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 $T(Id, D_1, \ldots, D_d)$, the skycube of T is the set of skylines with respect to to all nonempty subsets (subspaces) of the set of all dimensions $\{D_1, \ldots, D_d\}$. To optimize the evaluation of any skyline query, the solutions proposed so far in the literature either (i) precompute all of the skylines or (ii) 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 with respect to 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 article, 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 with respect to 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 datasets show that our solutions generally outperform state-of-the-art algorithms.

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

Additional Key Words and Phrases: Materialization, implementation

ACM Reference Format:
Sofian Maabout, Carlos Ordonez, Patrick Kamnang Wanko, and Nicolas Hanusse. 2016. Skycube materialization using the topmost skyline or functional dependencies. ACM Trans. Database Syst. 41, 4, Article 25 (November 2016), 40 pages.
DOI: http://dx.doi.org/10.1145/2955092

1. INTRODUCTION

Multidimensional database analysis has been a hot research topic over the past decade. Precomputation 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

This work was 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 was 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. K. Wanko, and N. Hanusse, LaBRI-University of Bordeaux, 351 cours de la Libération, 33405 Talence, France; emails: {maabout, pkamnang, hanusse}@labri.fr; C. Ordonez, University of Houston, Department of computer science, TX, 77204, USA; email: ordonez@cs.uh.edu.

DOI: http://dx.doi.org/10.1145/2955092

intuitively represents the set of aggregation queries with all potential subsets of grouping attributes. After an initial series of works concentrating on efficient ways to fully materialize a data cube, e.g., see Agarwal et al. [1996] and Zhao et al. [1997], it was rapidly recognized that this solution was unfeasible in practice due to the large amount of memory space and the processing time needed since the number of queries is exponential with respect to the number of dimensions. Therefore, the question 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 constraints considered. For example, Harinarayan et al. [1996], Agrawal et al. [2000], and 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] consider the problem differently: given a prescribed queries evaluation time, find the minimal part of the data cube that 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 that 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 preorder among the values of every attribute and specify 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 navigate through the multidimensional space by asking skyline queries over chosen sets of attributes. To optimize the response times, several algorithms aiming at fully materializing the skycube, i.e., precompute all possible skylines, have been proposed (see 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 of a closed skycube. The authors propose an algorithm for identifying equivalent skylines to save memory space by storing just one copy of the same query result. A second solution is the so-called 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—that is, 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.

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 in identifying important skyline 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

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

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 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 with respect to 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) is faster to obtain, and (iii) significantly reduces query processing times. Moreover, our algorithms get asymptotically better than competing algorithms when the number of dimensions or data size grows. Therefore, our solution is a good trade-off between fast and space-efficient skycube materialization and efficient query evaluation.

Article organization. The following section gives the main definitions and notations used throughout the article. 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 FDs. To do so, we show that the existence of FDs implies an inclusion between skylines and 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 Table I, which describes 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—that is, 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 that r_1 *dominates* r_3) because it is closer to the beach, wider, and not worse in terms of price. However, r_1 and r_2 are incomparable: r_1 is better with respect to the price, whereas r_2 is better in terms of distance. The skyline of T with respect to 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 with respect to D and S is $\{r_2\}$. Note that the best rooms with respect to just the price is $\{r_1, r_3\}$. This example shows that depending on the attributes that we consider (users preferences), we obtain different sets of best rooms. Moreover, we observe that there

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 $Sky(X)$	Skyline of T with respect to X
$\pi_X(T)$	Projection of T on X with set semantics
$\|X\|$	Cardinality of $\pi_X(T)$
$\|\|X\|\|$	Number of attributes of X
$\mathcal{S}(T)$ or simply $\mathcal{S}$	Skycube of T
$\|Sky(X)\|$	Size of $Sky(X)$
S	Subskycube of $\mathcal{S}$
k	Number of distinct values per dimension

may be no inclusion relationships between these sets: neither the skyline with respect to P is included in that with respect to P, D, and S nor the converse even if we have an inclusion of attributes sets, i.e., $\{P\} \subseteq \{P, D, S\}$. This nonmonotony of skyline 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 with respect to P from the precomputed one with respect to P, D, and S or vice versa without accessing the input table.

2.2. Definitions

We now introduce basic definitions used throughout the article. Let T be a table whose set of attributes $Att(T)$ is divided into two subsets $\mathcal{D}$ and $Att(T)\backslash\mathcal{D}$. $\mathcal{D}$ 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 with respect to X, or t' X-dominates t, noted $t' \prec_X t$, if and only if for every $X_i \in X$ we have $t'[X_i] \leq t[X_i]$ and there exists $X_j \in X$ such that $t'[X_j] < t[X_j]$. The skyline of T with respect to $X \subseteq \mathcal{D}$ is defined as $Sky(T, X) = \{t \in T \mid \nexists t' \in T \text{ such that } t' \prec_X t\}$. To simplify notation and when T is understood from the context, sometimes we omit T and 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 article.

Table III. 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 with respect to 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*—that is, neither $X \subseteq X' \Rightarrow Sky(X) \subseteq Sky(X')$ nor $X \subseteq X' \Rightarrow Sky(X) \supseteq Sky(X')$ is true. For instance, $Sky(ABD) \not\subseteq Sky(T, ABCD)$ and $Sky(D) \not\supseteq Sky(AD)$. This makes partial materialization of skycubes harder than classical data cubes.

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 to minimize both its storage space and computation time. Before formally stating the problem that we address, we exhibit some properties holding between the subspace skylines of a skycube.

3.1. Skyline 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 $\forall t \in T$, and 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 $\forall v \in T \backslash Sky(T, Y)$, $\exists u \in Sky(T, Y), u \prec_Y v$. 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 that $Sky(Sky(T, BC), B) = \{t_2\}$ (because $t_2[B] = 3 < t_4[B] = 4$), and this does not correspond to $Sky(T, B)$ since the latter actually has an extra tuple, which is t_1. However, note 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, we have the following theorem.

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 and then fill all missing dimensions with the special symbol $*$. These generalized tuples define the skyline *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.

Pei et al. [2005] introduced the *skyline groups lattice*, which is quite different from the 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 subspace 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 Section 6.

Lemma 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, with respect to X, from $Sky(Y)$ to obtain S_1. The second step consists of computing the skyline of S_1 by considering all of its dimensions, i.e., X. This gives the set S_2, which is then joined with table T.

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, Lemma 3.1 guarantees that all $t' \in Sky(X)$ are retrieved. Hence, all and only $t' \in Sky(X)$ are returned.

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

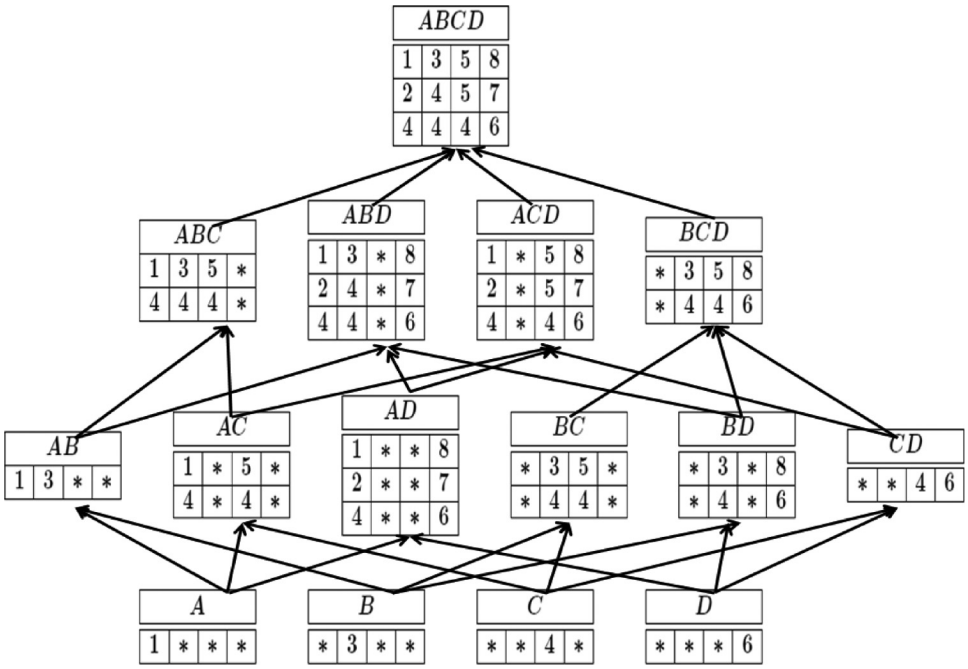


Fig. 1. Lattice of the skycube patterns.

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 $T \bowtie S_2$;

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' subject to $t'[AB] = \langle 1, 3\rangle$. 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, for example, 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 had better perform $Sky(T, X)$, whose complexity is $O(|T|.|Sky(X)|)$.

3.2. Topmost Skyline–Based Materialization

Lemma 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

ALGORITHM 2: SemiNaivePartialSkyCube

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 $\mathcal{I}$;

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, i.e., $|Sky(\mathcal{D})| = O(\sqrt{n})$. 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 this is the case, then every tuple in the topmost 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 $X \subseteq \mathcal{D}$, $t \in Sky(X)$ if and only if $t[X] = \langle 0, \ldots, 0\rangle$. 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* 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 seminaive 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 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} with respect to every D_i 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 topmost 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

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) if and only if 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 if and only if it contains a sufficient set of skycuboids that 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 ICSs. If, for example, S_1 is materialized, then the query $Sky(T, A)$ can be evaluated as $Sky(Sky(ABD), A)$ using four tuples instead of using table T, which contains six tuples. Note that $\mathcal{S}_3 = \{Sky(ABCD)\}$ is not an ICS because, for example, 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 ICSs.

Definition 3.10 (*Minimal Information Completeness*). S is a *minimal information-complete subskycube* (MICS) if and only if 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 to answer all 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 $(X, Y) \in E$ if and only if $X \subset Y$ 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 remainder of this section, we devise a method leveraging the FD concept to avoid the full materialization. For this, we show that the presence of some FDs implies the inclusion of skylines. Hence, we can avoid to compute some of them.

3.4. Using FDs to Discover Inclusion Between Skylines

Recall that the FD $X \rightarrow Y$ is satisfied by T if and only if 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 FD can be used to identify a monotonic inclusion between two skylines.

Theorem 3.13. *Let $X \rightarrow Y$ 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 $t \in Sky(T, X) \backslash Sky(T, XY)$. Since $t \notin Sky(T, XY)$, there must exist t' that dominates t with

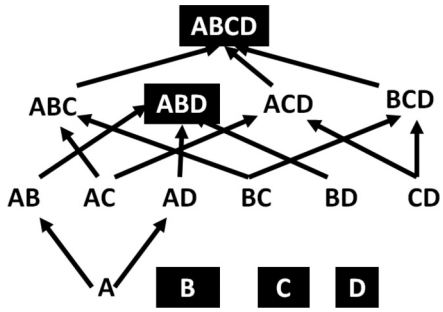


Fig. 2. Inclusions between skycuboids captured by FDs.

respect to XY, i.e., $t' \prec_{XY} t$. t' cannot dominate t with respect to X because otherwise t cannot belong to $Sky(T, X)$. We conclude that $t[X] = t'[X]$. On the other 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 with respect to XY. □

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

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

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 that 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 if and only if $|\pi_{D_i}(T)| = n$. In other words, all 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 preceding 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 FD terminology. In particular, $\forall X, Y$ and 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 skyline 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 $(X, Y) \in \mathcal{E}$ if and only if $Sky(X) \subseteq Sky(Y) \in \mathcal{I}$. 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. □

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 $ClosedSub = ClosedSub \backslash \{X\}$;
9 Break;
10 **Return** $ClosedSub$ and their respective skylines;

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.

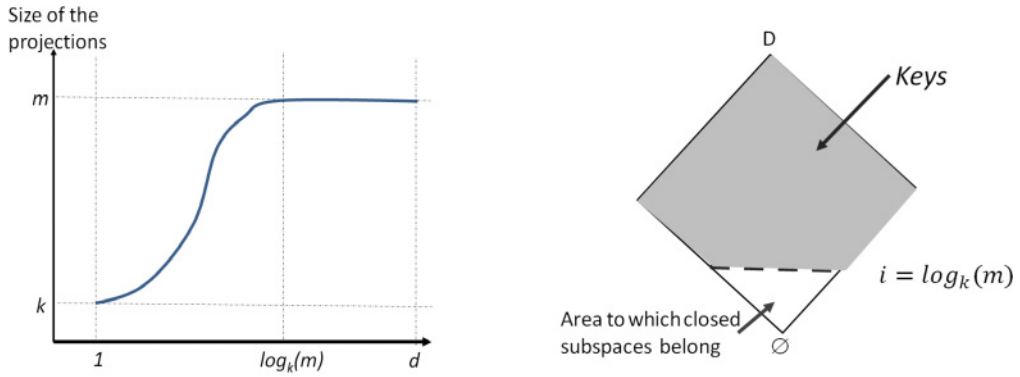
Now we provide some properties of the elements of Γ that allow us to identify them efficiently. By Theorem 3.13, we conclude that a subspace X belongs to Γ if and only if there is no subspace Y such that $X \cap Y = \emptyset$ and the FD $X \rightarrow Y$ is satisfied by T. In the FD literature, these subspaces are classically called *closed* sets of attributes. We recall briefly the definition and suggest referring to Maier [1983] and 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 with respect to F, noted X_F^+ or simply X^+, is the largest subspace Y such that $F \vdash X \rightarrow Y$, where $\vdash$ represents the *implication* between FDs. X is *closed* if and only if $X^+ = X$.

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

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 FDs that we consider are those that hold in the instance T. These are not supposed to be known beforehand. Thus, we distinguish between those FDs that act as constraints that 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 with respect to the size of $\mathcal{F}$ [Mannila and Räihä 1994]. The description of the algorithm that we propose for this purpose is presented in Appendix A.

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



(a) Size of projections with respect to dimensionality $||X||$

(b) Closed subspaces in the subspace lattice

Fig. 3. Keys and closed subspace behaviors.

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 $Sky(B) \subseteq Sky(ABD)$, $Sky(C) \subseteq Sky(ABCD)$, $Sky(D) \subseteq Sky(ABD)$, 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, e.g., $||ABC|| = 3$, 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 if and only if $|X| = m$. 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 with respect to 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 illustrated in Figure 3(b). Note

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

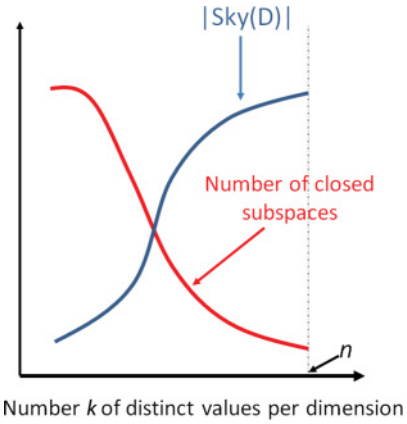


Fig. 4. Evolution of skyline size and number of closed subspaces with respect to cardinality k.

that the area to which closed subspaces belong may contain nonclosed subspaces as well.

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 a 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 in Section 3.2. The following theorem formalizes this result.

THEOREM 3.20. *Let n and d be fixed. Let $\mathcal{T}_k$ 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 in the skylines of all tables $T_k \in \mathcal{T}_k$. 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 preceding 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 five 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 five tuples among these allows 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 evolve with respect to the number of distinct values k when $n < k^d$. The main conclusion is that when the number of closed subspaces increases, the size of the skyline tends to

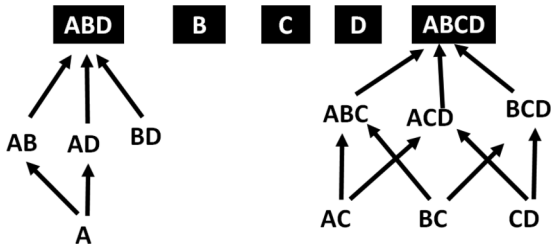


Fig. 5. Partial skycube.

decrease. This is why in the case of no FDs, we are almost sure that the topmost skyline is small.

4. QUERY EVALUATION

In this section, we show how skyline queries are efficiently evaluated when the MICS is materialized. To simplify the presentation, we assume that materialized skycuboids are those associated to X, where X is closed. In other words, 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 $Sky(T, A)$ from T, thus using six tuples, we rely on $Sky(T, ABD)$ containing only four tuples. The closure of every X can either be hard coded or 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 dataset with n tuples have a worst-case time complexity in $O(n^2)$ that reflects the number of comparisons. Thus, 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 for $|Sky(X^+)|$ to 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 $Sky(X)$ with respect to their Y values—that is, $t_i \equiv t_j$ if and only if $t_i[Y] = t_j[Y]$. 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 $t_i' \in Sky(Sky(X), Y)$ if and only if $t_i \in Sky([\mathcal{P}_Y(X)], Y)$.*

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 $Sky(Y)$ as well.

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 returns $\{t_2\}$. Since t_1 is equivalent to t_2, then t_1 is also in $Sky(A)$.

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

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

PROOF. Because $T \models X \rightarrow Y$ if and only if $\mathcal{P}_X(T) = \mathcal{P}_{XY}(T)$. □

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

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 us first recall the definition of skycuboids of Type I.

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

In other words, $Sky(X)$ is of Type I if and only if 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 $Y \rightarrow X$, 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 with respect to X gives only one part. Therefore, every tuple of this part is necessarily in $Sky(Y)$. Since $Y \subseteq X$, and $Y \rightarrow X$, then $Sky(Y) \subseteq Sky(X)$, and we conclude that $Sky(X) = Sky(Y)$. □

Example 4.10. Since $Sky(AD)$ is of Type I and since $A \rightarrow D$ 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 that we developed in the previous section are no longer valid. Thus, we need to use some standard skyline algorithm for evaluating $Sky(X)$ using a materialized $Sky(Y)$ without all of the optimization that we have seen so far.

Example 4.11. Table III shows that for closed subspaces C and $ABCD$, we have, for example, $Sky(C) \subseteq Sky(ABCD)$. 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. Thus, the partition with respect to $ABCD$ contains three parts, whereas the partition of these tuples with respect to C contains only two tuples, namely $\langle 3 \rangle$ and $\langle 4 \rangle$.

The preceding 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 naive solution that consists of evaluating every skyline from T, previous works exhibit derivation properties and cases where computation sharing among skylines is possible so as to speed up this process.

Since this materialization evaluates every possible skyline query, the previous properties that we identified can easily be exploited in this context. Hence, we propose FMC (short 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; (ii) then it computes their respective skylines; and (iii) finally, for every nonclosed 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 the case of a small topmost skyline, all we need 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 saw in Section 3.5. Despite its simplicity, FMC turns out to be very efficient in practice and outperforms state-of-the-art algorithms, which is shown in Section 7.

ALGORITHM 4: FMC Algorithm

```
   Input: Table T
   Output: Skycube of T
1  Let S_D = Sky(T, D);
2  if |S_D| is small then
3      T_clean = cleanup(T);
4      create index I;
5      for every X ⊂ D do
6          // Loop executed in parallel
7          Let S = Sky_X_from_Sky_Y(T_clean, S_D, X);
8          Skycube_T = Skycube_T ∪ {S};
9      Return Skycube_T;
10 else
11     Closed = ClosedSubspaces(T);
12     foreach X ∈ 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 ⋃_{X∈2^D} Sky(X);
```

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_D = Sky(T, D);
6      if |S_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 I;
10         Return S_D;
11     else
12         // We need to minimize the memory storage space
13         Return MICS(T);
```

5. MAIN ALGORITHM: ASSEMBLING ALL COMPONENTS TOGETHER

In this section, we summarize our proposal by assembling the different procedures that we introduced so far. On the one hand, we show how skycube materialization is addressed, and on the other, 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 the case of the second option, the algorithm may decide to store only the topmost skyline if it is 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.

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_D| is small then
6          // S_D is always materialized
7          Return Sky_X_From_Sky_Y(T, S_D, X);
8      else
9          if X+ is materialized then
10             Return Sky(Sky(X+), X);
11         else
12             // i.e., only skylines in MICS are stored
13             Let Y be the remaining ancestor of X+ in MICS;
14             Return Sky(Sky(Y), X);
```

However, with respect to query evaluation, it depends on how the skycube is materialized. Algorithm 6 shows how skyline queries are handled. In the 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 FDs, 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 with respect to the size of the cover set [Mannila and Räihä 1992]. Additionally, when computing MICS, a skyline $Sky(X)$ of some closed X is removed if and only if there exists some closed Y such that Y is an *ancestor* of X. If it is the case, we need to keep this information to be able to answer $Sky(X)$ and 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], and Lee and won Hwang [2010]. Some algorithms have been specifically tailored to the case where dimensions have low cardinalities, e.g., see Morse et al. [2007]. All of these algorithms have $O(n^2)$ worst-case complexity where n is the size of table T. The pioneering work of Börzsönyi et al. [2001] considered external memory cost and showed the inadequacy of SQL to efficiently evaluate skyline queries. Nonetheless, the algorithms proposed there suffer from the polynomial time complexity as well. 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 preprocessing 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 that we propose in the present article do not rely on any skyline computation algorithm. Thus, all 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 2010, 2014; Pei et al. 2005; Yuan et al. 2005]. To avoid the naive 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 of 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 $Sky(X)$. 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 experimentally compare 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 datasets show, however, that in practice, MICS is generally computed 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 of 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 these pairs is that t belongs to all 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 with respect to $ABCD$, 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, whereas 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. With regard to storage space, the two methods seem incomparable in general. For example, if $Sky(A) \subset Sky(ABC)$ and $Sky(A) \not\subseteq Sky(AB)$, then MICS does not store $Sky(A)$, whereas the skyline group lattice stores some information about the tuples in $Sky(A) \backslash Sky(ABC)$. However, 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, whereas the skyline group can avoid storing 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 speed up 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 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 $Sky(X)$ if and only if there is no $Y \subset X$ such that $t \in Sky(Y)$. $min\_sky(X)$ denotes the minimal skyline tuples of $Sky(X)$. The CSC consists simply of 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, $t \in Sky(T, X)$ if and only if $\exists Y \subseteq X$ 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 that the traversal of all subsets of X, an exponential number, collect all tuples and then execute a standard skyline query to remove those that are dominated with respect to X. As shown later in Section 7, 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 out 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 to which it belongs. Evaluating $Sky(X)$ consists 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 latter 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 by those authors show that in general, hashcubes compress the skycube storage space by a factor 10 but is about 10 times slower for query evaluation. However, we note 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 computing the whole skycube. However, achieving a 10% compression ratio means that the hashcube structure has an exponential growth with respect to d since the skycube size grows exponentially. Our empirical study shows a linear growth of MICS size with respect to d, providing evidence that MICS scales better with respect to d.

Even if FDs have already been used in semantic query optimization [Chakravarthy et al. 1990; Godfrey et al. 2001] and in data cubes used in 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. However, 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 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, whereas the cornerstone for skyline queries is the different orders defined among the different attributes.

Some works have been proposed to estimate the skyline size, e.g., Godfrey et al. [2007], Shang and Kitsuregawa [2013], and Chaudhuri et al. [2006]. We could have used those results to show the relationship that we established between the number

Table IV. Compared Algorithms With Respect to Tasks

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

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] consider the same data distribution as the one that we used (independence of dimensions), whereas Shang and Kitsuregawa [2013] focus on anticorrelated data. Chaudhuri et al. [2006] does not consider any data distribution hypothesis, but the estimator is not a closed formula. A noticeable exception is in the work of Godfrey [2004], where repeated values in dimensions are allowed. However, Godfrey's 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 the distinct values hypothesis. But this result does not show the skyline size trend with respect to 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 assist in 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 reoptimized 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 three directions. First, we compare our solution to the previous works targeting the skycube full materialization. In this scope, we analyze scalability with respect to both d (dimensionality) and n (table size). Our solution outperforms state-of-the-art algorithms for full materialization when both data and dimensions get larger. Second, 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) memory space used by the structures, and (c) query execution time. Third, 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 3.3GHz hexacores processors, and a 1Tb disk under Redhat Entreprise Linux OS.

7.1. Datasets

We used both real and synthetic datasets. Table V describes NBA, IPUMS, and MBL real data, which are well known in the skyline literature. As they are relatively small, we artificially augmented their respective sizes by just copying them multiple times. For example, NBA50 is obtained from the NBA dataset by copying every tuple 50 times. We call these three sets and their variations *benchmark* datasets. In addition, we used two larger real datasets, namely USCensus and Householders. We used them because of their larger size, but more importantly, we used them because they satisfy a very different number of FDs in either dataset (almost zero in USCensus but many in Householder). Moreover, we use synthetic data generated by a software available at `pubzone.org` and provided by the authors of Börzsönyi et al. [2001]. It takes as

Table V. Real Datasets

Dataset	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

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. Attribute 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 four threads just by adding a compilation directive (pragma) to the source code as follows:

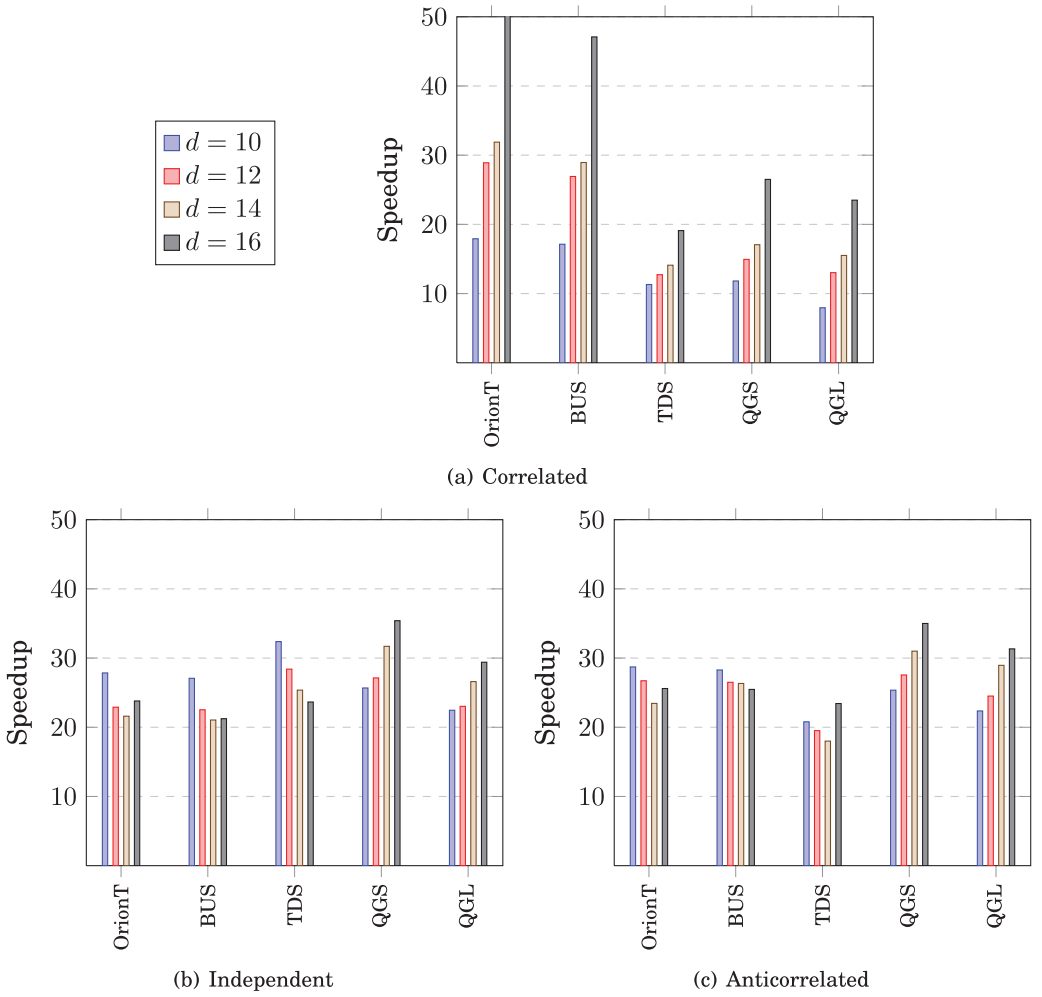
```
♯pragma omp parallel for num_threads(4)
  for(int i=0; i<1000; i++){f(i); }
```

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 $f(i_2)$, and 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, e.g., by using locks, so some threads need to wait until others free their locks, which penalizes the parallel execution. Moreover, following Amdahl'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] and OrionTail [Raïssi et al. 2010], as well as the most recent proposals of QSkyCubeGS and QSkyCubeGL reported in Lee and won Hwang [2014]. We used the authors' implementations without any change.[3] All of 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

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



(a) Correlated

(b) Independent

(c) Anticorrelated

Fig. 6. Speedup with respect to dimensionality d ($n = 100K$).

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, as its sequential execution cannot be 12 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 rationale behind this choice is to be sure that skyline evaluation complexity becomes linear in n.

7.3.1. Synthetic Data: Scalability With Respect to 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 Figure 6. Figure 6(a) shows that the speedup obtained by FMC increases with respect to d. This tends to demonstrate that for this kind of data distribution, FMC is more appropriate to use when d gets larger. A more

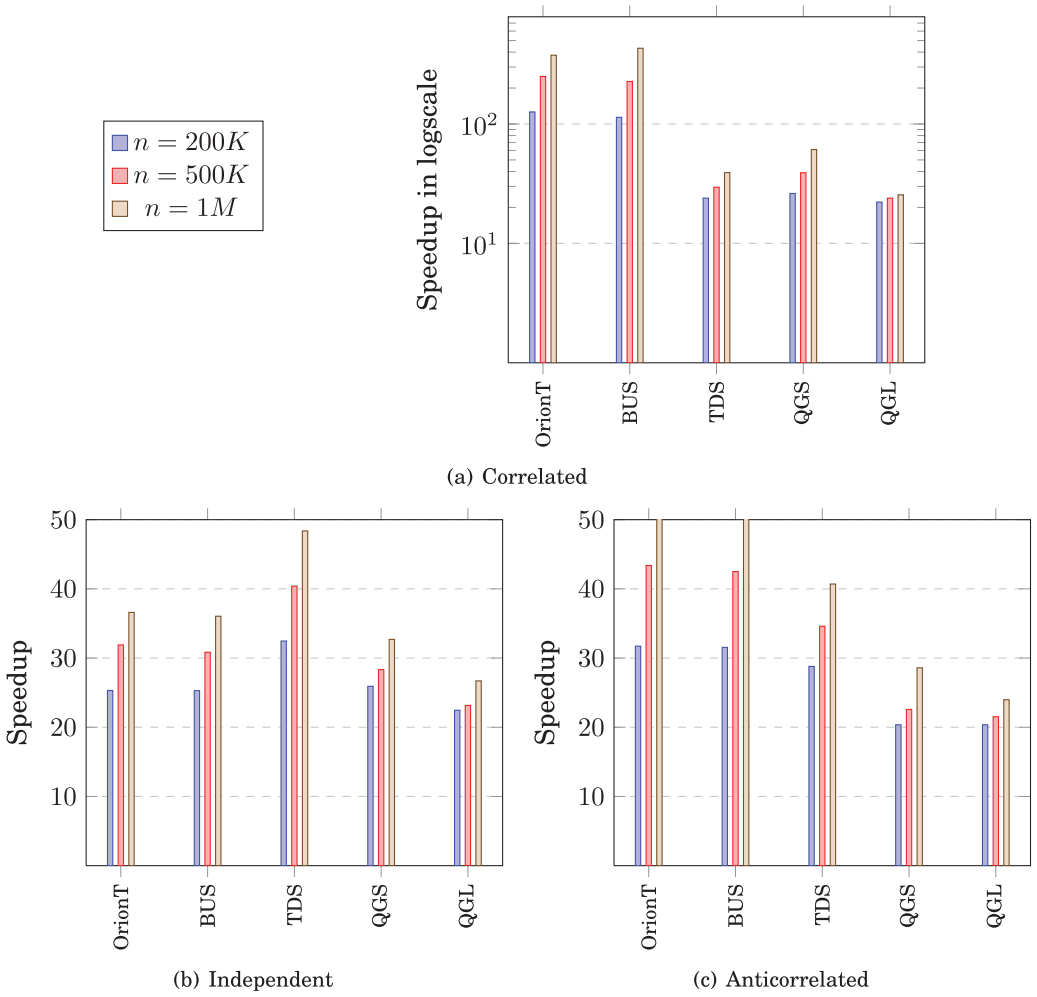


Fig. 7. Speedup with respect to data size n ($d = 16$).

detailed look to the performance of QSkycubeGL shows that when $d = 10$, the speedup 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, 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, Figure 6(b) and (c) show that with $d = 10$, we already have a speedup larger than 12. Even if we note that all algorithms do not have the same behavior with respect to d growth, FMC is consistently much more efficient than them in all cases.

7.3.2. Synthetic Data: Scalability With Respect to 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 speedup increases uniformly when n increases. However, 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 datasets 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. Thus, 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 anticorrelated and even independent data. Actually, in the former situations, FMC almost turns out to be the naive algorithm, as 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 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 shoboth the dimensionality and number of distinct values per dimension increase.

Remark 7.2. FMC can be easily optimized by better exploiting skyline 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 subspace lattice.

7.3.3. Real Data Analysis. To avoid the biases introduced by the way synthetic data are generated, we performed the same kinds of experiments as before by using three real datasets. USCensus is a well-known dataset 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, the dataset is interesting because of its real data distribution. We picked 20 columns at random for our experiments. The second dataset is publicly available at the Web site of the French National Institute of Statistics and Economic Studies and describes `householders` in southwest region in France. It has more than $2.5 * 10^6$ tuples described by 67 variables. Here we picked 20 attributes as well, but by contrast to USCensus, these columns have a meaningful ranking semantics with regard to the number of persons living in the house, number of rooms, and so on. Although both datasets have almost the same number of tuples and the same dimensionality, their respective data distributions are radically different: whereas 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 $d \geq 10$.

USCensus dataset. 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 were too time consuming. Nevertheless, we note that starting from $d = 16$, both QSkyCubeGS and QSkyCubeGL saturated the total 24Gb of available memory and started to swap to disk during the computation, which is why we stopped their execution, as 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. Thus, memory saturation is not due to the size of the skycube but rather to

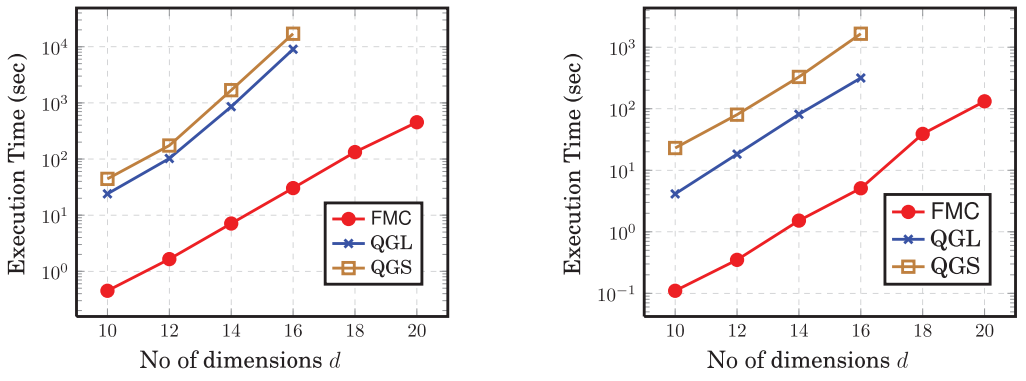


Fig. 8. Full skycube computation with large real datasets.

the STreeCube shared data structure used by those algorithms to speed up execution time. Note the rapid growth of the speedup when d increases, reaching three orders of magnitude when $d = 16$ with QSkyCubeGL and almost 600 with QSkyCubeGS. A specificity of this dataset is that its dimensions have a very small number of distinct values: about 10 distinct values each. This makes FDs difficult to satisfy when the number of tuples is quite large. In fact, this dataset 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 3,873 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 seminaive skycube computation that is chosen by FMC (lines 2 through 6 in Algorithm 4). Hence, computing $Sky(X)$ is performed by a join operation that simply consists of 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 dataset. 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 dataset and the previous one is that many FDs are satisfied. The number of closed subspaces is around 29,000 out of the $2^{20} - 1$ subspaces. The number of distinct values per dimension varies from 2 to 4,248. The topmost skyline has 154,752 tuples. This shows the effectiveness of using the skylines of the closed subspaces to compute the rest of the skylines. Indeed, even if the speedups that we obtain are less impressive than those with the previous dataset, FMC is still 50 times faster than QSkyCubeGL, and even more comparatively to QSkyCubeGS.

Benchmark real datasets. NBA, IPUMS, and MBL are relatively small sets. For these datasets, 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. However, note that when the number of dimensions is small,

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

Table VI. Execution Times (in Seconds) for Small Real Datasets

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

Note: FMC is executed with 1 and 12 threads.

Table VII. Execution Times (in Seconds) for Artificially Enlarged Real Datasets

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

Note: FMC is executed with 1 and 12 threads.

which is the case with IPUMS, the speedup of FMC(12) with respect to 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 naive algorithm with IPUMS: for every subspace X, compute $Sky(T, X)$. This loop was executed using 12 threads as well. The algorithm terminates after 0.68 seconds, providing a speedup of 3.8 over QSkyCubeGL. Thus, the question of whether using shared data structures and/or computations is worthwhile remains open.

We artificially enlarged these datasets by copying their content multiple times. Table VII shows the execution times. We note that FMC scales almost linearly with respect to data size. QSkyCubeGS does not scale well. QSkyCubeGL seems to be the algorithm with the best behavior regarding data size growth. However, note that it is the algorithm that is more memory consuming. For example, we tried to execute QSkyCubeGL with NBA100 and it saturated the memory, whereas FMC handled this dataset 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 to those of QSkyCubeGS and QSkyCubeGL. The speedup of our algorithm is almost constant.

7.4. Partial Skycube Materialization

7.4.1. MICS Versus Closed Skycubes. We compare our solution to OrionClos, the algorithm proposed in Raïssi et al. [2010], to compute the closed skycubes. We make the

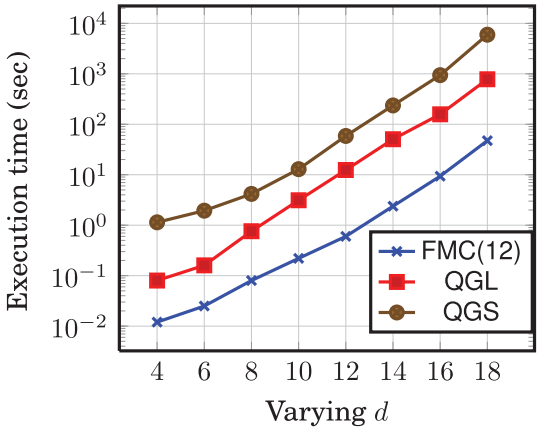


Fig. 9. Full materialization with MBL10.

Table VIII. MICS Versus Orion

NBA		
Technique	Time to Build (sec)	Materialized Skycuboids
MICS	12.8	5,304
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	29,155
Orion	11,069	43,075

comparison by considering (i) the storage space usage and (ii) the speed for materializing the subskycube. We consider the three real datasets used by those authors: 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 that 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 to increase with data size n. In addition to these datasets, we tried to test OrionClos with larger datasets, but it was unable to terminate in a reasonable time. For example, 36 hours were not sufficient to process a correlated dataset with $d = 20$ and $n = 100K$. By contrast, it took 20 seconds to find and materialize the MICS of the same dataset. This is because synthetic data generators tend to return distinct values, making the computation of equivalence classes required by OrionClos harder.

7.4.2. Compressed Skycubes. In this section, we compare our proposal to the CSC approach of Xia et al. [2012] following three criteria: time to build, storage space, and query execution time. For this purpose, we use the three real datasets, NBA,

Table IX. Partial Skycube Versus 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	47,654	921,911	7,212
Partial Skycube	10	78	58
IPUMS			
Technique	Time to Build (sec)	Storage	Query Time (sec)
CSC	3,776	129,812	142
Partial Skycube	6	73,382	2
Correlated			
Technique	Time to Build (sec)	Storage	Query Time (sec)
CSC	4,533	498	10
Partial Skycube	0.015	2	0.03
Independent			
Technique	Time to Build (sec.)	Storage	Query Time (sec)
CSC	36,123	921,911	6,212
Partial Skycube	35	83,650	78
Anticorrelated			
Technique	Time to Build (sec.)	Storage	Query Time (sec)
CSC	51,263	4,556,789	11,972
Partial Skycube	62	121,347	129

MBL, and IPUMS, as well as three synthetic datasets with 16 independent/correlated/anticorrelated dimensions and $100K$ tuples. Regarding query evaluation performance, we consider a workload of all $2^d - 1$ possible skyline queries 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 relies on MICS. The results are depicted in Table IX. As can be observed, our proposal outperforms CSC in all criteria. Note that for real datasets NBA and MBL, as well as the correlated set, our solution requires the storage of the topmost skylines, which contain 3, 78, and 2 tuples, respectively.

7.4.3. Partial Skycube Versus Hashcube. We use two datasets, namely `NBA` and `Householder`, to illustrate the advantages and limitations of the hashcube structure [Bøgh et al. 2014]. Whereas the first dataset is small and correlated, the second one is much larger and noncorrelated. We report the size of the hashcube and the partial skycube that 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.

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

For the second dataset, 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

Table X. Partial Skycube Versus Hashcube

NBA		
Technique	Storage	Query Time (sec)
Hashcube	541,316	0.029
Partial	3	3.5
Householder		
Technique	Storage	Query Time (sec)
Hashcube	2,431,675,126	_
Partial	49,022,451	128

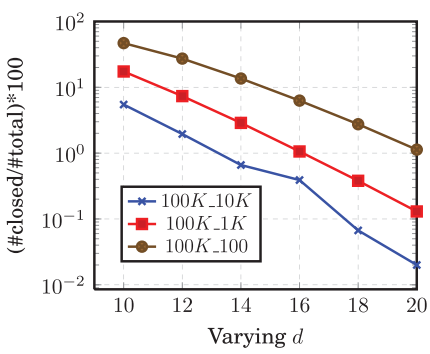
continue the experiment for the query part, as it would had taken too much time. Note that our solution is about 300× smaller than the skycube.

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.

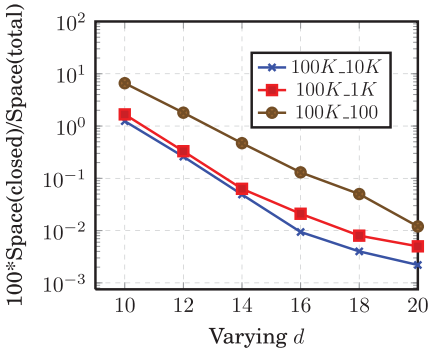
7.4.4. Storage Space Analysis. In the second part of the experiments, we generated a set of independent synthetic data by fixing the following parameters: d is the number of attributes, n is the number of tuples, and k is the average number of distinct values taken by each attribute. For example, $100K\_10K$ designates a dataset where $n = 100K$ and the number k of distinct values per dimension is $10K$. Since the data generator returns floats in a $[0, 1]$ interval, it suffices to keep the first f decimal digits to get a dataset where every dimension has $k = 10^f$ distinct values on average. For example, 0.0123 is replaced by 12 if $k = 10^3$. In doing so, the correlation between the different columns is preserved. Even if we report only the results obtained with the independent datasets, we should mention that we performed the same experiments with correlated and anticorrelated data. The conclusion was the same.

We first investigate the number of closed subspaces comparatively with 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 dataset $100K\_10K$ when $d = 10$. About 7% of the subspaces out of the total $2^{10} - 1$ 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 $100K\_10K$, whereas 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 with respect to the closed subspaces never exceeds 10% of the whole skycube size. For example, even if we materialize 84.5% of the skylines of the skycube related to the $100K\_100$ dataset when $d = 10$, this storage space represents less than 10% of the related skycube size. This would indicate that either our proposal tends to avoid the materialization of heavy skycuboids 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.

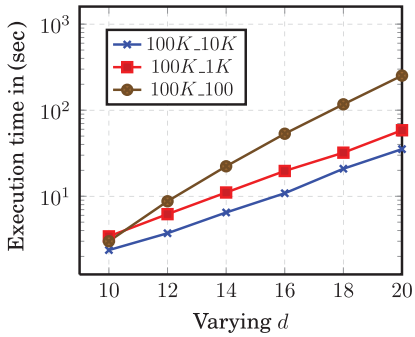
Finally, Figure 10(c) shows the execution times for the datasets we have considered so far. We stress the fact that these times represent (i) the extraction of the closed subspaces and (ii) their respective skylines. Clearly, the less distinct values per dimension, the more closed subspaces there are and the more time is needed to compute them.



(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.

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)—that is, when k decreases, the number of closed subspaces increases, whereas their respective sizes decrease (see Theorem 3.20).

Therefore, the main lesson that we retain from the preceding experiments is that when the number of closed subspaces increases, the size of the skylines decreases. Thus, even if our solution stores more skylines, this does not necessarily mean that it uses more storage space. The second lesson that 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.

7.5. Query Evaluation

In this section, 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` dataset, vary d from 16 to 20, and vary n from $500K$ to $2M$. We generate 1,000 distinct skyline queries among the $2^d - 1$ possible queries as follows. The $2^d - 1$ subspaces are listed and sorted with respect to a

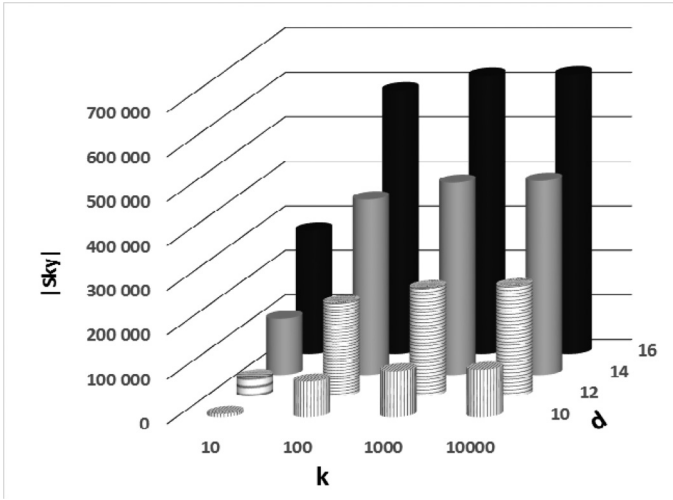


Fig. 11. Skyline size evolution with respect to k and d with $n = 10^6$.

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%)

lexicographic order in a vector Q. We randomly and uniformly pick 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 1,000. 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 pieces of 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 1,000 queries from the materialized skycuboids, whereas it takes 99.92 seconds when T is used. The execution time is therefore divided by more than 2,000. 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 in 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. CONCLUSIONS AND FUTURE WORK

In this article, we show how the classical concept of FDs may be 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 attribute 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 dataset), and (iii) the speed to 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 datasets, 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 that 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 recomposition constraint with the help of FDs or maybe with other classes of dependencies, such as order dependencies. In this article, 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. We plan to investigate 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 believe that it can also be useful in the context of incremental maintenance. Since FDs are invariant on isomorphic data transformations, the inclusions detected by the FDs are invariant with respect to 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.

APPENDIXES

A. COMPUTING THE CLOSED SUBSPACES

The naive algorithm for finding the closed sets of attributes consists simply of 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 that do not determine A. Thanks to the antimonotonicity of functional dependencies (FDs), 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. To efficiently find these maximal subspaces, our algorithm tries to exploit this antimonotonic property by avoiding to test[5]

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

the obviously not closed subspaces, as well as testing all potentially closed subspaces. In the remainder of this section, we formalize these observations.

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

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

LEMMA A.1. *For each $X \in 2^{\mathcal{D}_i}$, if there exists $X' \in Det_{A_i}$ subject to $X' \subseteq X$, then X is not closed.*

PROOF. $X' \in Det_{A_i}$ means that $T \models X' \rightarrow A_i$. Hence, $T \models X \rightarrow A_i$, 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, for example, $BCD \supset BD$.

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

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

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

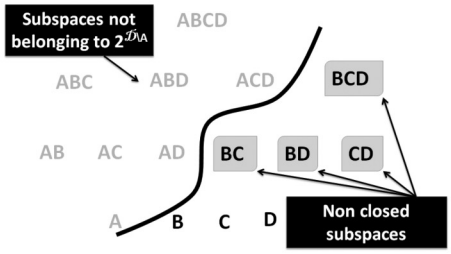
PROOF. The first item is obviously true, and thus we prove the second one. Let us first show that the implication $X \in 2^{\mathcal{D}} \backslash \mathcal{C}^- \Rightarrow X$ is closed. For the sake of contradiction, let $X \in 2^{\mathcal{D}} \backslash \mathcal{C}^-$, $X \neq \mathcal{D}$, and suppose 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 $\Rightarrow X \in 2^{\mathcal{D}} \backslash \mathcal{C}^-$. Assume the contrary, i.e., X is closed and $X \notin 2^{\mathcal{D}} \backslash \mathcal{C}^-$. 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. Figure 12 shows a part of the nonclosed subspaces that we infer from the sets of attributes determining A and B, respectively.

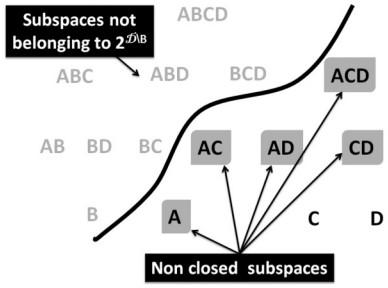
The procedure ClosedSubspaces (see Algorithm 8) takes as input the table T and returns the closed subspaces. As one may see, the most critical part of this algorithm is the statement in line 2, which consists of computing a set of violated FDs.

A.1. Extracting Maximal Violated FDs

As we have seen previously, all subsets of the left-hand side of violated FDs are potentially closed sets of attributes. Thus, given an attribute A_i, the first part of our procedure consists of extracting the maximal X's such that $X \rightarrow A_i$ is not satisfied. There are several algorithms for computing *minimal* FDs in the literature, e.g. Yao



(a) Nonclosed sets with respect to A: B, C, and D are the maximal sets not determining A. Hence, all of their supersets not containing A are not closed.



(b) Nonclosed sets with respect to B: C and D are the maximal sets not determining B. All of their supersets not containing B together with A are not closed.

Fig. 12. Pruned sets with respect to 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 $A \rightarrow B$ is found satisfied, and AC is not part of $2^{D\setminus A}$ but can be tested with D. The same goes for AD, which can be tested with C.

and Hamilton [2008], Lopes et al. [2000], Huhtala et al. [1999], and 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 MaxNFD, (short for maximal left-hand side of nonfunctional 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 $T \not\models X \rightarrow A_i$. First, if 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 the next iteration. The *parent* of X is simply the successor superset of X in the lexicographic order. For example, the parent of BDF is $BDFG$. Second, if X does not pass the test, i.e., $T \models X \rightarrow A_i$, then (i) its *children* are candidates for the next iteration, and (ii) 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 than 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 subspaces belonging 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 if and only if X does not determine any $A_i \in X$. It is not sufficient to make the intersection of these sets, because, e.g., no such set X relative to A_i does contain A_i, still there may exist closed sets containing A_i. ClosedSubspaces exploits the previous results to infer the closed subspaces.

ALGORITHM 7: MaxNFD

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

ALGORITHM 8: ClosedSubspaces

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

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

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

THEOREM B.4. *We have the following:*

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

In other words, the preceding 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. To prove the previous theorem, the following definitions and lemmas are presented.

Definition B.5. Let f be a relation that 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 that 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, $|\mathscr{T}_{2k}| = (2k)^{n \cdot d}$. However, 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 subject to each cell (i,j) of T_{2k} 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 $T \neq T'$, 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 dimensions the skyline size will also increase. Without loss of generality, assume that 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 $\Pi(T)$, and let 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, $t \in Sky(\Pi(T))$, and thus $u \not\prec t$. We have to consider the following three cases:

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

- $u[1] < t[1] \Rightarrow f(u)[1] < f(t)[1]$. However, $f(u)[2 \ldots d] = u[2 \ldots d]$ and $f(t)[2 \ldots d] = t[2 \ldots d]$. 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 ($S^{(0)} = S(\Pi(T))$, $S^{(1)} = S(\Pi(T^{(1)}))$, $\ldots$, $S^{(d)} = S(\Pi(T^{(d)}))$). Consider $T = T_k$ and then we have the following:

$$S(\Pi(T_k)) = S^{(0)} \leq S^{(1)} \leq S^{(2)} \leq \cdots \leq S^{(d)} = S(\Pi(T_{2k})). \quad \square$$

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.

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

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. $\square$

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 article.

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 *Proceedings of the VLDB Conference.*

Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. 2000. Automated selection of materialized views and indexes in SQL databases. In *Proceedings of the VLDB Conference*.

Ilaria Bartolini, Paolo Ciaccia, and Marco Patella. 2008. Efficient sort-based skyline evaluation. *ACM Transactions on Database Systems* 33, 4, Article No. 31.

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 Information and Knowledge Management (CIKM'14)*.

Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. 2001. The skyline operator. In *Proceedings of the ICDE Conference*.

Upen S. Chakravarthy, John Grant, and Jack Minker. 1990. Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems* 15, 2, 162–207.

Surajit Chaudhuri, Nilesh N. Dalvi, and Raghav Kaushik. 2006. Robust cardinality and cost estimation for skyline operator. In *Proceedings of the ICDE Conference*.

Thomas Eiter and Georg Gottlob. 1995. Identifying the minimal transversals of a hypergraph and related problems. *SIAM Journal on Computing* 24, 6, 1278–1304.

Eve Garnaud, Sofian Maabout, and Mohamed Mosbah. 2012. Using functional dependencies for reducing the size of a data cube. In *Proceedings of the FoIKS Conference*. 144–163.

Parke Godfrey. 2004. Skyline cardinality for relational processing. In *Proceedings of the FoIKS Conference*.

Parke Godfrey, Jarek Gryz, and Calisto Zuzarte. 2001. Exploiting constraint-like data characterizations in query optimization. In *Proceedings of the SIGMOD Conference*.

Parke Godfrey, Ryan Shipley, and Jarek Gryz. 2007. Algorithms and analyses for maximal vector computation. *VLDB Journal* 16, 15–28.

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, 29–53.

Nicolas Hanusse and Sofian Maabout. 2011. A parallel algorithm for computing borders. In *Proceedings of the CIKM Conference*.

Nicolas Hanusse, Sofian Maabout, and Radu Tofan. 2009. A view selection algorithm with performance guarantee. In *Proceedings of the EDBT Conference*. ACM, New York, NY, 946–957.

Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. 1996. Implementing data cubes efficiently. In *Proceedings of the SIGMOD Conference*.

Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. 1999. TANE: An efficient algorithm for discovering functional and approximate dependencies. *Computer Journal* 42, 2, 100–111.

Jongwuk Lee and Seung won Hwang. 2010. BSkyTree: Scalable skyline computation using a balanced pivot selection. In *Proceedings of the EDBT Conference*.

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

Daniel Lemire, Owen Kaser, and Eduardo Gutarra. 2012. Reordering rows for better compression: Beyond the lexicographic order. *ACM Transactions on Database Systems* 37, 3, Article No. 20.

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

Stéphane Lopes, Jean-Marc Petit, and Lotfi Lakhal. 2000. Efficient discovery of functional dependencies and Armstrong relations. In *Proceedings of the EDBT Conference*.

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

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, 83–99.

M. D. Morse, J. M. Patel, and H. V. Jagadish. 2007. Efficient skyline computation over low-cardinality domains. In *Proceedings of the VLDB Conference*.

Noël Novelli and Rosine Cicchetti. 2001. FUN: An efficient algorithm for mining functional and embedded dependencies. In *Proceedings of the ICDT Conference*.

Carlos Ordonez. 2010. Statistical model computation with UDFs. *IEEE Transactions on Knowledge and Data Engineering* 22, 12, 1752–1765.

Carlos Ordonez, Wellington Cabrera, and Achyuth Gurram. 2016. Comparing columnar, row and array DBMSs to process recursive queries on graphs. *Information Systems*. Accepted.

Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. 2005. Progressive skyline computation in database systems. *ACM Transactions on Database Systems* 30, 141–82.

Jian Pei, Wen Jin, Martin Ester, and Yufei Tao. 2005. Catching the best views of skyline: A semantic approach based on decisive subspaces. In *Proceedings of the VLDB Conference*.

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 Transactions on Database Systems* 31, 4, 1335–1381.

Chedy Raïssi, Jian Pei, and Thomas Kister. 2010. Computing closed skycubes. In *Proceedings of the VLDB Conference*.

Atish Das Sarma, Ashwin Lall, Danupon Nanongkai, and Jun Xu. 2009. Randomized multi-pass streaming skyline algorithms. In *Proceedings of the VLDB Conference*.

Haichuan Shang and Masaru Kitsuregawa. 2013. Skyline operator on anti-correlated distributions. *Proceedings of the VLDB Endowment* 6, 9, 649–660.

Cheng Sheng and Yufei Tao. 2012. Worst-case I/O-efficient skyline algorithms. *ACM Transactions on Database Systems* 37, 4, Article No. 26.

Tian Xia, Donghui Zhang, Zheng Fang, Cindy X. Chen, and Jie Wang. 2012. Online subspace skyline query processing using the compressed skycube. *ACM Transactions on Database Systems* 37, 2, Article No. 15.

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

Yidong Yuan, Xuemin Lin, Qing Liu, Wei Wang, Jeffrey Xu Yu, and Qing Zhang. 2005. Efficient computation of the skyline cube. In *Proceedings of the VLDB Conference*.

Yihong Zhao, Prasad Deshpande, and Jeffrey F. Naughton. 1997. An array-based algorithm for simultaneous multidimensional aggregates. In *Proceedings of the SIGMOD Conference*.

Received July 2015; revised April 2016; accepted June 2016