Skycube Materialization Using the Topmost Skyline or Functional Dependencies

Sofian Maabout, University of Bordeaux
Carlos Ordonez, University of Houston
Patrick Kamnang Wanko, University of Bordeaux
Nicolas Hanusse, University of Bordeaux - CNRS

Given a table $T(Id, D_1, \dots, D_d)$, the skycube of T is the set of skylines w.r.t. to all non-empty subsets (subspaces) of the set of all dimensions $\{D_1, \dots, D_d\}$. In order to optimize the evaluation of any skyline query, the solutions proposed so far in the literature either (i) pre-compute all the skylines or (ii) they use compression techniques so that the derivation of any skyline can be done with little effort. Even though solutions (i) are appealing because skyline queries have optimal execution time, they suffer from time and space scalability because the number of skylines to be materialized is exponential w.r.t d. On the other hand, solutions (ii) are attractive in terms of memory consumption but, as we show, they also have a high time complexity. In this paper we make contributions to both kinds of solutions. We first observe that skyline patterns are monotonic. This property leads to a simple yet efficient solution for full and partial skycube materialization when the skyline w.r.t all dimensions, the *topmost* skyline, is *small*. On the other hand, when the topmost skyline is large relative to the size of the input table, it turns out that functional dependencies, a fundamental concept in databases, uncover a monotonic property between skylines. Equipped with this information, we show that *closed attributes sets* are fundamental for partial and full skycube materialization. Extensive experiments with real and synthetic data sets show that our solutions generally outperform state of the art algorithms.

CCS Concepts: •**Information systems** → **Data management systems; Database query processing; Query optimization;**

General Terms: Skyline, Functional dependencies, Algorithms, Performance

Additional Key Words and Phrases: Materialization, Implementation

ACM Reference Format:
Sofian Maabout, Carlos Ordonez, Patrick Kamnang Wanko and Nicolas Hanusse, 2015. Skycube Materialization Using the Topmost Skyline or Functional Dependencies. *ACM Trans. Datab. Syst.* V, N, Article XXXX (2015), 40 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Multidimensional database analysis has been a hot research topic during the last decade. Pre-computation is a common solution to optimize multidimensional queries. An early proposal of such approach is the so-called data cube [Gray et al. 1997] which, intuitively, represents the set of aggregation queries with all potential subsets of

This work has been carried out with financial supports from the French State, in the frames of the "Investments for the future" Programme IdEx Bordeaux - CPU (ANR-10-IDEX-03-02) and SPEEDDATA research project. It has also been partially supported by CNRS under the MASTODONS-PETASKY initiative. This work was initiated while the second author was visiting the University of Bordeaux in 2013.
Authors' addresses: S. Maabout, P. Kamnang Wanko and N. Hanusse, {maabout, pkamnang, hanusse}@labri.fr, LaBRI - University of Bordeaux, France; C. Ordonez, ordonez@cs.uh.edu, Department of cumputer science, University of Houston, Texas, USA.

DOI: http://dx.doi.org/10.1145/0000000.0000000

grouping attributes. After an initial series of works concentrating on efficient ways to fully materialize a data cube, see e.g., [Agarwal et al. 1996; Zhao et al. 1997], 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 w.r.t. the number of dimensions. Therefore, the question that is raised is 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. For example, [Harinarayan et al. 1996; Agrawal et al. 2000; Li et al. 2005] consider some budget, typically memory space, and try to find a part of the data cube that fits in the available space and minimizes the overall queries response time, whereas [Hanusse et al. 2009] considers the problem differently: given a prescribed queries evaluation time, find the minimal part of the data cube which guarantees that performance. More recently, *skyline* queries [Börzsönyi et al. 2001] were proposed to express multidimensional data ranking. Classically, users are required to provide a mapping function which assigns a numeric score to each tuple. Such score is then used to rank the query result and return the *best* tuples, e.g., *Top-K*. In some situations, it is not easy to define such mapping. Skyline queries prevent users from giving this function. Instead, they are required to provide a pre-order among the values of every attribute and specifying a preference among comparable values. The *best* tuples are those that are not *worse* than any other tuple on every attribute.

In the literature the *skycube* structure has been independently proposed in [Pei et al. 2005] and [Yuan et al. 2005]. This structure aims at helping users to navigate through the multidimensional space by asking skyline queries over chosen sets of attributes. In order to optimize the response times, several algorithms aiming at fully materializing the skycube, i.e., pre-compute all possible skylines, have been proposed. See e.g., [Pei et al. 2006] and [Lee and won Hwang 2014]. By contrast to data cubes, very few solutions have been proposed for partially materializing skycubes. To the best of our knowledge there are two such proposals: [Raïssi et al. 2010] introduced the concept *closed skycube*. The authors propose an algorithm for identifying equivalent skylines in order to save memory space by storing just one copy of the same *query* result. A second solution is the so called *compressed skycube* (CSC) structure proposed in [Xia et al. 2012]. Instead of reasoning at a whole skyline level as closed skycubes do, this technique operates at a tuple level, i.e., a tuple belonging to several skylines may not need to be stored together with every such skyline. We give more details about these two solutions in Section 6 devoted to the related work.

Outline of Contributions: In a nutshell, we propose novel algorithms to materialize skycubes (partially or fully), to efficiently process skyline queries on any subset of dimensions of a table T. When the skyline on all dimensions (topmost skyline) is *small* (much smaller than the input table), we show that by exploiting a monotonic property of skyline inclusion, this single skyline can be used to efficiently compute all other skylines. On the other hand, when the topmost skyline is *large* (relative to the size of the input table), we exploit a fundamental concept in database systems, namely functional dependencies (FDs), to accelerate skycube materialization. Specifically, we establish a connection between FDs discovered on the input table and the monotony of skylines. In other words, FDs help identifying important skylines inclusions. Using this knowledge, we introduce novel, but simple, algorithms to materialize the skycube, either fully or partially. Thereafter, this set of materialized skylines can be used to answer *all* remaining skyline queries without resorting to the underlying input table T. Along our study, we also identify a close connection between the number of FDs holding in a table and the size of skylines. More precisely, the less distinct values there exist

Table I. Hotels

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

per attribute, the less FDs (thus less detected inclusions), and the less the number of *distinct tuples* that may appear in every skyline (i.e., smaller projections). In particular, if there exist few FDs satisfied by the input table, then there is a high probability that the topmost skyline is small and its projections w.r.t subspaces are even smaller. Therefore, materializing the topmost skyline is sufficient for multidimensional skyline query optimization. Our solutions are implemented and experimentally compared to state of the art algorithms. For the skycube full materialization, we show that our algorithms outperform previous algorithms by orders of magnitude. On the other hand, for partial skycube materialization, we show that our solution (i) generally requires less storage space, (ii) it is faster to obtain and (iii) it significantly reduces query processing times. Moreover, our algorithms get asymptotically better than competing algorithms when the number of dimensions or the data size grow. Therefore, our solution is a good trade-off between fast and space-efficient skycube materialization and efficient query evaluation.

Paper organization. The following section gives the main definitions and notations used throughout the paper. Next we give a first solution of partial materialization of skycubes based on the topmost skyline. This solution uses a monotonic property of skyline patterns and is efficient only when the topmost skyline is small. Then, we provide a complementary solution based on functional dependencies. To do so, we show that the existence of functional dependencies implies an inclusion between skylines and we analyze the query evaluation from the materialized part of the skycube. We summarize our contributions by providing two algorithms: one for the materialization, full or partial, and the other for query evaluation. We compare our proposal to some related works and terminate by a series of experiments aiming to show the efficiency of our solutions. We conclude by giving some directions for future work.

2. PRELIMINARIES

2.1. Motivating Example

Let us first motivate skyline queries via a practical example. Consider the following Table I describing some characteristics of hotels rooms. These are described by their respective price, distance from the beach and size. A user may ask for the *best* rooms, i.e., those minimizing the price and the distance and maximizing the surface. Clearly, room r_3 does not belong to the set of best rooms because r_1 is better than it (we say r_1 *dominates* r_3) because it is closer to the beach, wider and not worse in terms of price. r_1 and r_2 are however *incomparable*: r_1 is better w.r.t. the price while r_2 is better in terms of distance. The skyline of T w.r.t. the three attributes P, D and S is the set $\{r_1, r_2\}$. Now, consider a rich tourist who does not care about the price. This user wants the best rooms by considering just the distance and the size of the rooms. In this case, r_2 is better than both r_1 and r_3 and thus, the skyline w.r.t. D and S is $\{r_2\}$. Note that the best rooms w.r.t. just the price is $\{r_1, r_3\}$. This example shows that depending on the attributes we consider (users preferences), we obtain different sets of *best* rooms. Moreover, we observe that there may be no inclusion relationships between these sets: neither the skyline w.r.t. P is included in that w.r.t. P, D and S nor the converse even if we have an inclusion of attributes sets, i.e., $\{P\} \subseteq \{P, D, S\}$. This *non monotony* of

Table II. Notations

Notation	Definition
T	Relational table
$\mathcal{D}$	Attributes/dimensions used for skylines
d	$\|\mathcal{D}\|$ number of dimensions
D_i, A_i	i^{th} dimension
n, m	number of tuples, size
$X, Y \ldots$	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
$\mathcal{S}(T)$ or simply $\mathcal{S}$	skycube of T
$\|Sky(X)\|$	the size of $Sky(X)$
S	a subskycube of $\mathcal{S}$
k	number of distinct values per dimension

skylines queries,also pointed in previous work [Pei et al. 2006; Xia et al. 2012; Pei et al. 2005; Yuan et al. 2005], makes their optimization harder than aggregation queries in the context of data cubes. More precisely, we cannot compute the skyline w.r.t. P from the pre-computed one w.r.t. P, D and S or vice versa without accessing the input table.

2.2. Definitions

We now introduce basic definitions used throughout the paper. Let T be a 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 *(multidimensional) space*. If $X \subseteq \mathcal{D}$, then X is a *subspace*. $t[X]$ denotes the projection of tuple t on subspace X. We denote by d the number of dimensions. Without loss of generality, for each $D_i \in \mathcal{D}$ we assume a total order $<$ between the values 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 w.r.t. $X \subseteq \mathcal{D}$ is defined as $Sky(T, X) = \{t \in T | \nexists t' \in T$ such that $t' \prec_X t\}$. To simplify notation and when T is understood from the context, sometimes we omit T and we use $Sky(X)$ instead. The *skycube* of T, denoted by $\mathcal{S}(T)$ or simply $\mathcal{S}$ is the set of all $Sky(T, X)$ where $X \subseteq \mathcal{D}$ and $X \neq \emptyset$. Formally, $\mathcal{S}(T) = \{Sky(T, X) \mid X \subseteq \mathcal{D}$ and $X \neq \emptyset\}$. Each $Sky(T, X)$ is called a *skycuboid*. d represents 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 II summarizes the different notations used throughout the paper.

Table III. The set of all skylines.

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	$\{t_4\}$		

Example 2.1. We borrow the toy table T from [Raïssi et al. 2010] and use it as our running example.

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

$Att(T) = \{Id, A, B, C, D\}$. 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 $\mathcal{D}$, i.e., the set of all skycuboids are depicted in Table III. This example shows that the skyline results are not *monotonic*, i.e., neither $X \subseteq X' \Rightarrow Sky(X) \subseteq Sky(X')$ nor $X \subseteq X' \Rightarrow Sky(X) \supseteq Sky(X')$ are true. For instance, $Sky(ABD) \nsubseteq Sky(T, ABCD)$ and $Sky(D) \nsupseteq Sky(AD)$. This makes partial materialization of skycubes harder than classical data cubes.

3. PARTIAL MATERIALIZATION OF SKYCUBES

The main goal of the present work is to devise a solution to the partial materialization of skycubes. The most important requirement is that the partial skycube should be *as small as possible* in order to minimize both its storage space and its computation time. Before formally stating the problem we address, we exhibit some properties holding between the subspace skylines of a skycube.

3.1. Skylines Patterns Monotony

Even if the skyline query is not monotonic, we can exhibit a monotonic property between $Sky(T, X)$ and $Sky(T, Y)$ whenever $X \subseteq Y$. Intuitively, this property is based on the following observation: for a tuple t to belong to $Sky(T, X)$ it is necessary and sufficient that there exists some tuple $t' \in Sky(Sky(T, Y), X)$, i.e., the skyline over X by considering only tuples in $Sky(T, Y)$, such that $t'[X] = t[X]$. This observation is stated in the following lemma.

LEMMA 3.1. *Let* $X \subseteq Y$. *Then* $\forall t \in T$ *we have that* $t \in Sky(T, X) \Leftrightarrow \exists t' \in Sky(Sky(T, Y), X)$ *s.t.* $t'[X] = t[X]$.

PROOF. We prove the two implications.

❶ $\Rightarrow$: Let $t \in Sky(T, X)$. If $t \in Sky(Sky(T, Y), X)$ then the statement is clearly satisfied. Assume now that $t \notin Sky(T, Y)$. Then there exists $t' \in Sky(T, Y)$ such that

$t' \prec_Y t$ but $t' \not\prec_X t$ because t belongs to $Sky(T, X)$. This implies that $t'[X] = t[X]$ so t' also belongs to $Sky(T, X)$. Therefore $t' \in Sky(Sky(T, Y), X)$.

❷ $\Leftarrow$: t' $\in$ Sky(Sky(T, Y), X) $\Leftrightarrow \forall u \in Sky(T, Y), u \not\prec_X t'$. On the other hand we have $\forall v \in T \setminus Sky(T, Y)$, $\exists u \in Sky(T, Y), u \prec_Y v$. Knowing that $u \not\prec_X t'$ and $u \preceq_X v$ because $X \subseteq Y$, we have that $v \not\prec_X t' \Leftrightarrow t' \in Sky(T, X)$. Since $t[X] = t'[X]$ then $t \in Sky(T, X)$.

□

Example 3.2. Consider the subspaces B and BC. As shown in Table III, $Sky(T, BC) = \{t_2, t_4\}$. From this set of tuple, we deduce $Sky(Sky(T, BC), B) = \{t_2\}$ (because $t_2[B] = 3 < t_4[B] = 4$) and this does not correspond to $Sky(T, B)$ since the later has actually an extra tuple which is t_1. Note however that (i) this missing tuple t_1 coincides with t_2 on B and (ii) it is the only one that does satisfy this property.

As a consequence, we obtain an inclusion relationship between the tuples belonging to $Sky(X)$ and those in $Sky(Y)$ whenever $X \subseteq Y$. More precisely,

THEOREM 3.3. *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 3.4. $Sky(A) = \{t_1, t_2\}$ 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 $\pi_A(Sky(AC))$=$\{\langle 1\rangle; \langle 4\rangle\}$.

Theorem 3.3 shows that the skyline points of every subspace skyline form a lattice: 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 3.5. Take the subspace B. $Sky(B)$ contains two tuples t_1 and t_2. The single pattern defining the tuples in $Sky(B)$ is the generalized tuple $\langle *, 3, *, *\rangle$. Figure 1 shows the patterns of the skycube of the running example.

In [Pei et al. 2005] the authors introduced the *skyline groups lattice* which is quite different from skyline patterns lattice. Indeed, the former is more targeting skyline membership queries, i.e., given tuple t, which are the subspaces X such that $t \in Sky(T)$. The set of these subspaces X is *summarized* by a set of upper and lower bound subspaces pairs and the tuples that share the same pair and same values in some dimensions are in the same group. For example, tuples t_1 and t_2 in the running example share the same values of A and B respectively. Both tuples belong to $Sky(AB)$, $Sky(A)$ and $Sky(B)$. They are then in the same group defined by the upper subspace AB and the lower subspaces A and B. More details about this structure and a comparison with the skyline patterns lattice are given in the related work Section 6.

Theorem 3.1 shows that whenever $X \subseteq Y$, the computation of $Sky(X)$ can benefit from the fact that $Sky(Y)$ is already materialized. Algorithm 1 shows how this can be done.

In simple terms, Algorithm 1 first eliminates the duplicates, w.r.t. X, from $Sky(Y)$ to obtain S_1. The second step consists of computing the skyline of S_1 by considering all its dimensions, i.e., X. This gives the set S_2 which is then joined with table T.

[1]We consider the canonical set semantics of the projection.

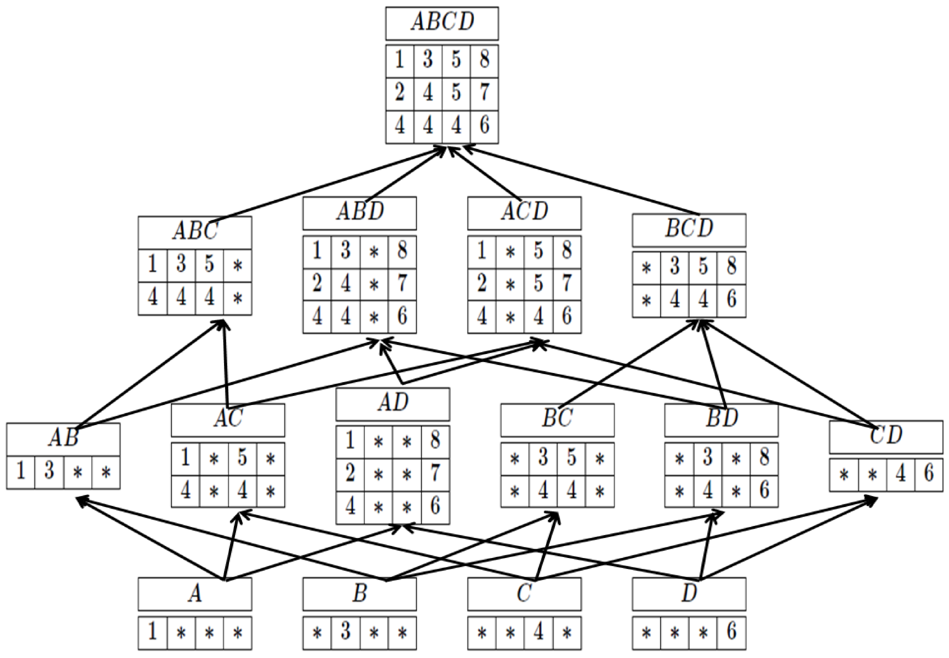


Fig. 1. Lattice of the skycube patterns

ALGORITHM 1: Sky_X_from_Sky_Y

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

Algorithm 1 is correct because since S_2 has no duplicates, we have the guarantee that a tuple $t' \in T$ can be returned at most once. Moreover, Theorem 3.1 guarantees that all $t' \in Sky(X)$ are retrieved. Hence, all and only $t' \in Sky(X)$ are returned.

Example 3.6. Suppose that $Sky(ABC) = \{t_2, t_4\}$ is already materialized and we want to compute $Sky(AB)$. Algorithm **Sky_X_from_Sky_Y** first projects $Sky(ABC)$ onto AB and obtains $S_1 = \{\langle 1,3\rangle; \langle 4,4\rangle\}$. Then line 2 of the algorithm returns $S_2 = Sky(S_1, AB) = \{\langle 1,3\rangle\}$. Finally (line 3), T is joined with S_2, i.e., we return those t' s.t $t'[AB] = \langle 1,3\rangle$. There are two such tuples: t_1 and t_2. Hence, $Sky(AB) = \{t_1, t_2\}$.

Time complexity of **Sky_X_from_Sky_Y** . The projection in line 1 can be performed in $O(|Sky(Y)|)$ using hashing. Line 2 (skyline computation) can be performed in $O(|S_1|.|S_2|)$, that is, every input tuple in S_1 is compared to every output tuple in S_2. The join operation between S_2 and T can be performed in $O(|T| + |S_2|)$ with, e.g., a hash-join algorithm: traverse $S_2 = Sky(S_1, X)$ and insert each t in a hash-table H, then traverse T, hash every $t'[X]$ and check in $O(1)$ whether the value belongs to H. If that is the case, then return t'. To sum up, the overall computation time is bounded by $O(|Sky(Y)|) + O(|S_1|.|S_2|) + O(|T| + |S_2|)$. Thus, if $|S_1|$ is close to $|T|$, we better perform $Sky(T, X)$ whose complexity is $O(|T|.|Sky(X)|)$.

ALGORITHM 2: SemiNaïvePartialSkyCube

```
Input: Table $T$
Output: $Sky(\mathcal{D}), T_{clean}$, index $\mathcal{I}$
1 Let $S_{\mathcal{D}} = Sky(T, \mathcal{D})$;
2 for every $t \in T$ do
3     for every $t' \in S_{\mathcal{D}}$ do
4         if $\exists D_i$ such that $t[D_i] = t'[D_i]$ then
5             insert $t$ into $T_{clean}$;
6             break;
7 create a bitmap index $\mathcal{I}$ on every $D_i \in T_{clean}$;
8 Return $S_{\mathcal{D}}$, $T_{clean}$, index $\mathcal{I}$;
```

3.2. Topmost Skyline Based Materialization

Theorem 3.1 allows us to derive an ad hoc solution for the partial materialization of skycubes. Indeed, it suffices to materialize $Sky(T, \mathcal{D})$, i.e., the skyline over all dimensions and use it to answer every submitted skyline query $Sky(X)$ by calling **Sky_X_from_Sky_Y**$(T, Sky(\mathcal{D}), X)$. This solution is particularly efficient whenever $Sky(\mathcal{D})$ is small, for instance $|Sky(\mathcal{D})| = O(\sqrt{n})$. This may happen when the dimensions are correlated and/or when there are few distinct values per dimension. Moreover, once $Sky(\mathcal{D})$ is computed, we know that every tuple $t \in T$ that does not match any $t' \in Sky(\mathcal{D})$ on any dimension can be removed from T because we are sure that it does not belong to any skycuboid. The next example illustrates this discussion.

Example 3.7. Suppose that we have four dimensions and each of them has three distinct values $\{0, 1, 2\}$. The number of possible values is then equal to $3^4 = 81$. If the number of tuples in T is n where $n \gg 81$, then there is a good chance that the tuple $\langle 0, 0, 0\rangle$ is present. If it is the case, then every tuple in the top most skyline must be equal to $\langle 0, 0, 0\rangle$. In this extreme situation, the size of $Sky(\mathcal{D})$ is equal to one whatever is the number of duplicates. Moreover, for every tuple $t \in T$ and $X \subseteq \mathcal{D}$, $t \in Sky(X)$ iff $t[X] = \langle 0, \ldots, 0\rangle$. In other words, if t has only values different from 0, then we can safely remove it from T: we know that it does not belong to any skyline. Hence, once $Sky(\mathcal{D})$ is computed, T can be *cleansed* so as to obtain $T_{clean} \subseteq T$ where T_{clean} contains only those tuples that have a chance to belong to at least one skyline.

The semi-naïve solution for skycube partial materialization is described in Algorithm 2. Cleansing T can actually be done in $O(n)$: for each dimension D_i, we create a hash table H_i corresponding to $\pi_{D_i}(Sky(\mathcal{D}))$. We obtain d hash tables. Then, we traverse T and we check for every $t \in T$ whether $t[D_i] \in H_i$. If that is the case, then t is added to T_{clean}. The complexity of this cleaning is in $O(dn)$ and if d is fixed then it is in $O(n)$.

In Line 8, we create a bitmap index on T_{clean} w.r.t. every D_i in order to optimize the join operation when skyline queries are submitted, i.e., line 3 of Algorithm 1. More precisely, every skyline query $Sky(X)$ is evaluated by calling **Sky_X_from_Sky_Y**$(T_{clean}, Sky(\mathcal{D}), X)$. Line 3 of Algorithm 1, i.e., $S_2 \bowtie T_{clean}$, is performed as follows: for every $t \in S_2$, select from T_{clean} those tuples t' satisfying $t[X] = t'[X]$. Thanks to the bitmap index $\mathcal{I}$, these tuples are retrieved efficiently.

When the skyline over $\mathcal{D}$ is large, materializing just the top most skyline together with an index is clearly inefficient. Indeed, the cost of the first step of query evaluation, i.e., $Sky(Sky(T, \mathcal{D}), X)$, is almost the same as that of evaluating $Sky(T, X)$. Therefore, and from a query optimization perspective, we need to precompute more skycuboids than just $Sky(\mathcal{D})$.

The next section addresses the problem of partial skycube materialization when $Sky(\mathcal{D})$ is large. Our objective will be to materialize a portion of the skycube sufficient to answer every skyline query *without making access to* the underlying data.

3.3. Minimal Information-Complete Subskycube (MICS)

Before giving the formalization of the addressed problem, we first give some definitions. We start with the *information-completeness* property of partial skycubes.

Definition 3.8 (*Information-Complete Subskycube*). Let S be a subskycube of $\mathcal{S}$. S is an *Information-Complete Subskycube* (ICS) iff for every subspace X, there exists a subspace Y such that $X \subseteq Y$, $Sky(Y) \in S$ and $Sky(X) \subseteq Sky(Y)$.

Intuitively, S is an ICS iff it contains a sufficient set of skycuboids which is able to answer every Skyline query. Recall that $Sky(T, X) \subseteq Sky(T, Y) \Rightarrow Sky(T, X) = Sky(Sky(T, Y), X)$. In other words, $Sky(X)$ can be evaluated using $Sky(Y)$ instead of T.

Example 3.9. 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. If for example, S_1 is materialized, then the query $Sky(T, A)$ can be evaluated as $Sky(Sky(ABD), A)$ using 4 tuples instead of using table T which contains 6 tuples. Note that $\mathcal{S}_3 = \{Sky(ABCD)\}$ is not an ICS because, e.g., there is no superset Y of ABD such that $Sky(ABD) \subseteq Sky(Y)$.

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

Definition 3.10 (*Minimal Information Completeness*). S is a *minimal* ICS (MICS) iff there exists no other ICS S' such that $S' \subset S$.

Example 3.11. $S_1 = \{Sky(ABCD), Sky(ABD)\}$ 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 $\mathcal{D}$, find an MICS of T that will be materialized in order to answer all the skyline queries over subsets of $\mathcal{D}$.

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

PROPOSITION 3.12. *Given T, there is a unique MICS of T.*

PROOF. Consider the directed graph $G = (V, E)$ where $V = 2^{\mathcal{D}}$ and $E \subseteq V \times V$ such that $(X, Y) \in E$ iff $X \subset Y$ and $Sky(X) \subseteq Sky(Y)$. Let Σ be the set of nodes without any outgoing edge. Then clearly $S = \{Sky(X) | \; X \in \Sigma\}$ is the unique MICS. □

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

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

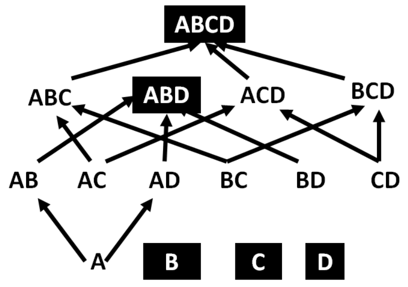


Fig. 2. Inclusions between skycuboids captured by FDs

3.4. Using FDs to discover Inclusion between Skylines

Recall that the functional dependency $X \to Y$ is satisfied by T iff for every pair of tuples $t_1, t_2 \in T$, if $t_1[X] = t_2[X]$ then $t_1[Y] = t_2[Y]$. The following theorem represents our main result. It shows how an existing functional dependency can be used to identify a monotonic inclusion between two skylines.

THEOREM 3.13. *Let $X \to Y$ be an FD satisfied by T. Then $Sky(T, X) \subseteq Sky(T, XY)$.*

PROOF. Suppose that the claim of the theorem is not true and let $t \in Sky(T, X) \setminus Sky(T, XY)$. Since $t \notin Sky(T, XY)$, there must exist t' which dominates t w.r.t. XY, i.e., $t' \prec_{XY} t$. t' cannot dominate t w.r.t. X because otherwise t cannot belong to $Sky(T, X)$. We conclude that $t[X] = t'[X]$. On another hand, $X \to Y$ is satisfied by hypothesis and from $t[X] = t'[X]$ we derive that $t[Y] = t'[Y]$ and thus $t[XY] = t'[XY]$ which contradicts the fact that t' dominates t w.r.t. XY. $\square$

Example 3.14. Turning back to the running example, the set of minimal functional dependencies satisfied by T are

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

The inclusions of skylines identified by the FDs satisfied by T are depicted in Figure 2. Each node X represents $Sky(T, X)$. An edge from X to Y represents an inclusion. Paths also represent inclusions. For example, we can see that $Sky(T, A) \subseteq Sky(T, AB) \subseteq Sky(ABD)$. One should notice that these are only the inclusions we can deduce from the FDs satisfied by T. There may be other inclusions that are not detected by FDs. For example, $Sky(C) = \{t_4\} \subseteq Sky(BC) = \{t_2, t_4\}$, and this inclusion is not captured by the FDs.

The *distinct values* property has been used in previous work [Xia et al. 2012; Pei et al. 2006; Lee and won Hwang 2014] to optimize the skycube computation. More precisely, T satisfies the distinct values property iff $|\pi_{D_i}(T)| = n$. In other words, all the values appearing in each dimension are distinct. The following result shows that when T satisfies this property, it guarantees that skylines are monotonic.

THEOREM 3.15. *[Pei et al. 2006] If T satisfies the distinct values property then for every subspaces X and Y, the following implication holds: $X \subseteq Y \Rightarrow Sky(X) \subseteq Sky(Y)$.*

The above result can be seen as a consequence of Theorem 3.13. Indeed, under the distinct values hypothesis, every single attribute determines all other attributes: it is a *key* in FDs terminology. In particular, $\forall X, Y$ we have $X \to Y$. By theorem 3.13 we deduce that $Sky(X) \subseteq Sky(XY)$.

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

PROPOSITION 3.16. *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 directed graph where $V = 2^{\mathcal{D}}$ and $\mathcal{E} \subseteq V \times V$ such that $(X, Y) \in \mathcal{E}$ iff $Sky(X) \subseteq Sky(Y) \in \mathcal{I}$. Let $\Gamma = \{X \in V \mid X$ has no outgoing edge$\}$. Then Γ is an ICS.*

PROOF. It suffices to show that Γ includes the unique minimal ICS. Since the set of inclusions $\mathcal{I}$ is a subset of the actual inclusions, a node in the graph G, as defined in the proof of Proposition 3.12, has no outgoing edge only if it has no outgoing edge in $\mathcal{G}_{\mathcal{I}}$. This shows that Γ contains the MICS. □

As a consequence of the previous proposition, we can conclude that having Γ is sufficient to infer the MICS. For example, in Figure 2, the nodes in a black box form Γ. For example, both B and ABD belong to Γ. From Table III, one can see that $Sky(B) \subseteq Sky(ABD)$. Therefore, $Sky(B)$ is removed from Γ.

Now, we provide some properties of the elements of Γ that allow to identify them efficiently. By Theorem 3.13, we conclude that a subspace X belongs to Γ iff there is no subspace Y such that $X \cap Y = \emptyset$ and the FD $X \to Y$ 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 [Maier 1983; Mannila and Räihä 1992] for more details.

Definition 3.17 (*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 $F \vdash X \to Y$ where $\vdash$ represents the *implication* between FDs. X is *closed* iff $X^+ = X$.

Example 3.18. Consider our running example. A is not closed because there exists another attribute determined by A. For example, T satisfies $A \to D$. ABD is closed because the only remaining attribute is C and T does not satisfy $ABD \to C$.

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. To the best of our knowledge, no algorithm has been proposed so far in the literature for addressing the problem of finding the closed attributes sets. Previous research proposed rather efficient algorithms for computing the closure of a set $\mathcal{F}$ of FDs, i.e., all FDs that are logically implied by $\mathcal{F}$. Interestingly, it is shown that this closure can be computed in linear time w.r.t. the size of $\mathcal{F}$ [Mannila and Räihä 1994]. The description of the algorithm we propose for this purpose is postponed to Appendix A.

[Previous Section 3.8] Now we can describe the procedure which, given a table T, returns its corresponding MICS. It is depicted in Algorithm 3. It first finds the closed subspaces, then it computes their respective skylines. The third step consists in removing every subspace X whose skyline is included in that of some Y provided that $X \subset Y$.

Example 3.19. Recall that the closed subspaces of the running example are $ABCD, ABD, B, C$ and D. Algorithm 3 first computes their respective skylines. Then it discovers the inclusions $Sky(B) \subseteq Sky(ABD)$, $Sky(C) \subseteq Sky(ABCD)$, $Sky(D) \subseteq$

ALGORITHM 3: MICS

```
Input: Table $T$
Output: MICS
1  Let $ClosedSub = $ ClosedSubspaces($T$);
2  for every $X \in ClosedSub$ do
3      // Loop executed in parallel
4      compute $Sky(T, X)$;
5  for every $X \in ClosedSub$ do
6      for every $Y \in ClosedSub$ s.t $X \subset Y$ do
7          if $Sky(T, X) \subseteq Sky(T, Y)$ then
8              $ClosedSub = ClosedSub \setminus \{X\}$;
9              Break;
10 Return $ClosedSub$ and their respective skylines;
```

$Sky(ABD)$ and $Sky(D) \subseteq Sky(ABCD)$. Hence, only $Sky(ABCD)$ and $Sky(ABD)$ belong to MICS. This is the minimal set of skycuboids that must be materialized.

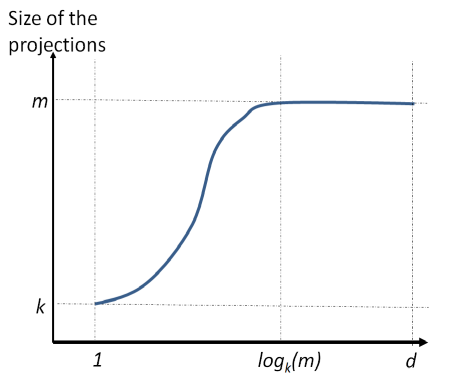
3.5. Analysis of the number of closed subspaces

Obviously, the more dependencies are satisfied by table T, the less there are closed subspaces. The number of distinct values per dimension is an important parameter that has been identified in the FD mining literature in that it impacts the number of valid FDs[2]. Intuitively, when this parameter is large, statistically it becomes hard to find two tuples sharing the same value on X and different values on Y making $X \rightarrow Y$ violated. To illustrate, suppose that each dimension has n distinct values, i.e., the number of rows in T. In this extreme case, it is impossible to violate any FD. As another extreme case, suppose now that every dimension has only two distinct values. If n is large enough, i.e., $n \gg 2^d$ then for every X, Y there is a high probability that there exist two tuples sharing the same value on X and different values on Y making $X \rightarrow Y$ violated. This intuitive discussion can be formalized under some data distribution hypothesis.

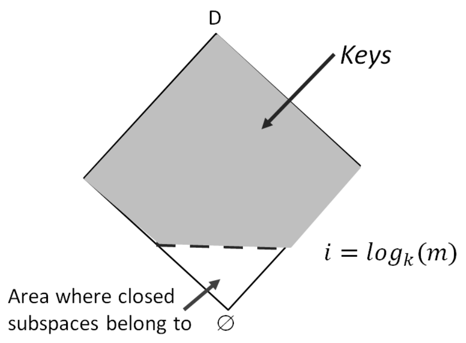
Let $||X||$ denote the number of attributes in X, for example $||ABC|| = 3$, and let k be the number of distinct values of every dimension and m be the number of distinct tuples in table T. X is a key of T iff $|X| = m$. Clearly, if X is a key then every Y such that $Y \supseteq X$, Y is not closed (apart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

Assuming a uniform distribution, if $||X|| \geq \log_k(m)$ then X has a high probability for being a key [Harinarayan et al. 1996]. Moreover, since $X \supseteq Y \Rightarrow |X| \geq |Y|$, the probability for X to be a key increases when $||X||$ increases. We conclude that: the larger k, the smaller $||X||$ making X to be a key, the larger the number of supersets of X and the fewer the closed subspaces. Figure 3(a) shows the evolution of projection sizes w.r.t. the number of dimensions: when this number reaches $log_k(m)$ the projections reach the same number of distinct 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 3(b). Note that the area where closed subspaces belong to may contain non closed subspaces too.

[2]It is often called *correlation factor* in the FD mining literature.



(a) Size of projections w.r.t. dimensionality $||X||$



(b) Closed subspaces in the subspaces lattice

Fig. 3. Keys and closed subspaces behaviors

3.6. Analysis of Skyline Size

When the cardinality k of dimensions decreases, the number of closed subspaces increases and may reach $2^d - 1$. From partial materialization point of view, this is a bad situation: all skylines are materialized. In fact, 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 the one for the topmost subspace, i.e., $Sky(D)$. This fact is sufficient to answer all skyline queries efficiently using Algorithm 1 as we described it in Section 3.2. The following theorem formalizes this result.

THEOREM 3.20. *Let n and d be fixed. Let $\mathscr{T}_k$ denote the set of tables with d independent dimensions, n tuples and at most k distinct values per dimension uniformly distributed. Let S_k denote the average number of distinct tuples int the skylines of all tables $T_k \in \mathscr{T}_k$. Then we have that: $k \leq k' \Rightarrow S_k \leq S_{k'}$*

PROOF. See Appendix B for a detailed proof. □

In other words, the above theorem states that for fixed n and d, the size of the skyline (in terms of the number of distinct tuples) tends to increase when the number of distinct values per dimension grows. The following example gives a more intuitive illustration.

Example 3.21. For k and d, the number of possible tuples is k^d. Let $n = 5$ and $d = 2$. When $k = 2$, we are sure that there are duplicates among the 5 tuples. Moreover, there is a high probability that the tuple $\langle 0, 0\rangle$ is present and in this case, this is the unique distinct tuple in the skyline. Consider now the case where $k = 4$. Here the number of possible tuples is 16. Choosing 5 tuples among these lets small chances that $\langle 0, 0\rangle$ does appear. Hence, the number of tuples in the skyline tends to be larger than 1.

Figure 4 illustrates how both the number of closed subspaces and skyline size evolves w.r.t. the number of distinct values k when $n < k^d$. The main conclusion is that when the number of closed subspaces increase, the size of skyline tends to decrease. This is why in case of no FDs, we are almost sure that the topmost skyline is small.

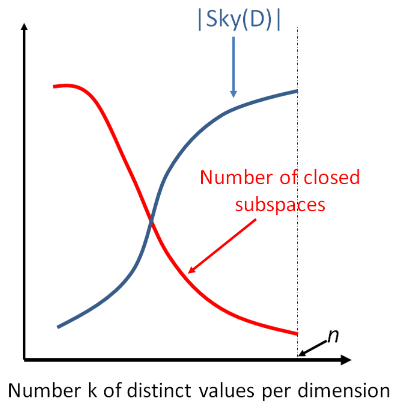


Fig. 4. The evolution of skyline size and number of closed subspaces w.r.t. cardinality k

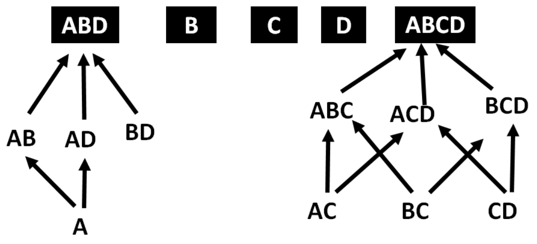


Fig. 5. The Partial Skycube

4. QUERY EVALUATION

In this section we show how skyline queries are efficiently evaluated when the MICS is materialized. To simplify presentation, we assume that materialized skycuboids are those associated to X, where X is closed. That is, this is a superset of the MICSs, and is denoted **SkycubeC**(T).

4.1. Evaluating $Sky(Sky(X^+), X)$

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 5 represents the materialized part of the skycube (nodes in black boxes) 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 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.

4.2. Using Partitions

In this section, we exhibit some properties that can be derived from those FDs holding in T for optimizing skyline query evaluation. These properties are based on skyline partitions.

Definition 4.1 (*Partition*). Let Y be a subspace such that $Y^+ = X$. Then $\mathcal{P}_Y(X) = \{p_1, \ldots, p_m\}$ denotes the partition of the tuples in $Sky(X)$ w.r.t their Y values, i.e., $t_i \equiv t_j$ iff $t_i[Y] = t_j[Y]$. Let $\tau_i \in p_i$, then $[\mathcal{P}_Y(X)] = \cup_{i=1\ldots m}\{\tau_i\}$ is a *representative* of $\mathcal{P}_Y(X)$.

Example 4.2. $A^+ = ABD$ and $Sky(ABD) = \{t_1, t_2, t_3, t_4\}$. $\mathcal{P}_A(ABD) = \{\{t_1, t_2\}; \{t_3\}; \{t_4\}\}$. $[\mathcal{P}_A(ABD)] = \{t_1, t_3, t_4\}$ and $[\mathcal{P}_A(ABD)] = \{t_2, t_3, t_4\}$ are both representatives of $\mathcal{P}_A(ABD)$.

The following lemma shows that computing $Sky(Y)$ can be evaluated from $[\mathcal{P}_Y(X)]$ instead of $Sky(X)$.

LEMMA 4.3. *Let Y be such that $Y^+ = X$ and let $[\mathcal{P}_Y(X)]$ be some representative. Let $t_i \in [\mathcal{P}_Y(X)]$ and t'_i be a tuple in $Sky(X)$ such that $t_i[Y] = t'_i[Y]$. Then $t'_i \in Sky(Sky(X), Y)$ iff $t_i \in Sky([\mathcal{P}_Y(X)], Y)$.*

In other words, it suffices to evaluate $Sky(\pi_Y(Sky(X)), Y)$ and if t_i is in the result then all of its equivalent tuples in $Sky(X)$ belong to $Sky(Y)$ too.

Example 4.4. Suppose that for evaluating $Sky(A)$ the representative $[\mathcal{P}_A(ABD)] = \{t_2, t_3, t_4\}$ is used. This means that query $Sky([\mathcal{P}_A(ABD)], A)$ is evaluated and it returns $\{t_2\}$. Since t_1 is equivalent to t_2 then t_1 is also in $Sky(A)$.

The above lemma would suggest that for each query $Sky(Y)$, we should partition $Sky(X)$ w.r.t. Y. In fact, it is much easier than that because it suffices to partition once the tuples of $Sky(X)$ w.r.t X and this partition is exactly the same as those w.r.t every Y such that $Y^+ = X$.

PROPOSITION 4.5. *Let $Y^+ = X$. Then $\mathcal{P}_Y(X) = \mathcal{P}_X(X)$.*

PROOF. Because $T \models X \to Y$ iff $\mathcal{P}_X(T) = \mathcal{P}_{XY}(T)$. □

For example, since $A^+ = ABD$ the partitions $\mathcal{P}_A(ABD)$ and $\mathcal{P}_{ABD}(ABD)$ are equal. The above result shows that the complexity of retrieving a non materialized skycuboid does not depend on the number of tuples in its materialized ancestor but rather on the size of its partition. Note however Proposition 4.5 does not completely answer our previous question since a priori, the number of parts could be equal to $|Sky(X)|$ and since $|Sky(X)|$ could itself be equal to $|T|$ then we would have no gain by using $Sky(X)$ instead of T. The following proposition shows that in fact this could happen in only one case, namely when $X = \mathcal{D}$. More precisely,

PROPOSITION 4.6. *Let X be a closed subspace and $\mathcal{P}_X(X)$ be the partition $\{p_1, \ldots, p_m\}$. If $X \neq \mathcal{D}$ then $m < |T|$.*

PROOF. Suppose that the number of parts is equal to $|T| = n$. This means that there are n distinct values when the tuples in $Sky(X)$ are projected onto X. Therefore, X is a key of T. Hence, $X = \mathcal{D}$, which contradicts the hypothesis. □

In [Raïssi et al. 2010], a special class of skycuboids is identified, namely the class of skycuboids of *Type I*. The authors use this class for inferring the content of other skycuboids. By Lemma 4.3 we infer some skylines without any computation when skycuboids of Type I are used. Let's first recall the definition of skycuboids of Type I.

Definition 4.7. $Sky(X)$ is of Type I iff $\forall t_1, t_2 \in Sky(X)$, $t_1[A_i] = t_2[A_i]$ for every $A_i \in X$.

In other words, $Sky(X)$ is of Type I iff the projection of $Sky(X)$ over X, i.e., $\pi_X(Sky(X))$ has a single element.

Example 4.8. In the running example, $Sky(AD) = \{t_1, t_2\}$ and it is of Type I because $t_1[AD] = t_2[AD]$.

The following proposition shows that under some conditions on FDs, some skylines can be derived without effort.

PROPOSITION 4.9. *If $Sky(X)$ is of Type I then for every $Y \subseteq X$ such that $Y \to X$, we have $Sky(Y) = Sky(X)$.*

PROOF. This is a direct consequence of Lemma 4.3. Indeed, if $Sky(X)$ is of Type I, then the partition of its elements w.r.t. X gives only one part. Therefore, every tuple of this part is necessarily in $Sky(Y)$. Since $Y \subseteq X$, and $Y \to X$ then $Sky(Y) \subseteq Sky(X)$ then we conclude that $Sky(X) = Sky(Y)$. □

Example 4.10. Since $Sky(AD)$ is of Type I and since $A \to D$ is valid FD in T, we conclude that $Sky(A) = Sky(AD)$.

4.3. Evaluating $Sky(Sky(Y), X)$

When only the skycuboids belonging to MICS are materialized, it may happen that $Sky(X^+)$ is not materialized for some X because we found that $Sky(X^+) \subseteq Sky(Y)$ and $X \subset Y$ for some closed subspace Y. In this case, the properties we developed in the previous section are no longer valid. So, we need to use some standard skyline algorithm for evaluating $Sky(X)$ using a materialized $Sky(Y)$ without all the optimization we have seen so far.

Example 4.11. Table III shows that for closed subspaces C and $ABCD$, we have e.g., $Sky(C) \subseteq Sky(ABCD)$. Therefore, $Sky(C)$ does not belong to MICS(T). Hence, if a user asks for $Sky(C)$, we need to evaluate $Sky(Sky(ABCD), C)$. Note that all tuples in $Sky(ABCD)$ are distinct from each other. So the partition w.r.t. $ABCD$ contains three parts while the partition of these tuples w.r.t. C contains only two tuples, namely $\langle 3 \rangle$ and $\langle 4 \rangle$.

The above example shows that trying to reduce the storage space by not materializing some skycuboids of closed subspaces comes with query evaluation price. Nonetheless, it is always better than evaluating the queries from the underlying table T.

4.4. Full Skycube 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 speedup 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

ALGORITHM 4: FMC algorithm

```
Input: Table $T$
Output: Skycube of $T$
1  Let $S_{\mathcal{D}} = Sky(T, \mathcal{D})$;
2  if $|S_{\mathcal{D}}|$ is small then
3      $T_{clean} = cleanup(T)$;
4      create index $\mathcal{I}$;
5      for every $X \subset \mathcal{D}$ do
6          // Loop executed in parallel
7          Let $S =$ Sky_X_from_Sky_Y$(T_{clean}, S_{\mathcal{D}}, X)$;
8          $Skycube\_T = Skycube\_T \cup \{S\}$;
9      Return $Skycube\_T$;
10 else
11     $Closed$ = ClosedSubspaces($T$);
12     foreach $X \in Closed$ do
13         // Loop executed in parallel
14         Compute $Sky(T, X)$;
15     foreach subspace $X$ do
16         // Loop executed in parallel
17         Compute $Sky(Sky(X^+), X)$;
18     Return $\bigcup_{X \in 2^{\mathcal{D}}} Sky(X)$;
```

FMC (for Full Materialization with Closed subspaces) as a procedure for solving this problem. It is described in Algorithm 4.
FMC first computes the topmost skyline $Sky(\mathcal{D})$. If this skyline is *small* enough then, as we have seen previously, every $Sky(X)$ can be efficiently obtained from $Sky(\mathcal{D})$ and T_{clean}. If $Sky(\mathcal{D})$ is not small, then FMC proceeds in three main steps: (i) it finds the closed subspaces, then (ii) it computes their respective skylines, and finally (iii) for every non closed subspace X, it computes $Sky(Sky(X^+), X)$.

The main advantage of FMC is that its main computation steps can benefit from a parallel execution. The second advantage is its low memory consumption: in case of a small topmost skyline, all what we need is to keep in memory is this skyline together with the bitmap index. In the opposite case, we keep the skylines over *all* closed subspaces but those are not many as we have seen in Section 3.5. Despite its simplicity, FMC turns to be very efficient in practice and outperforms state of the art algorithms as this is shown in Section 7.

5. MAIN ALGORITHM: ASSEMBLING ALL COMPONENTS TOGETHER

In this section we summarize our proposal by assembling the different procedures we introduced so far. On one hand, we show how skycube materialization is addressed and on another hand we describe how query evaluation is handled. Algorithm 5 describes the materialization of skycubes. It takes as input a table T and an indication of whether a full or a partial materialization is requested. In case of the second option, the algorithm may decide to store only the topmost skyline if it *small*. This property, being small, can be an input of the user, e.g., a certain percentage of the underlying table T or hard coded in the algorithm.

On the other hand, with respect to query evaluation, it depends on how the skycube is materialized. Algorithm 6 shows how skyline queries are handled. In case of partial materialization, we see that for computing $Sky(X)$ we need to know X^+ (Line 10). This information can be stored offline during the computation of the functional dependen-

ALGORITHM 5: Skycube Materialization

```
Input: Table $T$, TypeMat ∈ {full, partial}
Output: Partial or full Skycube of $T$
1  if TypeMat=full then
2      Return FMC($T$);
3  else
4      // The user asks for a partial materialization
5      Let $S_\mathcal{D} = Sky(T, \mathcal{D})$;
6      if $|S_\mathcal{D}|$ is small then
7          // The small condition can be input by the user. E.g., as a percentage of
               $size(T)$
8          $T_{clean} = cleanup(T)$;
9          create index $\mathcal{I}$;
10         Return $S_\mathcal{D}$;
11     else
12         // We need to minimize the memory storage space
13         Return MICS($T$);
```

ALGORITHM 6: Skyline Query Evaluation

```
Input: Table $T$, TypeMat ∈ {full, partial}, subspace $X$
Output: $Sky(X)$
1  if TypaMat=full then
2      Return $Sky(X)$;
3      // No required computation
4  else
5      if $|S_\mathcal{D}|$ is small then
6          // $S_\mathcal{D}$ is always materialized
7          Return Sky_X_From_Sky_Y($T, S_\mathcal{D}, X$);
8      else
9          if $X^+$ is materialized then
10             Return $Sky(Sky(X^+), X)$;
11         else
12             // i.e., only skylines in MICS are stored
13             Let $Y$ be the remaining ancestor of $X^+$ in MICS;
14             Return $Sky(Sky(Y), X)$;
```

cies, i.e., a table that maps every X to X^+. A second option would be to store a minimal cover of the mined FDs and use it online to compute X^+. Recall that computing the closure of X can be done in linear time w.r.t. the size of the cover set [Mannila and Räihä 1992]. Also, when computing MICS, a skyline $Sky(X)$ of some closed X is removed iff there exists some closed Y such that Y is an *ancestor* of X. If it is the case, we need to keep this information in order to be able to answer $Sky(X)$ as well as all queries $Sky(Z)$ such that $Z^+ = X$.

6. RELATED WORK

Many algorithms for computing skylines have been proposed in the literature. The complexity of most of them is analyzed in RAM cost setting, e.g., [Godfrey et al. 2007; Bartolini et al. 2008; Lee and won Hwang 2010]. Some algorithms have been specifically tailored to the case where dimensions have low cardinalities, see e.g., [Morse et al. 2007]. All these algorithms have $O(n^2)$ worst case complexity where n is the size of table T. The pioneering work [Börzsönyi et al. 2001] considered external memory cost and showed the inadequacy of SQL to efficiently evaluate skyline queries. Nonetheless, the algorithms proposed there do *suffer* from the polynomial time complexity too. Recently, [Sheng and Tao 2012] proposed an I/O aware algorithm guaranteeing, in the worst case, a polylog number of disk accesses. [Sarma et al. 2009] proposed a randomized algorithm requiring $O(\log(n))$ passes over the data to find an approximation of the skyline with high probability. Other works make use of some pre-processing like multidimensional indexes. For example, [Papadias et al. 2005] proposed to use R-trees to optimize skyline points retrieval in a progressive way. We emphasize that the optimization techniques we propose in the present paper do not rely on any skyline computation algorithm. So every progress that can be made in skyline algorithms will benefit our approach.

As for multidimensional skylines, most previous works considered the full materialization of skycubes [Lee and won Hwang 2014; 2010; Pei et al. 2005; Yuan et al. 2005]. In order to avoid the naïve solution consisting of 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 $Sky(Y)$. For example, if $Y \subset X$ then a tuple t_i cannot belong to $Sky(Y)$ if it does not match some t_j in $Sky(X)$. In order to take advantage of this property, one must keep $Sky(X)$ in memory. Since several skylines must be kept in that way, this may become a real bottleneck when data are large. 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 [Xia et al. 2012; Raïssi et al. 2010], thereby to reduce the storage space occupied by the entire skycube.

[Raïssi et al. 2010] 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 of T are incomparable in general. Our experiments with various data sets show however, that in practice, **MICS** is computed in general much faster and consumes less memory space than closed skycube. 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.

[Pei et al. 2005] proposed the skyline group lattice which can be seen as a skycube partial materialization technique. We recall that structure intuitively rather than using formal definitions. This lattice consists in storing for every tuple t, a set of pairs of the form ⟨max-subspace, min-subspaces⟩. The elements of each pair act as borders: the first is an upper border and the second is a lower one. The semantics of theses pairs is that t belongs to all the skylines relative to subspaces included in the max-subspace and including at least one of the min-subspaces. For example, suppose that the pair $p_1 = \langle ABCD, \{AC, AD\}\rangle$ is associated to tuple t. p_1 means that t belongs to skylines w.r.t $ABCD$, ABC, ACD, AC and AD but neither to those related to BCD, BC or BD because none of them contains AC or AD (the lower frontier) nor to those related to A, C and D (the immediate subsets of the lower border) and nor to the subspaces of the form $ABCDE_i$, i.e., an immediate superset of $ABCD$ (the upper border). All tuples sharing some pair and having the same value in the max-space form a skyline group and the set of all skyline groups form a lattice. **This technique reasons on a tuple level in that it tries to reduce the number of times a tuple is stored while our approach uses a subspace level reasoning by trying to store the least number of skycuboids.** From a different angle, the rich semantics of this lattice comes with a price: it is harder to compute than the full skycube. As regards storage space, the two methods seem incomparable in general. For example, if $Sky(A) \subset Sky(ABC)$ and $Sky(A) \nsubseteq Sky(AB)$ then **MICS** does not store $Sky(A)$ while the skyline group lattice stores some information about the tuples in $Sky(A) \setminus Sky(ABC)$. On another hand, if for some X there is no Y such that $X \subseteq Y$ and $Sky(X) \subseteq Sky(Y)$ then $Sky(X)$ is stored entirely while Skyline group can avoid to store information about all tuples in $Sky(X)$. Finally, it is interesting to note that the inclusions between skylines that we are able to identify thanks to FDs can speedup the computation of the skyline group lattice in the following way: We know that all tuples belonging to $Sky(X) \cap Sky(X^+)$ belong to every $Sky(Z)$ such that $X \subseteq Z \subseteq X^+$. This information can simplify skyline group lattice construction, e.g., if $Sky(X) = Sky(X^+)$.

[Xia et al. 2012] 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 lossless in the sense 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 $Sky(min\_sky(Y), X)$. Therefore, the dimensionality d can become a bottleneck for query evaluation. Indeed, evaluating $Sky(X)$ requires the traversal of all subsets of X, an exponential number, collect all the tuples then execute a standard skyline query to remove those that are dominated w.r.t X. As it is shown in the experimental section, CSC does not provide competitive query evaluation performance. From the storage space point of view, CSC and **MICS** are incomparable in general. In our experiments, it turns that **MICS** is always less space consuming, and, perhaps more importantly, much faster to obtain.

It is interesting to note that in the case where all dimensions values are distinct, CSC, skyline groups and MICS store exactly the same tuples: all and only those tuples belonging to the topmost skyline. MICS stores them once (the topmost skyline) and every $Sky(X)$ is obtained by evaluating $Sky(Sky(\mathcal{D}), X)$. CSC and skyline groups store every skyline point t in $Sky(Y)$ such that Y is a minimal subspace for t. Hence, every tuple is replicated as many times as there are minimal subspaces where it belongs to. Evaluating $Sky(X)$ turns to make the union of $Sky(Y)$ such that $Y \subseteq X$. Hence, in this very special situation, the storage space is larger for CSC and Skyline groups than MICS but for query evaluation, the later are optimal.

Recently, [Bøgh et al. 2014] proposed the Hashcube structure to encode the skycube by using bit strings for the storage and boolean operations for query evaluation. The experiments presented in that reference show that in general, hashcubes compress the skycube storage space by a factor 10 while it is about 10 times slower for query evaluation. We note however that the authors do not provide a specialized procedure for building the hashcube directly from the underlying data. Instead, the whole skycube is assumed to be already available. Recall that computing **MICS** does not require to compute the whole skycube. On the other hand, achieving a 10% compression ratio means that the hashcube structure has an exponential growth w.r.t d since the skycube size grows exponentially. Our empirical study shows a linear growth of **MICS** size w.r.t d giving an evidence that **MICS** scales better w.r.t d.

Even if FDs have already been used in semantic query optimization [Chakravarthy et al. 1990; Godfrey et al. 2001] and in data cubes partial materialization [Garnaud et al. 2012], to the best of our knowledge, we are the first to show their usefulness in the context of skylines. From another side, it is interesting to note that since the early works dealing with skyline [Börzsönyi et al. 2001], the statistical correlation has been identified as a key parameter impacting both the skyline size and query execution time. FDs can be seen as a special case of correlation and the present work shows their impact on multidimensional skylines. Our findings may appear surprising because FDs are agnostic to data semantics and much less to any order based on any attributes while the corner stone for skyline queries is the different orders defined among the different attributes.

Some works have been proposed in order to estimate the skyline size, e.g., [Godfrey et al. 2007; Shang and Kitsuregawa 2013; Chaudhuri et al. 2006]. We could have used those results in order to show the relationship we established between the number of distinct values per dimension and the skyline size. Unfortunately, most of those works assume the distinct values hypothesis which was not helpful for us. [Godfrey et al. 2007] considers the same data distribution as the one we used (independence of dimensions) while [Shang and Kitsuregawa 2013] focuses on anti-correlated data. [Chaudhuri et al. 2006] does not consider any data distribution hypothesis but the estimator is not a closed formula. A noticeable exception is [Godfrey 2004] where repeated values in dimensions are allowed. However, their main result is an upper bound of the skyline size. More specifically, it is shown that for fixed n and d, the skyline size when we have k distinct values is bounded by the skyline size when $k = n$, i.e., the case of distinct values hypothesis. But this result does not show the skyline size trend w.r.t k.

An important research direction in data mining and cube processing has been the integration of analytic algorithms into a DBMS with User-Defined Functions (UDFs) [Ordonez 2010] or SQL queries [Ordonez et al. 2016]. UDFs could help finding closed subspaces in the skycube or in skycube materialization. Furthermore, it is worthwhile to investigate the impact of column-based storage to optimize multidimensional skyline queries following a similar approach to how recursive queries were re-optimized in a columnar DBMS [Ordonez et al. 2016]. This could be done in combination with the partitions based storage described in Section 4.2.

7. EXPERIMENTAL EVALUATION

We conduct 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 scalability w.r.t. both d (dimensionality) and n (table size). It turns that our solution outperforms state of the art algorithms for full materialization when both data and dimensions get larger; (ii) we compare our partial materialization proposal to state of the art techniques by considering three parameters: (a) the time required to build the partial skycube, (b) the memory space used by the structures, and (c) the query execution time. Finally, (iii) we analyze the impact of materialization in query cost reduction. Table IV summarizes the compared algorithms regarding full and partial materialization of skycubes. All experiments were conducted on a machine with 24G of RAM, two 3.3 GHz hexacores processors, and a 1 Tb disk under Redhat Entreprise Linux OS.

7.1. Data sets

We used both real and synthetic data sets. Table V describes NBA, IPUMS and MBL real data which are well known in the skyline literature. As they are relatively small, we augmented artificially their respective sizes by just copying them multiple times.

Table IV. Compared algorithms w.r.t. tasks

Materialization	Our algorithm	Other Algorithms
Full	FMC	OrionTail, BUS, TDS, QSkyCubeGS, QSkyCubeGL
Partial	MICS	Orion, CSC, Hashcube

Table V. Real data sets

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

For example, NBA50 is obtained from NBA data set by copying every tuple 50 times. We call these three sets and their variations as *benchmark* data sets. In addition, we used two larger real data sets, namely USCensus and Householders. We use them because of their larger size, but more importantly, because they satisfy a very different number of functional dependencies in either data set (almost zero in USCensus, many in Householder). Moreover, we use synthetic data generated by a software available at `pubzone.org` and p rovided by the authors of [Börzsönyi et al. 2001]. 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 distribution. Attributes values are real numbers normalized into $[0, 1]$.

7.2. Parallel Processing

We implemented our solutions using `C++` language together with `OpenMP` to benefit from parallelism. This API makes it very easy to spawn several threads in parallel. For example a loop of the form:

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

can be executed by 4 threads just by adding a compilation directive (pragma) to the source code as follows:

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

In theory, its execution time is then divided by 4 if we have four processing units. In practice this is rarely the case because (i)the execution of $f(i_1)$ can take more or less time than that of $f(i_2)$ thus we do not have a perfect load balancing and (ii) if both $f(i_1)$ and $f(i_2)$ need to access a shared structure then we need to control this concurrency, for example by using locks, so some threads need to wait until others free their locks and this penalizes the parallel execution. Moreover, following Amdhal's law, algorithms have sequential parts that cannot benefit from parallelism. Thus whatever is the number of available processing units, these parts are necessarily executed by a single thread.

7.3. Full Skycube Materialization

In this section, we compare the execution time of FMC to state of the art algorithms, namely BUS and TDSG [Pei et al. 2006], OrionTail [Raïssi et al. 2010], as well as the most recent proposals QSkycubeGS and QSkyCubeGL reported in [Lee and won

Hwang 2014]. We used the authors implementations without any change[3]. All these algorithms take as input a table and return its respective skycube. All of them are implemented in C++. For FMC, we make use of the BSkyTree implementation [Lee and won Hwang 2010] for computing skylines. Unless otherwise stated, for every execution of FMC we fixed the number of threads to 12 (number of available cores). Since all previous algorithms are sequential, and to make the comparison faithful, we sometimes report the speedup of FMC over its competitors instead of their execution times. More precisely,

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

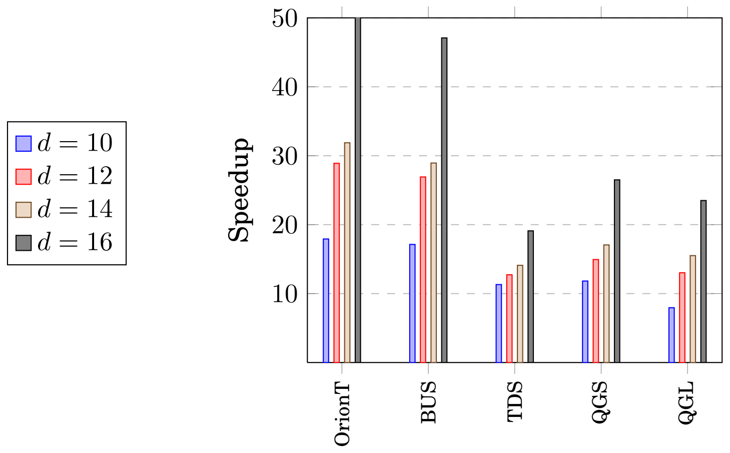
If the speedup is greater than 12, we can safely conclude that FMC outperforms its competitor since its sequential execution cannot be twelve times slower than its parallel execution. For a topmost skyline to be considered as *small* we used the condition that its size should be less than $\sqrt{n}$. The rational behind this choice is to be sure that skyline evaluation complexity becomes linear in n.

7.3.1. Synthetic data: scalability w.r.t dimensionality d. To analyze the effect of dimensionality growth, we start with synthetic data. 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 6(a)-6(c). Figure 6(a) shows that the speed-up obtained by FMC increases w.r.t. d. This tends to demonstrate that, for this kind of data distribution, FMC is more appropriate to use when d gets larger. A more detailed look to the performance of QSkycubeGL shows that when $d = 10$, the speed-up of FMC is *only* about 7. One may be tempted to conclude that executing FMC with a single thread, i.e., sequential execution, is slower than QSkycubeGL. Actually, it turns that both algorithms perform equally in that case. We should also note that the actual execution time is about 2 seconds. For the other two distributions, Figures 6(b) and 6(c) show that with $d = 10$, we already have a speed-up larger than 12. Even if we note that all algorithms do not have the same behavior w.r.t d growth, FMC is consistently much more efficient than them in all cases.

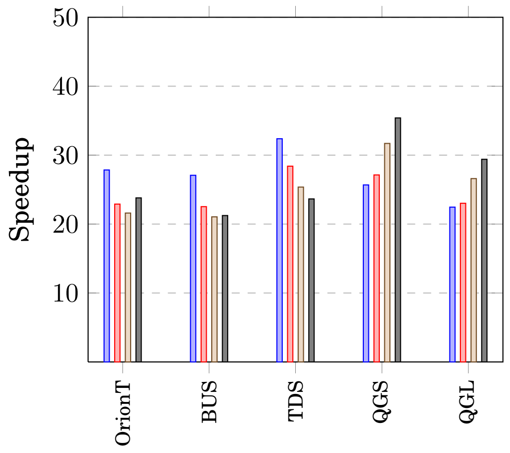
7.3.2. Synthetic data: scalability w.r.t. data size. Here we fix d to 16 and vary the value of n by considering $200K$, $500K$ and $1M$ tuples. The results are reported in Figure 7. The experiments show that FMC outperforms all algorithms in all cases. Moreover, the speed-up increases uniformly when n increases. Even though, it is interesting to note that QSkyCubeGL and QskyCubeGS are the most scalable algorithms.

Remark 7.1 (*On Synthetic Data Generation and* FMC *Extension*). 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 is because the values in each dimension are reals picked uniformly at random from $[0, 1]$ dense interval. So, the probability that the same value is picked twice from this, theoretically infinite, set is close to zero. This case tends to reduce the set of closed subspaces to just $\mathcal{D}$ and some single dimensions. 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 and even independent data. Actually, in the former situations, FMC almost turns to become the naïve algorithm since the size of $Sky(\mathcal{D})$ is close to that of T and most skycuboids are computed from $Sky(\mathcal{D})$. Nevertheless, as the previous experiments have shown, FMC is still competitive in those cases essentially for two

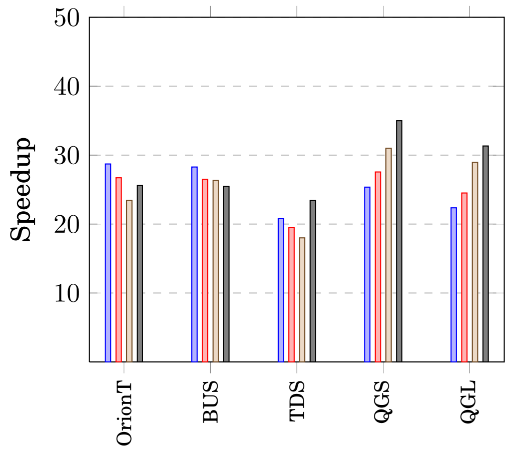
[3]We are grateful to the authors of [Lee and won Hwang 2014] who provided us with an implementation of all these algorithms.



(a) Correlated



(b) Independent

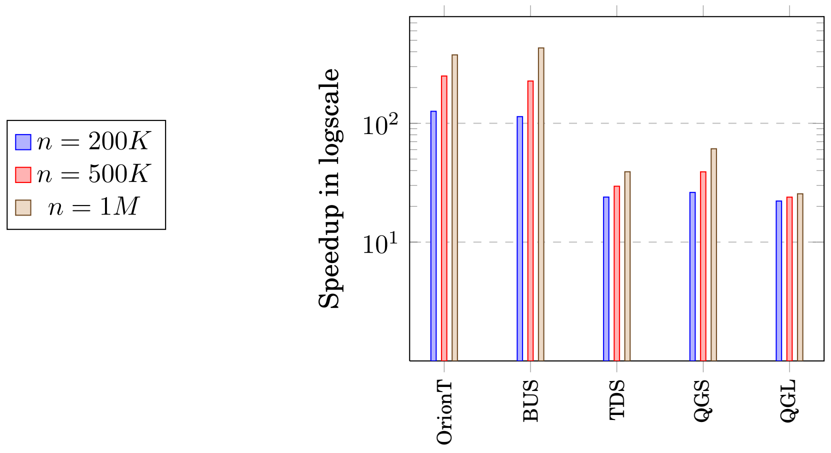


(c) Anticorrelated

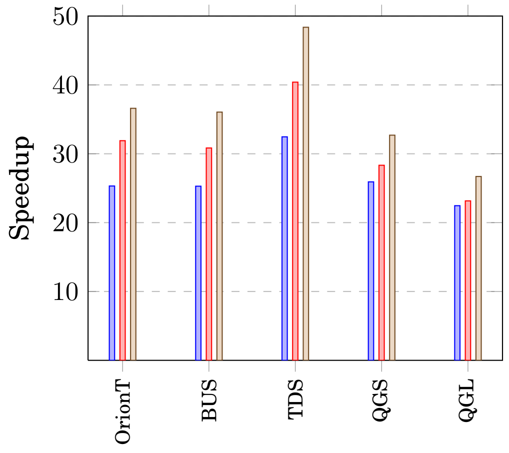
Fig. 6. Speedup w.r.t dimensionality d ($n = 100K$)

reasons (i) its parallel execution and (ii) the fact that we do not make any extra computation aiming to retrieve potentially missed tuples of $Sky(T, X)$ when we evaluate $Sky(Sky(\mathcal{D}), X)$. This second point represents a bottleneck of previous works. Indeed, they make unnecessary and wasteful checks. More importantly, the previous experiments show that previous solutions perform *worse* than the naïve algorithm when the dimensionality as well as the number of distinct values per dimension increase.

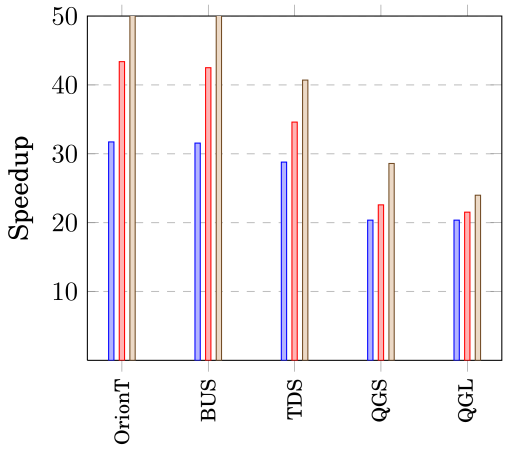
Remark 7.2. FMC can be easily optimized by better exploiting skylines inclusions that are induced by FDs. For example, suppose that $\mathcal{D} = \{A, B, C, D\}$ and that the distinct values property holds in T. Then for every subspace $X \neq ABCD$, FMC evaluates $Sky(Sky(ABCD), X)$. A better choice would be to evaluate the skyline of X from the skyline of one of its *immediate* parents, e.g., $Sky(A)$ from $Sky(AB)$, $Sky(AC)$ or $Sky(AD)$. This can be performed by making either breadth or depth first traversal of the subspaces lattice.



(a) Correlated

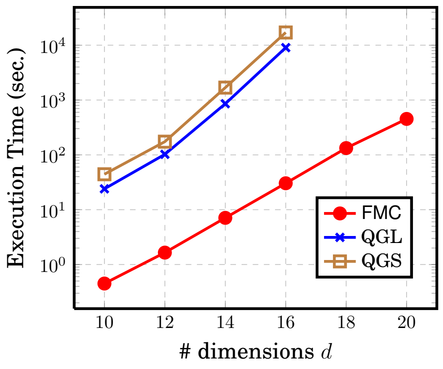


(b) Independent

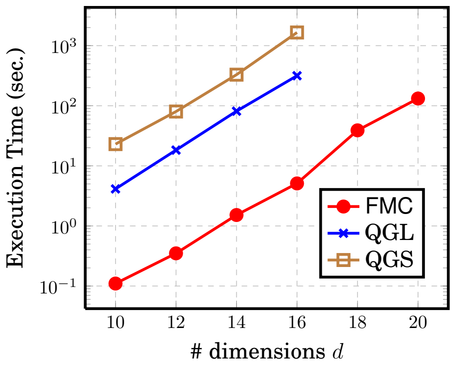


(c) Anticorrelated

Fig. 7. Speedup w.r.t data size n ($d = 16$)



(a) USCensus



(b) Householders

Fig. 8. Full Skycube Computation with large real data sets

7.3.3. 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 data sets. `US Census` is a well known data set used in machine learning and available from the UCI repository. 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. 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 by contrast to USCensus, these columns have a meaningful ranking semantics (e.g., number of persons living in the house, number of rooms, and so on). 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` all subspaces are closed meaning that there are no FDs, with `householders` the proportion of closed subspaces is very small (less than 2%) when $d \geq 10$.

US Census data set. Figure 8(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 to disk during the computation. That 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 were careful 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 `STreeCube` shared data structure used by those algorithms in order to speedup execution time. Note the rapid growth of the speedup 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 makes functional dependencies hard to be satisfied while the number of tuples is quite large. In fact this data set does not satisfy any FD. Therefore all subspaces are closed. However, the topmost skyline is quite *small*. For example, for $d = 10$, $Sky(\mathcal{D})$ contains *only* 3873 tuples. Moreover, these tuples have exactly the same values in every dimension; $Sky(\mathcal{D})$ is of Type I. This makes the computation of any skyline easy.

It is the *semi-naïve* skycube computation that is chosen by FMC (lines 2-6 in Algorithm 4). Hence, computing $Sky(X)$ is performed by a join operation which simply consists in retrieving those tuples t from T such that $t[X] = t'[X]$ where t' is the *unique* representative of $Sky(\mathcal{D})$. To optimize this join operation, in our implementation we make use of a bitmap index[4][Lemire et al. 2012]. This experiment empirically confirms the relationship between the number of distinct values per dimension, the number of FDs and the size of the topmost skyline.

Householders data set. The results are reported in Figure 8(b). Here again, both QSkyCubeGS and QSkyCubeGL were unable to handle the cases where $d \geq 16$. A noticeable difference between this data set and the previous one is that many FDs are satisfied. The number of closed subspaces is around 29000 out of the $2^{20} - 1$ subspaces. The number of distinct values per dimension varies from 2 to 4248. The topmost skyline has 154752 tuples. This shows the effectiveness of using the skylines of the closed

[4]We used the `EWAH` bitmap index implementation of D. Lemire which is available from `https://github.com/lemire/EWAHBoolArray`

Table VI. Execution times (sec.) for small real datasets. FMC is executed with 1 and 12 threads

NBA			
FMC(1)	FMC(12)	QGL	QGS
1.01	0.22	8.15	4.57

IPUMS			
FMC(1)	FMC(12)	QGL	QGS
1.36	0.45	2.27	10.24

MBL			
FMC(1)	FMC(12)	QGL	QGS
12.3	4.5	63.85	267.1

Table VII. Execution times (sec.) for artificially enlarged real datasets FMC is executed with 1 and 12 threads

NBA50			
FMC(1)	FMC(12)	QGL	QGS
23	7	59	336

IPUMS10			
FMC(1)	FMC(12)	QGL	QGS
22	6	16	336

MBL10			
FMC(1)	FMC(12)	QGL	QGS
311	47	780	5963

subspaces to compute the rest of the skylines. Indeed, even if the speedups we obtain are less impressive than those with the previous data set, FMC is still 50 times faster than QSkyCubeGL and even more comparatively to QSkyCubeGS.

Benchmark real data sets. NBA, IPUMS and MBL are relatively small sets. For these data sets, we executed FMC twice: In the first execution, denoted by FMC(1), we used only one thread, i.e., a sequential execution , and in the second one, denoted by FMC(12), we used 12 threads. We did not make any modification to the program. Table VI shows the execution times. Note however that when the number of dimensions is *small* which is the case with IPUMS, the speed-up of FMC(12) w.r.t. QSkycubeGL is rather weak (about 5.04). It is much larger with NBA and MBL which have 17 and 18 dimensions respectively. This tends to show that parallel FMC 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 speedup of 3.8 over QskycubeGL. So the question of when using shared data structures and/or computations is really worthwhile remains open.

We artificially enlarged these data sets by just copying their content multiple times. Table VII shows the execution times. We note that FMC scales almost linearly w.r.t data size. QSkyCubeGS does not scale well. QSkyCubeGL seems to be the algorithm having the best behavior regarding data size growth. Note however that it is the algorithm which is more memory consuming. For example, we tried to execute QSkyCubeGL with NBA100 and it saturated the memory while FMC handled this data set and finished its execution after 13 seconds.

Finally, we analyzed the behavior of the three algorithms w.r.t dimensionality. We used MBL10 and varied d from 4 to 18. Figure 9 compares FMC(12) execution times

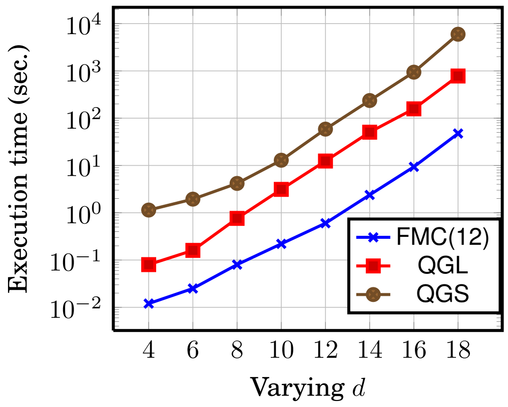


Fig. 9. Full materialization with MBL10

Table VIII. MICS vs Orion

NBA		
Technique	time to build (sec.)	Materialized skycuboids
MICS	12.8	5304
Orion	9	5304
IPUMS		
Technique	time to build (sec.)	Materialized skycuboids
MICS	2.1	11
Orion	322	738
MBL		
Technique	time to build (sec.)	Materialized skycuboids
MICS	172	29155
Orion	11069	43075

to those of QSkycubeGS and QSkycubeGL. The speed-up of our algorithm is almost constant.

7.4. Partial Skycube Materialization

7.4.1. MICS vs Closed Skycubes:. We compare our solution to OrionClos the algorithm proposed in [Raïssi et al. 2010] to compute the *Closed Skycubes*. We make the comparison by considering (i) the storage space usage and (ii) the speed for materializing the sub-skycube. We consider the three real data sets used in that reference: NBA, IPUMS and MBL. 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. For both techniques, we also report the total computation time. The results are presented in Table VIII.

In general, we note that 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. By contrast, our solution is often much 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 36 hours were not sufficient to process a correlated data set with $d = 20$ and

Table IX. Partial skycube vs CSC

NBA			
Technique	time to build (sec.)	storage	query time (sec.)
CSC	631	57571	150
Partial Skycube	0.0008	3	3.5
MBL			
Technique	time to build (sec.)	storage	query time (sec.)
CSC	47654	921911	7212
Partial Skycube	10	78	58
IPUMS			
Technique	time to build (sec.)	storage	query time (sec.)
CSC	3776	129812	142
Partial Skycube	6	73382	2
Correlated			
Technique	time to build (sec.)	storage	query time (sec.)
CSC	4533	498	10
Partial Skycube	0.015	2	0.03
Independent			
Technique	time to build (sec.)	storage	query time (sec.)
CSC	36123	921911	6212
Partial Skycube	35	83650	78
Anti-correlated			
Technique	time to build (sec.)	storage	query time (sec.)
CSC	51263	4556789	11972
Partial Skycube	62	121347	129

$n = 100K$. By contrast, it took 20 seconds to find and materialize the MICS of the same data set. 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.

7.4.2. Compressed Skycubes (CSC). In this section, we compare our proposal to the compressed skycubes (CSC) approach of [Xia et al. 2012] following three criteria: (i) time to build, (ii) storage space and (iii) query execution time. For this purpose, we use the three real data sets: NBA, MBL and IPUMS as well as three synthetic data sets with 16 independent/correlated/anti-correlated dimensions and 100k tuples. Regarding query evaluation performance, we consider a workload of all the $2^d - 1$ possible skyline queries in order to get an idea about the average execution time for queries. To make a fair comparison, our algorithms are executed sequentially. Recall that our solution either stores only the topmost skyline if its size is found small or it relies to MICS. The results are depicted in Table IX. As it can be observed, our proposal outperforms CSC in all criteria. Note that for real data sets NBA and MBL as well as correlated set, our solution requires the storage of the topmost skylines which contain respectively 3, 78 and 2 tuples.

7.4.3. Partial Skycube vs Hashcube. We use two data sets, namely `NBA` and `Householder`, to illustrate the advantages as well as the limitations of the Hashcube structure [Bøgh et al. 2014]. While the first data set is small and correlated, the second one is much larger and non correlated. We report the size of the Hashcucube and the partial skycube we obtain in terms of the number of tuples they store. We also report the execution time to answer the $2^d - 1$ possible skyline queries. Table X summarizes our findings.

Table X. Partial skycube vs Hashcube

NBA		
Technique	storage	query time (sec.)
Hashcube	541316	0.029
Partial	3	3.5

Householder		
Technique	storage	query time (sec.)
Hashcube	2431675126	_
Partial	49022451	128

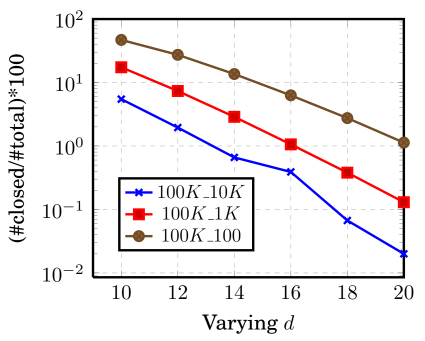
With `NBA`, we see that Hashcube requires too much storage space compared to our solution. We recall that the skycube itself requires to store about $4.7*10^6$ tuples. Hence, Hashcube achieves, as it was reported in [Bøgh et al. 2014], a compression ratio of almost 10%. We note however that the query execution time with hashcube is $100\times$ faster.

For the second data set, the size of the hashcube structure was too large to fit into the available memory. The skycube contains around $15 * 10^9$ tuples. Hence, we did not continue the experiment for the query part because it would had taken too much time. Note that our solution is about $300\times$ smaller than the skycube.

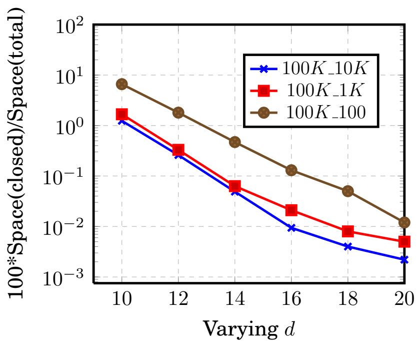
To conclude, one should use Hashcubes in the following situation: (i) the Skycube is already available and its size is too large to fit into memory and, (ii) the hashcube structure makes it possible to fit into memory.

7.4.4. Storage Space Analysis. In the second part of the experiments, we generated a set of independent synthetic data by fixing the following parameters: d is the number of attributes, n is the number of tuples, k is the average number of distinct values taken by each attribute. For example, $100K\_10K$ designates a data set where $n = 100K$ and the number k 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 $k = 10^f$ distinct values. For example, 0.0123 is replaced by 12 if $k = 10^3$. Doing so, the correlation between the different columns is preserved. Even if we we report only the results obtained with the independent data sets, we should mention that we performed the same experiments with correlated and anti-correlated data. The conclusion were the same.

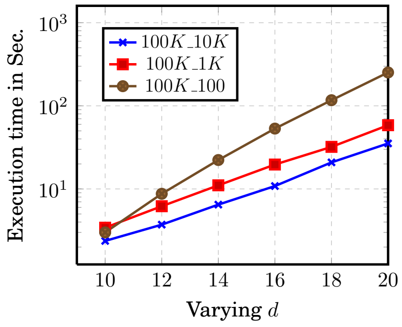
We firstly investigate the number of closed subspaces comparatively to the total number in the skycube. The results are reported in Figure 10(a). We note that the proportion of closed subspaces decreases when the number of attributes increases. For example, consider the data set $100K\_10K$ when $d = 10$. There are about 7% of the subspaces out of the total $2^{10} - 1$ that are closed. This proportion falls to 0.035% when $d = 20$. In addition, the number of closed subspaces grows when the number of distinct values taken by each attribute decreases. 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 10(b)) shows that this is not systematic. Here, we compute a ratio between the total numbers of tuples that should be stored when only the closed subspaces are materialized over the total number of skycube tuples. We see that in all cases, the memory space needed to materialize the skylines w.r.t. the closed subspaces 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



(a) Percentage of materialized skycuboids



(b) Ratio of consumed storage space



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

Fig. 10. Quantitative Analysis of the Closed Subspaces

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 next empirically show that it is rather because the size of skylines decreases.

Finally, Figure 10(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. Clearly, the less distinct values per dimension, the more closed subspaces there are and the more time is needed to compute them.

We conducted a second series of experiments aiming to show the evolution the skyline size when both the cardinality k and the dimensionality d vary while the size of data n is kept fixed. The results are shown in Figure 11. We observe that the size of the skyline increases uniformly regarding k whatever is d. This gives a clear explanation of the behavior noticed in Figure 10(b), i.e., when k decreases, the number of closed subspaces increases while their respective sizes decrease (See Theorem 3.20).

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 does not mean necessarily that it uses more storage space. The second lesson we derive is that when k 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 1. This is in concordance with the analytic study developed in Sections 3.5 and 3.6.

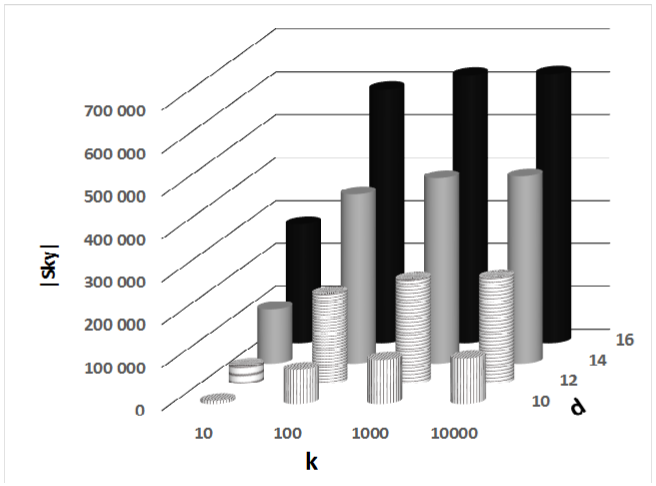


Fig. 11. Skyline size evolution w.r.t. k and d with $n = 10^6$

7.5. 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 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 XI. 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.

8. CONCLUSION AND FUTURE WORK

In this paper we show how the classical concept of FDs may be used used to identify inclusions between skylines over different subspaces. Thanks to this information, we investigate the partial materialization of skycubes. We also leverage the FDs to the full materialization case. This is 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 for partial materialization can be seen as a trade-off among (i) the space consumption (we try to store as least as possible), (ii) the query execution time (we avoid reading the entire data set) and (iii) the speed to

Table XI. Query execution times in seconds: optimized/not optimized and (Proportion of materialized skycuboids)

$n \backslash d$	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%)

obtain the partial skycube. We have experimentally compared our proposal to state of the art implementations. The conclusion is that our solutions scales better when data and dimensionality grow. On real data sets, by materializing a small fraction of the skycube, we gained orders of magnitude on query evaluation time.

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 in that direction, it is not clear whether we need global or local FDs. Just like the join operator and the lossless decomposition theory, it is tempting to come up with a decomposition theory under skyline re-composition constraint with the help of FDs or maybe with other classes of dependencies like *Order dependencies*. In this paper we identified the minimal set of skycuboids to be materialized (MICS). To further reduce 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. We also 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. As we have seen, the topmost skyline plays an important role in capturing the *shape* of the different skylines. We think that it can also be useful in the context of incremental maintenance. Since FDs are invariant upon isomorphic data transformations, the inclusions detected by the FDs are invariant w.r.t the chosen orders on dimensions. We are wondering whether our findings can be extended to contexts where users can dynamically define their own preferences. For example, flight companies can be (partially) ordered in different ways depending on users. Therefore, the best tickets for one user can be different from those for another user just because they do not consider flight companies in the same order.

APPENDIX

A. COMPUTING THE CLOSED SUBSPACES

The naïve algorithm for finding the closed sets of attributes consists simply in computing the size of every $\pi_X(T)$. If there exists $Y \supset X$ such that $|\pi_X(T)| = |\pi_Y(T)|$ then X is not closed because this means that $X \rightarrow Y \setminus X$ holds. Otherwise, X is closed. This algorithm is inefficient because it requires 2^d projections. In this section we develop a more efficient algorithm whose aim is to prune the search space by avoiding unnecessary projections.

Our algorithm can be summarized as follows: suppose that for some attribute A, we are given the maximal subspaces X which do not determine A. Thanks to the anti-monotonicity of FD's, i.e., if $X \not\rightarrow A$ then $Y \not\rightarrow A$ for every $Y \subseteq X$, we can conclude that all subsets of those maximal subspaces are potentially closed and all others are certainly not closed. In order to efficiently find these maximal subspaces, our algorithm tries to exploit this anti-monotonic property by avoiding to test [5] the obviously not closed subspaces as well as testing all potentially closed subspaces. In the remaining of this section, we formalize these observations.

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 $T \models X \to A_i$.
— $\mathcal{D}_i = \mathcal{D} \setminus A_i$.

LEMMA A.1. *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.*

PROOF. $X' \in Det_{A_i}$ means that $T \models X' \to A_i$. Hence, $T \models X \to A_i$ and therefore $X^+ \ni A_i$ showing that X is not closed. □

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

PROPOSITION A.3. *Let $\mathcal{C}_i^-$ be the set of non closed subspaces derived by Lemma A.1 and $\mathcal{C}^- = \bigcup_i \mathcal{C}_i^-$. Then X is closed iff*

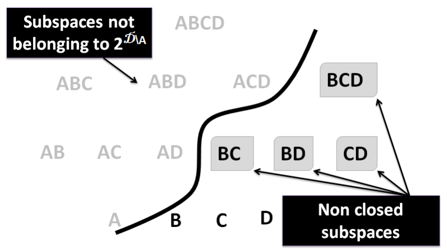
— *$X = \mathcal{D}$ (all the attributes) or*
— *$X \in 2^{\mathcal{D}} \setminus \mathcal{C}^-$.*

PROOF. The first item is obviously true thus we prove the second one. Let us first show the implication $X \in 2^{\mathcal{D}} \setminus \mathcal{C}^- \Rightarrow X$ is closed: For the sake of contradiction, let $X \in 2^{\mathcal{D}} \setminus \mathcal{C}^-$, $X \neq \mathcal{D}$ and suppose that X is not closed. In this case, there must exist some $A_i \notin X$ such that $A_i \in X^+$. Hence, there should exist $X' \subseteq X$ such that $X' \in Det_{A_i}$ which implies that $X \in \mathcal{C}_i^-$. We get a contradiction. Let us now show X closed $\Rightarrow X \in 2^{\mathcal{D}} \setminus \mathcal{C}^-$. Assume the contrary, i.e., X is closed and $X \notin 2^{\mathcal{D}} \setminus \mathcal{C}^-$. Thus, there must exist A_i such that $X \in \mathcal{C}_i^-$. By Lemma A.1 we conclude that X is not closed which contradicts the hypothesis. □

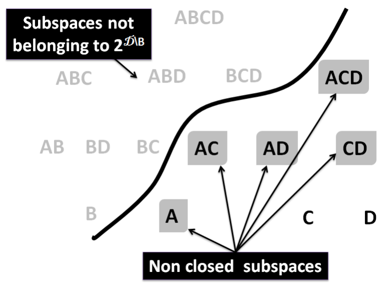
Example A.4. For the running example we have $Det_A = \{BD, BC, CD\}$, $Det_B = \{A, CD\}$, $Det_C = \emptyset$ and $Det_D = \{A, BC\}$. From these sets, we derive $\mathcal{C}_A^- = Det_A \cup \{BCD\}$, $\mathcal{C}_B^- = Det_B \cup \{AC, AD, ACD\}$, $\mathcal{C}_C^- = Det_C \cup \emptyset$ and $\mathcal{C}_D^- = Det_D \cup \{AC, ABC\}$. 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 12(a) and 12(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 8) 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.

[5]Testing X is costly because it means testing whether $X \to A$.



(a) Non closed sets w.r.t. A: B, C and D are the maximal sets not determining A. Hence, all their supersets not containing A are not closed.



(b) Non closed sets w.r.t. B: C and D are the maximal sets not determining B. All their supersets not containing B together with A are not closed.

Fig. 12. Pruned sets w.r.t. attributes A and B: BCD is pruned by A and is not part of any of the search spaces associated to B, C and D. Hence, $\pi_{BCD}(T)$ is never computed. ACD is pruned by B. Therefore, $\pi_{ACD}(T)$ is never computed. $\pi_{AC}(T)$ is pruned by B because $A \to B$ is found satisfied and AC is not part of $2^{D\setminus A}$ but it can be tested with D. The same with AD which can be tested with C.

A.1. Extracting Maximal Violated Functional 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 $X \to A_i$ is not satisfied. There are several algorithms for computing *minimal* functional dependencies in the literature, e.g. [Yao and Hamilton 2008; Lopes et al. 2000; Huhtala et al. 1999; Novelli and Cicchetti 2001]. Inferring from these sets the *maximal* violated dependencies can be performed by computing the minimal hypergraphs transversals, also called minimal hitting sets [Eiter and Gottlob 1995]. 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 an algorithm proposed in [Hanusse and Maabout 2011] for mining Maximal Frequent Itemsets. **MaxNFD** is depicted in Algorithm 7. It can be described as follows: let X be a candidate for which we want to test whether $T \not\models X \to A_i$. 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., $T \models X \to A_i$, 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 [Hanusse and Maabout 2011] 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 7. The correctness of the algorithm is already proven in [Hanusse and Maabout 2011]. An important property of **MaxNFD** is its amenability to be executed in a parallel way. Indeed, the **Foreach** loop in Line 4 of the algorithm shows that all the subspaces which belong to the set $Candidates[k]$ can be tested in parallel.

A.2. 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

ALGORITHM 7: MaxNFD

```
Input: Table $T$, Target attribute $A_i$
Output: Maximal $X$ s.t $T \not\models X \to A_i$
Candidates[1] ← {A_1};
k ← 1;
while $k \le (2d-1)$ do
    foreach $X \in Candidates[k]$ do
        // Loop executed in parallel
        if $\nexists Y \in$ Max st $Y \supseteq X$ then
            if $T \not\models X \to A_i$ then
                Add $X$ to Max;
                Remove the subsets of $X$ from Max;
                Add the parent of $X$ to $Candidates[k+1]$;
            else
                $Candidates[k+1] \uplus RightChildren(X)$;
                $Candidates[k+2] \uplus RightSibling(X)$;
    $k \leftarrow k+1$;
Return Max;
```

ALGORITHM 8: ClosedSubspaces

```
Input: Table $T$
Output: Closed subspaces
for $i = 1$ to $d$ do
    $\mathcal{L}^- =$ MaxNFD($T, A_i$);
    $\mathcal{L}^- =$ SubsetsOf($\mathcal{L}^-, A_i$);
    if $i = 1$ then
        $Closed = \mathcal{L}^-$;
    else
        $Closed_i = \{X \in Closed \mid X \ni A_i\}$;
        $Closed = (Closed \cap \mathcal{L}^-) \bigcup Closed_i$;
Return $Closed$;
```

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.

B. THE INFLUENCE OF DIMENSIONS CARDINALITY IN SKYLINE SIZE

In this section we provide a detailed proof of Theorem 3.20. We first give some definitions and notations.

Definition B.1. Let $\mathscr{T}_k$ denote the set of all tables T with d independent dimensions, n tuples and at most k distinct values per dimension (the values of each dimensions belong to $\{1, 2, \ldots, k\}$). The tuples of T are not necessarily distinct.

Definition B.2. Let $\Pi(T)$ denote a table with the same tuples as T but in which each tuple appears only once.

Let $Sky(T)$ denote the skyline of T over all its dimensions, i.e., $Sky(T) = Sky(T, \mathcal{D})$, and $S(T)$ denote the size of $Sky(T)$, i.e., $S(T) = |Sky(T)|$.

Definition B.3. Let $\overline{S(T)}_{T\in\mathscr{T}_k}$ denote the average (the expected value) of $S(T)$ where T belongs to $\mathscr{T}_k$.

THEOREM B.4. *We have the following*

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

In other words, the above theorem states that for fixed n and d, the number of distinct tuples appearing in the skyline tends to increase when the number of distinct values per dimension grows. Note that this is exactly the same result as Theorem 3.20. We rephrase it here for the sake of rigor. In order to prove the previous theorem the following definitions and Lemmas are presented.

Definition B.5. Let f be a relation which associates to each integer a the random value $2a - X$ where X is a random variable following Bernoulli distribution with a parameter equal to $\frac{1}{2}$. In other words, $f(a)$ can be equal to $2a-1$ or $2a$ with the same probability. Let $F(T)$ denote the set of all tables which can be obtained by applying f to each cell of T. Let $T_k \in \mathscr{T}_k$. Then
$F(T_k) = \{T_{2k} \in \mathscr{T}_{2k} : T_{2k}[i,j] = f(T_k[i,j]), \forall_{i,j} 1 \leq i \leq n \text{ and } 1 \leq j \leq d\}$

LEMMA B.6. *The set* $\{F(T_k), T_k \in \mathscr{T}_k\}$ *is a partition of* $\mathscr{T}_{2k}$.

PROOF. To each table $T_k \in \mathscr{T}_k$ is assigned a set $F(T_k)$ of tables $T_{2k} \in \mathscr{T}_{2k}$ such that T_{2k} results from T_k by applying f to each value. Reciprocally, each table $T_{2k} \in \mathscr{T}_{2k}$ is assigned to a single table $T_k \in \mathscr{T}_k$ (we get T_k by dividing and rounding each value of T_{2k} by 2, $f^{-1}(a) = \lceil \frac{a}{2} \rceil$). The size of $\mathscr{T}_k$ is given by $|\mathscr{T}_k| = k^{n\cdot d}$. Indeed, each cell (i,j) in T_k can take k values and since there are $n \times d$ cells, we obtain $k^{n\cdot d}$. Hence, $|\mathscr{T}_{2k}| = (2k)^{n\cdot d}$. On another hand, for every T_k, we have that $|F(T_k)| = 2^{n\cdot d}$ because each value of T_k has two possible mappings in T_{2k}. Clearly, $\bigcup_{T_k} F(T_k) \subseteq \mathscr{T}_{2k}$. Now let $T_{2k} \in \mathscr{T}_{2k}$. We prove that there exists T_k such that $T_{2k} \in F(T_k)$. Let T be s.t each cell (i,j) of T_{2k} is replaced by $\lceil T_{2k}[i,j]/2 \rceil$. T is in $\mathscr{T}_k$ and clearly $T_{2k} \in F(T)$. Let $T \neq T' \in \mathscr{T}_{2k}$. We show that $F(T) \cap F(T') = \emptyset$. Since $T \neq T'$ then there exists a cell (i,j) such that $T[i,j] \neq T'[i,j]$. Let $a = T[i,j]$ and $a' = T'[i,j]$. For the sake of contradiction, let $T" \in F(T) \cap F(T')$. This means that both values a and a' can be mapped to the same value $a"$ in $T"$. Thus, from T we have that $a" \in \{2a-1, 2a\}$ and from T' we get $a" \in \{2a'-1, 2a'\}$. This contradicts the fact that $a \neq a'$. □

LEMMA B.7. $\forall\, T_k \in \mathscr{T}_k, \forall\, T_{2k} \in F(T_k)$*, the skyline size* $S(\Pi(T_k))$ *is less or equal to* $S(\Pi(T_{2k}))$.

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

PROOF. Assume the general case where each dimension j of a table T contains k_j distinct values. If $S = S(\Pi(T))$ increases when the cardinality of only one dimension increases then by repeating the process for all the dimensions the skyline size will also increase. Without loss of generality, assume f is applied to D_1. Let $T^{(1)}$ be the obtained table. The first dimension of $T^{(1)}$ contains $2k_1$ distinct values. Let $S^{(1)} = S(\Pi(T^{(1)}))$ be the number of distinct tuples of skyline of $T^{(1)}$. Let t be a tuple belonging to $\Pi(T)$, t' and $\widehat{t}$ be the two possible images by f of t such that $t'[D_1] = 2t[D_1]$ and $\widehat{t}[D_1] = 2t[D_1] - 1$, in other words $f(t) \in \{t', \widehat{t}\}$. Hereinafter, $f(t)$ means independently t' or $\widehat{t}$ appearing as image by f of t. It is obvious that if $t \in Sky(\Pi(T)) \Rightarrow f(t) \in Sky(\Pi(T^{(1)}))$ then $S^{(1)} \geq S$. Therefore, it suffices to show that $t \in Sky(\Pi(T)) \Rightarrow f(t) \in Sky(\Pi(T^{(1)}))$.

By contradiction

$f(t) \notin Sky(\Pi(T^{(1)})) \Rightarrow \exists u \in \Pi(T)$ such that $f(u) \prec f(t)$. However $t \in Sky(\Pi(T))$ thus $u \not\prec t$. We have to consider the following three cases:

- $u[1] = t[1]$. We know that $f(u)[2\dots d] = u[2\dots d]$, $f(t)[2\dots d] = t[2\dots d]$ and $u[2\dots d] \not\prec t[2\dots d]$ (otherwise u dominates t). We conclude that $f(u)[2\dots d] \not\prec f(t)[2\dots d]$. Therefore $f(u)[2\dots d] \not\prec f(t)[2\dots d]$ and $f(u) \prec f(t) \Rightarrow f(u)[2\dots d] = f(t)[2\dots d] \Rightarrow (u = t)$. This represents a contradiction because $u \neq t$ since all tuples are distinct in $\Pi(T_k)$.
- $u[1] < t[1] \Rightarrow f(u)[1] < f(t)[1]$. However, $f(u)[2\dots d] = u[2\dots d]$ and $f(t)[2\dots d] = t[2\dots d]$. Then $f(u) \prec f(t) \Rightarrow u \prec t$ which represents a contradiction because $u \not\prec t$.
- $u[1] > t[1] \Rightarrow f(u)[1] > f(t)[1]$. Then $f(u) \not\prec f(t)$ which represents a contradiction because $f(u) \prec f(t)$.

We can conclude that $S \leq S^{(1)}$. Let $S^{(j)}$ be the size of the skyline of table $\Pi(T^{(j)})$ which denotes the projection on D of table T in which f has been applied on all j first dimensions ($S^{(0)} = S(\Pi(T)), S^{(1)} = S(\Pi(T^{(1)})), \dots, S^{(d)} = S(\Pi(T^{(d)}))$). Consider $T = T_k$, then we have the following

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

□

PROOF OF **THEOREM**B.4. We show that the expected value $\overline{S(\Pi(T))}_{T\in\mathscr{T}_k}$ of the random variable $S(\Pi(T_k))$ is less than $\overline{S(\Pi(T))}_{T\in\mathscr{T}_{k'}}$ for every $k' = k(1+\alpha)$ where α is a positive integer. For convenience, we show the result for $\alpha = 1$. The same proof scheme can be used for arbitrary α.

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

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

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

By applying E (Expected value) to both sides, we obtain

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

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

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

Which concludes the proof of the theorem. □

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions, which were helpful to improve the clarity of the manuscript.

REFERENCES

Sameet Agarwal, Rakesh Agrawal, Prasad Deshpande, Ashish Gupta, Jeffrey F. Naughton, Raghu Ramakrishnan, and Sunita Sarawagi. 1996. On the Computation of Multidimensional Aggregates. In *proc. of VLDB conf.*

Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. 2000. Automated Selection of Materialized Views and Indexes in SQL Databases. In *proc. of VLDB conf.*

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

Kenneth S. Bøgh, Sean Chester, Darius Sidlauskas, and Ira Assent. 2014. Hashcube: A Data Structure for Space- and Query-Efficient Skycube Compression. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM*.

Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. 2001. The Skyline Operator. In *Proc. of ICDE conf.*

Upen S. Chakravarthy, John Grant, and Jack Minker. 1990. Logic-Based Approach to Semantic Query Optimization. *ACM Trans. Database Syst.* 15, 2 (1990), 162–207.

Surajit Chaudhuri, Nilesh N. Dalvi, and Raghav Kaushik. 2006. Robust Cardinality and Cost Estimation for Skyline Operator. In *ICDE*.

Thomas Eiter and Georg Gottlob. 1995. Identifying the Minimal Transversals of a Hypergraph and Related Problems. *SIAM J. Comput.* 24, 6 (1995), 1278–1304.

Eve Garnaud, Sofian Maabout, and Mohamed Mosbah. 2012. Using Functional Dependencies for Reducing the Size of a Data Cube. In *Proceedings of FoIKS conference*. Springer, 144–163.

Parke Godfrey. 2004. Skyline Cardinality for Relational Processing. In *Proc. of FoIKS Conference*.

Parke Godfrey, Jarek Gryz, and Calisto Zuzarte. 2001. Exploiting Constraint-Like Data Characterizations in Query Optimization. In *SIGMOD Conference*.

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

Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals. *Data Mining and Knowledge Discovery* 1, 1 (1997), 29–53.

Nicolas Hanusse and Sofian Maabout. 2011. A parallel algorithm for computing borders. In *Proc. of CIKM conf.*

Nicolas Hanusse, Sofian Maabout, and Radu Tofan. 2009. A view selection algorithm with performance guarantee. In *Proceedings of EDBT conference*, Vol. 360. ACM, 946–957.

Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. 1996. Implementing Data Cubes Efficiently. In *SIGMOD conf.*

Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. 1999. TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. *Computer Journal* 42, 2 (1999), 100–111.

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

Jongwuk Lee and Seung won Hwang. 2014. Toward efficient multidimensional subspace skyline computation. *VLDB Journal* 23, 1 (2014), 129–145.

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

Jingni Li, Zohreh A. Talebi, Rada Chirkova, and Yahya Fathi. 2005. A formal model for the problem of view selection for aggregate queries. In *Proc. of ADBIS conf.*

Stéphane Lopes, Jean-Marc Petit, and Lotfi Lakhal. 2000. Efficient Discovery of Functional Dependencies and Armstrong Relations. In *proc. of EDBT conf.*

David Maier. 1983. *The Theory of Relational Databases*. Computer Science Press.

Heikki Mannila and Kari-Jouko Räihä. 1992. *Design of Relational Databases*. Addison-Wesley.

Heikki Mannila and Kari-Jouko Räihä. 1994. Algorithms for Inferring Functional Dependencies from Relations. *Data and Knowledge Engineering* 12, 1 (1994), 83–99.

Michael D. Morse, Jignesh M. Patel, and H. V. Jagadish. 2007. Efficient Skyline Computation over Low-Cardinality Domains. In *Proceedings of VLDB conf.*

Noël Novelli and Rosine Cicchetti. 2001. FUN: An Efficient Algorithm for Mining Functional and Embedded Dependencies. In *Proc. of ICDT conf.*

Carlos Ordonez. 2010. Statistical Model Computation with UDFs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 22, 12 (2010), 1752–1765.

Carlos Ordonez, Wellington Cabrera, and Achyuth Gurram. 2016. Comparing Columnar, Row and Array DBMSs to Process Recursive Queries on Graphs. *Information Systems* (2016).

Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. 2005. Progressive skyline computation in database systems. *ACM Trans. Database Syst.* 30, 1 (2005).

Jian Pei, Wen Jin, Martin Ester, and Yufei Tao. 2005. Catching the Best Views of Skyline: A Semantic Approach Based on Decisive Subspaces. In *Proc. of VLDB conf.*

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

Chedy Raïssi, Jian Pei, and Thomas Kister. 2010. Computing Closed Skycubes. *Proc. of VLDB conf.* (2010).

Atish Das Sarma, Ashwin Lall, Danupon Nanongkai, and Jun Xu. 2009. Randomized Multi-pass Streaming Skyline Algorithms. In *Proceeding of VLDB*.

Haichuan Shang and Masaru Kitsuregawa. 2013. Skyline Operator on Anti-correlated Distributions. *PVLDB* 6, 9 (2013).

Cheng Sheng and Yufei Tao. 2012. Worst-Case I/O-Efficient Skyline Algorithms. *ACM Trans. Database Syst.* 37, 4 (2012).

Jaroslaw Szlichta, Parke Godfrey, and Jarek Gryz. 2012. Fundamentals of Order Dependencies. *Proc. of VLDB conf* (2012).

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

Hong Yao and Howard J. Hamilton. 2008. Mining functional dependencies from data. *Data Mining and Knowledge Discovery* 16, 2 (2008), 197–219.

Yidong Yuan, Xuemin Lin, Qing Liu, Wei Wang, Jeffrey Xu Yu, and Qing Zhang. 2005. Efficient Computation of the Skyline Cube. In *Proc. of VLDB conf.*

Yihong Zhao, Prasad Deshpande, and Jeffrey F. Naughton. 1997. An Array-Based Algorithm for Simultaneous Multidimensional Aggregates. In *Proc. of SIGMOD conf.*