15%

Figure 2.18 – Execution time speed up

denotes the number of attributes in X. For each target attribute A_i, let Γ_i and Γ'_i be respectively the exact and approximate (as returned by HLL) minimal sets of attributes determining A_i. Let $\sigma'_i \subseteq \Gamma'_i$ be the non minimal FDs in Γ'_i. Let X' be LHS of σ'_i. Then $dist(X', \Gamma_i) = \min_{X \in \Gamma_i}\{d(X', X) | X \subseteq X'\}$. From *dist*, one can define the distance between Γ_i and Γ'_i by, *e.g.*, averaging or taking the maximal value. We vary the accuracy factor of HLL from 0.01 for a relatively precise solution using vectors of size 2^{14}, to 0.1 for a loose solution using a vector of size 2^7. For this experiment, the data set has these parameters : $NA = 32$, $NT = 10^6$ and $CF = 10\%$. Moreover, if $\widehat{|X|}$ denotes the approximation of $|X|$, then in the implementation, we consider $X \to A_i$ as valid iff $\widehat{|X|} \geq \widehat{|XA_i|}$. The results are shown below :

HLL accuracy	% of non valid FDs	Avg(conf.)	Min(conf.)	distance
1%	2.19%	0.9987	0.9985	0.2
5%	1.98%	0.9986	0.9975	0.23
10%	5.31%	0.9467	0.923	0.38

We note that the ratio of non valid FDs is quite small (around 3%). For those non valid FDs, the confidence is close to 1 making them *almost* valid. For non minimal FDs (last column), it suffices to remove about one attribute out of 5 to get a minimal FD. It is also important to note that since HLL is associative, our experiments show that using this technique for data parallelism together with task parallelism is possible when an approximate solution is sufficient. We plan to extend our implementation to this last setting in order to mine FDs from distributed data stores when data shipment is impossible either because of their size or for security reasons. Even in a centralized setting, this could be helpful. Indeed, if during the execution we find that the number of candidates is not sufficiently large for task parallelism, we could turn to data parallelism in order to get full exploitation of the available processors.



(a) 100K5% (b) 100K15% (c) 1M5%

(d) 1M15%

Figure 2.19 – Average number of candidates per round

2.16 Conclusion

In this chapter we proposed parallel algorithms for mining functional dependencies as well as some of their variations. We believe that in the near future, most of computers will be multiprocessors machines. However, it is not obvious to adapt sequential solutions to this new setting if we want to maximize the benefit from this extra computation power. Our present proposals show that simulating a depth first traversal of a search space in the special case we considered is not that obvious. Furthermore, in addition to the algorithmic side, the implementation is more intricate than in a sequential world. For example, one has to take into account cache management to avoid multiple communications between threads just because they write concurrently in the same chunks (cache line) of the memory. Data placement is also important. For our programs, the execution times between our first implementation and the current one gained a factor of more than 100. This was one of the main lessons we learned about parallel programming during the implementation phase.

Some extensions of the present work are immediate. For example, to deal with very large data sets, one can sample the data. Since, FDs are monotone i.e., $T_i \not\models X \rightarrow Y$ then for each $T_j \supseteq T_i$, $T_j \not\models X \rightarrow Y$. Hence, we may execute our programs on small parts of the data in order to obtain rapidly a set of FDs for which we are sure they do not hold in the whole data than we execute our algorithms on the entire data while exploiting the previous knowledge about hopeless candidates. We also intend to implement our solution in a distributed framework (Map-Reduce or MPI). This will raise new problems due to communication costs. Hence, a more in depth analysis of our algorithms is required. Finally, we envision to extend our algorithms to other data formats than relational tables, e.g. XML or RDF [9, 89].

The work presented in this chapter appeared in [32, 45].

Chapitre 3

Optimisation des requêtes dans les cubes de données

In On Line Analytical Processing applications the focus is to optimize query response time. To do so, we often resort to pre-computing or, equivalently, materializing query results. However, due to space or time limitations, we cannot store the result of all queries. So, one has to select the *best* set of queries to materialize. In the multidimensional model, more precisely when considering datacubes, relationships between the views can be used in order to define what is the best set of views. A view selection algorithm in the context of datacubes takes as input a fact table and returns a set of views to store in order to speed up queries. The performance of the view selection algorithms is usually measured by three criteria : (1) the amount of memory to store the selected views, (2) the query response time and (3) the time complexity of this algorithm. The two first measurements deal with the output of the algorithm. Most of the works proposed in the literature consider the problem of finding the *best* data to store in order to optimize query evaluation time while the memory space needed by these data does not exceed a certain limit fixed by the user. There are some variants of this problem depending on (1) the nature of data that can be stored, e.g only datacube views or views and indexes, (2) the chosen cost model e.g minimize not only the query response time but also the maintenance time for the stored views or (3) the set of possible queries e.g the user may ask all possible queries or just a subset of them. In this last case, the problem may be refined by taking into account the frequency of queries. To the best of our knowledge, none of the existing solutions give good trade-off between memory amount and queries cost with a reasonable time complexity. Generally, the performance is stated by experiments results without theoretical proof. In this chapter, we propose algorithms whose output, the selected views to be stored, provide some guarantees. The main criteria we consider to assess the quality of the output is the queries response time with respect to the optimal solution.

3.1 Preliminaries

Let $T(Dim_1, \ldots, Dim_n, M)$ be a fact table where attributes $Dim_i, 1 \leq i \leq n$ are *dimensions* and M is a *measure*. The data cube (introduced by [40]) $\mathcal{C}$ obtained from T is the result of the query

```
SELECT    Dim_1, ..., Dim_n, agg(M)
FROM      T
CUBE BY   Dim_1, ..., Dim_n
```

where $agg(M)$ is an algebraic aggregate function, *e.g.,* `COUNT`, `MIN`, `MAX` or `SUM`. The above query is equivalent to the union of the set of all the 2^n `GROUP BY` queries each of which uses a subset of $\{Dim_1, \ldots, Dim_n\}$, *i.e.,*

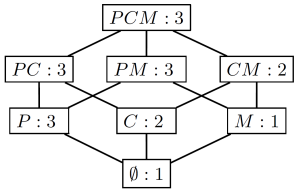
```
SELECT    D, agg(M)
FROM      T
CUBE BY   D
```

where $D \subseteq \{Dim_1, \ldots, Dim_n\}$. Every such query defines a *cuboid*. $Dim(Cube)$ denotes the set $\{Dim_1, \ldots, Dim_n\}$. If Let c and c' be two cuboids of $\mathcal{C}$. We note $c \prec c'$ iff $Dim(c) \subset Dim(c')$. $\langle \mathcal{C}, \prec \rangle$ defines a lattice. For a cuboid c of $\mathcal{C}$ we define : (i) its set of ancestors $A_c = \{c' \in \mathcal{C} |\ \prec'\}$, (ii) its set of descendants : $D_c = \{c' \in \mathcal{C} |\ ' \prec\}$ and (iii) its set of parents : $P_c = \{c' \in A_c\ |Dim(c')| = |Dim(c)| + 1\}$ where $|Dim(c)|$ denotes the cardinality of the set $Dim(c)$.

Example 13. *Consider the fact table Sales in Figure 3.1(a). The first tuple of* Sales *means that customer C_1 bought 50 units of product P_1 and this product is manufactured by M_1. Sales has three dimension attributes :* Product, Customer *and* Manufacturer, *and one measure attribute* nbSales. *Figure 3.1(b) represents its associated data cube. Each node represents a cuboid labeled by its dimensions (for readability, each dimension is designated by its first letter) and its size (number of tuples).*

Sales			
(P)roduct	*(C)ustomer*	*(M)anufacturer*	*nbSales*
P_1	C_1	M_1	*50*
P_2	C_1	M_1	*30*
P_3	C_2	M_1	*40*

(a) FactTable



(b) Cube

Figure 3.1 – Running example

The top most cuboid, i.e., the one with all dimensions, is called the base cuboid *and is noted c_b. It plays a special role as we'll see latter.*

The $\prec$ order defines not only the inclusion relation between dimensions but also a *computability* relation. Indeed, since we are considering only algebraic aggregate functions, we have $c_1 \prec c_2$ implies that c_1 can be computed from c_2 [1].

Example 13 (Continued)**.** *Suppose that the aggregate function is* `SUM`, i.e., *the query defining $\mathcal{C}$ is*

```
SELECT   P, C, M, SUM(nbSales) As S
FROM     Sales
CUBE BY  P, C, M
```

The content of cuboid `CM` *is*

C	M	S
C_1	M_1	80
C_2	M_1	40

Clearly, cuboid M can be computed from CM by the query

```
SELECT    M, SUM(S) As S
FROM      CM
GROUP BY  M
```

1. As an example, the `Median` aggregation function is not algebraic.

We consider the following scenario :

1. The user first submits a cubbing query (i.e., a `CUBE BY`), then
2. he/she will interact with (or navigate through) the cube by submitting specific queries.

The first step allows the user to specify the dimensions he/she is interested in (the analysis axis) together with the measures. The second part represents the analysis process which is more or a less a data mining task, i.e., the user *explores* the content of the cube by submitting a series of queries.

Suppose that the queries users may ask are only those of the form `select * from c` where c is a cuboid from $\mathcal{C}$ or equivalently `select * from T group by c` [2]. There are two extreme situations that can be considered here :

- The first one is that where only the base cuboid c_b is stored (materialized). In this case, every query requires the use of this cuboid and hence has a time cost proportional to the size of c_b.
- The other situation is that where all cuboids are materialized. In this latter case, the evaluation of each query consists just in scanning the corresponding cuboid making its cost proportional to the actual size of the cuboid. Of course, this last situation is quite unrealistic since in practice, we often do not have enough memory (or time) to compute and store the whole data cube.

What is often found in practice is rather a partial materialization of the data cube. Hence, the main problem is how to chose the *best* part of the cube to be computed and stored. There are different notions of what one considers as the *best* part of a data cube. We shall recall some of the definitions that have been considered in the literature and introduce our own definition and argue for its relevance. Before that, let us first introduce some notations.

Let $\mathcal{S} \subseteq \mathcal{C}$ be the set of materialized cuboids and v be a cuboid. Then, $\mathcal{S}_v = \{w \in \mathcal{S} | v \preceq w\}$ is the set of materialized cuboids from which v can be computed. We define the cost of evaluating a query v w.r.t a set $\mathcal{S}$ as follows : if $\mathcal{S}$ does not contain any ancestor of v then $cost(v,\mathcal{S}) = \infty$ otherwise $cost(v,\mathcal{S}) = \min_{w \in \mathcal{S}_v} size(w)$. That is, a query is evaluated by using one of its stored ancestors. The chosen ancestor is the one with fewer tuples. This is the measure usually used to estimate the time complexity (see e.g [49, 87, 91]). Note that when $v \in \mathcal{S}_v$ then $cost(v,\mathcal{S}) = size(v)$. This is the most advantageous situation for v. We also define the cost of a set $\mathcal{S}$ as the cost of evaluating all queries w.r.t $\mathcal{S}$. More precisely, $cost(\mathcal{S}) = \sum_{c \in \mathcal{C}} cost(c,\mathcal{S})$. When $\mathcal{S} = \mathcal{C}$ i.e all cuboids are stored, we have $cost(\mathcal{S}) = \sum_{c \in \mathcal{C}} size(c)$. This is the minimal cost and will be denoted $MinCost$. If $\mathcal{S} = \{c_b\}$ then $cost(\mathcal{S}) = |\mathcal{C}| * M$ where M is the size of c_b. This is the maximal cost and will be denoted $MaxCost$. Thus for every $\mathcal{S}$, we have $\sum_{c \in \mathcal{C}} size(c) \leq cost(\mathcal{S}) \leq |\mathcal{C}| * M$. Note that since the set of possible queries includes c_b then $\mathcal{S}$ should contain c_b, otherwise $cost(\mathcal{S}) = \infty$. Indeed, the base cuboid can be computed only from the fact table. Thus, in the definition of $MaxCost$ we have considered only the sets $\mathcal{S}$ that contain c_b.

The usual performance measures of a view selection algorithm (an algorithm that selects a subset of queries to materialize) $\mathcal{A}$ are :

- *the memory* : $Mem(\mathcal{S}) = \sum_{c \in \mathcal{S}} size(c)$, the amount of memory required to store $\mathcal{S}$;
- *the query cost* or *cost* : $cost(\mathcal{S})$ is proportional to the time to answer the 2^D possible grouping/aggregate queries ;
- *the time complexity* : of the view selection algorithm.

We introduce a new quality measure called the *performance factor*.

Definition 6 (Performance factor)**.** *Let $\mathcal{S}$ be the set of materialized cuboids and c be a cuboid of $\mathcal{C}$.* The performance factor *of $\mathcal{S}$ with respect to c is defined by* $f(c,\mathcal{S}) = \frac{cost(c,\mathcal{S})}{size(c)}$. The average

2. For notation convenience, we use AB to designate both the dimensions `A, B` and the name of the cuboid defined by these dimensions.

performance factor *of* $\mathcal{S}$ *with respect to* $\mathcal{C}' \subseteq \mathcal{C}$ *is defined by* $\tilde{f}(\mathcal{C}', S) = \frac{\sum_{c \in \mathcal{C}'} f(c,\mathcal{S})}{|\mathcal{C}'|}$

Intuitively, the performance factor measures the query response time of a cuboid using a given materialized sample $\mathcal{S}$ with respect to the query time whenever the whole datacube is stored. In other words, for a query c, we know that the minimal cost to evaluate it corresponds $size(c)$. This is reached when c itself is materialized. When c is not materialized, it is evaluated by using one of its ancestors present in $\mathcal{S}$. Thus, the performance factor for c measures how far is the time to answer c from the minimal time. The goal is to obtain the answer to a query with a time proportional to the size of the answer.

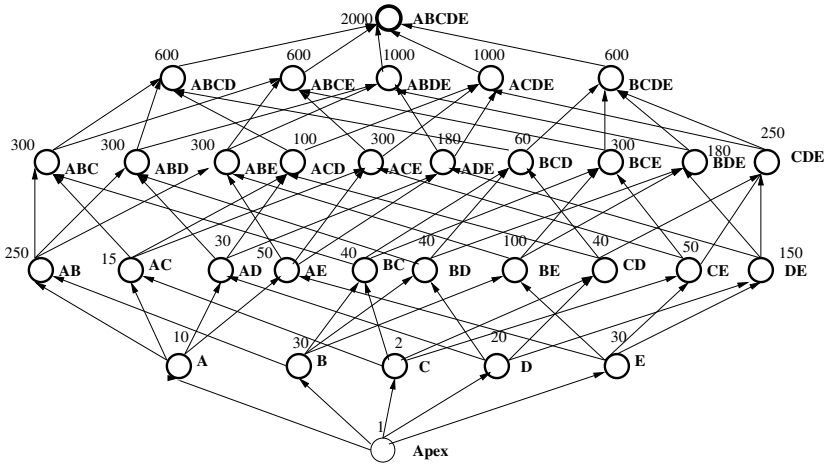


Figure 3.2 – A datacube example

Example 14. *Consider the graph of Figure 3.2. It represents the datacube lattice obtained from a fact table* T *whose dimensions are* A, B, C, D *and* E*. The measure attribute(s) are omitted. Each node is a cuboid and is labeled with its dimensions together with its size. There is an edge from* c_1 *to* c_2 *iff* c_2 *is a parent of* c_1 *so there is a path from* c_1 *to* c_2 *iff* c_2 *is an ancestor of* c_1*. Finally,* c_1 *can be computed from* c_2 *iff there is a path from* c_1 *to* c_2*.*

The top most cuboid is the base cuboid and corresponds to the fact table. The minimal cost for evaluating all queries corresponds to the case where each cuboid is pre-computed and stored. Thus, $MinCost = \sum_{i=1}^{2^5} size(c_i) = 8928$*. In contrast, the maximal cost corresponds to the situation where only the base cuboid is stored. In this case, every query is computed from* $ABCDE$ *and thus has a cost proportional to the base cuboid size. Hence* $MaxCost = 2^5 * size(ABCDE) = 2000 * 32 = 64000$*. Notice however that this is the minimal amount of memory we must use in order to be able to answer all queries.*

Assume now that $\mathcal{S} = \{ABCDE, BE\}$*. The performance measures of* $\mathcal{S}$ *are as follows : The memory required to store* $\mathcal{S}$ *is* $Mem(\mathcal{S}) = size(ABCDE) + size(BE) = 2000 + 100 = 2100$*. The cost for evaluating all the* 2^5 *possible queries is calculated as follows. First, consider the stored cuboid* BE*. It can be used to compute the queries* BE*,* B*,* E *and Apex* [3]*. All these queries can be computed from* $ABCDE$ *too. However this second alternative will require more time than the first one. Thus, the cost of* $\mathcal{S}$ *corresponds to the sum of costs of evaluating* BE*,* B*,* E *and Apex from the cuboid* BE *and all other queries (i.e* $2^5 - 4$*) from* $ABCDE$*. Hence,* $Cost(\mathcal{S}) = 4 * size(BE) + 28 * size(ABCDE) = 56400$*.*

3. *Apex* is the cuboid with no dimensions.

Let us now consider the cuboids BE and BC. Their respective performance factors w.r.t $\mathcal{S}$ are $f(BE,\mathcal{S}) = \frac{cost(BE,\mathcal{S})}{size(BE)} = 100/100 = 1$ and $f(BC,\mathcal{S}) = \frac{cost(BC,\mathcal{S})}{size(BC)} = size(ABCDE)/40 = 2000/40 = 50$. This means that by storing $ABCDE$ and BE, the cost for evaluating the query BE is exactly the minimal cost, but for evaluating the query BC the cost is 50 times the minimal one.

3.1.1 Problem Statement

We address the following problem :

Definition 7 (**View Selection under Query Constraint (VSQC)**). *Given a real number $f \geq 1$, find a set of cuboids $\mathcal{S}$ of minimum size so that $cost(\mathcal{S}) \leq f * MinCost$.*

So we suppose that the user wants a set $\mathcal{S}$ of cuboids which when materialized, the evaluation cost of queries does not exceed f times the minimal cost. Moreover $\mathcal{S}$ should be of minimal size. Notice that the standard way in which the view selection problem is stated consists in fixing the maximal available memory space and selecting a set $\mathcal{S}$ that respects this constraint and provides a good performance.

Definition 8 (**View Selection under Resource Constraint (VSRC)**). *Given a memory space limit space, find a set of cuboids $\mathcal{S}$ whose size is less than space and which provides a minimal cost.*

The obvious solution to our problem **VSQC** consists simply in considering all subsets $\mathcal{S} \in 2^{\mathcal{C}}$, compute their respective costs, keep those $\mathcal{S}$ satisfying $Cost(\mathcal{S}) \leq MinCost * f$ and then return $\mathcal{S}$ whose size is the smallest. Of course this algorithm is unpractical because of its complexity. In fact, from a theoretical perspective, we cannot do better. Indeed, we have the following result stating the hardness of **VSQC**.

Theorem 5. **VSQC** *is NP-Hard.*

So all what we can do is to propose some heuristics that, hopefully, will perform well in practice. Before presenting our solution, we first recall some previous work in partial data cube materialization.

3.1.2 Related Work

Several solutions have been proposed in order to solve the **VSRC** problem (see Def. 8) which is also proven to be NP-Hard. In [49], Ullman *et al* propose a greedy algorithm that returns a subset of views with a guarantee about the *gain* of the solution. This notion is defined as follows :

Definition 9 (Gain). *Let $\mathcal{S} \subset \mathcal{C}$ be a set of cuboids. The gain of $\mathcal{S}$ is defined by $cost(\{c_b\}) - cost(\mathcal{S})$.*

Intuitively, the gain represents the difference between the cost of the *worst* situation, i.e., the one where only the base cuboid c_b is materialized (hence, *MaxCost*) and the cost of the solution. [49] shows that finding the optimal $\mathcal{S}$, i.e., the one which maximizes the gain and respects the space constraint is an NP-hard problem. The greedy approximation algorithm they proposed [4] guarantees that the gain of the returned solution cannot be less than 63% of that of the optimal solution S^*. In other words, $\frac{cost(c_b) - cost(\mathcal{S})}{cost(c_b) - cost(S^*)} \geq 0.63$. Notice that this notion of performance is obtained by comparing the returned solution to the "worst" solution, while our performance factor is relative to the "best" solution. This remark has already been done in [53] where it is shown that maximizing the gain does not mean necessary optimizing query response time. Another weakness of [49] is its time complexity

4. We will call it HRU algorithm.

which is $k * n^2$ where k is the number of iterations (corresponds to the number of selected views) and n is the total number of cuboids. Since $n = 2^D$, so this algorithm is of little help when D is large [5]. To overcome this problem, [87] proposed a simplification of the former algorithm and called it PBS (Pick By Size). This algorithm simply picks the cuboids in size ascending order until there is no enough memory left. They show empirically that even its simplicity and its small complexity (linear), the solutions returned by PBS competes those of [49] in terms of gain. However, the query cost of its solutions suffers from the same problem as that of [49] i.e., no performance guarantee. [91] model the optimization problem as an integer linear problem (ILP). Hence, they succeed to solve exactly moderate sized problems (few dimensions). In order to handle large cases, the authors proposed some simplifications aiming to reduce the search space, thus the number of variable but still they do not provide any theoretical guarantee about the result.

As a final remark, it is worthwhile to note that all these methods suppose a prior knowledge of cuboids sizes. This information is considered as part of their input and it is either computed or estimated. This represents a real limitation of these works since when the number of dimensions is large, the number of cuboids growing exponentially, they become intractable. This is in contrast to our proposal. Indeed, the algorithm PickBorders we propose does not need this knowledge even if during its execution, it does compute the size of some cuboids. We also may cite [63] as another proposal for selecting materialized views. They consider the situation where the number of dimensions is so large, that none of the previous propositions can work. Their pragmatic proposition consists simply in storing cuboids of just 3 or 4 dimensions. The other cuboids can be answered by using a careful data storage technique. They justify their choice by the fact that usually people do not ask queries requiring too many dimensions. However, no performance study has been conducted along that work apart some experiments showing the feasibility of their method.

Dynamat [55] dealt with both data and workload dynamic. Its principle can be described as follows : each time a query is submitted, first find the best plan to evaluate it then decide whether its result could be kept among the already materialized views. In a sense, the system computes a new workload each time a new query is evaluated. In another side, when *batch* updates arrive, the system may have a time constraint in order to perform the propagation to the views. Hence, it choses the *most beneficial* that it can update within the allowed window time interval. This is constrained by the available memory and may trigger the removal of old views.

There is a large body work in selecting views to materialize in larger settings than the data cubes. We refer to the book of Chirkova and Yan [20] and the references there for a comprehensive survey.

3.2 PickBorders

In this section, we present an algorithm that reduces the size of the solution returned by the first algorithm while guaranteeing the fact that the cost still be below $MinCost * f$. Let us first recall some properties of the cuboids sizes.

Lemma 1. *Let c_1 and c_2 be two cuboids such that c_2 is an ancestor of c_1. Then $Size(c_2) \geq Size(c_1)$.*

This lemma simply says that cuboids size is a monotone property. This means that given a size threshold s, we can define a *border* of a data cube. More precisely,

Definition 10. *Let $s \geq 1$ be a natural number and $\mathcal{C}$ be a data cube. Then $Border(\mathcal{C}, s)$ is the set of cuboids $c \in \mathcal{C}$ such that :*

5. Note that HRU is polynomial w.r.t the total number of cuboids by opposition to the naive algorithm which is exponential.

- $Size(c) \leq s$ *and*
- *there is no* c' *ancestor of* c *such that* $Size(c') \leq s$

In other words, $Border(\mathcal{C}, s)$ is the set of *maximal* cuboids (in terms of dimensions set inclusion) whose respective sizes are less than s.

The main idea behind our algorithm *PickBorders* consists in just repeating the computation of borders by considering different values of s obtained by dividing the size of the base cuboid c_b by successive powers of the prescribed performance factor f, i.e., $f^0, f^1, f^2, \ldots, f^k$ until reaching k such that $Size(c_b)/f^k \leq 1$. The union of the so obtained borders forms a partial data cube that will be materialized. Concretely, the algorithm proceeds as follows :

$\mathcal{S} = \emptyset$
for$(i = 1; i \leq \lfloor \log_f(M) \rfloor; i++)$**do**
  $\mathcal{S} = \mathcal{S} \cup Border(\mathcal{C}, Size(c_b)/f^i)$
endfor
$\mathcal{S} = \mathcal{S} \cup \{c_b\}$
Return $\mathcal{S}$

As one may notice, computing the different borders can be performed by a slight modification of the algorithm **MineWithRounds** presented in the previous chapter.

Example 15. *To illustrate our proposition, let us continue with Figure 3.3. The curves represent the borders relative to different powers of the factor* f *when* $f = 10$. *The filled circles represent elements of at least one border. These cuboids are the only ones to be stored. The total size of the datacube is 8928 which also represents the minimal cost for computing all cuboids. The maximal cost is 32*2000=64000. By keeping only the elements of the borders, this will occupy a memory whose size is 2607 and the total cost is 27554. One should notice here that even if we have fixed* $f = 10$, *the cost ratio between the cost of PickBorders solution and MinCost is* $\frac{27554}{8927} = 3.09$. *This means that in average, query response time w.r.t* $\mathcal{S}$ *is 3 times the minimal cost. If we execute the HRU algorithm of [49] with a criterion of maximal available space fixed to 2607, then ABCDE and BCDE are the only cuboids that are returned. Note that with this solution, the total cost is 41600. Thus, the cost ratio is* $\frac{41600}{8927} = 4.67$. *This is worse than the performance of PickBorders. In another hand, if we execute PBS of [87] again with maximal available space fixed to 2607, 16 cuboids will be stored (the sixteen first cuboids ordered by their respective size). In this case, the total cost is 34518. The cost ratio now is* $\frac{34518}{8927} = 3.87$.

The solution returned by PickBorders guarantees the fact that the cost of every query is bounded by its minimal cost times the factor f fixed by the user. As a consequence, the total cost is also bounded. More formally,

Theorem 6. *Let* $f \geq 1$ *and* $\mathcal{S} = PickBorders(\mathcal{C}, f)$. *Then*

1. *For all* $c \in \mathcal{C}$, $cost(c, \mathcal{S}) \leq size(c) * f$.
2. $cost(\mathcal{S}) \leq MinCost * f$

The previous theorem doesn't say any thing about the memory occupied by $\mathcal{S}$. In fact we also have a guarantee with this respect. Let us first define what is an *optimal solution* w.r.t some memory constraint.

Definition 11 (Optimal solution)**.** *Let* Mem *denote a storage space amount. Let* $\mathcal{S} \subseteq \mathcal{C}$. $\mathcal{S}$ *is a* possible *partial datacube iff (1)* $size(\mathcal{S}) \leq Mem$ *and (2)* $cost(\mathcal{S}) \neq \infty$. *The set of possible partial datacubes w.r.t* Mem *is denoted* $Pos(Mem)$. $\mathcal{S}^*$ *is* optimal *w.r.t* Mem *iff (1)* $\mathcal{S}^* \in Pos(Mem)$ *and (2)* $cost(\mathcal{S}^*) = min_{\mathcal{S} \in Pos(Mem)} cost(\mathcal{S})$.

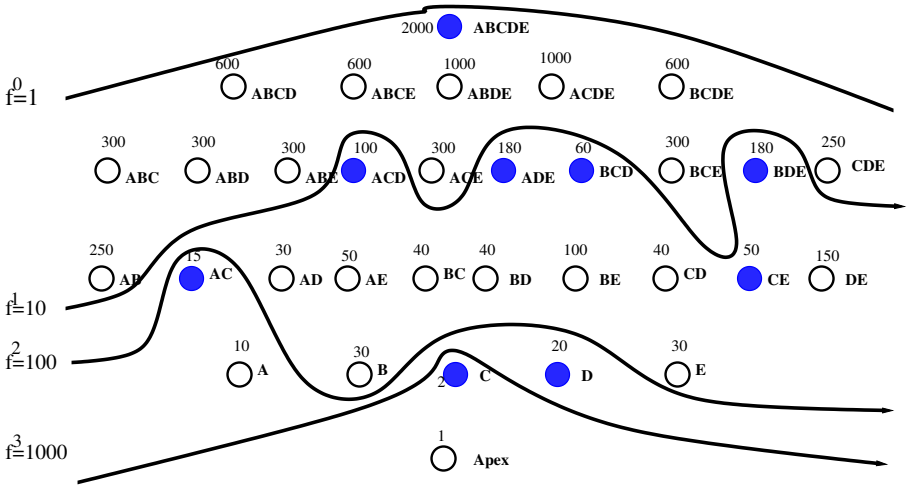


Figure 3.3 – PickBorders : the borders w.r.t $f = 10$

Proposition 3. *Let $f > 1$. Let $\mathcal{S}$ be the solution of PickBorders algorithm. Let $Mem = size(\mathcal{S})$. Let $\mathcal{S}^*$ be the optimal partial datacube w.r.t Mem. Then $cost(\mathcal{S}) \leq f * cost(\mathcal{S}^*)$.*

In other words, this says that if we consider among the sets of cuboids whose total size is less than the size of $\mathcal{S}$, then the cost provided by $\mathcal{S}$ cannot be larger than f times the cost of the optimal solution which respect the memory constraint $Size(\mathcal{S})$.

3.3 Workload optimization

In the previous section we provided a solution aiming at optimizing *all* possible queries. In this section we consider a situation where a subset of queries is consider as important regarding some criteria (e.g., the most frequent queries). For this new setting, we propose both exact and approximate solution depending on the search space as well as the algorithm we consider.

The problem we want to solve is formalized as follows : given (i) a real number $f \geq 1$ and (ii) a set of queries $\mathcal{Q}$, find a set of cuboids $\mathcal{S}$ such that(1) for each $q \in \mathcal{Q}$, $pf(q, \mathcal{S}) \leq f$ and (2) $size(\mathcal{S})$ is minimal. We denote this problem VSPF (View Selection under Performance Factor constraint). Intuitively, we want to find the smallest (in terms of storage space) set $\mathcal{S}$ that guarantees a performance factor for each target query $q \in \mathcal{Q}$.

3.3.1 View Selection as Minimal Weighted Vertex Cover

Theorem 7. *The VSPF problem is NP-Hard.*

Hence, an approximate solution is more viable. For this purpose, we show that our solution is actually the solution of a Minimal Weighted Vertex Cover (MWVC) instance. We first give some definitions. For each $q \in \mathcal{Q}$ we denote by $\mathcal{A}_f(q)$ the set of cuboids c such that (1) $q \preceq c$ and (2) $size(c) \leq f * size(c_q)$. We call this set the f_ancestors of q. $\mathcal{A}_f(\mathcal{Q}) = \bigcup_{q \in \mathcal{Q}} \mathcal{A}_f(q)$. Clearly, the solution of our problem belongs to $\mathcal{A}_f(\mathcal{Q})$.

Definition 12 (Search Graph)**.** *Let $G(f, \mathcal{Q}) = (V, E, \mathbf{w})$ be the graph defined as follows : $V = \mathcal{A}_f(\mathcal{Q})$, $(v_1, v_2) \in E$ iff (i) $v_2 \in \mathcal{Q}$ and (ii) $v_1 \in \mathcal{A}_f(v_2)$. $\mathbf{w}$ is a weight function defined as $\mathbf{w}(v) = size(v)$. We denote by $V_\mathcal{Q}$ the nodes of G that correspond to an element of $\mathcal{Q}$.*

The set of out-neighbors of a vertex v in G is noted $\Gamma(v) = \{v' \in \mathcal{Q} | (v, v') \in G\}$ and the weight of a set of nodes $\mathcal{S} \subseteq V$, denoted $\mathbf{w}(\mathcal{S})$, is equal to $\sum_{v \in \mathcal{S}} \mathbf{w}(v)$. A set $\mathcal{S}$ **covers** $V_{\mathcal{Q}}$ iff $\bigcup_{v \in \mathcal{S}} \Gamma(v) \supseteq V_{\mathcal{Q}}$. So the solution to our problem consists in finding a subset $\mathcal{S} \subseteq V$ such that $\mathcal{S}$ covers $V_{\mathcal{Q}}$ and $\mathcal{S}$ is of minimal weight. This is an instance of the MWVC which is known to be NP-Hard.

3.3.2 Exact Solution

In this section, we propose an ILP program to solve our problem. Let us first give some notations : For each $c_i \in \mathcal{A}_f(\mathcal{Q})$, the constant s_i designates the size of c_i, for each $c_i \in \mathcal{A}_f(\mathcal{Q})$, the variable $x_i \in \{1,0\}$ means respectively that c_i belongs to the solution or not, $y_{ij} \in \{1,0\}$ means that the query $q_i \in \mathcal{Q}$ uses cuboid $c_j \in \mathcal{A}_f(q_i)$ or not. The linear program is :

$$\min \sum_{j : c_j \in \mathcal{A}_f(\mathcal{Q})} x_j * s_j \tag{3.1}$$

$$\forall i : q_i \in \mathcal{Q} \sum_{j : c_j \in \mathcal{A}_f(q_i)} y_{ij} = 1 \tag{3.2}$$

$$\forall i : q_i \in \mathcal{Q}, \forall j : c_j \in \mathcal{A}_f(q_i)\ y_{ij} \leq x_j \tag{3.3}$$

$$\forall j : c_j \in \mathcal{A}_f(\mathcal{Q})\ x_j \in \{0,1\} \tag{3.4}$$

$$\forall i : c_i \in \mathcal{Q},\ \forall j : c_j \in \mathcal{A}_f(q_i)\ y_{ij} \in \{0,1\} \tag{3.5}$$

The program above is denoted $ILP(G(f, \mathcal{Q}))$. The objective function is the minimization of the solution's size. Constraint (2) imposes that q_i uses exactly one materialized cuboid and constraint (3) means that the query q_i cannot use c_j whenever c_j is not materialized. Constraints (4) and (5) say that the variables are binary. The following is a straightforward result characterizing the exact solution of our problem.

Proposition 4. *Let $G = G(f, \mathcal{Q})$ be a search graph. Let Sol be a solution of $ILP(G)$, i.e. Sol is assignment function of $x_i's$ and $y_{ij}'s$ variables of $ILP(G)$. Let $\mathcal{S}^* = \{c_i \in \mathcal{A}_f(\mathcal{Q}) | Sol(x_i) = 1\}$. Then $\mathcal{S}^*$ is an optimal solution.*

For notation convenience, we consider $\mathcal{S}^* = ILP(G(f, \mathcal{Q}))$. It is an optimal solution. The solution to our problem is the set of $c_j \in \mathcal{A}_f(\mathcal{Q})$ such that $x_j = 1$. Current solvers cannot handle these linear programs when the number of variables is too large (this number grows exponentially w.r.t. f and $|\mathcal{Q}|$) rapidly reach thousands). This motivates a different approach that is more efficient in terms of execution time but returns an approximate solution.

3.3.3 Approximate Solution

In this section, we borrow the greedy algorithm of [22] to solve our problem. We first define the **load** of a vertex $v \in V$, noted $\ell(v)$, as $\frac{\mathbf{w}(v)}{|\Gamma(v)|}$.

Function PickFrom$f$Ancestors$(G(f,\mathcal{Q}))$
V = nodes of $G(f,\mathcal{Q})$
$\mathcal{S} = \emptyset$
While $V_{\mathcal{Q}} \neq \emptyset$
  $c^* = \arg\min_{c \in V} \ell(c)$
  $\mathcal{S} = \mathcal{S} \cup \{c^*\}$
  $V = V \setminus \Big(\Gamma(c^*) \cup \{c^*\}\Big)$
End While
Return $\mathcal{S}$
End.

The algorithm choses at each iteration the vertex with minimal load and adds it to the solution. The following theorem is a direct consequence of [22] result.

Theorem 8. *Let $f \geq 1$, $\mathcal{S}$ =PickFrom$f$Ancestors$(G(f,\mathcal{Q})$ and $\mathcal{S}^* = ILP(G)$. Then :*
- *For each $q \in \mathcal{Q}$, $pf(q,\mathcal{S}) \leq f$;*
- *$size(\mathcal{S}) \leq (1+\ln\Delta) * size(\mathcal{S}^*)$ where Δ is the maximal out-degree of G ;*

Even if the complexity of this algorithm is polynomial in the size of $G(f,\mathcal{Q})$, the size itself may be exponential depending on f and the cardinality of $\mathcal{Q}$. Thus, reducing the search space is important in both cases (ILP and PickFrom$f$Ancestors). The first obvious simplification consists in removing all candidates c_j such that $size(c_j) > size(\Gamma(c_j))$. This simplification, which has also been suggested in [91], does not change the solutions of both techniques. Still, there may be too many remaining candidates. In the next section, we propose an additional reduction of the search space.

3.3.4 Reducing the Search Graph

Intuitively, the simplification we consider here consists in keeping for each query q among its f_ancestors, only those that are maximal. More precisely, $\mathcal{B}_f(q) \subseteq \mathcal{A}_f(q)$ denotes the maximal elements of $\mathcal{A}_f(q)$, i.e if $c' \in \mathcal{B}_f(q)$ then for each $c \in \mathcal{A}_f(q)$, either $size(c) > size(c')$ or $c' \not\preceq c$. $\mathcal{B}_f(q)$ is the f_border of q and $\mathcal{B}_f(\mathcal{Q})$ is the union of the f_borders. The intuition behind this heuristic is that keeping maximal f_ancestors tends to keep the ancestors that cover the maximal number of queries (this of course is not always true). The partial search graph is now defined as follows.

Definition 13 (Partial Search Graph)**.** *Let $G_p(f,\mathcal{Q})$ be the graph $(V_p, E_p, \mathbf{w})$ where $V_p = \mathcal{B}_f(\mathcal{Q}) \cup \mathcal{Q}$,$(v_1,v_2) \in E_p$ iff $v_2 \in \mathcal{Q}$ and $v_1 \in \mathcal{B}_f(v_2)$ and $\mathbf{w} : V_p \to \mathbb{N}$ defined as $\mathbf{w}(c) = size(c)$.*

Example 16. *Figure 3.4 shows the search graph $G(f,\mathcal{Q})$ where $f = 10$ and $\mathcal{Q} =$* {`select*fromB`, `select*fromC`, `select* fromD`}*. The dimensions of the underlying datacube are A,B,C and D. All the nodes do not belong to $G(f,\mathcal{Q})$. They are present just for a sake of clarity. Dashed arrows are not present in the partial graph.*

If we consider G_p as the search space then both solutions of $ILP(G_p)$ and of PickFrom$f$Ancestors(G_p) guarantee (1) a performance factor less than f and (2) an approximation factor of the solution's size. Indeed,

Theorem 9. *Let $\mathcal{S}^* = ILP(G)$ be the optimal solution. Let $\mathcal{S}_1 = ILP(G_p)$ and $\mathcal{S}_2 =$ PickFrom$f$Ancestors(G_p). Then (1) $size(\mathcal{S}_1) \leq f * size(\mathcal{S}^*)$ and (2) $size(\mathcal{S}_2) \leq f * (1+\ln\Delta) * size(\mathcal{S}^*)$ where Δ is the maximal out degree of G. Recall that $|\Delta| \leq |\mathcal{Q}|$.*

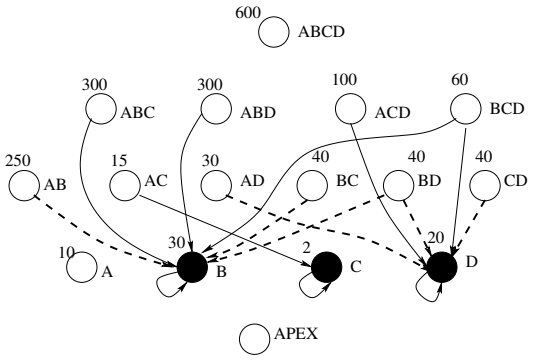


Figure 3.4 – Global and partial search graphs.

3.4 Dynamic Maintenance

In our solutions, we assume that the running time of the view selection algorithms is not a problem. However, whenever some updates on the fact table or the dimensions are performed, not only we should propagate them but may be we have to compute a new set of views in order to guarantee a performance factor below f for target request. We first analyze the *stability* of our solution. Intuitively, this property tells that the query performance factor of a solution computed at time t remains almost unchanged at time $t+1$ after some updates. Thus, the set of materialized views $\mathcal{S}$ can remain unchanged (one only have to maintain it). At the end of this section, we show how to handle this property in order to refresh and to maintain dynamically the views selection with a light cost of computation and materialization.

3.4.1 Stability

Let us consider a query q belonging to $\mathcal{Q}$ and its smallest ancestor $c \in \mathcal{S}$ in term of size. We know that $pf(q,\{c\}) \leq f$. We aim at computing the number of tuples to insert into (or to delete from) c so that $pf(q,\{c\})$ becomes greater than $f+1$. We first start with easy lemmas in order to prove in Theorem 10 some sufficient conditions ensuring the stability of our solutions.

Definition 14 (Stability)**.** *Let T and T' be two successive instances of a fact table. T' is obtained from T by performing some insertions and/or deletions. Let $\mathcal{Q}$ and $\mathcal{S}$ such that $\forall q \in \mathcal{Q}$, $pf(q,\mathcal{S}) \leq f$. $\mathcal{S}$ is* stable *between T and T' if and only if $\forall q \in \mathcal{Q}$, $pf(q,\mathcal{S}) \leq f+1$ wrt T'.*

The insertion of n tuples in c implies the insertion of m tuples in c_q with $0 \leq m \leq n$. The worst case, from query performance perspective, is when $m = 0$. Indeed, in this case, the size of c increases and that of c_q remains unchanged. The following lemma gives the minimal number of tuples to insert into c to break the stability of c.

Lemma 2. *Let c be a cuboid and q be a query such that $pf(q,\{s\}) \leq f$. The insertion of at least $size(c_q)$ tuples into c is required in order to get $pf(q,\{c\}) \geq f+1$.*

Using the same argument as before, we can easily see that the deletion of n tuples from c may trigger the deletion of m tuples from c_q with $0 \leq m \leq n$. From the performance factor point of view, the worst situation corresponds to the case $m = n$.

Lemma 3. *Let c be a cuboid and q be a query such that $pf(q,\{c\}) \leq f$. The deletion of at least $\frac{size(c_q)}{f}$ tuples from c is required in order to get $pf(q,\{c\}) \geq f+1$.*

In our experiments, it turns out that, for the majority of the target queries, the solution is stable after a very large number of updates. This phenomenon can also be explained theoretically by the following result. Let us first give some definitions. Let $Dom(c)$ denotes the domain of a cuboid c and $m(c) = |Dom(c)|$ denotes its cardinality. Clearly, $size(c) \leq m(c)$. c is *saturated* iff $size(c) = m(c)$. c is a *small cuboid* wrt a parameter f iff $size(c) < \frac{|T|}{2f \ln 4f}$. Under some assumption of data distribution and given T :

- There exists a threshold β_1 such that for any *small* cuboid $q \in \mathcal{Q}$ of size larger than β_1, any insertion of tuples does not break the stability property of the solution $\mathcal{S}$;
- There exists a threshold β_2 such that if $|T| \geq \beta_2$, then for any $q \in \mathcal{Q}$ if c_q is a *small* cuboid then the solution $\mathcal{S}$ is stable whatever the number of tuples we insert.

More precisely, without any attempt of optimization of the constant factor β, we have :

Theorem 10. *Let $\{Dom_i\}_{i\in[1,D]}$ be a multi-set. Set $\beta = 64f^2$. Let T be a fact table in which tuples are chosen uniformly at random within the Cartesian product $\Pi_{i=1}^{D} om_i$. Let $\mathcal{Q}$ be a set of queries and $\mathcal{S}$ a set of cuboids such that $\forall q \in \mathcal{Q}$, $pf(q, \mathcal{S}) \leq f$ and c_q is a* small *cuboid. After any sequence of insertions into T and with probability $1 - 2/|\mathcal{Q}|$:*

- *If $\forall q \in \mathcal{Q}$, $size(c_q) > \beta \ln |\mathcal{Q}|$, then $\mathcal{S}$ is* stable.
- *If $|T| > \beta f \ln |\mathcal{Q}|(\ln \beta + \ln \ln |\mathcal{Q}|)$, then $\mathcal{S}$ is* stable.
- *If we have less than $\frac{|T|}{2f \ln 4f}$ insertions of tuples in T, then $\mathcal{S}$ is* stable.

3.5 Some Connections with Functional Dependencies

In this section we give some hints on the interplay between the functional dependencies that may hold in the fact table and the view selection from the data cubes. As a first result we show that the set of *closed* sets of attributes corresponds exactly to the minimal solution for the view selection problem when the performance factor f is set to 1. Let us first recall the definition of closed sets of attributes.

Definition 15. *Given a fact table T. Let $X \subset \mathcal{D}$. X is closed iff there is no attribute/dimension $A \notin X$ such that T satisfies the dependency $X \to A$.*

Theorem 11. *Let T be a fact table, $\mathcal{C}$ be its corresponding data cube ans $\mathcal{S}$ be the set of closed attributes sets of T. The minimal set of cuboids in $\mathcal{C}$ to materialize in order to guarantee a performance factore $f = 1$ for every query on $\mathcal{C}$ is equal to $\mathcal{S}$.*

The above theorem shows that storing the whole data cube or *just* the cuboids corresponding the the closed attributes sets will provide exactly the same performance. In some situations, this can save much useless storage space without sacrificing query execution time.

Now one may wonder how to extend the previous result to arbitrary values of f. The extension is not direct. Let us first define the *strength* of *approximate* functional dependencies.

Definition 16 (Strength)**.** *Let T be some fact table. The strength of $X \to Y$ is given by $|X|/|XY|$ where $|X|$ is the size of $\pi_X(T)$.*

Clearly, $0 < Stronth(X \to Y) \leq 1$ and $X \to Y$ is valid (or exact) iff its strength is equal to 1. Otherwise, the dependency is approximate. Intuitively, $Stronth(X \to Y) = \alpha$ means that, on average, every value of X is associated to $\frac{1}{\alpha}$ distinct values of Y. We may note that $Strength(X \to Y) \geq \alpha$ iff XY is an $\frac{1}{\alpha}$_ancestor of X. By analogy with *closed* attributes sets, one can be tempted to extend this notions to approximate dependencies in order to characterize the minimal sets of cuboids to materialize while respecting f. Unfortunately, we show a less general result.

Definition 17 (f_closed X)**.** *Let X be a set of attributes and $f \geq 1$ be a real number. X is f_closed iff there is no $A \notin X$ such that strength$(X \rightarrow A) \geq \frac{1}{f}$*

Theorem 12. *Given T and f. $\mathcal{S}_1$ and $\mathcal{S}_f$ denote respectively the set of closed and f_closed attributes/dimensions of T. Let $\mathcal{S}^*$ be a minimal solution of the view selection problem under f constraint. Then there exists some set of cuboids $\mathcal{S}'$ satisfying the f constraint such that $\mathcal{S}_f \subseteq \mathcal{S}' \subseteq \mathcal{S}_1$ and $Size(\mathcal{S}') = Size(\mathcal{S}^*)$.*

Intuitively, the above theorem shows that, even if $\mathcal{S}_f$ doesn't characterize exactly the solution of our problem, it may help in that it is included in some optimal solution which itself is included in $\mathcal{S}_1$. From a computational point of view, this can reduce drastically the search space and simplify the computation.

In our works, we identified other connections between some *summarization* techniques of data cubes (see e.g., [57, 92]) and a special case of FD's, namely the conditional FD's. Simply said, these techniques try to select the tuples of the data cube to be stored instead of whole cuboids. We do not report on this work in the present manuscript and refer to [34] for an extensive study.

3.6 Experiments

An experimental validation is given in this section. We consider the following data sets :

- USData10 : contains data with 2.5 millions of tuples with 10 attributes corresponding to the eleven first attributes (excluding the first one representing a rowid) of USData set. This dataset is US Census 1990 data available from `http://kdd.ics.uci.edu/`
- USData13 : it is the same dataset as USData10 but with 13 attributes (adding the next three attributes to USData10).
- Objects : contains data with (only) 8000 tuples with 12 attributes dealing with objects found in archaeological mining. This example represents a case for which the number of attributes is relatively large with respect to the size of the dataset.
- Zipf10 : this dataset is synthetic. It contains 10^6 rows and 10 dimensions. Many observations showed that the attributes values of real datasets do not follow - in general - a uniform distribution but often a power law distribution. That is, if we sort the values of a given attribute in the decreasing order, then the frequency of the value of rank i is proportional to $\frac{1}{i^\alpha}$, $\alpha > 0$. α belongs mostly to the range $[2,3]$. In our experiments, we have considered a power law of parameter $\alpha = 2$.

Table 3.1 summarizes some characteristics of these data sets. It sums up MinCost (whenever the whole datacube is stored) and MaxCost (whenever only the base cuboid is stored).

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

Table 3.1 – MinCost and MaxCost of Datasets

3.6.1 Cost and memory

In our experiments, since we make no assumption on the way the views are physically stored, the amount of memory is expressed as the number of rows of the materialized views set. For PickBorders,

we run the algorithm taking $f = 1.5, f^2 = 2.25, f^3 = 3.38, \ldots$ for all datasets. Each execution leads to a pair Memory/Cost. We then used this Memory value as the space limit parameter for both HRU and PBS. Again, each time we obtain a pair Memory/Cost. This experiments results are depicted respectively in Figures 3.5, 3.6, 3.7 and 3.8. For instance, for USData10, when $f = 1.5$, PickBorders needs 2160000 units of memory for a cost of 4750000 whereas for $f = 3.38$, PickBorders takes 828000 units of memory and has a cost of 7100000.

In all experiments, PBS has the worst performance in terms of cost and memory. In general, HRU has the best performance but PickBorders is a very good challenger. We can also remark that PickBorders is very competitive whenever the amount of available memory is not too low (for $M > 10^5$ in USData10, for $M > 10^6$ in Objects and for $M > 15.10^6$ in ZIPF10).

Due to time required to run HRU, we stopped computation before adding all cuboids to S as soon as the cost function is close to MinCost.

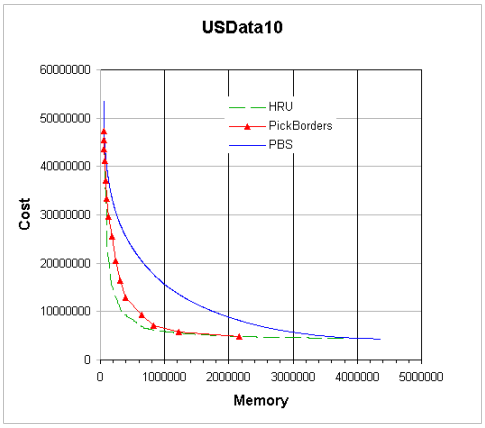


Figure 3.5 – Cost/Memory : USData10.

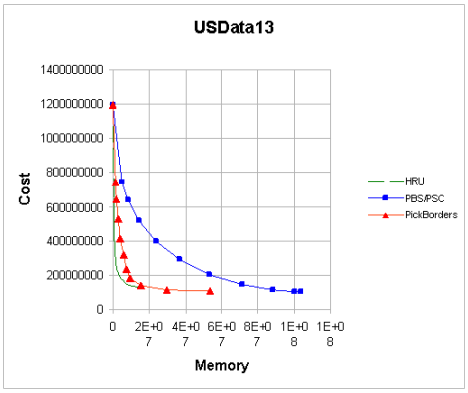


Figure 3.6 – Cost/Memory : USData13.

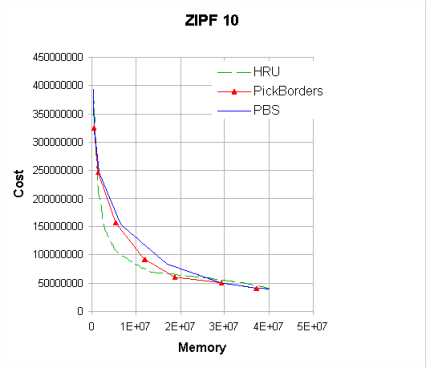


Figure 3.7 – Cost/Memory : ZIPF10.

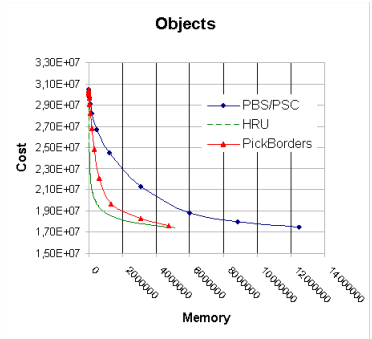


Figure 3.8 – Cost/Memory : Objects.

3.6.2 Performance factor

Now we compare our approach to HRU and PBS with respect to their query evaluation performances in order to assess the *quality* of the returned solution. For this experiment, we executed PickBorders by varying the value of f. For each so obtained solution $\mathcal{S}$ we run both HRU and

PBS with $Size(\mathcal{S})$ as their hard input constraint. We then compare the performance by which each solution can optimize the queries by reporting the respective performance factors. For example, Figure 3.9 shows that PickBorders has a better average performance factor than PBS and HRU for USData10 (for $f = 3.38$ and $f = 11.39$).

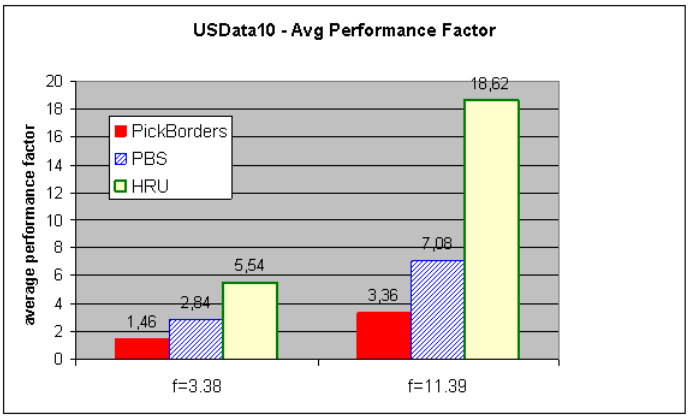


Figure 3.9 – Performance factor with $f = 3.38$ and $f = 11.39$.

A more careful look at the distribution of the performance factors in Figures 3.10 and 3.11 explains this fact by showing that some views should be computed from materialized views whose size is very large. For instance, running HRU with 830000 units of memory, there are 7 (not materialized) cuboids whose smallest materialized ancestor is of size at least 100 times larger. We ran PBS and HRU with 309000 and 830000 units of memory limit corresponding respectively to the amount of memory used by PickBorders with $f = 3.38$ and $f = 11.39$. The first category represents the set of cuboids with performance factor 1 (materialized views and views whose least materialized ancestor have the same size). Second category counts the cuboids with a performance factor in $]1,2]$. The third category counts the cuboids with a performance factor in $]2,3.5]$ and so on. Note that the spirit of each algorithm is sketched by the first category : PBS stores many small cuboids, HRU tends to materialize few large cuboids and PickBorders chooses a combination of cuboids of different sizes.

We terminate this section by noting that the experiments we have presented in this paper aimed to show the quality of the solutions returned by PickBorders compared to other algorithms. We do not report the execution times of the three algorithms. In fact, while PickBorders and PBS often took few seconds to obtain a solution, HRU required hours.

We used the *US Census 1990 data*. Here after, the time parameter represents the accumulated time for constructing the search graph and the resolution time for obtaining a solution $\mathcal{S}$. The different performance measures of a solution $\mathcal{S}$ depend heavily on the target queries $\mathcal{Q}$. We studied three random generation methods of $\mathcal{Q}$. Each of which corresponds to special properties : (i) **Uniform generation (UNIF)** : All possible queries $q \in \mathcal{C}$ have the same probability to belong to $\mathcal{Q}$; (ii) **Queries generated between level 1 and level d_{max} (DMAX)** : We fix the maximal number of dimensions then we iterate over the levels 1 and $dmax$ each time we pick uniformly a query from level i and (iii) **DESC** : We use the same principle as DMAX but here, each time we pick a query q from level i, we add to $\mathcal{Q}$ the 2^i queries $\{q_j\}_{j\in[1,2^i]}$ where q is an ancestor of q_j. In order to check the effectiveness of the approximate algorithms, we compared their results to the exact solutions in terms of computation time and storage space. We generated $\mathcal{Q}$ using UNIF and compared the space memory required for storing $\mathcal{Q}$ with the memory needed by the exact and that of PickFrom$f$Ancestors$(G(f,\mathcal{Q}))$ with $f = 10$. Figure 3.12 shows that PickFrom$f$Ancestors (solution S) behaves very well w.r.t the optimal solution (S^*) in terms of memory gain while being much

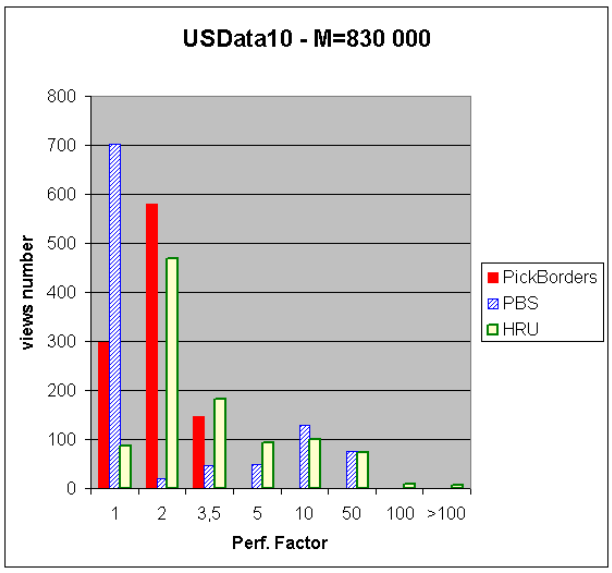


Figure 3.10 – Performance factor distribution with $f = 3.38$.

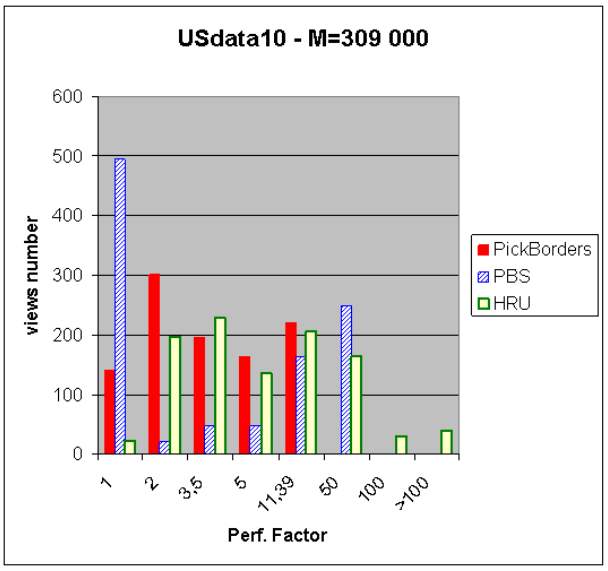


Figure 3.11 – Performance factor distribution with $f = 11.39$.

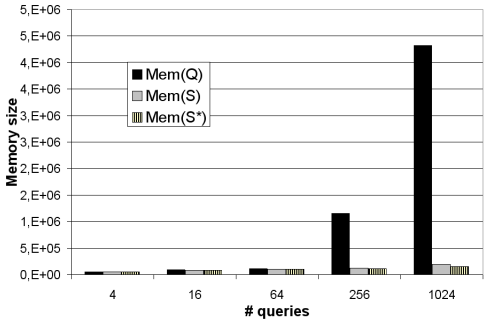
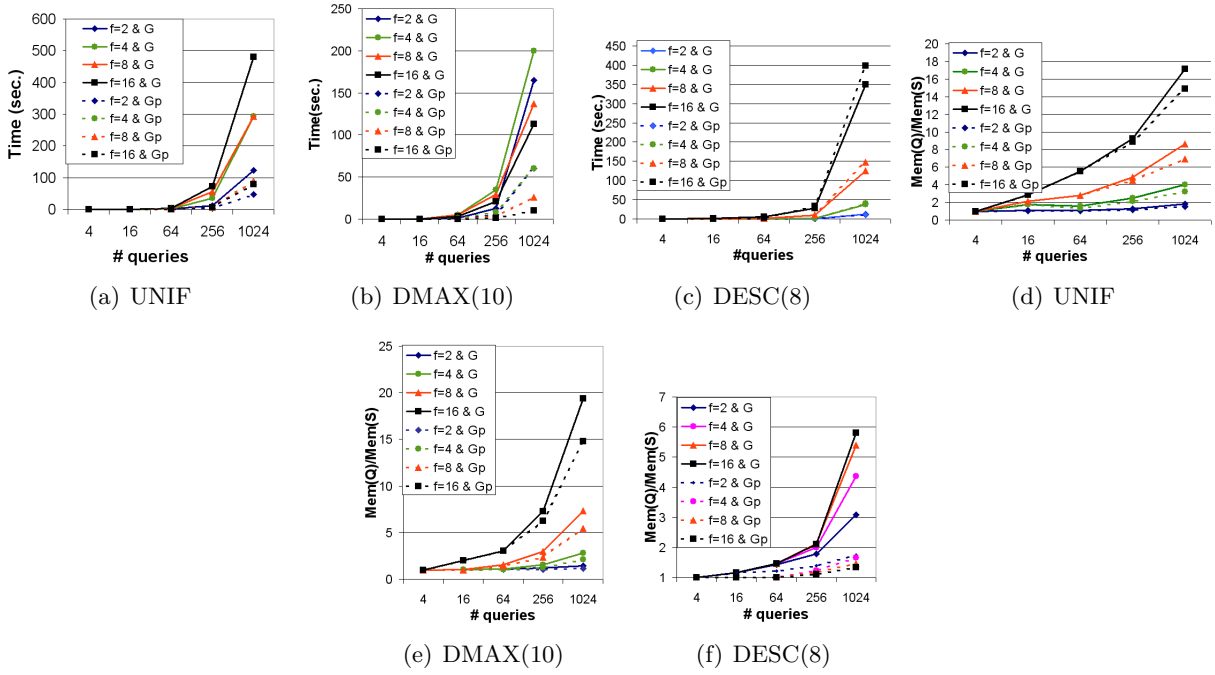


Figure 3.12 – Storage space of the query targets ($\mathcal{Q}$), the approximate (S) and the exact ($S*$) solutions.

faster : it took 2 seconds to find S and more than an hour for S^* using the CPlex software. We should mention that this experiment was performed with only 10 dimensions. With more than 10, the computation time of S^* prohibitive. The base cuboid has 500K rows and 20 dimensions. For



(a) UNIF (b) DMAX(10) (c) DESC(8) (d) UNIF

(e) DMAX(10) (f) DESC(8)

Figure 3.13 – Execution time and memory gain

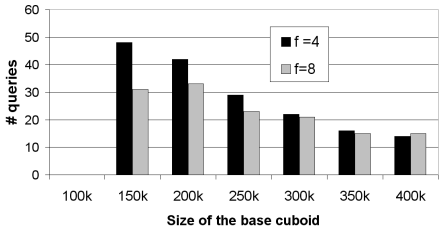
each query generation method, we varied $|\mathcal{Q}|$ and f. For each combination of the three preceding parameters, we computed a solution with both G and G_p.

The execution times (expressed in seconds) are illustrated in Figure 3.13(a) to 3.13(c). The partial search graph offers an interesting compromise in terms of computation time and the memory gain. Indeed, the execution time is about 4 times less than that of G while keeping the memory gains comparable. We encountered however one exception (cf. Figures 3.13(f) and 3.13(c)). In that case, the partial search graph does not summarize well the set $\mathcal{Q}$ because it finds a solution with too much cuboids. Since the computation time is in $O(\Delta|V(G)| \cdot |\mathcal{S}|)$, it depends on the number of returned cuboids. This explains why, in this case, the computation time of G_p is larger than that of

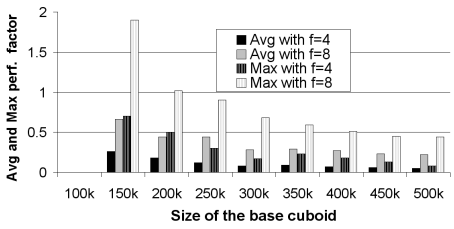
G. It also gives a hint about the importance of the way the workload is built. The results depicted in Figures 3.13(d) to 3.13(f) concern the same experiments as previously. First, it is clear that with G, the storage space is always more reduced. We also note that in most cases (the first three), the reduction ratios obtained with G or G_p are comparable. The last experience exhibits a different behavior (Figure 3.13(f)) since we have a drastic difference. Recall that this case is also the one where the execution time with G is better than that with G_p (see 3.13(c)).

3.6.3 Stability Analysis

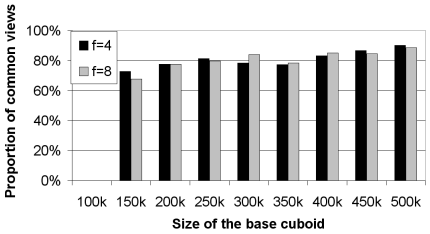
In order to analyze the stability of our solutions, we conducted some experiments whose principle can be described as follows : We fix the factor f, the number of target queries and the generation method. We execute the approximation algorithm PickFrom$f$Ancestors$(G(f,\mathcal{Q}))$ on different data sets $file_1,\ldots,file_n$ such that $file_1 \subset \cdots \subset file_n$. For each $1 < i \leq n$, we obtain a solution $\mathcal{S}_i$ and for each $q \in \mathcal{Q}$, we compute $pf(q,\mathcal{S}_{i-1})$. This allows us to verify in what extent the performance of our solutions worsen from $file_i$ to $file_{i+1}$. The retained criteria of comparison are (1) the number of target queries of $\mathcal{Q}$ whose performance factor become beyond f (2) and for these queries, we measure the difference between the new pf and the fixed f. This represents the amount of overtaking. We present the obtained results when $|\mathcal{Q}| = 512$. The queries are generated using UNIF and the performance threshold takes two values $f = 4$ and $f = 8$. Figure 3.14(a) shows the number of



(a) Queries whose performance factor is beyond the threshold.



(b) Deviations from the fixed performance factor threshold.



(c) Percentage of common views.

Figure 3.14 – Stability analysis.

queries whose performance factor becomes larger than the threshold f. We note that this number is decreasing while the size of the base cuboid increases. Figure 3.14(b) illustrates the deviations from the fixed threshold, it shows the maximal and the average deviations, i.e., average and maximal values of $pf(q,\mathcal{S})$ for those target queries q whose pf is greater than the fixed f. Again, we note that these parameters (max and average) decrease while the size of $file_i$ increases. Figure 3.14(c) shows the proportion of views that are kept in the next solution. We measured $|\mathcal{S}_i \cap \mathcal{S}_{i+1}|/|\mathcal{S}_{i+1}|$. Around 80% of the views selected in the previous solution belong to the next one. This shows that even when we have to recompute a new solution, only few views will be calculated from scratch ; the majority will need at worst to be refreshed.

3.7 Conclusion and Future Work

We presented a new formalization of the materialized view selection problem in the context of data cubes. Some extensions of the present work are straightforward, e.g. integrating dimensions hierarchies is made easy because hierarchies are themselves lattices. Furthermore, it is not required to have a unique f for all queries. It suffices to consider $\mathcal{A}_{f_i}(q_i)$ so that, without changing the algorithms, the obtained solutions guarantee $pf(q_i, \mathcal{S}) \leq f_i$ where f_i reflects the importance of the query (lower is f_i more important is q_i). As future research, we plan to analyze the stability property more in depth depending on data distributions. A prior knowledge of data distribution and/or dimension dependencies could be helpful in this case [23]. [8, 12] provided a solution to the selection of binary join indexes to optimize star join queries under storage space constraint. We believe our solution can easily be adapted in order to select instead of cuboids, the join indexes to materialize. The work presented in this chapter appeared in [33, 34, 47, 48].

Chapitre 4

Optimisation des requêtes skyline

Given a table $T(Id, D_1, \ldots, D_d)$, the skycube of T is the set of skylines wrt to every subsets of dimensions (subspace) $\{D_1, \ldots, D_d\}$. In order to optimize these skyline queries, the solutions proposed so far in the literature either (i) precompute all the skylines or (ii) they propose compression techniques so as the derivation of every skyline is performed with little effort. Clearly, solutions (i) do not scale when d is large because of their exponential number of skylines. Solutions (ii) are appealing in terms of skyline derivation but they too suffer from a high computation complexity making them unfeasible in practice. In this paper, we propose a new formalization of the optimization problem : find a minimal set of skylines sufficient to answer every remaining skyline queries while avoiding to use of the, possibly large, underlying data. Our solution can be seen as a trade off between valuation, computational cost for finding what to materialize and the memory space usage. To solve this problem, we need to know if the skyline wrt some dimension set X is included into that wrt Y provided X is included into Y. Because of the non-monotonic nature of skylines, it is hard to establish this relationship. By exploiting the classical concept of functional dependencies, we identify cases where the inclusion holds. Equipped with this information, we show how to find the smallest set of skylines to materialize. We conduct a thorough experimental study showing that with the help of a small number of materialized skylines, we drastically reduce the execution time of the remaining skyline queries. We also propose an algorithm for the full materialization of skycubes and compare it to state of the art solutions. We show that our proposal outperforms previous work both on real and synthetic data especially with large dimensionality and/or data size. Finally, we compare our proposal to *the closed skycube* technique and we show empirically that in general our solution uses less storage space and more importantly, it is orders of magnitude faster to obtain when the number of dimensions increases.

4.1 Introduction

Multidimensional database analysis has been a hot research topic during the last decade. Precomputation is a common solution to optimize multidimensional queries. An early proposal of such approach is the so-called data cube [40] which, intuitively, represents a set of aggregation queries with all potential subsets of the attributes. After an initial series of works concentrating on efficient ways to fully materialize a data cube, see e.g., [5], [102], it was rapidly recognized that this solution was unfeasible in practice due to the large amount of memory space as well as the processing time needed since the number of queries is exponential with respect to the number of dimensions. Therefore, the question that raised was how to materialize just a subset of queries while satisfying some prescribed user requirements. This problem has been largely studied in the literature and different solutions have been proposed depending on the objectives and the constraints that are considered, see e.g. [49], [7], [62], [46].

Besides, skyline queries [14] have been proposed to express multidimensional data ranking. In order to rank the data with respect to all possible combinations of the attributes, the *skycube* structure has been independently proposed in [80], [100]. This laid to several proposals of efficient algorithms aiming at fully materializing the skycube, see e.g., [81], [59]. Little work has been proposed for partially materializing skycubes. For example, [82] proposed to materialize the *closed skycubes* which identifies equivalent skylines in order to save memory space by storing just one copy of the same *query* result. Another summarization technique of skycubes is the *compressed skycube* (CSC) structure proposed in [95] which tries to reduce the number of materialized copies of each skyline *point* while closed skycubes reason on the skylines level by trying to reduce the number of stored copies of the same set of skyline query result. These techniques will be described more precisely in the related work section. What we can note however is that our experiments show that their computation time is prohibitive when the number of attributes is large.

In the past, functional dependencies (FDs) have been successfully used in order to detect redundant information, e.g., [69], [71], [2]. They also have been used in semantic query optimization, e.g., [17], [36]. Inspired by [33] where FDs have been proven useful for defining data cube partial materialization, in the present paper we propose to use them as a way to select the *minimal set* of skyline queries to be materialized in order to answer efficiently every skyline query of the skycube without using the underlying data. This appears counter-intuitive and surprising because FDs do not carry any information about any ordering among the attributes values, whereas skyline queries are based on tuples ranking.

Our objective is to provide a solution that in practice can be computed in a reasonable time while guaranteeing a reasonable execution time for every skyline query.

An important motivation for proposing a skycube structure is to provide the user with a set-based representation that can help him/her choose dimensions that lead to *a convenient* set of non dominated objects. Among the criteria that define the convenience of the returned set, its size is in general required. For example, suppose that a website of a car dealer proposes to use skyline queries to rank the vehicles using a subset combination of three parameters : price, mileage and fuel consumption. Suppose that price and mileage are first chosen by a user and it turns that the number of vehicles in the skyline is too large. There is no evidence that adding fuel consumption or removing mileage will reduce the number of returned objects. Hence, providing the user with the inclusion information will help him/her to navigate through this multidimensional space. As another case where the inclusion information is useful, suppose that we know that the skyline w.r.t. price is included in the skyline w.r.t. mileage and price, then once the later is computed, thus cached, if the user *drills down* to the skyline w.r.t. price, there is no need to use the entire data set, it suffices to use the previously cached result.

Contributions : In this paper we make several contributions towards multidimensional skyline queries optimization. More precisely,

- identifying, with the help of FDs, cases where the inclusion relationship between two skylines holds. This makes the study of partial materialization of skycubes feasible and well founded.
- Moreover, this identification turns out to be useful when full materialization of the skycube is envisioned by defining an order following which skylines can be computed efficiently.
- To provide a complete solution to the skyline query problem, we had to come up with an efficient algorithm for finding closed sets of attributes from a table. Even though this concept is classical in the theory of FDs, we found no algorithm for computing this set.
- Since our solution relies on FD's that hold, it appears that the less distinct values per dimension we have, the less FD's that are satisfied and thus the more, in theory, skylines to be materialized. However, we show that in this case, the size of skylines tends to decrease giving

new optimization opportunities.
- Our solutions are implemented and compared to state of the art algorithms. We show that (i) it outperforms state of the art algorithms aiming at fully materializing skycubes. Moreover, (ii) we compare our solution wrt to the most recent technique for skycubes summarization, namely *closed skycubes* and show that our proposal consumes less storage space, can be computed much faster and doesn't sacrifice too much query response time. This gives evidence that our proposal is a good trade-off between space usage and query optimization. Finally, (iii) we show that the superiority of our approach is clearer when either the number of dimensions or the data size get larger.

Chapter organization : The next section gives the main definitions and notations used through out the paper. Next we formalize the skycube partial materialization problem and provide our solution. Then we show how skyline queries are answered efficiently from the materialized part of the skycube. We compare our proposal with some related works and terminate by a series of experiments showing the efficiency of our solution. We conclude with directions for future work.

4.2 Preliminaries

Let T be a relational table whose set of attributes $Att(T)$ is divided into two subsets $\mathcal{D}$ and $Att(T) \setminus \mathcal{D}$. $\mathcal{D}$ is the subset of attributes (dimensions) that can be used for ranking the tuples. In the skyline literature $\mathcal{D}$ is called a *space*. If $X \subseteq \mathcal{D}$, then X is a *subspace*. $t[X]$ denotes the projection of tuple t on X. We denote by d the number of dimensions. For each $D_i \in \mathcal{D}$ we assume a total order $<$ between the elements of the domain of D_i. We say that t' dominates t w.r.t. X, or t' X-dominates t, noted $t' \prec_X t$, iff for every $X_i \in X$ we have $t'[X_i] \leq t[X_i]$ and there exists $X_j \in X$ such that $t'[X_j] < t[X_j]$. The skyline of T with respect to $X \subseteq \mathcal{D}$ is defined as $Sky(T,X) = \{t \in T \,|\, \nexists t' \in T$ such that $t' \prec_X t\}$. To simplify the notation and when T is understood from the context, we sometimes omit T and use $Sky(X)$ notation instead. The *skycube* of T, noted $\mathcal{S}(T)$ or simply $\mathcal{S}$ is the set of all $Sky(T,X)$ where $X \subseteq \mathcal{D}$ and $X \neq \emptyset$. Formally, $\mathcal{S}(T) = \{Sky(T,X) \,|\, X \subseteq \mathcal{D}$ and $X \neq \emptyset\}$. Each $Sky(T,X)$ is called a *skycuboid*. $d = |\mathcal{D}|$ is the dimensionality of $\mathcal{S}(T)$. There are $2^d - 1$ skycuboids in $\mathcal{S}(T)$. S is a *subskycube* of the skycube $\mathcal{S}$ if it is a subset of $\mathcal{S}$. Table 4.1 summarizes the different notations used throughout the paper.

Example 17. *We borrow the toy table T from [82] and use it as our running example.*

RiD	*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

$Att(T) = \{Rid, A, B, C, D\}$. Let $\mathcal{D} = ABCD$ and let $X = ABCD$, then $t_2 \prec_X t_1$. Indeed, $t_2[X_i] \leq t_1[X_i]$ for every $X_i \in \{A, B, C, D\}$ and $t_2[C] < t_1[C]$. In this example $d = 4$. The skylines w.r.t. each subspace of $\mathcal{D}$, the set of all skycuboids, are depicted in Table 4.2. This example shows that the skyline results are not monotonic*, i.e., neither $X \subset X' \Rightarrow Sky(X) \subseteq Sky(X')$ nor $X \subset X' \Rightarrow Sky(X) \supseteq Sky(X')$ are true. For example, $Sky(ABD) \not\subseteq Sky(T, ABCD)$ and $Sky(D) \not\supseteq Sky(AD)$. This makes partial materialization of skycubes harder than classical data cubes.*

Notation	Definition
T	Relation instance
$\mathcal{D}$	Attributes/Dimensions used for skylines
d	$\|\mathcal{D}\|$ number of dimensions
n, m	number of tuples
$X, Y \ldots$	subset of dimensions/subspace
XY	$X \cup Y$
$t[X]$	projection of tuple t on X
$t_1 \prec_X t_2$	$t_1[X]$ dominates $t_2[X]$ or t_1 X-dominates t_2
$Sky(T, X)$ or simply $Sky(X)$	the skyline of T w.r.t X
$\pi_X(T)$	projection of T on X with set semantics
$\|X\|$	the cardinality of $\pi_X(T)$
$\|\|X\|\|$	number of attributes of X
$t_1 \equiv_X t_2$	$t_1[X] = t_2[X]$
$\mathcal{S}(T)$ or simply $\mathcal{S}$	skycube of T
S	a subskycube of $\mathcal{S}$

Table 4.1 – Notations

4.3 Partial Materialization of Skycubes

The main objective of our present work is to devise a solution to the partial materialization of skycubes under some constraints. The first and most important one is that the partial skycube should be *as small as possible* in order to minimize both its storage space and its computation time.

Before stating formally the problem we address, we exhibit some properties holding between the subspace skylines of a skycube.

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 4.2 – The set of all skylines

4.3.1 Properties of Subspace Skylines

Even if the skyline query is not monotonic, we can exhibit a monotonic property between the set of tuples belonging to the skyline over some subspace X and that over Y whenever $X \subseteq Y$. More precisely,

Proposition 5. *Let $X \subseteq Y$ and let $\pi_X(Sky(X))$ be the projection*[1] *on X of the tuples belonging to $Sky(X)$. Then, $\pi_X(Sky(X)) \subseteq \pi_X(Sky(Y))$.*

Example 18. *$Sky(A) = \{t_1, t_2\}$ and $Sky(AC) = \{t_2, t_4\}$. Although $Sky(A) \not\subseteq Sky(AC)$, we have $\pi_A(Sky(A)) \subseteq \pi_A(Sky(AC))$. Indeed, the projection $\pi_A(Sky(A))$=$\{\langle 1\rangle\}$ and $\pi_A(Sky(AC))$=$\{\langle 1\rangle; \langle 4\rangle\}$.*

The above result has been identified by previous work, see eg. [81]. As a consequence we obtain a relationship between the tuples belonging to $Sky(T,X)$ and those in $Sky(T,Y)$ whenever $X \subseteq Y$. More precisely,

Proposition 6. *Let $X \subseteq Y$. Then $t \in Sky(T,X)$ iff $\exists t' \in Sky(Sky(T,Y),X)$ such that $t'[X] = t[X]$.*

The proposition above shows that the skyline points of every subspace skyline form a lattice. Indeed, it suffices to project every skyline on the dimensions defining its subspace then fill all missing dimensions with the special symbol $*$. These generalized tuples define the skylines *patterns*.

Example 19. *Take the subspace B. $Sky(B)$ contains two tuples t_1 and t_2. The unique pattern defining the tuples in $Sky(B)$ is the generalized tuple $\langle *, 3, *, * \rangle$. Figure 4.1 shows the patterns of the skycube of the running example.*

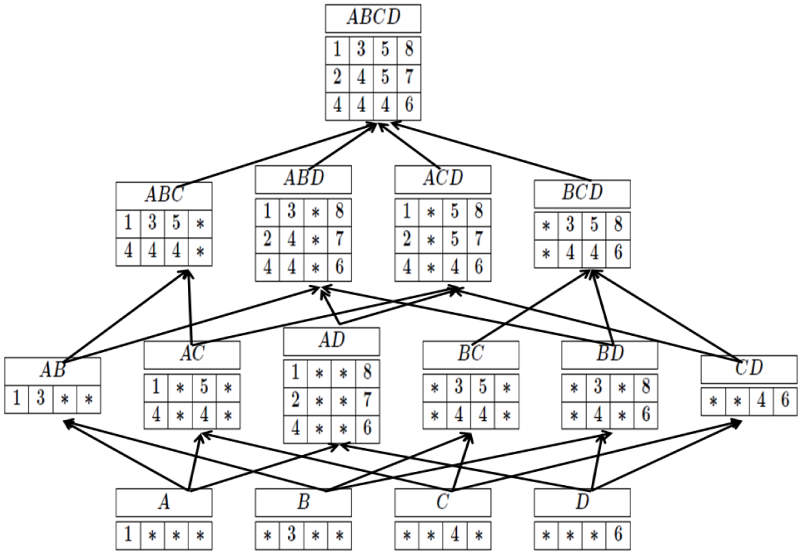


Figure 4.1 – Lattice of the skycube patterns

This proposition shows that when $X \subseteq Y$, the computation of $Sky(X)$ can benefit from the fact that $Sky(Y)$ is already materialized as described in Algorithm 5. It has two main steps :

1. We consider the set semantics of projection.

1. compute $S_Y^X = Sky(Sky(Y), X)$ which represents a subset of $Sky(X)$. Then
2. *retrieve* from T those tuples t that are X-*joinable* with some $t' \in S_Y^X$, i.e., $t[X] = t'[X]$. The retrieved tuples form $Sky(X)$.

Algorithm 5: Sky_X_from_Sky_Y

Input: Table T, $Sky(Y)$
Output: $Sky(X)$

```
1 if X ⊂ Y then
2 |   Let S1 = πX(Sky(Y));
3 |   Let S2 = Sky(S1, X);
4 |   Return T ⋈ S2;
5 else
6 |   Return Sky(T, X) ;
```

Example 20. *Suppose that $Sky(ABC) = \{t_2, t_4\}$ is already materialized and we want to get $Sky(AB)$. First, we compute $S_{ABC}^{AB} = Sky(Sky(ABC), AB)$. This contains only one tuple which is t_2 ($t_2 \prec_{AB} t_4$). Then, we retrieve from T those t' s.t $t'[AB] = t_2[AB] = \langle 1, 3\rangle$. There is one such tuple, apart t_2 itself, which is t_1. Hence, $Sky(AB) = \{t_1, t_2\}$.*

The second step of the above procedure is not very costly. Indeed, it is actually a join operation between $Sky(Sky(Y), X)$ and T. It can be performed in $O(|T| + |Sky(Sky(Y), X)|$ with a hash-join algorithm : traverse $Sky(Sky(Y), X)$ and insert each $t[X]$ in a hash-table H, then traverse T, hash every $t'[X]$ and check in $O(1)$ whether the value belongs to H. If it is the case, then return t'. An even more efficient procedure would be to use a bitmap index of T by considering the values appearing in $Sky(D)$. Doing so, recovering the tuples in T that match those in S_2 can be done efficiently if the size of the bitmap index is not too large, i.e., too many distinct values to consider [60], [93]. Nonetheless, it is still interesting to know *a priori* whether this second step is necessary. In the next section we give a sufficient condition for identifying cases where the join operation can be avoided. Before giving the formalization of the addressed problem, we first give some definitions. We start with the *information-completeness* property of partial skycubes.

Definition 18 (Information complete subskycube)**.** *Let S be subskycube of $\mathcal{S}$. S is an* Information-Complete Subskycube *(ICS) iff for every subspace X, there exists a subspace $Y \in 2^{\mathcal{D}}$ such that $X \subseteq Y$, $Sky(Y) \in S$ and $Sky(X) \subseteq Sky(Y)$.*

Intuitively, S is an ISC iff it contains a sufficient set of skycuboids that is able to answer every Skyline query without resorting to the underlying data T.

Example 21. *One can easily verify that the subskycubes $S_1 = \{Sky(ABCD),\ Sky(ABD)\}$ and $S_2 = \{Sky(ABCD),\ Sky(ABD), Sky(AC)\}$ are both ICS's. Note that $\mathcal{S}_3 = \{Sky(ABCD)\}$ is not an ICS because, e.g., there is no superset of ABD whose skyline is a superset of that of ABD. If for example, S_1 is materialized, then $Sky(A)$ could be evaluated from $Sky(ABD)$, whose size is 4, instead of using table T with 6 tuples.*

From the storage space usage perspective, it is natural to try to identify smallest ICS's.

Definition 19 (Minimal Information Completeness)**.** *S is a* minimal *ICS (MICS) iff there exists no other ICS S' such that $S' \subset S$.*

Example 22. *$S_1 = \{Sky(ABCD), Sky(ABD)\}$ is smaller than $S_2 = \{Sky(ABCD), Sky(ABD), Sky(AC)\}$. One can easily verify that S_1 is the unique MICS of T.*

Now we are ready to formalize the problem of partial materialization of skycubes as we address it.

Problem Statement : Given a table T and its set of dimensions D, find an MICS of T that will be materialized in order to answer all the skyline queries over subsets of D.

The following proposition shows that actually, every table T admits a unique MICS.

Proposition 7. *Given T, there is a unique MICS of T.*

Identifying the unique MICS is easy when the full skycube is available. However this is inefficient from a practical point of view.

In the rest of this section we devise a method leveraging the functional dependencies concept in order to avoid the full materialization. Indeed, we show that the presence of some functional dependencies implies the inclusion of skylines. Hence, we can avoid to compute some of them.

4.3.2 The Interplay Between FDs and Skylines

Recall that the functional dependency $X \rightarrow Y$ holds iff for every pair of tuples t_1, t_2, if $t_1[X] = t_2[X]$ then $t_1[Y] = t_2[Y]$. The following theorem represents our main result in this paper. It shows how functional dependencies can be used in order to identify inclusion cases between related skylines.

Theorem 13. *Let $X \rightarrow Y$ be an FD satisfied by T. Then $Sky(T, X) \subseteq Sky(T, XY)$.*

Example 23. *Turning back to the running example, the set of minimal functional dependencies satisfied by T are*

$$A \rightarrow B \quad A \rightarrow D \quad BD \rightarrow A \quad CD \rightarrow B$$
$$BC \rightarrow A \quad BC \rightarrow D \quad CD \rightarrow A$$

From these FDs, we derive, among others, the following inclusions $Sky(T, A) \subseteq Sky(T, AB) \subseteq Sky(ABD)$. The inclusion relationships between the different skycuboids of $Skycube(T)$ are depicted in Figure 4.2. Each node X represents $Sky(T, X)$. An edge from X to Y represents an inclusion. Paths also represent inclusions. The red nodes represent skycuboids without outgoing edges. One should notice that these are only the inclusions we can deduce from the FDs satisfied by T. For example, $Sky(C) = \{t_4\} \subseteq Sky(BC) = \{t_2, t_4\}$, but this inclusion is not captured by the FDs.

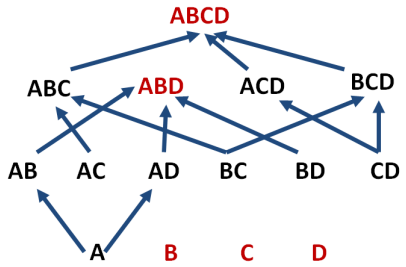


Figure 4.2 – Inclusions between skycuboids

The following proposition shows how the FDs can be used to derive an ICS.

Proposition 8. *Let $\mathcal{I}$ be the set of skylines inclusions derived from the FDs satisfied by T and let $\mathcal{G}_{\mathcal{I}}(V,\mathcal{E})$ be the oriented graph where $V = 2^{\mathcal{D}}$ and $\mathcal{E} \subseteq V \times V$ such that $(X,Y) \in \mathcal{E}$ iff $Sky(X) \subset Sky(Y) \in \mathcal{I}$. Let $\Gamma = \{X \in V \mid X$ has no outgoing edge$\}$. Then Γ is an ICS.*

As a consequence of the previous proposition, we can conclude that having Γ is sufficient to infer the MICS. Indeed, it is sufficient to check the inclusion relationship between the skylines of its elements and those whose skyline is not included into any skyline of their ancestors are elements to find the MICS. For example, the red nodes in Figure 4.2 form Γ. We see that only $Sky(ABD)$ and $Sky(ABCD)$ are not included in any superset. As we have seen, they form the MICS.

Now, we provide some properties of the elements of Γ that allow to identify them efficiently. By Theorem 13, we conclude that a subspace X belongs to Γ iff there is no subspace Y such that $X \cap Y = \emptyset$ and the FD $X \rightarrow Y$ is satisfied by T. In the functional dependency literature, these subspaces are classically called *closed* sets of attributes. We recall briefly the definition and suggest, e.g., references [69], [71] for more details.

Definition 20 (Closed Subspace)**.** *Let F be a cover set of the FDs satisfied by T and X be a subspace of T. The* closure *of X w.r.t. F, noted X_F^+ or simply X^+, is the largest subspace Y such that $F \vdash X \rightarrow Y$ where $\vdash$ represents the* implication *between FDs. X is* closed *iff $X^+ = X$.*

The elements of Γ are the closed subspaces. Hence, having the FDs satisfied by T, it becomes easy to find its MICS. It is important to note that the functional dependencies we consider are those that hold in the instance T. These are not supposed to be known beforehand. So we distinguish between those FDs that act as constraints which are always satisfied by the instances of T and those that are just satisfied by the present instance. Therefore, we need an efficient algorithm to *mine* the FDs satisfied by the instance T from which we can derive the closed subspaces. This will be discussed next.

Before that, we make some remarks about the importance of Theorem 13. We point out that it allows to derive some previous results that were hard to prove without resorting to FDs. Perhaps the most used property from which many optimization techniques were derived is that related to the *distinct value condition* hypothesis.

Theorem 14. *[81] If $\forall D_i \in \mathcal{D}$ and $\forall t_1, t_2 \in T$, $t_1[D_i] \neq t_2[D_i]$ then for every subspaces X and Y, $X \subseteq Y \Rightarrow Sky(X) \subseteq Sky(Y)$.*

The above result is a consequence of Theorem 13. Indeed, under the distinct value hypothesis, every single attribute determines, in the FDs sens, all other attributes. In particular, $\forall X, Y : X \rightarrow Y$. Therefore $Sky(X) \subseteq Sky(XY)$.

4.3.3 Analysis of the number of closed subspaces

Let $||X||$ denote the number of attributes in X, for example $||ABC|| = 3$, and let k be the number of distinct values of every dimension and m be the number of distinct tuples in table T. X is a key of T iff $|X| = m$. Clearly, if X is a key then every Y such that $Y \supseteq X$, Y is not closed (apart the special case where Y is the set of all dimensions).

Under a uniform distribution, if $||X|| \geq \log_k(m)$ then X is a key [49]. As a particular case, if $k = m$ then $\log_k(m) = 1$ meaning that every single attribute is a key by itself. As an example, if $d = 20$, $m = 10^7$ and $k = 100$ then most subspaces X with $||X|| \geq 4$ are keys, hence not closed. Even if the hypothesis of uniform distribution as well as the the fact that all dimensions have the same number of distinct values k are rarely met in real data, it however gives an intuition about the relationship that exists between the active domain sizes of every attribute and the size, in terms of

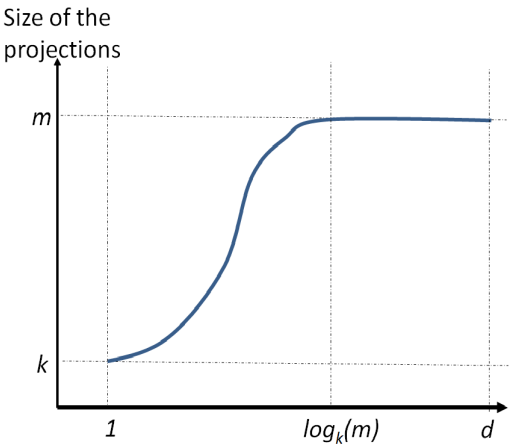


Figure 4.3 – Size of projections wrt dimensionality $||X||$

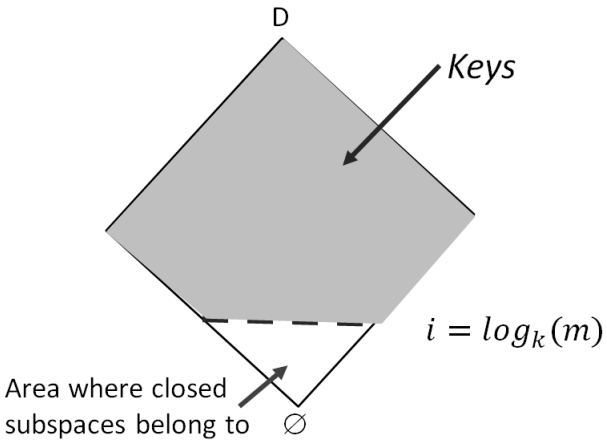


Figure 4.4 – Closed subspaces in the subspaces lattice

number of attributes, of the keys : the larger k, the smaller $||X||$ the larger the number of supersets of X and the fewer the closed subspaces. Figure 4.3 shows the evolution of projection sizes wrt the number of dimensions we consider : when this number reaches $log_k(m)$ the projections reach the same number of distinction tuples in T meaning that these are keys. The more we have keys, the less closed subspaces there are as it is illustrated in Figure 4.4.

4.3.4 Skyline size analysis

When the cardinality k of dimensions decreases, the number of closed subspaces increases and maybe reaching $2^d - 1$. From partial materialization point of view, this is a bad situation : all skylines are materialized. In fact and by contrast to the number of closed subspaces, the size of skylines is proportional to k. This indicates that when k is small, we do not need to materialize other skycuboids than that of the topmost subspace, i.e., $Sky(D)$. This latter is sufficient to answer all skyline queries efficiently using Algorithm 5. The following theorem formalizes this result.

Claim 1. *Let $\mathscr{T}(d,n,k)$ denote the set of tables with d independent dimensions, n tuples and k distinct values per dimension uniformly distributed. Let $T_k \in \mathscr{T}(d,n,k)$ and $T_{k'} \in \mathscr{T}(d,n,k')$ where*

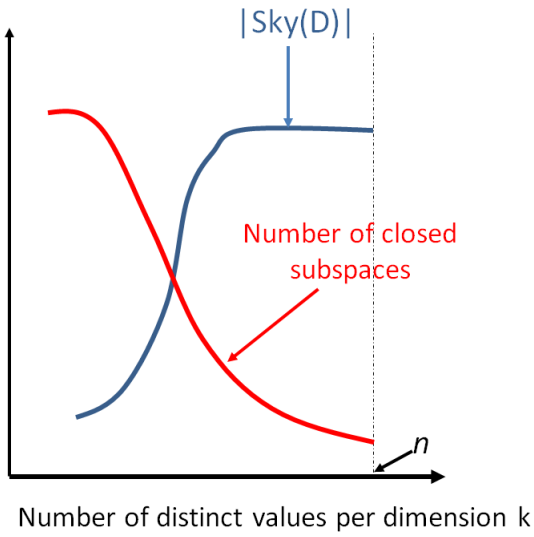


Figure 4.5 – Skyline size evolution wrt cardinality k

$k' \geq k$. Let $S(T_k)$ or S_{T_k} denote the size of the skyline of T_k over the d dimensions. Let S_k denote the average size of $S(T_k)$. The tuples are distinct ($n < k^d$). We have the following

$$k \leq k' \Rightarrow S_k \leq S_{k'}$$

In other words, the above theorem states that for fixed n and d, the size of the skyline tends to increase when the number of distinct values per dimension grows.

Figure 4.5 illustrates how both the number of closed subspaces and skyline size evolve with respect to the number of distinct values k when $n > k^d$.

4.3.5 Data Dynamics

In this section we analyze the skycube evolution when data are inserted or deleted. Our main concern is to study how the partial skycube should be maintained. The first result established an inclusion relationship between closed subspaces when data are inserted or removed.

Proposition 9. *Let $T \subseteq T'$, $Closed(T)$ be the closed subspaces in T and $Closed(T')$ be those closed in T and T'. Then, $Closed(T) \subseteq Closed(T')$.*

A consequence of the previous proposition is that when tuples are deleted, it suffices to maintain the already materialized skycuboids to answer all skyline queries. More precisely,

Corollary 1. *Let $T \subseteq T'$ and S be the set of skycuboids $Sky(T, X)$ such that $X \in Closed(T')$. Then S is an ICS.*

If the removed tuples do not belong to $Sky(T', X)$ then $Sky(T', X) = Sky(T, X)$. This shows that in this case, no re-computation is needed.

Let us consider the insertion case and suppose that $\tau = T' \setminus T$ be the inserted tuples. Consider some subspace X. What is the new content of $Sky(T', X)$ regarding its previous value $Sky(T, X)$? The first remark we can notice is that all previously dominated tuples, i.e., those in $T \setminus Sky(T, X)$, are still dominated in T'. Therefore, we can conclude that $Sky(T', X) \subseteq Sky(T, X) \cup \tau$. This remark suggests a first solution for handling insertions which consists in keeping exactly the same

closed subspaces as those in $Closed(T)$. When some query $Sky(T', X)$ is submitted, we evaluate $Sky(Sky(T, X^+) \cup \tau, X)$ where X^+ is the closure of X w.r.t. T. Even if this solution doesn't require any special maintenance of the skycube and even if the query evaluation performance is already better than evaluating skylines from T', there are some properties that may reduce query evaluation time by reducing the input data size. Again, we make use of FDs.

Proposition 10. *Let X_T^+ denote the closure of X w.r.t. T. Suppose that $Sky(T, X_T^+) \cup \tau$ satisfies the FD $X \rightarrow (X_T^+ \setminus X)$. Then $Sky(T', X) \subseteq Sky(T', X_T^+)$.*

The proposition simply states when the inserted tuples do not violate the previously valid FDs then skyline inclusion still holds. Note that FDs violation is not tested over the whole T' table but just on $Sky(T, X_T^+) \cup \tau$.

We do not develop much more the partial skycube maintenance and let this for future work.

4.3.6 Computing the Closed Subspaces

We start with some lemmas letting us to characterize the closed subspaces. We use the following notations :

- Det_{A_i} is the set of minimal sets of attributes X such that $T \models X \rightarrow A_i$.
- $\mathcal{D}_i = \mathcal{D} \setminus A_i$.

Det_{A_i} represents in fact a frontier separating the elements of $2^{\mathcal{D}_i}$ which determine A_i from those which do not.

Lemma 4. *For each $X \in 2^{\mathcal{D}_i}$, if there exists $X' \in Det_{A_i}$ s.t $X' \subseteq X$ then X is not closed.*

Example 24. *From the running example, we have $Det_A = \{BD, CD, BC\}$. $BCD \in 2^{\mathcal{D}_A}$ is not closed because, e.g., $BCD \supset BD$.*

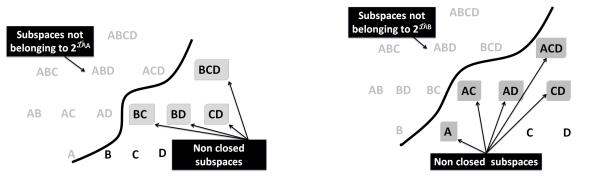
The converse of the previous lemma doesn't hold. Indeed, even if when some $X \in 2^{\mathcal{D}_i}$ does not include any element of Det_{A_i} then $X \not\rightarrow A_i$, this does not imply necessarily that X is closed. Indeed, it is possible that there exists $A_j \neq A_i$ such that $X \in 2^{\mathcal{D}_j}$ and $T \models X \rightarrow A_j$ making X not closed. In fact, we have the following necessary and sufficient condition for X being closed.

Proposition 11. *Let $\mathcal{C}_i^-$ be the set of non closed subspaces derived from Lemma 4 and $\mathcal{C}^- = \bigcup_i \mathcal{C}_i^-$. Then X is closed iff*

- *$X = \mathcal{D}$ (all the attributes) or*
- *$X \in 2^{\mathcal{D}} \setminus \mathcal{C}^-$.*

Example 25. *For the running example we have $Det_A = \{BD, BC, CD\}$, $Det_B = \{A, CD\}$, $Det_C = \emptyset$ and $Det_D = \{A, BC\}$. From these sets, we derive $\mathcal{C}_A^- = Det_A \cup \{BCD\}$, $\mathcal{C}_B^- = Det_B \cup \{AC, AD, ACD\}$, $\mathcal{C}_C^- = Det_C \cup \emptyset$ and $\mathcal{C}_D^- = Det_D \cup \{AC, ABC\}$. Notice for example that $ABD \notin \mathcal{C}_A^-$ even if it is a superset of BD. Recall that for $\mathcal{C}_A^-$ we consider only elements from $2^{\mathcal{D}_A}$ and ABD does not belong to this set. Figures 4.6(a) and 4.6(b) show a part of the non closed subspaces that we infer from the sets of attributes determining, respectively, A and B.*

Procedure **ClosedSubspaces** (c.f. Algorithm 7) takes as input the table T and returns the closed subspaces. As one may see, the most critical part of this algorithm is the statement in Line 2 which consists in computing a set of violated functional dependencies.



(a) Non closed sets w.r.t. A

(b) Non closed sets w.r.t. B

Figure 4.6 – Pruned sets w.r.t. attributes A and B

Extracting Maximal Violated Dependencies

As we have seen previously, all subsets of the left hand side of violated functional dependencies are potentially closed sets of attributes. So, given an attribute A_i the first part of our procedure consists to extract the maximal X's such that $X \rightarrow A_i$ is not satisfied. There are several algorithms for computing *minimal* functional dependencies in the literature, e.g. [98], [66], [52], [75]. Inferring from these sets the *maximal* violated dependencies can be performed by computing the minimal hypergraphs transversals, also called minimal hitting sets [25]. Since the minimal transversals computation is in general hard, its precise complexity is still an open problem and no known polynomial algorithm for the general case has been proposed so far, we use instead **MaxNFD** (for Maximal left hand side of Non Functional Dependencies). It is an adaptation of the parallel algorithm MineWithRounds proposed in [45] for mining borders of theories [70]. The main principles of **MaxNFD** are presented in Algorithm 6. It can be described as follows : let X be a candidate for which we want to test whether $T \not\models X \rightarrow A_i$. If (i) X passes the test then it is possibly a maximal not determining set. Hence it is added to **Max** and its *parent* is generated as a candidate for next iteration. The *parent* of X is simply the successor superset of X in the lexicographic order. For example, the parent of BDF is $BDFG$. If (ii) X does not pass the test, i.e., $T \models X \rightarrow A_i$, then (a) its *children* are candidates for the next iteration and (b) its *sibling* is a candidate for the iteration after the next one. For example, the *children* of BDF are BF and DF, i.e., all subsets of BDF containing one attribute less but the prefix (BD is not a child of BDF). The *sibling* of BDF is BDG, i.e., F is replaced by its successor G. If $|\mathcal{D}| = d$, it is shown in [45] that at most $2d-1$ iterations are needed for each attribute A_i to find the maximal X's that do not determine A_i. This explains the **While** loop in Line 3 of Algorithm 6. The correctness of the algorithm is already proven in [45].

Inferring Closed subspaces

The subsets returned by the previous procedure are *potentially* closed. Indeed, X is closed iff X does not determine any $A_i \in X$. It is not sufficient to make the intersection of these sets because, e.g., no such set X relative to A_i does contain A_i, still there may exist closed sets containing A_i. **ClosedSubspaces** exploits the previous results to infer the closed subspaces.

These algorithms have been implemented and turned to be very efficient to find rapidly the closed subspace as this is shown in Section 4.6 where we relate our experiments results.

4.4 Query evaluation

As a first solution to the partial materialization of skycubes, we propose to select all and only those skycuboids $Sky(T, X)$ such that $X^+ = X$, i.e., X is a closed subspace. We denote this set of skycuboids by **SkycubeC**(T).

Algorithm 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          if ∄Y ∈ Max st Y ⊇ X then
6              if T ⊭ X → A_i then
7                  Add X to Max;
8                  Remove the subsets of X from Max;
9                  Add the parent of X to Candidates[k+1];
10             else
11                 Candidates[k+1] ⊎ RightChildren(X);
12                 Candidates[k+2] ⊎ RightSibling(X);
13     k ← k+1;
14 Return Max;
```

Now, when a query $Sky(T, X)$ is submitted, we first compute the closure X^+ to find the materialized ancestor skycuboid from which the query is to be evaluated. For example, the query $Sky(T, A)$ is computed from $Sky(T, A^+) = Sky(T, ABD)$. As a matter of fact, $Sky(T, ABD) = \{t_1, t_2, t_3, t_4\}$. Hence, instead of computing $Sky(T, A)$ form T, thus using 6 tuples, we rely on $Sky(T, ABD)$ containing *only* 4 tuples. The closure of every X can either be hard coded or it can be computed on the fly by using the available set of FDs that have already been mined.

For the running example, Figure 4.7 represents the materialized part of the skycube (in red) as well as the closure relationships.

Let us analyze in more depth the query evaluation complexity. First, we recall that all algorithms proposed so far for evaluating a skyline query from a data set with n tuples have a worst case time complexity in $O(n^2)$ which reflects the number of comparisons. So we expect that by partially

Algorithm 7: ClosedSubspaces

Input: Table T
Output: Closed subspaces

```
1  for i = 1 to n do
2      L⁻ = MaxNFD(T, A_i);
3      L⁻ = SubsetsOf(L⁻, A_i);
4      if i = 1 then
5          Closed = L⁻;
6      else
7          Closed_i = {X ∈ Closed | X ∋ A_i};
8          Closed = (Closed ∩ L⁻) ⋃ Closed_i;
9  Return Closed;
```

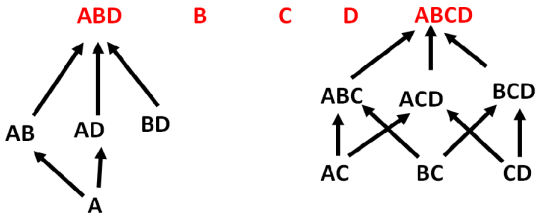


Figure 4.7 – The Partial Skycube

materializing a skycube, the cost of evaluating skyline queries should be less than $O(n^2)$. Suppose that the query is $Sky(X)$ and $Sky(X^+)$ is materialized. Therefore, evaluating $Sky(X)$ is performed with an $O(|Sky(X^+)|^2)$ time complexity. Is it possible that $|Sky(X^+)|$ be equal to n, which means that we get no gain in computing $Sky(X)$ from $Sky(X^+)$ rather than computing it from T? We show that, unless $X^+ = \mathcal{D}$, the cost is strictly smaller than $O(n^2)$ even if the size of $Sky(X^+)$ is equal to n.

4.4.1 Full Materialization

A special case of query evaluation is when we want to compute all skyline queries. This is equivalent to the full materialization of skycubes. To deal with this case and to avoid the naïve solution which consists in evaluating every skyline from T, previous works exhibit derivation properties and cases where computation sharing among skylines is possible so as to speed up this process.

Since this materialization turns to evaluate every possible skyline query, the previous properties we identified can easily be exploited in this context. Hence, we propose FMC (for Full Materialization with Closed subspaces) as a procedure for solving this problem. It is described in Algorithm 8.

Algorithm 8: FMC algorithm

Input: Table T
Output: Skycube of T
1 $Closed$ = **ClosedSubspaces**(T);
2 **foreach** $X \in Closed$ **do**
3 Compute $Sky(T, X)$;
4 **foreach** *subspace* X **do**
5 Compute $Sky(Sky(X^+), X)$;
6 **Return** $\bigcup_{X \in 2^{\mathcal{D}}} Sky(X)$;

FMC proceeds in three main steps : (i) it first finds the closed subspaces, then (ii) it computes their respective skylines from T, and finally, (iii) for every non closed subspace X, it computes $Sky(Sky(X^+), X)$.

The first advantage of FMC is that the input data used for every skyline over a non closed subspace is a subset of T. The second one is that it is easily amenable to a parallel execution. Indeed, both **foreach** loops in the algorithm (lines 2 and 4) as well as algorithm **ClosedSubspaces** (Line 1) can be executed in parallel to benefit from multi-processor machines that are the standard nowadays. Finally, FMC doesn't need to keep the whole skycube into RAM memory to achieve the computation since, by opposition to related work (see Section 4.5), no data structure sharing is required. Despite its simplicity, FMC turns to be very efficient in practice and outperforms state of the art algorithms as this is shown in Section 4.6.1.

4.5 Related Work

Several algorithms for computing skylines have been proposed in the literature. The complexity of most of them is analyzed in RAM cost setting, e.g., [37], [10], [58]. Some algorithms have been specifically tailored to the case where dimensions have low cardinalities, see e.g., [73]. All these algorithms have worst case complexity in $O(n^2)$ where n is the size (number of tuples) of the underlying table T. The pioneering work of [14] considered external memory cost and showed the inadequacy of SQL to evaluate efficiently skyline queries. However, the algorithms proposed there do *suffer* from the polynomial time complexity. Recently, [86] proposed an I/O aware algorithm guaranteeing, in the worst case, a polylog number of disk accesses. [84] proposed a randomized algorithm requiring $O(log(n))$ passes over the data to find an approximation of the the skyline with high probability. Other works make use of some pre-processing like multidimensional indexes. For example, [78] proposed R-trees as a convenient data structure to optimize a skyline points retrieval in a progressive way.

All progresses that can be made in single skyline query evaluation can benefit to multidimensional skyline queries. Our present work is then independent of any specific skyline algorithm.

When one is interested by multidimensional skylines, most of the previous works tackled the problem by considering the full materialization of skycubes [59], [58], [80], [100]. In order to avoid the naïve solution consisting in computing every skyline from the underlying table, they try to amortize some computations by using either (i) *shared structures* facilitating the propagation/elimination of skyline points of $Sky(X)$ to/from $Sky(Y)$ when $Y \subseteq X$ or (ii) *shared computation*. More precisely, the idea here consists in devising some derivation rules that help in finding $Sky(X)$ by using some parts of $Sky(Y)'s$. For example, if $Y \subset X$ then a tuple t_i cannot belong to $Sky(Y)$ if it doesn't match some t_j in $Sky(X)$, i.e., $t_j[Y] = t_i[Y]$. In order to take advantage of this property, one must keep $Sky(X)$ into memory which can become a real bottleneck when data are large. Note that several skylines must be kept in that way. In the next section, we compare experimentally FMC implementation and these state of the art algorithms and show that they do not scale with large dimensions and/or large data.

Due to the exponential number of skylines, some works tried to devise compression techniques [95],[82], thereby to reduce the storage space occupied by the entire skycube.

[82] proposed the closed skycube concept as a way to summarize skycubes. Roughly speaking, this method partitions the $2^d - 1$ skycuboids into equivalent classes. Two skycuboids are equivalent if their respective skylines are equal. Hence, the number of materialized skylines is equal to the number of equivalent classes. Once the equivalent classes are identified and materialized, query evaluation is immediate : for each query $Sky(T, X)$, return the skyline associated to the equivalence class of X. Given T, the size of **MICS** or even **SkycubeC(T)** and that of the closed skycube T of are incomparable in general. For instance, **SkycubeC(T)** of the running example requires the storage of 11 tuples while the closed skycube, illustrated in Figure 4.8, requires 12. Moreover, the experiments we conduct show that in practice **MICS** and **SkycubeC(T)** are in general computed much faster than the closed skycube as it is shown in Section 4.6.2. This shows that our proposal is a reasonable trade-off between skyline query optimization, storage space and the speed by which the solution is computed.

[95] proposed the Compressed SkyCube (CSC) structure. CSC can be described as follows. Let t be a tuple belonging to $Sky(X)$. Then t is in the *minimum* skyline of $Sky(X)$ iff there is no $Y \subset X$ such that $t \in Sky(Y)$. $min\_sky(X)$ denotes the minimal skyline tuples of $Sky(X)$. The compressed skycube consists simply in storing with every X the set $min\_sky(X)$. The authors show that this structure is losseless in that $Sky(T, X)$ can be recovered from the content of $min\_sky(Y)$ such that $Y \subseteq X$. More precisely, $t \in Sky(T, X)$ iff $\exists Y \subseteq X$ such that t belongs to $Sky(min\_sky(Y), X)$. Here

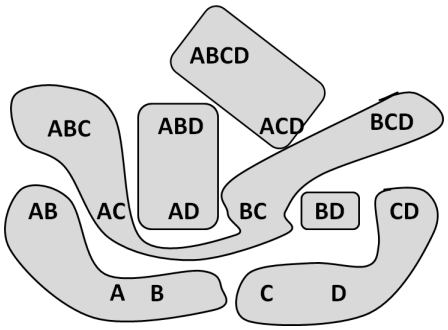


Figure 4.8 – The partition of the closed Skycube

again, the two solutions are incomparable in terms of space usage. For instance, the compressed skycube of the running example stores $\{B\langle t_1, t_2\rangle; D\langle t4\rangle; AD\langle t_3\rangle; A\langle t_1, t_2\rangle; C\langle t_4\rangle\}$, hence 7 tuples. This is not always true. For example, take a table T with just one tuple $t_1 = \langle A:1, B:1, C:1, D:1\rangle$. t_1 belong to the *min_sky* of A, B, C and D. It is then stored four times. Our solution stores $Sky(ABCD) = \{t_1\}$, hence t_1 is stored once.

Several works have been proposed in order to estimate the skyline size. [37] considers the same data distribution as the one we used (independence of dimensions) while [85] focuses on anti-correlated data. [18] doesn't consider any data distribution hypothesis but the estimator is not a closed formula. It is noticeable that all these works consider distinct value hypothesis, the cardinality of columns is equal to the number of rows. So these estimators cannot be used in situations where this hypothesis doesn't hold which is often the case in many practical situations.

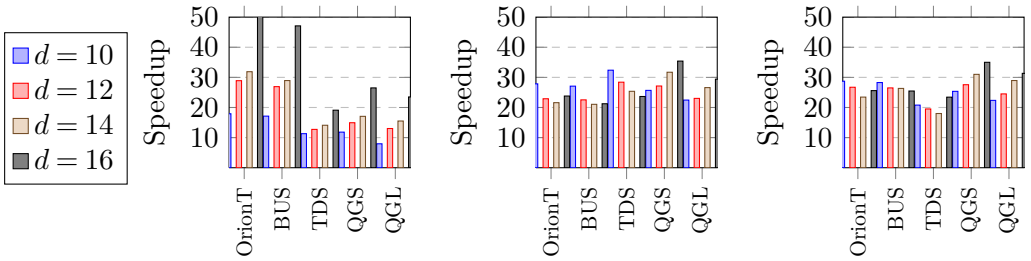
4.6 Experiments

We conduct a series of experiments aiming to illustrate the strengths and the weaknesses of our approach. For this purpose, we consider 3 directions : (i) We compare our solution to the previous works targeting the skycube full materialization. In this scope, we analyze the scalability w.r.t. both dimensionality and data size growth. It turns that our solution outperforms state of the art algorithms for full materialization when both data and dimensions get large ; (ii) we investigate the *summarization power* of our technique by taking into account the number of distinct values appearing in every attribute. As we shall see, this parameter has a great impact in the size of the returned summaries. From this perspective, we compare our solution to the *closed skycube* proposal of [82]. On real data, our solution appears to be both less space consuming than closed skycubes and it is obtained much faster ; (iii) finally, we analyze the impact of materialization in query cost reduction.

All experiments were conducted on a machine whose characteristics are : 24Gb of RAM, two 3.3 GHz hexacores processors, and a disk of 1Tb under Redhat Entreprise Linux OS. We implemented our solutions using `C++` language together with `OpenMP` API to benefit from parallelism.

We use both synthetic and publicly available real data sets. For synthetic data, we used the data generator software available at `pubzone.org` which follows the indications of [14]. It takes as input the values of n and d as well as a data distribution (correlated, independent or anticorrelated) and returns n tuples of d dimensions respecting the prescribed data distribution. The attributes values are real numbers normalized into $[0, 1]$ interval.

The values of d we consider in these experiments do not exceed 20 not because we cannot handle larger dimensions but this value was sufficient to show the scalability of our solutions. Note that



(a) Correlated (b) Independent (c) Anticorrelated

Figure 4.9 – Speedup w.r.t dimensionality d ($n = 100K$)

the number of dimensions reported in related works experiments rarely exceeds 10.

4.6.1 Full Skycube Materialization

In this section, we compare the execution time of FMC to state of the art algorithms, namely BUS and TDSG [81], OrionTail [82], as well as the most recent proposals QSkycubeGS and QSkyCubeGL reported in [59]. We used the authors implementations without any change [2]. All these algorithms take as input a table and return its respective skycube. They all are implemented in C++. For FMC, we make use of the BSkyTree implementation for computing skylines and presented in [58]. For every execution, we fixed the number of threads to 12 (number of available cores). Since all previous algorithms are sequential, and to make the comparison meaningful, we report the speedup of FMC over its competitors instead of their execution times. More precisely,

$$\text{Speedup}=\frac{(execution\,time\,of\,algorithm\,i)}{(execution\,time\,of\,\mathbf{FMC})}$$

If the speedup is greater than 12, we can safely conclude that FMC outperforms its competitor since its sequential execution cannot be twelve times slower than its parallel execution.

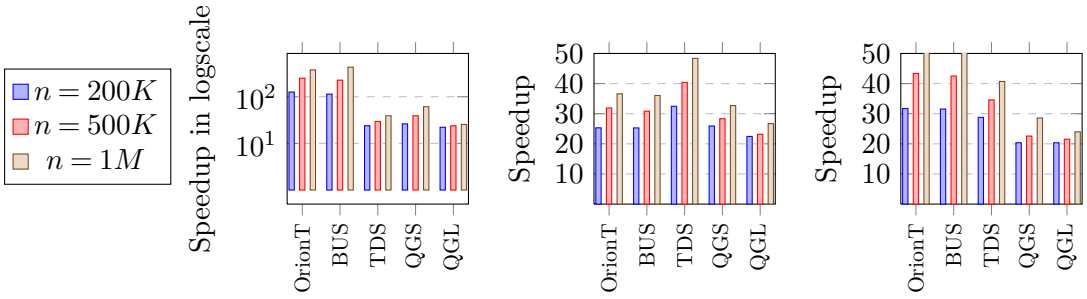
Scalability w.r.t Dimensionality

To analyze the effect of dimensionality growth, we fix n to 100K tuples (a relatively small value of n) and vary d from 10 to 16. We consider the three kinds of data correlations. The results are reported in Figures 4.9(a)-4.9(c). For example, 4.9(a) shows that FMC execution is *only* about 7 times faster than OrionTail when $d = 10$. A general remark is that while for the correlated data, the speedup increases uniformly with d, this is not the case for the other distributions. When $d = 16$, whatever is the data distribution (correlated, independent or anti-correlated), the speedup is greater than 12 for every implementation. This gives an evidence that FMC becomes more interesting when d increases.

Scalability w.r.t. Data Size

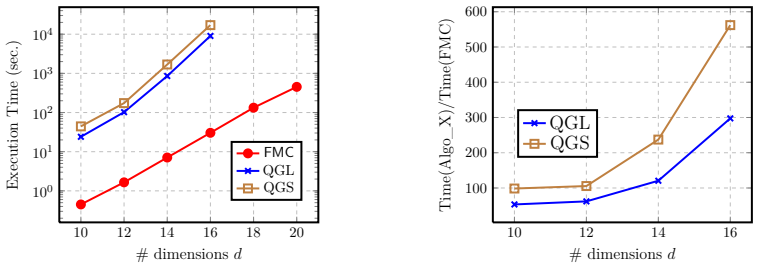
Here we fix d to 16 and vary the value of n by considering $200K$, $500K$ and $1M$ tuples. The results are reported in Figure 4.10. The experiments show that FMC outperforms all algorithms in all cases. Even though, it is interesting to note that QSkyCubeGL is the most scalable algorithm since the speedup of FMC over it does not exceed 13 in all cases.

2. We are grateful to the authors of these references who made available their software packages.



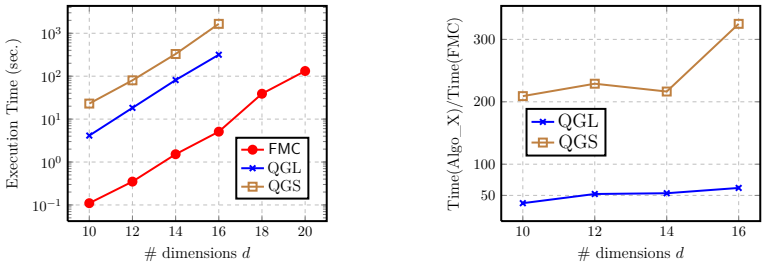
(a) Correlated (b) Independent (c) Anticorrelated

Figure 4.10 – Speedup w.r.t data size n ($d = 16$)



(a) Skycube Full Materialization (b) SpeedUp

Figure 4.11 – Real data set `USCensus` ($n = 2458285$)



(a) Skycube Full Materialization (b) SpeedUp

Figure 4.12 – Real data set `Householders` ($n = 2628433$)

NBA ($d = 17, n = 20493$)			IPUMS ($d = 10, n = 75836$)			Baseball ($d = 18, n = 92797$)		
FMC	QGL	QGS	FMC	QGL	QGS	FMC	QGL	QGS
0.22	8.15	4.57	0.45	2.27	10.24	4.5	63.85	267.1
Min Speed Up : 20.78			Min Speed Up : 5.04			Min Speed Up : 14.19		

Figure 4.13 – Execution times for small real datasets

Some Remarks on Synthetic Data Generation In the course of our experiments, we found that the synthetic data sets tend to satisfy the distinct values property, i.e., every dimension has almost n distinct values. This case tends to reduce the set of closed subspaces to just one, namely $\mathcal{D}$. Therefore, during the execution of FMC most of the skylines are evaluated from $Sky(\mathcal{D})$. In the case of correlated data, this is beneficial because the size of $Sky(\mathcal{D})$ is small compared to that of T. This is not the case with anti-correlated data. Nevertheless, as the previous experiments have shown, FMC is still competitive in that case, thanks to its parallel execution style.

Real Data Analysis

In order to avoid the biases introduced by the way synthetic data are generated, we performed the same kinds of experiments as before by using three real datasets, (i) the well known `US Census` data set used in machine learning ($n \sim 2.5 * 10^6$ and $d = 65$). All attributes are positive integer valued and even if not all of them have a meaningful ranking, still, the data set is interesting because of its real data distribution. We picked 20 columns at random for our experiments. (ii) The second data set is publicly available from the french INSEE institute website (statistics and economic institute) and describes `householders` in south-west region in France. It has more than $2.5 * 10^6$ tuples described by 67 variables. Here too, we picked 20 attributes but these columns have a meaningful ranking semantics (e.g., number of persons living in the house, number of rooms, etc ...). Although both data sets have almost the same number of tuples and the same dimensionality, their respective data distributions are radically different : while with `USCensus` data all subspaces are closed (there are no FD's), with `householders` the proportion of closed subspaces is very small (less than 2%) when $d \geq 10$. (iii) We also used three relatively small data sets that we find regularly in skyline literature, namely the `NBA` data set with about $20K$ tuples and 17 dimensions, the `IPUMS` set with around $100K$ tuples and 10 dimensions and the `Baseball` set with around $75K$ tuples and 18 dimensions. These sets give another insight about the performance of our proposal.

US Census data set Figure 4.11(a) shows the execution times needed by FMC, QSkyCubeGS and QSkyCubeGL to fully materialize the skycube by varying d from 10 to 20. We consider only these two competitors because the others took too much time. Nevertheless, we note that starting from $d = 16$ both QSkyCubeGS and QSkyCubeGL saturated the total 24 Gb of available memory and started to swap with disk during the computation. This why we stopped their execution otherwise it would have taken too much time. This shows the limit of data structure sharing of skycube full materialization techniques. We should mention that we took care to modify the original source codes of those algorithms so that as soon as a skycuboid is computed, its content is cleared. So memory saturation is not due to the size of the skycube but rather to the shared data structure, namely the `StreeCube`.

Figure 4.11(b) shows the speed up of FMC over its competitors. We note the rapid growth of the speed up when d increases reaching 3 orders of magnitude when $d = 16$ with QSkyCubeGL and almost 600 with QSkyCubeGS.

A specificity of this data set is that its dimensions have a very small number of distinct values : about 10 distinct values per each. This make functional dependencies hard to be satisfied while the number of tuples is quite large. In fact this data set doesn't satisfy any FD. Therefore all subspaces are closed. However, the topmost skyline, i.e., $Sky(D)$ contains *only* 3873 tuples. Moreover, these tuples have exactly the same values in every dimension ; $Sky(D)$ is of Type I. This makes the computation of any skyline very easy. This empirically confirms the relationship between the number of distinct values per dimension, the number of FD's and the size of the topmost skyline.

Householders data set The results are reported in Figure 4.12(a). Here again, both QSkyCubeGS and QSkyCubeGL were unable to handle the cases where $d \geq 16$. A noticeable difference however between this data set and the previous one is that the number of closed subspaces is quite small : 2880 out of the $2^{20} - 1$ subspaces. The number of distinct values per dimension varies from 2 to 4248. This shows the effectiveness of using the skylines of the closed subspaces to compute the rest of the skylines. Indeed, even if the speed ups we obtain are less impressive than those with the previous data set (see Figure 4.12(b)), FMC is still 50 times faster than QSkyCubeGL and even more comparatively to QSkyCubeGS.

Small real datasets With these quite small data sets (see Fig. 4.13), FMC is always faster than its competitors. Note however that when the number of dimensions is *small* which is the case with IPUMS dataset, the speed up is rather weak (only 5.04). Recall that we used 12 threads with FMC. This tends to show that our solution is rather more appropriate when the number of dimensions becomes large. Interestingly, we executed the naïve algorithm with IPUMS : for every subspace X, compute $Sky(T, X)$. This loop was executed using 12 threads too. The algorithm terminates after 0.68 seconds providing a speed up of 3.8 over QskycubeGL. So the question of when using shared data structures and/or computations is really worthwhile remains open.

Extending FMC

As the previous experiments have shown, the cost of first mining the closed sets in order to use their respective skylines for materializing the full skycube is amortized in general. Indeed, we found out that doing so, we are at least as efficient as state of the art algorithms and for some cases, FMC is orders of magnitude faster. Note that FMC does not use any optimization related to the sharing computation principle. Indeed, all skylines are computed independently from each others. This reduces memory consumption which is a severe bottleneck of previous proposals when data get large. For relatively small data, FMC can be optimized in the following way. Let $Cover(Y)$ denote the subspaces X such that $X^+ = Y$. Then instead of computing each $Sky(X)$ from $Sky(X^+)$, we reverse the loop by iterating over the closed subspaces as suggested in Algorithm 9.

Algorithm 9: FMC+ algorithm

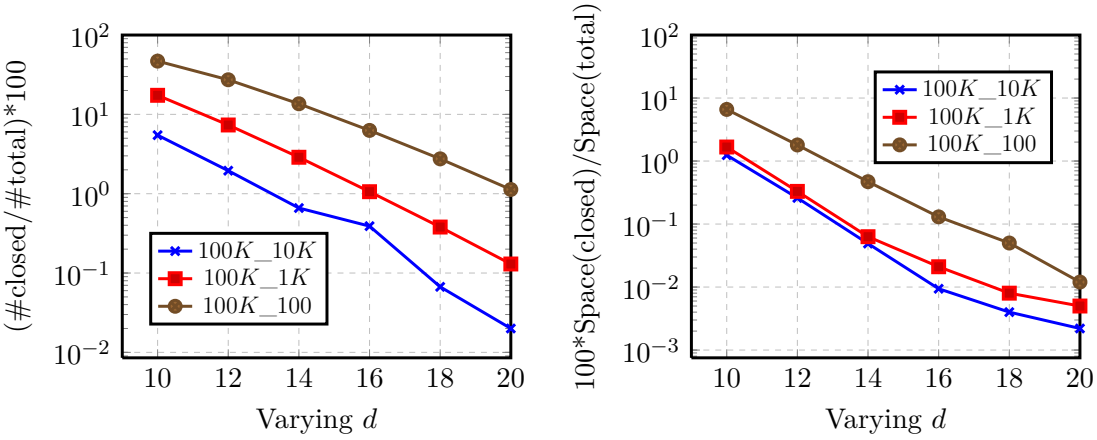
Input: Table T
Output: Skycube of T

```
1 Closed = Closed sets of attributes in T;
2 foreach X ∈ Closed do
3 |  Compute Sky(X) from T;
4 foreach closed subspace Y do
5 |  Compute the skylines of Cover(Y);
6 Return ⋃_{X∈2^D} Sky(X);
```

Since for every $Y_1, Y_2 \in Cover(Y)$ we have $Y_1 \subseteq Y_2 \Rightarrow Sky(Y_1) \subseteq Sky(Y_2)$, we can use for example a levelwise procedure in order to avoid unnecessary computations. Another solution consists in adapting the previous works that do use sharing computations.

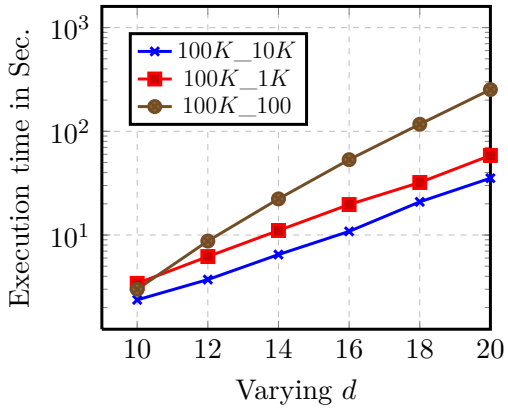
4.6.2 Storage Space Analysis

In the second part of the experiments, we generated a set of synthetic data by fixing the following parameters : d is the number of attributes, n is the number of tuples, c is the average number



(a) Percentage of materialized skycuboids

(b) Ratio of consumed storage space



(c) Running time for finding closed subspaces and materializing their respective skylines

Figure 4.14 – Quantitative Analysis of the Closed Subspaces

of distinct values taken by each attribute. The importance of this last parameter has already been identified, e.g., [73] by noting that dimensions with small number of distinct values deserve specialized skyline algorithms. For example, $100K\_10K$ designates a data set where $n = 100K$ and the number c of distinct values per dimension is $10K$. Since the data generator returns floats in $[0,1]$ interval, it suffices to keep the first f decimal digits to get a data set where every dimension has on average $c = 10^f$ distinct values. For example, 0.0123 is replaced by 12 if $c = 10^3$. Doing so, the correlation between the different columns is preserved. Since all the conclusions we derived are the same whatever is the kind of correlation, we report here only the results obtained with the independent data sets.

We firstly investigate the number of closed subspaces comparatively to the total number in the skycube. The results are reported in Figure 4.14(a). Some conclusions can be derived at this stage : (i) the proportion of closed subspaces decreases when the number of attributes increases. For example, consider the data set $100K\_10K$ when $d = 10$. There are about 7% of the subspaces out of the total $2^{10} - 1$ that are closed. This proportion falls to 0.035% when $d = 20$. The second lesson we may draw from this experiment is that (ii) the number of closed subspaces grows when the number of distinct values taken by each attribute increases. For example, when $d = 10$, only 7% of the subspaces are closed with $100K\_10K$ while there are 84.5% closed subspaces with $100K\_100$. This second case could indicate that the memory saving when storing only the skycuboids associated to closed subspaces is marginal. The second experiment (see Figure 4.14(b)) shows that this is not systematic. Here, we compute ratio between the total numbers of tuples that should be stored when only the closed subspaces are materialized over the the total number of skycube tuples. We see that in all cases, the memory space needed to materialize the skylines wrt the closed subpaces never exceed 10% of the whole skycube size. For example, even if we materialize 84.5% of the skylines of the skycube related to $100K\_100$ data set when $d = 10$, this storage space represents less than 10% of the related skycube size. This would indicate that either our proposal tends to avoid the materialization of *heavy* skycuboids, i.e., or the size of skylines tend to decrease when the number of closed subspaces increase. We empirically show that it is rather because of the second explanation (see Figure 4.15).

Finally, Figure 4.14(c) shows the execution times for the data sets we considered so far. We stress the fact that these times represent (i) the extraction of the closed subspaces and (ii) their respective skylines.

We conducted a second series of experiments aiming to show the evolution the the skyline size when both the cardinality c and the dimensionality d vary while the size of data n is kept fixed. The results are shown in Figure 4.15. We observe that the size of the skyline increases uniformly regarding c whatever is d. This gives a clear explanation of the behavior noticed in Figure 4.14(b), i.e., when c decreases, the number of closed subspaces increases while their respective sizes decrease.

Therefore, the main lesson we retain from the above experiments is that when the number of closed subspaces increases, the size of the skylines decreases. So, even if our solution stores more skylines, this doesn't mean necessarily that it uses more storage space. The second lesson we derive is that when c is small, every skycuboid tends to be small. Therefore, from a pragmatic point of view, materializing just the topmost skyline is sufficient to efficiently answer every skyline query by using Algorithm 5. This confirms the analytic study developed in Sections 4.3.3 and 4.3.4.

MICS vs Closed Skycubes : We compare our solution to the *Closed Skycubes* [82] in terms of (i) storage space usage and (ii) the speed for materializing the sub-skycube. We consider the three real data sets used in that reference : NBA ($n = 20493, d = 17$), IPUMS ($n = 75836, d = 10$) and MBL ($n = 92797, d = 18$). We compare the number of equivalence classes, i.e., the number of effectively stored skylines in the closed skycube and the number of skylines we store in the MICS.

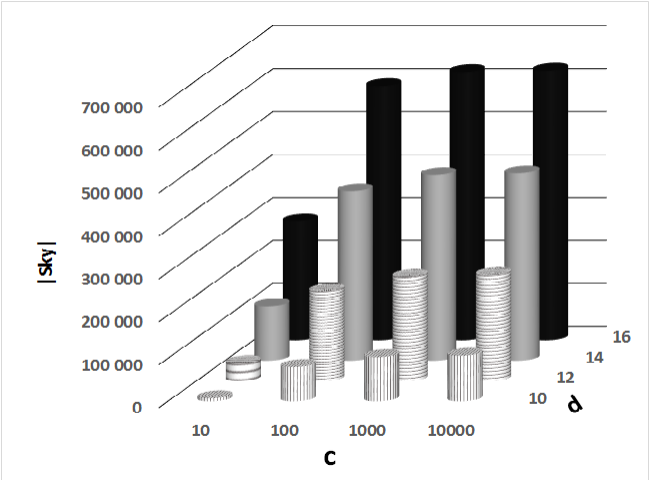


Figure 4.15 – Skyline size evolution wrt c and d with $n = 10^6$

	NBA	IPUMS	MBL
MICS	5304 (12.8 sec.)	11 (2.1 sec.)	29155 (172 sec.)
Closed	5304 (9 sec.)	738 (322 sec.)	43075 (11069 sec.)

Table 4.3 – Number of materialized skycuboids

For every technique, we also report the total computation time. The results are presented in Table 4.3.

In general, the MICS requires less skycuboids than closed skycubes. For **NBA** the solutions are identical. For **IPUMS**, we store 73 382 tuples, corresponding to 11 skycuboids, and the closed skycube requires 530 080. For **MBL**, we store 118 640 340 versus 133 759 420. The storage space ratio is not that large. However, our solution is often faster than its competitor. More precisely, the speedup ratio seems increasing with data size n. In addition to these data sets, we tried to test OrionClos with larger data sets but it was unable to terminate in a reasonable time. For example, after 36 hours, the correlated data set with $100K$ tuples and 20 attributes was not processed yet. Our implementation finds the closed subspaces and their respective skylines after 20 seconds. This is due to the fact that synthetic data generators tend to return distinct values making the computation of equivalence classes required by OrionClos harder.

For each data set, we count the number of skycuboids of Type 1 [3]. Note that computing a skyline from a skycuboid of Type 1 does not require any computation. For NBA all the 5304 are of Type 1, for IPMUS 4 out of 11, and for MBL 22960 out of 29155.

4.6.3 Query Evaluation

In this part, we analyze the efficiency of our proposal in terms of query evaluation time after the partial materialization of the skycube, i.e., once the skylines of the closed subspaces are computed. We use the `Householder` data set, vary d from 16 to 20, and vary n from $500K$ to $2M$. We generate 1000 distinct skyline queries among the $2^d - 1$ possible queries as follows : the $2^d - 1$ subspaces are listed and sorted w.r.t. a lexicographic order in a vector Q. We pick randomly and uniformly an integer number i lying between 1 and $2^d - 1$. $Sky(Q[i])$ is part of the workload if it does not

3. A skyline is of Type 1 iff all its elements are identical [82].

$n\backslash d$	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 4.4 – Query execution times in seconds : optimized/not optimized and (Proportion of materialized skycuboids)

correspond to a closed subspace. We repeat this process until the total number of distinct skyline queries reaches 1000. The obtained workload contains distinct queries of different dimensions. Each time we pick a value of i, the probability that it corresponds to a query with δ dimensions is $\frac{\binom{d}{\delta}}{2^d-1}$. The results are presented in Table 4.4. For every combination (n, d) we report three important information : (i) the total execution time when materialized skycuboids are used, (ii) the total execution time when skylines are evaluated from T and (iii) the proportion of skycuboids that are materialized. For example, when $n = 2M$ and $d = 20$, 0.049 seconds are sufficient to evaluate 1000 queries from the materialized skycuboids while it takes 99.92 seconds when T is used. The execution time is therefore divided by more that 2000. This performance is obtained by materializing *only* 0.31% out of the $2^{20} - 1$ skycuboids. What is remarkable is that in all cases, with a very small effort of materialization, the skyline queries are evaluated orders of magnitude faster from the materialized skycuboids than from T. We finally should mention that the overall partial skycube calculation takes only few seconds.

4.7 Conclusion and Future Work

In this paper we provided a solution for the partial materialization of skycubes. We showed how the classical concept of FDs may be used successfully to identify a *computability* relationship between skycuboids and then characterize those that should be materialized. This appears quite surprising because FDs are independent of the order used between the attributes values. In contrast, skyline queries are based on these orders. This shows the robustness of FDs. Our proposal represents akind of trade off between the size of storage memory space (we try to store as least as possible), the query execution time (we avoid using the whole data set) and the rapidity with which we obtain the partial skycube. Besides the theoretical results, we did our best to compare experimentally our proposal to state of the art implementations of related work when these were available. The conclusion is that our solutions scales better with data and dimensions growth. On real data sets, it turns that with a small portion of the skycube, we were able to gain orders of magnitude on query evaluation time.

We intend to investigate the incremental maintenance of the materialized skycuboids. When the insertions violate the FDs, the set of closed subspaces is updated. It is then interesting to come up with incremental solutions to discover the new closed subspaces and compute their content efficiently.

New directions for future work can be pursued thanks to the foundations we provide. For example, distributed skycubes have not been addressed so far. To extend our work to that setting, it is not clear whether we need global or local FDs. In the present work we identify the minimal

set of skycuboids to be materialized (MICS). To reduce further the skyline queries evaluation, it is tempting to materialize additional skycuboids. Investigating how given a storage space constraint, which is necessarily a multiple of the MICS size, we can find the best set of skycuboids satisfying the space budget constraint and minimizing query cost is one of our future research directions, or as we did with data cube, given a performance factor, provide a minimal set of skycuboids to materialize.

By contrast to FDs, the recently proposed class of *order* dependencies [90] do carry information about order. It is then tempting to investigate the usefulness of these dependencies for skycubes computation.

The results presented in this chapter appear in [68].

Chapitre 5

Conclusion générale

Dans ce manuscrit, nous avons présenté l'essentiel de nos contributions pour l'optimisation des requêtes d'analyse multidimensionnelles. La plupart des problèmes que nous avons abordés ont des complexités théoriques exponentielles en fonction du nombre de dimensions. Par exemple, extraire les dépendances valides à partir d'une table de 10^2 attributs et 10^3 tuples est beaucoup plus difficile que traiter le même problème à partir d'une table de 10^{10} tuples et 10 attributs. Or on entend souvent parler du problème du traitement des *big data* en terme de tailles mémoire mais moins en terme de dimensions. Ceci, à notre avis, peut parfois prêter à confusion surtout si des techniques d'échantillonnage éprouvées d'un point de vue statistique peuvent être appliquées pour réduire la taille. Ainsi, big data $\not\Rightarrow$ big problem.

Aussi, nous assistons ces dernières années à un foisonnement de systèmes spécialement conçus pour la manipulation de données volumineuses [1]. L'utilisateur néophyte a non seulement du mal à choisir lequel est le plus approprié pour son application mais a aussi tendance à croire que le système va résoudre, grâce a une certaine intelligence, son problème. Un cas typique de cette situation est ce que l'on rencontre avec un système tel que Hadoop implémentant le paradigme Map-Reduce. En effet, tel qu'il est généralement présenté, le système est capable à lui tout seul de distribuer les données, à synchroniser les traitement et à tenir compte des pannes éventuelles du système distribué. La seule chose que l'utilisateur doit fournir, ce sont les opérations de map et reduce et le système va se *débrouiller* avec. En réalité, les choses ne sont pas si simples que ça. Il suffit de voir par exemple les récents travaux d'Ullman *et al* (ex : [3]) sur l'analyse de la complexité, qui prend en compte le coût des communications, pour avoir une idée sur les limites de l'approche et de la nécessité de bien concevoir algorithmiquement les tâches successives des traitements. Un exemple simple que nous avons rencontré dans nos travaux est le test de validité d'un ensemble de dépendances fonctionnelles sur une table qui est distribuée horizontalement sur plusieurs nœuds. A priori, on pourrait penser que le fait de tester *en parallèle* sur des parties de la table pourrait être plus rapide que de faire le test en séquentiel sur la table entière. En réalité, ceci n'est pas toujours le cas à cause notamment des coûts de communication. Ainsi, faire du parallélisme rien que sur les données ne permet pas forcément d'avoir des garanties vis à vis de l'accélération du temps de traitement ; il faut que cela soit combiné avec du parallélisme de traitements.

Pour nos travaux futurs, nous comptons poursuivre nos investigations sur le traitement de données distribuées en considérant à chaque fois un type particulier de requêtes et voir dans quelle mesure celle-ci peut être optimisée. Par exemple, peu de travaux ont été élaborés pour le traitement des requêtes skyline à partir de données verticalement distribuées. Par exemple, un site qui décrit les chambres d'hôtel selon le prix et la proximité de la plage et un autre site qui les décrit selon la superficie et la disponibilité de wifi. Comment évaluer efficacement la requête qui cherche les meilleures chambres selon le prix et la superficie ?

1. Notamment avec l'émergence des systèmes dits NoSQL.

Un autre axe que nous n'avons pas abordé dans nos précédents est l'investigation des algorithmes dits *progressifs*. Il s'agit là d'algorithmes capables de retourner rapidement quelques premiers résultats mais qui au final peuvent être plus longs quant au résultat global. Dans certaines situations, l'utilisateur peut être intéressé par juste une partie du résultat surtout quand il débute l'exploration des données. En effet, un premier aperçu, même partiel, peut donner une indication sur l'orientation à prendre, choix des dimensions par exemple, pour poursuivre l'interaction avec les données.

Enfin, les données que nous avons considérées jusqu'à présent sont représentées sous forme tabulaire. Quand on veut intégrer des données à partir de différentes sources, une façon de les modéliser consiste à les voir comme des graphes étiquetés. Étendre nos travaux à des données du type graphe, notamment les données RDF, est une direction que nous comptons poursuivre dans nos travaux futurs.

Références bibliographiques

[1] Ziawasch Abedjan and Felix Naumann. Advancing the discovery of unique column combinations. In *Proc. of CIKM conf.*, 2011. 25

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. 60

[3] Foto N. Afrati, Anish Das Sarma, Semih Salihoglu, and Jeffrey D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. *PVLDB*, 6(4) :277–288, 2013. 85

[4] R. Agarwal, C. Aggarwal, and V. Prasad. Depth first generation of long patterns. In *SIGKDD Conf.*, 2000. 7

[5] Sameet Agarwal, Rakesh Agrawal, Prasad Deshpande, Ashish Gupta, Jeffrey F. Naughton, Raghu Ramakrishnan, and Sunita Sarawagi. On the computation of multidimensional aggregates. In *proc. of VLDB conf.*, 1996. 59

[6] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*. Morgan Kaufmann, 1994. 6, 27

[7] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *proc. of VLDB conf.*, 2000. 59

[8] K. Aouiche, J. Darmont, O. Boussaid, and F. Bentayeb. Automatic selection of bitmap join indexes in data warehouses. In *Proceedings of DaWaK*, 2005. 57

[9] Marcelo Arenas, Jonny Daenen, Frank Neven, Jan Van den Bussche, Martin Ugarte, and Stijn Vansummeren. Discovering xsd keys from xml data. In *SIGMOD*, 2013. 38

[10] Ilaria Bartolini, Paolo Ciaccia, and Marco Patella. Efficient sort-based skyline evaluation. *ACM Trans. Database Syst.*, 33(4), 2008. 73

[11] R. Bayardo. Efficiently mining long patterns from databases. In *SIGMOD Conf.*, 1998. 5, 7, 9

[12] L. Bellatreche and K. Boukhalfa. Yet another algorithms for selecting bitmap join indexes. In *Proceedings of DaWak*, 2010. 57

[13] Philip Bohannon, Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for data cleaning. In *Proceedings of ICDE conference*, pages 746–755. IEEE, 2007. 25

[14] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In *Proc. of ICDE conf.*, 2001. 3, 60, 73, 74

[15] Nicolas Bruno. *Automated Physical Database Design and Tuning*. CRC Press inc, 2011. 7

[16] D. Burdick, M. Calimlim, and J. Gehrke. MAFIA : A maximal frequent itemset algorithm for transactional databases. In *ICDE Conf.*, 2001. 5, 7, 10

[17] Upen S. Chakravarthy, John Grant, and Jack Minker. Logic-based approach to semantic query optimization. *ACM Trans. Database Syst.*, 15(2) :162–207, 1990. 60

[18] Surajit Chaudhuri, Nilesh N. Dalvi, and Raghav Kaushik. Robust cardinality and cost estimation for skyline operator. In *ICDE*, 2006. 74

[19] Dehao Chen, Chunrong Lai, Wei Hu, Wenguang Chen, Yimin Zhang, and Weimin Zheng. Tree partition based parallel frequent pattern mining on shared memory systems. In *Proceedings of IPDPS*, 2006. 9

[20] Rada Chirkova and Jun Yang. *Materialized views*, volume 4 of *Foundations and trends in databases*. Now publishing, 2012. 44

[21] Soon M. Chung and Congnan Luo. Efficient mining of maximal frequent itemsets from databases on a cluster of workstations. *Knowl. Inf. Syst.*, 16(3) :359–391, 2008. 9

[22] V. Chvàtal. A greedy heuristic for the set covering problem. *Mathematics of operation research*, 4(3) :233–235, 1979. 47, 48

[23] Paolo Ciaccia, Matteo Golfarelli, and Stefano Rizzi. On estimating the cardinality of aggregate views. In *DMDW*, 2001. 57

[24] Thierno Diallo, Noel Novelli, and Jean Marc Petit. Discovering (frequent) constant conditional functional dependencies. *International Journal of Data Mining, Modelling and Management*, 4(3) :205–223, 2012. 25

[25] Thomas Eiter and Georg Gottlob. Identifying the minimal transversals of a hypergraph and related problems. *SIAM J. Comput.*, 24(6) :1278–1304, 1995. 70

[26] Thomas Eiter, Georg Gottlob, and Kazuhisa Makino. New results on monotone dualization and generating hypergraph transversals. *SIAM J. Comput.*, 32(2) :514–537, 2003. 25

[27] M. El-Hajj and O. Zaïane. Parallel leap : Large-scale maximal pattern mining in a distributed environment. In *ICPADS Conf.*, 2006. 9

[28] Wenfei Fan, Floris Geerts, Jianzhong Li, and Ming Xiong. Discovering conditional functional dependencies. *IEEE Transactions on Knowledge and Data Engineering*, 23(5) :683–698, 2011. 6, 25

[29] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog : the analysis of a near-optimal cardinality estimation algorithm. In *Proceedings of AofA conference*, pages 127–146. DMTCS, 2007. 31, 36

[30] F. Flouvat, F. De Marchi, and J.M. Petit. A thorough experimental study of datasets for frequent itemsets. In *ICDM Conf.*, 2005. 20

[31] Frédéric Flouvat, Fabien De Marchi, and Jean-Marc Petit. A new classification of datasets for frequent itemsets. *Journal of Intelligent Information Systems*, 34(1) :1–19, 2010. 16

[32] Eve Garnaud, Nicolas Hanusse, Sofian Maabout, and Noël Novelli. Parallel mining of dependencies. In *Proceedings of HPCS conference*. IEEE, 2014. 38

[33] Eve Garnaud, Sofian Maabout, and Mohamed Mosbah. Using functional dependencies for reducing the size of a data cube. In *Proceedings of FoIKS conference*, pages 144–163. Springer, 2012. 57, 60

[34] Eve Garnaud, Sofian Maabout, and Mohamed Mosbah. Functional dependencies are helpful for partial materialization of data cubes. *Annals of Mathematics and Artificial Intelligence*, pages 1–30, 2014. 51, 57

[35] Amol Ghoting, Gregory Buehrer, Srinivasan Parthasarathy, Daehyun Kim, Anthony D. Nguyen, Yen-Kuang Chen, and Pradeep Dubey. Cache-conscious frequent pattern mining on modern and emerging processors. *VLDB J.*, 16(1) :77–96, 2007. 18, 23

[36] Parke Godfrey, Jarek Gryz, and Calisto Zuzarte. Exploiting constraint-like data characterizations in query optimization. In *SIGMOD Conference*, 2001. 60

[37] Parke Godfrey, Ryan Shipley, and Jarek Gryz. Algorithms and analyses for maximal vector computation. *VLDB Journal*, 16(1), 2007. 73, 74

[38] K. Gouda and M. Zaki. GenMax : An efficient algorithm for mining maximal frequent itemsets. *DMKD Journal*, 11(3), 2005. 5, 7, 10

[39] G. Grahne and J. Zhu. Fast algorithms for frequent itemset mining using FP-Trees. *IEEE TKDE journal*, 17(10), 2005. 5, 7

[40] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube : A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Mining and Knowledge Discovery*, 1(1) :29–53, 1997. 2, 39, 59

[41] D. Gunopulos, R. Khardon, H. Mannila, S. Saluja, H. Toivonen, and R.S. Sharm. Discovering all most specific sentences. *ACM TODS journal*, 28(2), 2003. 6, 7, 25

[42] Matthias Hagen. Lower bounds for three algorithms for transversal hypergraph generation. *Discrete Applied Mathematics*, 157(7) :1460–1469, 2009. 25

[43] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD Conf.*, 2000. 8

[44] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. Mining frequent patterns without candidate generation : A frequent pattern tree approach. *DMKD*, 8(1) :53–87, 2004. 6

[45] N. Hanusse and S. Maabout. A parallel algorithm for computing borders. In *Proceedings of CIKM'11 conference*. ACM, 2011. 38, 70

[46] N. Hanusse, S. Maabout, and R. Tofan. A view selection algorithm with performance guarantee. In *EDBT Conf.*, 2009. 6, 59

[47] Nicolas Hanusse, Sofian Maabout, and Radu Tofan. Algorithmes pour la sélection de vues à matérialiser avec garantie de performance. In *5èmes journées francophones sur les Entrepôts de Données et l'Analyse en ligne (EDA 2009), Montpellier*, volume B-5 of *RNTI*, pages 107–122, Toulouse, Juin 2009. Cépaduès. 57

[48] Nicolas Hanusse, Sofian Maabout, and Radu Tofan. Revisiting the partial data cube materialization. In *ADBIS conf.*, 2011. 57

[49] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In *SIGMOD '96 : Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 205–216, New York, NY, USA, 1996. ACM. 41, 43, 44, 45, 59, 66

[50] Arvid Heise, Jorge-Arnulfo Quiané-Ruiz, Ziawasch Abedjan, and Anja Jentzsch anFed lix Naumann. Scalable discovery of unique column combinations. In *Proc. of VLDB conference*, 2014. 26

[51] Arvid Heise, Jorge-Arnulfo Quiane-Ruiz, Ziawasch Abedjan, Anja Jentzsch, and Felix Naumann. Scalable discovery of unique column combinations. *PVLDB*, 7(4) :301–312, 2013. 25

[52] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. Tane : An efficient algorithm for discovering functional and approximate dependencies. *Computer Journal*, 42(2) :100–111, 1999. 6, 24, 70

[53] Howard Karloff and Milena Mihail. On the complexity of the view-selection problem. In *PODS '99 : Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 167–173, New York, NY, USA, 1999. ACM. 43

[54] Jyrki Kivinen and Heikki Mannila. Approximate inference of functional dependencies from relations. *Theoretical Computer Science*, 149(1) :129–149, 1995. 26

[55] Yannis Kotidis and Nick Roussopoulos. A case for dynamic view management. *ACM TODS*, 26 :2001, 2001. 44

[56] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *Journal of ACM*, 22(4) :469–476, October 1975. 3

[57] Laks V. S. Lakshmanan, Jian Pei, and Jiawei Han. Quotient cube : How to summarize the semantics of a data cube. In *VLDB*, pages 778–789, 2002. 51

[58] Jongwuk Lee and Seung won Hwang. BSkyTree : scalable skyline computation using a balanced pivot selection. In *Proc. of EDBT conf.*, 2010. 73, 75

[59] Jongwuk Lee and Seung won Hwang. Toward efficient multidimensional subspace skyline computation. *VLDB Journal*, 23(1) :129–145, 2014. 60, 73, 75

[60] Daniel Lemire, Owen Kaser, and Eduardo Gutarra. Reordering rows for better compression : Beyond the lexicographic order. *ACM Trans. Database Syst.*, 37(3), 2012. 64

[61] Eric Li and Li Liu. Optimization of frequent itemset mining on multiple-core processor. In *VLDB*, pages 1275–1285, 2007. 9, 18, 23

[62] Jingni Li, Zohreh Asgharzadeh Talebi, Rada Chirkova, and Yahya Fathi. A formal model for the problem of view selection for aggregate queries. In *ADBIS*, pages 125–138, 2005. 59

[63] Xiaolei Li, Jiawei Han, and Hector Gonzalez. High-dimensional olap : A minimal cubing approach. In *VLDB*, pages 528–539, 2004. 44

[64] D. Lin and Z. Kedem. Pincer search : A new algorithm for discovering the maximum frequent set. In *EDBT Conf.*, 1998. 7

[65] Jixue Liu, Jiuyong Li, Chengfei Liu, and Yongfeng Chen. Discover dependencies from data - a review. *IEEE Trans. Knowl. Data Eng.*, 24(2) :251–264, 2012. 25

[66] Stéphane Lopes, Jean-Marc Petit, and Lotfi Lakhal. Efficient discovery of functional dependencies and armstrong relations. In *EDBT Conf.*, 2000. 6, 70

[67] Stéphane Lopes, Jean-Marc Petit, and Lotfi Lakhal. Efficient discovery of functional dependencies and armstrong relations. In *Proceedings of EDBT conference*, pages 350–364. Springer, 2000. 24

[68] Sofian Maabout, Carlos Ordonez, Nicolas Hanusse, and Patrick Kamnang Wanko. Les dépendances fonctionnelles pour l'optimisation des requêtes skyline multi-dimensionnelles. In *Actes de la conférence BDA*, 2014. 83

[69] David Maier. *The Theory of Relational Databases*. Computer Science Press, 1983. 60, 66

[70] H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Mininig and Knowledge Discovery*, 1(3) :241–258, 1997. 2, 5, 6, 70

[71] Heikki Mannila and Kari-Jouko Räihä. *Design of Relational Databases*. Addison-Wesley, 1992. 26, 60, 66

[72] Muhammed Miah, Gautam Das, Vagelis Hristidis, and Heikki Mannila. Standing out in a crowd : Selecting attributes for maximum visibility. In *ICDE Conf.*, 2008. 6, 8

[73] Michael D. Morse, Jignesh M. Patel, and H. V. Jagadish. Efficient skyline computation over low-cardinality domains. In *Proceedings of VLDB conf.*, 2007. 73, 80

[74] S. Nedjar, A. Casali, R. Cicchetti, and L. Lakhal. Emerging cubes : Borders, size estimations and lossless reductions. *Information Systems*, 34(6), 2009. 6

[75] Noel Novelli and Rosine Cicchetti. Fun : An efficient algorithm for mining functional and embedded dependencies. In *Proceedings of ICDT conference*, volume 1973 of *LNCS*, pages 189–203. Springer, 2001. 6, 24, 35, 70

[76] OpenMP. www.openmp.org. 16

[77] Wim Le Page. *Mining Patterns in Relational Databases*. PhD thesis, University of Antwerp, 2009. 31

[78] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, 30(1), 2005. 73

[79] Odysseas Papapetrou, Wolf Siberski, and Wolfgang Nejdl. Cardinality estimation and dynamic length adaptation for bloom filters. *Distributed and Parallel Databases*, 28(2-3), 2010. 31

[80] Jian Pei, Wen Jin, Martin Ester, and Yufei Tao. Catching the best views of skyline : A semantic approach based on decisive subspaces. In *Proc. of VLDB conf.*, 2005. 60, 73

[81] Jian Pei, Yidong Yuan, Xuemin Lin, Wen Jin, Martin Ester, Qing Liu, Wei Wang, Yufei Tao, Jeffrey Xu Yu, and Qing Zhang. Towards multidimensional subspace skyline analysis. *ACM TODS*, 31(4) :1335–1381, 2006. 60, 63, 66, 75

[82] Chedy Raïssi, Jian Pei, and Thomas Kister. Computing closed skycubes. *Proc. of VLDB conf.*, 2010. 60, 61, 73, 74, 75, 80, 81

[83] Senjuti Basu Roy, Sihem Amer-Yahia, Ashish Chawla, Gautam Das, and Cong Yu. Constructing and exploring composite items. In *SIGMOD Conf.*, 2010. 6, 8

[84] Atish Das Sarma, Ashwin Lall, Danupon Nanongkai, and Jun Xu. Randomized multi-pass streaming skyline algorithms. In *Proceeding of VLDB*, 2009. 73

[85] Haichuan Shang and Masaru Kitsuregawa. Skyline operator on anti-correlated distributions. *PVLDB*, 6(9), 2013. 74

[86] Cheng Sheng and Yufei Tao. Worst-case i/o-efficient skyline algorithms. *ACM Trans. Database Syst.*, 37(4), 2012. 73

[87] Amit Shukla, Prasad Deshpande, and Jeffrey Naughton. Materialized view selection for multidimensional datasets. In *Proceedings of VLDB conference*, pages 488–499. Morgan Kaufmann, 1998. 41, 44, 45

[88] Yannis Sismanis, Paul Brown, Peter J. Haas, and Berthold Reinwald. GORDIAN : Efficient and scalable discovery of composite keys. In *Proc. of VLDB conf.*, pages 691–702, 2006. 25

[89] Dezhao Song and Jeff Heflin. Domain-independent entity coreference for linking ontology instances. *J. Data and Information Quality*, 4(2), 2013. 38

[90] Jaroslaw Szlichta, Parke Godfrey, and Jarek Gryz. Fundamentals of order dependencies. *Proc. of VLDB conf*, 2012. 83

[91] Zohreh Asgharzadeh Talebi, Rada Chirkova, Yahya Fathi, and Matthias Stallmann. Exact and inexact methods for selecting views and indexes for olap performance improvement. In *EDBT*, pages 311–322, 2008. 41, 44, 48

[92] W. Wang, J. Feng, H. Lu, and J.X. Yu. Condensed cube : An effective approach to reducing data cube size. In *Proceedings of ICDE conference*, pages 155–165. IEEE, 2002. 51

[93] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31(1), 2006. 64

[94] Catharine M. Wyss, Chris Giannella, and Edward L. Robertson. Fastfds : A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances. In *DaWaK Conf.*, 2001. 6, 24

[95] Tian Xia, Donghui Zhang, Zheng Fang, Cindy X. Chen, and Jie Wang. Online subspace skyline query processing using the compressed skycube. *ACM TODS*, 37(2), 2012. 60, 73

[96] X. Yan and J. Han. gSpan : Graph-based substructure pattern mining. In *ICDM Conf.*, 2002. 23

[97] G. Yang. The complexity of mining maximal frequent itemsets and maximal frequent patterns. In *SIGKDD Conf.*, 2004. 7

[98] H. Yao and H.J. Hamilton. Mining functional dependencies from data. *Data Mining and Knowledge Discovery*, 16(2) :197–219, 2008. 70

[99] Hong Yao and Howard J. Hamilton. Mining functional dependencies from data. *Data Min. Knowl. Discov.*, 16(2) :197–219, 2008. 24

[100] Yidong Yuan, Xuemin Lin, Qing Liu, Wei Wang, Jeffrey Xu Yu, and Qing Zhang. Efficient computation of the skyline cube. In *Proc. of VLDB conf.*, 2005. 60, 73

[101] X. Zeng, J. Pei, K. Wang, and J. Li. PADS : a simple yet effective pattern-aware dynamic search method for fast maximal frequent pattern. *KAIS Journal*, 20(3), 2009. 5, 7, 20, 35

[102] Yihong Zhao, Prasad Deshpande, and Jeffrey F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proceedings of SIGMOD conference*, pages 159–170. ACM Press, 1997. 59