

Refreshing the Sky: The Compressed Skycube with Efficient Support for Frequent Updates

Tian Xia
tianxia@ccs.neu.edu

Donghui Zhang^{*}
donghui@ccs.neu.edu

College of Computer and Information Science
Northeastern University
360 Huntington Avenue, Boston, MA 02115

ABSTRACT

The skyline query is important in many applications such as multi-criteria decision making, data mining, and user-preference queries. Given a set of d -dimensional objects, the skyline query finds the objects that are not dominated by others. In practice, different users may be interested in different dimensions of the data, and issue queries on any subset of d dimensions. This paper focuses on supporting concurrent and unpredictable subspace skyline queries in frequent updated databases. Simply to compute and store the skyline objects of every subspace in a skycube will incur expensive update cost. In this paper, we investigate the important issue of updating the skycube in a dynamic environment. To balance the query cost and update cost, we propose a new structure, the *compressed skycube*, which concisely represents the complete skycube. We thoroughly explore the properties of the compressed skycube and provide an efficient object-aware update scheme. Experimental results show that the compressed skycube is both query and update efficient.

1. INTRODUCTION

Given a set of d -dimensional objects, the *skyline query* returns the “best” objects that are not *dominated* by others. An object t_1 *dominates* another object t_2 if t_1 has equal or smaller values than t_2 in all dimensions, and has a smaller value in at least one dimension. In fact, the preference function “*dominate*” can also be defined in other ways as long as it is monotone on all dimensions. Without loss of generality, we use the *MIN* operation in this paper. A classic example of the skyline query is to find hotels in Nassau (a city in Bahamas) that are cheap and close to the beach. In Figure 1, suppose the 2-dimensional objects represent the hotels, and each dimension is an attribute of the data (e.g. the price). The skyline consists of objects t_1 , t_4 , t_6 and t_8 .

^{*}Partially supported by NSF CAREER Award IIS-0347600.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD 2006, June 27–29, 2006, Chicago, Illinois, USA.
Copyright 2006 ACM 1-59593-256-9/06/0006 ...\$5.00.

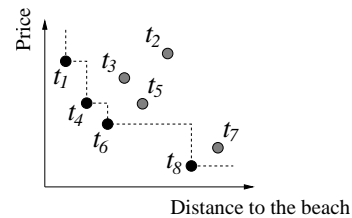


Figure 1: A classic example of the skyline query.

Since it was introduced by Börzsönyi *et al.* [7], the skyline query has attracted considerable attention, largely due to its importance in multi-criteria decision making, data mining [12], and user-preference queries [11]. Traditional skyline computation [2, 3, 8, 9, 10, 13, 15, 16, 19] is restricted to a fixed set of dimensions that are interesting to the skyline queries. Based on the a-priori information, indices can be used or data can be pre-sorted on the query dimensions to expedite the query processing. Recently, research [17, 21, 20] on the skyline query has shifted to the *subspace skyline* computation, which do not assume the knowledge of the query dimensions. Given a set of objects with d dimensions, different users may be interested in different dimensions of the data. Therefore, the skyline queries can be issued on any subset of the d dimensions. In general, a d -dimensional space contains $2^d - 1$ subspaces, and a *subspace skyline query* asks for the skyline in one of the $2^d - 1$ subspaces. In particular, the space of d dimensions is called *full-space*. In the previous example of the hotels in Nassau, suppose each hotel has three attributes, *distance* (to the beach), *price* and *rating*. A user interested in price and rating of the hotels can issue a skyline query on the subspace $\langle price, rating \rangle$.

In an online system that accepts multiple concurrent subspace skyline queries, the query response time is important, and on-the-fly approaches (e.g. [20]) are usually unsatisfactory. Due to the fact that the subspaces of users’ interests are unpredictable, Yuan *et al.* [21] proposed the concept of *Skycube*, which is the complete computation of all possible subspace skylines. Similar to the idea of the data cube [1] in the data warehouse, the skycube consists of the skylines of all possible subspaces. [21] provided bottom-up and top-down methods for the initial computation of skycube.

However, the previous work on the skycube overlooked an important fact that the data are *not* static. In many applications, objects change their attribute values; new objects are

added; and outdated data are deleted. Frequently updated databases are not uncommon in the real world, especially in online query processing systems. For example, in an online hotel information system (e.g. hotels.com), prices of hotels change according to time, availability and the prices of other facilities. Another example is the sensor network, where sensors report various statistics of different locations, such as temperature, humidity, wind speed and etc. Those attributes' values are changing over time. In addition, sensors may lose connections or reconnect to the server. Therefore, in a dynamic environment, the subspace skyline queries issued at different time may get different objects. The pre-computed results (e.g. the skycube) need to be updated to reflect the correct results.

Naively, the skycube is recomputed upon each update. This straightforward method is extremely inefficient and in turn affects the query performance of the online server, since the server needs to spend most of the time recomputing the correct skycube. In the frequently updated databases, the update cost is not negligible. A good update scheme is more important than the initial computation of the results, since the processing time of updates is much less tolerable than that of the initial computation. To the best of our knowledge, this paper is the first work addressing the efficient update support of the skycube in dynamic environments.

Updating the skycube is inherently expensive because it contains a huge number of duplicates and needs to maintain complete subspace skyline in every cuboid. In this paper, we first aim at improving the storage of the skycube to support efficient update. We propose a new structure called the *Compressed Skycube*, based on the concept of the *minimum subspaces*. The compressed skycube concisely represents the complete skycube and preserves the essential information of subspace skylines. Each skyline object is stored only in the cuboids which correspond to its minimum subspaces, and the compressed skycube contains only non-empty cuboids. Compared to the original skycube [21], the compressed skycube has much less duplicates among cuboids, and does not need to contain all cuboids. We thoroughly explore some interesting properties of the compressed skycube and provide an efficient query processing algorithm.

Recomputing everything upon updates is obviously unacceptable because of the expensive costs of cuboid computations and disk accesses in retrieving objects. To minimize such costs during the updates of objects, we propose the *object-aware* update scheme for the compressed skycube. More specifically, we differentiate various cases such as when an update needs to retrieve new objects from the disk, when existing objects in the compressed skycube are affected, and etc. The analyses of these cases are *not* trivial. Furthermore, given a new object t in the full-space skyline, we need to identify the minimum subspaces of t in order to insert t into the compressed skycube. As we will see, identifying the minimum subspaces of a full-space skyline object is challenging. We propose a novel solution to efficiently report the minimum subspaces.

To sum up, our key contributions are:

1. We propose the *compressed skycube* which concisely represents the complete skycube. We thoroughly explore the properties of the compressed skycube and provide an efficient query processing algorithm. Using the compressed skycube, a generic system is proposed to answer online concurrent skyline queries.
2. We propose the object-aware update scheme for the compressed skycube. We carefully design and analyze the scheme to avoid unnecessary disk accesses and cuboid computation, so that the update of the compressed skycube is both incremental and scalable.
3. It is non-trivial to find minimum subspaces for a new full-space skyline object. We propose a novel solution for identifying the *minimum subspaces* of a full-space skyline object, without generating any false positive subspaces.

The rest of this paper is organized as follows. Section 2 reviews the existing work on computing the skyline and subspace skyline queries. Section 3 presents the structure of the compressed skycube and its query processing system. In Section 4, we discuss how to maintain the compressed skycube upon updates of objects. Finally, Section 5 shows our experimental results and Section 6 concludes this paper.

2. RELATED WORK

The skyline query can be traced back from 1960s in the theory field, where the skyline is called the *Pareto set*, and the skyline objects are called *admissible points* [4] or *maximal vectors* [6]. The corresponding problem in the theory field is known as the *maximal vector problem* [14, 18]. Several main-memory algorithms [14, 6, 5] have been proposed to solve the maximal vector problem. However, in the database context, those main-memory algorithms are inefficient for the skyline query, due to the large sizes of data sets. In Section 2.1, we review the external algorithms for the full-space skyline query. In Section 2.2, we review subspace skyline queries and the skycube.

2.1 Full-space Skyline

Börzsönyi *et al.* [7] proposed first two algorithms for the skyline computation. The BNL (block nested loop) algorithm compares every object with others and produce a block of skyline objects in every iteration. The m -way DC (divide and conquer) algorithm recursively divides the objects into m partitions such that each partition fits into main memory. It produces the final skyline objects by merging the local skyline in each partition. Both BNL and m -way DC extend their main-memory counterparts by taking into consideration the memory size.

BNL makes many unnecessary comparisons between objects that are not in the skyline. To eliminate those comparisons, SFS [9] (sort filter skyline) sorts the entire data first, according to some monotonic function of the skyline dimensions. The skyline objects are output to a window. If the window is large enough, each object is compared only with the skyline objects, and objects put into the window are guaranteed to be in the skyline. Otherwise, some objects will be put in a temporary file as BNL does. SFS always takes the minimum number of iterations. Recently, Godfrey *et al.* [10] proposed another generic algorithm LESS (linear elimination sort for skyline), aimed at improving the asymptotic complexity. LESS also requires the data to be pre-sorted, while it eliminates some non-skyline objects in the external sort routine. LESS achieves $O(dn)$ average-case cost, where d is the dimensionality and n is the cardinality. Tan *et al.* [19] proposed an alternate method, *Bitmap*. *Bitmap* maps each object to a bit string, and the skyline is computed using efficient bit operations.

Parallel to the above generic algorithms that do not rely on any index, several index-based algorithms have also been proposed to provide pruning power, so that only a fraction of data need to be visited. The first index-based algorithm, *Index*, was proposed in [19]. There are d lists, and an object appears in i^{th} list if its i^{th} coordinate-value is the minimum among all the dimensions. The i^{th} list is sorted in ascending order of the i^{th} coordinate-value. Then *Index* scans the d lists sequentially and simultaneously from the first entries. If the current unexamined object in a list has the key value larger than the maximum coordinate-value of some object, the remaining of the list can be pruned.

NN [13] (nearest neighbor) and BBS [16] (branch and bound skyline) compute the skyline using nearest neighbor search, and prune the search space using the newly found nearest neighbor object. The difference is that NN issues multiple nearest neighbor queries, while BBS only traverse the index once. It can be proved that BBS achieves I/O optimal.

There have also been a number of other papers concerning the skyline query in some specific settings. Balke *et al.* [3] solved the skyline query in a distributed environment. Lin *et al.* [15] discussed the sliding window skyline queries over data stream. Chan *et al.* [8] studied the computation of skyline queries with partially-ordered attributes.

2.2 Subspace Skyline and the Skycube

Methods in Section 2.1 either explicitly or implicitly rely on the assumption that the query dimensions are fixed. Latest research [21, 17, 20] has shifted to a more general scenario, where the query dimensions can vary. Given a set of objects with d dimensions, a skyline query can be issued on any subset of the d dimensions. The interested subset of d dimensions is called *subspace* and the corresponding skyline query is called *subspace skyline query*. Full-space skyline approaches are optimized for a fixed set of dimensions, thus are not efficient for the general case.

Pei *et al.* [17] discussed the subspace skylines primarily from the view of the query semantics. They solved the skyline membership query, *why and in which subspaces is an object in the skyline*, by using the notion of *skyline group*. A skyline group G in subspace U is the set of objects that share the same values on U and are in the skyline of U . All objects in G do not share any value on any other dimension $u \notin U$, and no other object $o \notin G$ shares the same value with objects in G on U . The skyline membership query can then be answered by using the skyline group lattice, in which each skyline group forms a node. However, there is no guarantee that the number of skyline groups is smaller than the number of subspaces, as one subspace may contain many skyline groups, especially in dense data space.

Independently, Yuan *et al.* [21] proposed *Skycube* which is the complete computation of all possible subspace skylines. Similar to the idea of data cube [1] in the data warehouse, the skycube consists of the skylines of 2^d subsets of d dimensions, as shown in Figure 2(a). With the complete skycube, the subspace skyline query can be answered with little overhead cost. [21] focused on the initial construction of the skycube, where computations can be shared by different subspace skylines. Two approaches were proposed, the BUS (bottom-up skycube) and the TDS (top-down skycube). BUS extends SFS by sharing d sorted lists of objects (d is the dimensionality) during the computation. It com-

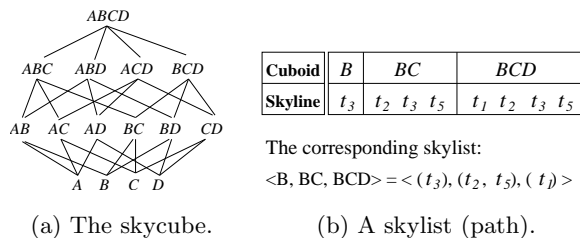


Figure 2: Illustration of the skycube and a skylist.

putes the skycube by levels from bottom to the top. Lower level cuboids are merged to form (part of) the upper level parent cuboids. TDS extends DC by sharing the partitions and merging along a path of cuboids. In particular, a structure called *skylist* is utilized in TDS. A skylist stores the skylines of a path without duplications. For example, the skylist of B, BC, BCD is shown in Figure 2(b). A cuboid u_i (e.g. BC) contains objects in the skyline of u_i but not in the skylines of its descendant subspaces (e.g. B). To compute the skycube, TDS constructs $\binom{d}{\lceil d/2 \rceil}$ skylists/paths, and the each cuboid can be computed along the path. However, [21] did not discuss how to maintain the skycube upon updates and how to balance the query and update costs, which are the focuses of our paper.

Recently, [20] proposed an index-based method, *SUBSKY*, to compute skylines in low-dimensionality subspaces (e.g. 2), while the total dimensionality may be high (e.g. 10). Based on the data distribution, SUBSKY creates an anchor point for each cluster, and builds a B+-tree on the L_∞ distance between each object to its corresponding anchor. There are two problems in SUBSKY, however. First, the anchor points are never modified after the initial computation. Since the pruning power is largely decided by the choice of the anchor points, SUBSKY is not suitable for dynamic data where the data distribution may change over time. Second, SUBSKY's pruning ability deteriorates fast with the increase of query dimensionality, which makes it inappropriate for computing the skylines of all subspaces.

3. THE COMPRESSED SKYCUBE

We assume our problem setting is an online system that accepts concurrent subspace skyline queries. The data objects have d attributes (dimensions), whose values may be frequently updated. We further assume the subspace skyline queries are *unpredictable*, such that a skyline query may be issued on any subset of the d dimensions. Due to the concurrentness and unpredictability of the skyline queries, our solution can efficiently answer any subspace skyline query *without* accessing the disk at the query time.

To balance the query and update costs, we propose a new structure, the *Compressed Skycube* (CSC), which concisely represents all subspace skylines. In this section, we thoroughly explore the properties of the compressed skycube and provide the query processing algorithm, while in Section 4, we design an object-aware update scheme for the efficient maintenance of the compressed skycube. Based on the compressed skycube, a generic framework of an online system for subspace skyline queries is proposed at the end of this section. Table 1 summarizes the notations used in this paper.

Notation	Definition
\mathcal{D}	full-space
U, V	subspace/cuboid
d	dimensionality of the full-space
u_i	one dimension ($1 \leq i \leq d$)
t	a tuple/an object
$sky(U)$	the skyline of subspace U
$mss(t)$	the set of minimum subspaces of t
$t(u_i)$	the value of t on dimension u_i

Table 1: Summary of notations.

3.1 Structure of the Compressed Skycube

A complete computation of all subspace skylines forms the skycube, where each subspace is also called a *cuboid*. Let us see a running example. Given a table of objects with 4 attributes (u_1 through u_4) in Figure 3, the corresponding skycube is shown in Figure 4. For clarity, we divide the table in Figure 3 into three parts. The first four tuples are in the full-space skyline, i.e. in the cuboid $\langle u_1, u_2, u_3, u_4 \rangle$. The next two tuples are not in the full-space skyline but appear in some subspace skylines (e.g. t_4 is in the cuboid $\langle u_4 \rangle$). The last three tuples are not skyline objects in any subspaces, thus do not appear in the skycube. To answer a skyline query of some subspace U , we can go to the cuboid U and return the result tuples immediately.

	u_1	u_2	u_3	u_4
t_1	3	4	2	5
t_5	2	2	3	1
t_6	6	1	1	3
t_7	1	3	4	1
t_4	4	3	6	1
t_9	2	2	3	7
t_2	4	6	7	2
t_3	9	7	5	6
t_8	6	5	3	8

Figure 3: Dataset.

Cuboid	Skyline
u_1	t_7
u_2	t_6
u_3	t_6
u_4	t_5, t_7, t_4
u_1, u_2	t_5, t_6, t_7, t_9
u_1, u_3	t_1, t_5, t_6, t_7, t_9
u_1, u_4	t_7
u_2, u_3	t_6
u_2, u_4	t_5, t_6
u_3, u_4	t_5, t_6
u_1, u_2, u_3	t_1, t_5, t_6, t_7, t_9
u_1, u_2, u_4	t_5, t_6, t_7
u_1, u_3, u_4	t_1, t_5, t_6, t_7
u_2, u_3, u_4	t_5, t_6
u_1, u_2, u_3, u_4	t_1, t_5, t_6, t_7

Figure 4: Cuboids of the skycube.

However, maintaining the skycube upon frequent updates is expensive. There are two main reasons why the skycube is inefficient for updates. First, there are a huge number of duplicates among the cuboids. For example, in Figure 4, t_6 is stored in 12 of the total 15 cuboids, and t_5 is stored in 10 cuboids. Thus, if the tuple t_5 is updated, large part of the skycube needs to be updated. Second, each cuboid needs to maintain complete results. If an object is updated, every *affected* cuboid needs to be recomputed to reflect the correct results, no matter whether the updated object is in the cuboid or not. The re-computation of a cuboid is expensive. Therefore, although there is little cost in retrieving the query results, updating the skycube becomes the bottle-neck

of the query processing in a dynamic environment with frequent updates. Incoming queries will be blocked when the system is updating the whole cube.

In frequently updated databases, balancing the query cost and the update cost is important. The skycube is one extreme case where the query cost is almost zero while the update cost is too much. Another extreme case is that we do not precompute any cuboid. Then the query cost is expensive as the query needs to access the whole dataset, while the update cost is small (assuming no complex index on the data). To combine the benefits of the two extreme cases, we aim at minimizing the storage of the skycube to support more efficient update, without compromising the query efficiency. We first introduce the concept of the *minimum subspace*, which is the basis of our new compressed structure. It makes possible the elimination of many duplicates.

Definition 1. Given an object t , the *minimum subspaces* of t , denoted as $mss(t)$, is a set of all subspaces, such that $\forall U \in mss(t), t \in sky(U)$, and $\forall V \subset U, t \notin sky(V)$.

Examples of the minimum subspaces is shown in Figure 5 (using the dataset in Figure 3). Take t_5 for instance. Tuple t_5 is a skyline object in the following 10 cuboids, $\langle u_4 \rangle, \langle u_1, u_2 \rangle, \langle u_1, u_3 \rangle, \langle u_2, u_4 \rangle, \langle u_3, u_4 \rangle, \langle u_1, u_2, u_3 \rangle, \langle u_1, u_2, u_4 \rangle, \langle u_1, u_3, u_4 \rangle, \langle u_2, u_3, u_4 \rangle$ and $\langle u_1, u_2, u_3, u_4 \rangle$. Among them, only three cuboids, $\langle u_4 \rangle, \langle u_1, u_2 \rangle, \langle u_1, u_3 \rangle$, are $mss(t_5)$. For most objects, especially the full-space skyline objects, the number of minimum subspaces is much smaller than the number of all subspaces where an object is a skyline object.

	Minimum Subspaces
t_1	$\langle u_1, u_3 \rangle$
t_5	$\langle u_4 \rangle, \langle u_1, u_2 \rangle, \langle u_1, u_3 \rangle$
t_6	$\langle u_2 \rangle, \langle u_3 \rangle$
t_7	$\langle u_1 \rangle, \langle u_4 \rangle$
t_4	$\langle u_4 \rangle$
t_9	$\langle u_1, u_2 \rangle, \langle u_1, u_3 \rangle$

Figure 5: Minimum subspaces.

Cuboid	Skyline
u_1	t_7
u_2	t_6
u_3	t_6
u_4	t_5, t_7, t_4
u_1, u_2	t_5, t_9
u_1, u_3	t_1, t_5, t_9

Figure 6: Cuboids of the compressed skycube.

Based on the minimum subspaces, we define the *compressed skycube* as follows.

Definition 2. The *compressed skycube (CSC)* consists of *non-empty* cuboids U , such that an object t is stored in a cuboid U if and only if $U \in mss(t)$.

Figure 6 shows the cuboids of the compressed skycube. Compared to 15 cuboids in the skycube, CSC contains only 6 non-empty cuboids. Each cuboid in CSC contains only those objects, of which the cuboid is one of their minimum subspaces. Thus, the set of objects stored in a cuboid in CSC is a subset of objects in the corresponding cuboid in the skycube. Also, due to the small number of minimum subspaces of most objects, CSC has much less number of duplicates among the cuboids than the skycube. For example, while t_6 appears in 12 cuboids in the skycube, it only appears in two cuboids, $\langle u_2 \rangle$ and $\langle u_3 \rangle$, in CSC. Furthermore, the number of objects in each cuboid of CSC is more balanced. Under the independent distribution assumption, the expected number of skyline objects is $O(\ln^{k-1} n / (k-1)!)$ [6], where

n is the number of total objects and k ($1 \leq k \leq d$) is the query dimensionality. Higher dimensional cuboids usually have much larger number skyline objects (assuming $n \gg d$). In CSC, since objects do not appear in the cuboids which are supersets of the (low dimensional) minimum subspaces, the cuboid loads are more balanced.

The complete CSC structure of the above example is shown in Figure 7. It consists of two parts: the non-empty cuboids corresponding to the minimum subspaces and two object lists. The cuboids are organized in up to d levels. Because the object size is usually large due to high dimensionality, we store the pointers of objects in the cuboid instead of the actual object. For clarity, in Figure 7, we highlight only the pointers between t_5 and its minimum subspaces using solid lines. As we will see, the full-space skyline objects ($sky(\mathcal{D})$) are particularly important in our update scheme. Therefore, we store $sky(\mathcal{D})$ in one list and other skyline objects in another list. To facilitate searching, the two lists can be sorted in ascending order by a monotone function on \mathcal{D} . For ease of discussion, we specify the monotone function to be the entropy scoring function [9] in the rest of this paper.

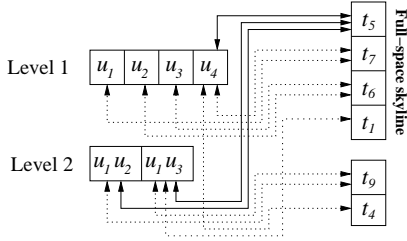


Figure 7: The complete CSC structure.

One property of CSC is that the number of non-empty cuboids is solely decided by $sky(\mathcal{D})$. In other words, there does not exist a cuboid which only contains objects *not* in $sky(\mathcal{D})$. This property is important as it implies that the full-space skyline plays a key role in our update scheme of CSC (to be discussed in Section 4). Intuitively, as long as the full-space skyline is unchanged, no new *cuboid* will be added to CSC. Theorem 1 provides the theoretical basis of this property.

THEOREM 1. *Given an object t and a subspace $U \in mss(t)$, at least one of the following two conditions holds: (1) $t \in sky(\mathcal{D})$; and (2) $\exists t' \in sky(\mathcal{D})$, such that t and t' share the same values on U and $U \in mss(t')$.*

PROOF. Since $U \in mss(t)$, by Definition 1, $t \in sky(U)$. As proved in [21, 17], either $t \in sky(\mathcal{D})$, or $\exists t' \in sky(\mathcal{D})$, t' shares the same values on U with t . If $t \in sky(\mathcal{D})$, we are done. Now we prove the case that if $t \notin sky(\mathcal{D})$, U is a minimum subspace of t' . Suppose $U \notin mss(t')$. Since $t' \in sky(U)$, by Definition 1, $\exists V$, such that $V \in mss(t')$ and $V \subset U$. Since t and t' share the same values on U , they also share the same values on V . Therefore, $t \in sky(V)$, which contradicts with the fact that $U \in mss(t)$. \square

COROLLARY 1.1. *Each non-empty cuboid in CSC contains at least one object in $sky(\mathcal{D})$.*

PROOF. By Definition 2, if an object t is stored in U , $U \in mss(t)$. By Theorem 1, either $t \in sky(\mathcal{D})$, or $\exists t' \in sky(\mathcal{D})$, such that $U \in mss(t')$, i.e. t' is stored in U . \square

Our compressed skycube is also superior than the skylists [21]. Recall that [21] divides the d -dimensional skycube into $\binom{d}{\lceil d/2 \rceil}$ paths (skylists) during the top-down computation of the skycube. The skylists can be used to store the final results, such that there is no duplicate along a path. Besides that the skylists have to keep all cuboids, we also prove in Theorem 2 that the skylists store more objects than CSC. Intuitively, the skylists can *not* avoid duplicates among $\binom{d}{\lceil d/2 \rceil}$ paths, while CSC can avoid duplicates in all cuboids which are supersets of the minimum subspaces.

THEOREM 2. *The compressed skycube has fewer duplicates than the skylists.*

PROOF. We prove the theorem in two steps. First, we show that any object in a cuboid of CSC must also appear in the same cuboid of a skylist. Given an object t and a subspace $U \in mss(t)$, by Definition 1, $\forall V \subset U$, $t \notin sky(V)$. Let P be the skylist in the skylists that contains U . By the definition of the skylist, U is the first cuboid in P such that $t \in sky(U)$. Therefore, t is stored in U in the skylist P .

Second, we show there exist objects that appears in cuboids of the skylists but not in these of CSC. Let P be the path in the skylists that contains only one cuboid U . Suppose an object t is in cuboid $V \subset U$, such that t is not dominated on subspace $U - V$ by any other object $\in V$. Since t is also not dominated on subspace V by any object not in cuboid V , $t \in sky(U)$. Since P contains only U , t is in cuboid U . However, in CSC, since $V \subset U$, t is not in cuboid U . \square

3.2 Querying the Compressed Skycube

In this section, we present the query processing algorithm on CSC. Our algorithm only visits CSC and does not access the disk or the whole dataset at the query time. Since CSC stores partial information (the minimum subspaces) in each non-empty cuboid, a query may need to visit multiple cuboids. Comparing to the skycube, we trade some of the query efficiency for better storage and more efficient maintenance of CSC upon updates. In practice, due to the small size of each cuboid and the *local-comparison* property (to be explained shortly), the query performance of CSC is only slightly affected. Figure 8 shows the algorithm *QueryCSC*.

Algorithm *QueryCSC* searches CSC by levels (step 4). By Lemma 1, we only check the cuboids which is a subset of

Algorithm *QueryCSC* (U_Q, l)

Input: Query subspace U_Q in level l .

Output: The skyline $sky(U_Q)$.

1. If U_Q is full-space, **return** the full-space skyline.
 2. $SK = \emptyset$. /* $sky(U_Q)$. */
 3. $FP = \emptyset$. /* false positives. */
 4. For each non-empty level i that $i \leq l$
 - 4.1 For each non-empty cuboid V that $V \subseteq U_Q$,
 - (1) if an object in V is in FP , continue.
 - (2) if an object in V is in SK , push objects dominated by it on U_Q into FP .
 - (3) compare the rest of the objects in V on dimensions U_Q , push skyline objects into SK and false positives into FP .
 5. **return** SK .
-

Figure 8: Algorithm *QueryCSC*.

the query subspace U_Q (step 4.1). Intuitively, if an object $t \in \text{sky}(U_Q)$, $\exists V \in \text{mss}(t)$ such that $V \subseteq U_Q$. Also, the visited cuboids may contain false positives. To filter the false positives, Lemma 2 guarantees that we only compare objects *locally* in the same cuboid (step 4.1(3)). With Lemma 2, the query performance on CSC can be efficient. Since the size of each cuboid in CSC is usually small, the *local-comparison* property ensures that the number of comparisons among objects is small. Furthermore, to avoid repetitive comparisons, we maintain the set of all false positives found so far, so that the false positives are compared only once (step 4.1(1)).

The correctness of our query algorithm *QueryCSC* can be easily proved by combining Lemma 1 and Lemma 2.

LEMMA 1. *Given a query subspace U_Q and an object t , if $\forall U_i \in \text{mss}(t)$, $U_i \not\subseteq U_Q$, then $t \notin \text{sky}(U_Q)$.*

PROOF. Suppose $t \in \text{sky}(U_Q)$. There exists a (smallest) subspace $V \subseteq U_Q$ such that $t \in \text{sky}(V)$ and $\forall V' \subset V$, $t \notin \text{sky}(V')$. By Definition 1, $V \in \text{mss}(t)$. It contradicts with the assumption that $\forall U_i \in \text{mss}(t)$, $U_i \not\subseteq U_Q$. \square

LEMMA 2. *Given a query subspace U_Q , an object t , and a cuboid V such that $V \in \text{mss}(t)$ and $V \subseteq U_Q$, if t is not dominated on U_Q by any other object $u \in V$, then $t \in \text{sky}(U_Q)$.*

PROOF. Since $V \in \text{mss}(t)$, by Definition 1, $t \in \text{sky}(V)$. Suppose $t \notin \text{sky}(U_Q)$ and is dominated by an object t' not in cuboid V . Since $V \subseteq U_Q$, t' also dominates t on V , which contradicts with the fact that $t \in \text{sky}(V)$. \square

Algorithm *QueryCSC* can be extended to handle multiple queries simultaneously, if the query subspaces have containment relation. A query of subspace U_Q will visit all the cuboids needed for the queries of U_Q 's subsets. They can be answered in the middle of computing the query of U_Q . Therefore, concurrent queries with containment relations can be grouped and answered together with very little additional cost.

Based on the compressed skycube, we further propose a generic system for online concurrent skyline queries. The illustration of the system is shown in Figure 9.

The CSC-based system has two major parts: the query buffer and the CSC structure. The query buffer stores the most frequently requested query results. If the requested query results are not in the buffer, the query buffer will issue a query-miss request to CSC, and new results are computed in CSC. As we mentioned before, the query buffer can group

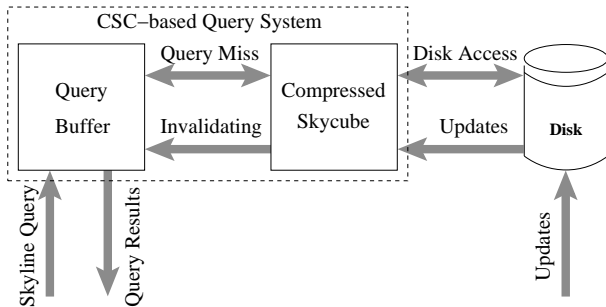


Figure 9: Illustration of the generic query system based on CSC.

a set of queries with containment relation in one query-miss request. The CSC part monitors the updates of objects. According to different objects, CSC decides whether it needs to access the disk to retrieve new objects, or just to update its cuboids. If some cuboids are updated, results in the query buffer may not be accurate any more. CSC then invalidates affected query results in the buffer.

The proposed system is both query-load aware and update efficient. The storage of the system is optimized for balancing the query cost and the update cost. In the next section, we discuss how to dynamically maintain CSC in frequently updated databases.

4. UPDATING COMPRESSED SKYCUBE

Naively, the complete skycube is recomputed upon each update. Such “blind” updating method is extremely inefficient. First, it needs to access disk pages to compute the cuboids on every update. Second, computation may be wasted as the skylines of many cuboids are not changed.

In this section, we propose our *object-aware* update scheme for CSC. Intuitively, accessing disk pages is needed only if the update of an object may introduce new objects to CSC. That is, previous non-skyline objects become skyline objects after the update. For those updates that can be proved not to introduce new objects, disk accesses should be completely avoided. Therefore, in order to minimize the number of data and disk accesses, it is crucial to decide of which object the update may need the retrieval of new objects for CSC. Theorem 3 guarantees that, for any update of the object which does not belong to the full-space skyline, disk accesses are not necessary.

THEOREM 3. *Given an object $t \notin \text{sky}(\mathcal{D})$, any update of t does not introduce new object ($\neq t$) to CSC.*

PROOF. Let the updated value of t be t_{new} . Suppose there is no update on other objects. Given that $t \notin \text{sky}(\mathcal{D})$, we prove the theorem in two cases.

If $t_{new} \notin \text{sky}(\mathcal{D})$, $\text{sky}(\mathcal{D})$ is not changed after the update of t . According to Corollary 1.1, no new cuboid is added to CSC. Since $\text{sky}(\mathcal{D})$ remains the same in the existing cuboids, it follows that any object ($\neq t$) not in CSC can not appear in any cuboid after the update of t .

If $t_{new} \in \text{sky}(\mathcal{D})$, we prove the theorem by contradiction. Let $U \in \text{mss}(t_{new})$, i.e. $t_{new} \in \text{sky}(U)$. Suppose there exists a new object t' not in CSC sharing the same values on U with t_{new} . Thus $t' \in \text{sky}(U)$, which is independent of t or t_{new} . This contradicts with the assumption that t' is not in CSC before the update of t . \square

Based on Theorem 3, we differentiate $\text{sky}(\mathcal{D})$ from other objects, since only the updates of objects in $\text{sky}(\mathcal{D})$ may require accessing the disk. Recall that in CSC, we store $\text{sky}(\mathcal{D})$ in a sorted array based on the entropy function on \mathcal{D} . Therefore, to identify whether an object is an existing full-space skyline object, we simply binary-search the array using the entropy value of the object.

In Section 4.1, we present the details of handling updates of objects not in the full-space skyline. Then in Section 4.2, we discuss how to handle updates of full-space skyline objects. An important operation during updating CSC is to identify the minimum subspaces of a new *full-space* skyline object. As we will see shortly, the minimum subspaces of a

new full-space skyline object is not restricted to the existing cuboids in CSC. In Section 4.3, we propose a novel algorithm which reports the minimum subspaces of a full-space skyline object without examining any other subspaces.

4.1 Handling Updates of Objects Not in $sky(\mathcal{D})$

Given an object t , let t_{new} be the updated value of t . Without loss of generality, we focus on value-change updates. Insertion of a new object and deletion of an old object are special cases of the value-change update. In this section, we assume t is not a full-space skyline object. By Theorem 3, we can safely remove t from every cuboid $U \in mss(t)$ if t is in CSC.¹ To deal with t_{new} , we further differentiate two cases based on whether t_{new} is in the $sky(\mathcal{D})$ or not. In both cases, no disk access is performed.

- **Case 1:** $t_{new} \notin sky(\mathcal{D})$.

An important observation in this case is that, if neither t nor t_{new} is in the full-space skyline, existing objects ($\neq t$) in CSC are not affected. According to Theorem 1, any object in a cuboid U is either a full-space skyline object, or its values on U overlap with some full-space skyline object. Therefore, as long as the set of full-space skyline objects is not changed, all other objects ($\neq t$) remain in the same cuboids.

Based on the above observation, we handle *Case 1* in two steps. First, to decide whether t_{new} is in $sky(\mathcal{D})$, we compare t_{new} with the existing objects in $sky(\mathcal{D})$ in ascending order of their entropy values. Since $t_{new} \notin sky(\mathcal{D})$, let t_f be the first full-space skyline object encountered to dominate t_{new} . This step finishes immediately after t_f is found.

The second step is to determine the minimum subspaces of t_{new} . It is proved in Theorem 4 that $mss(t_{new})$ is determined by the minimum subspaces of *any* full-space skyline object that dominates t_{new} . Theorem 4 also justifies why we do not need to search further than t_f in the first step. After checking $mss(t_f)$, if $mss(t_{new}) \neq \emptyset$, we add t_{new} to the cuboids in $mss(t_{new})$.

LEMMA 3. *Given two objects t and t' such that t and t' share the same values on a subspace U , $U \in mss(t)$ if and only if $U \in mss(t')$.*

PROOF. Two directions are symmetric. We prove only one direction, and the other one can be proved exactly in the same way. If $U \in mss(t)$, since t and t' share the same values on U , $t' \in sky(U)$. By Definition 1, $\exists V \subseteq U$, $V \in mss(t')$. Since t and t' also share the same values on V , $t \in sky(V)$. Thus $U \subseteq V$. Therefore, $U = V$ and $U \in mss(t')$. \square

THEOREM 4. *Given an object t and any full-space skyline t_f which dominates t , $mss(t) \subseteq mss(t_f)$.*

PROOF. Let U be the subspace on which t and t_f share the same values. By Lemma 3, $\forall V \subseteq U$, if $V \in mss(t_f)$, $V \in mss(t)$. Since t_f strictly dominates t on $\mathcal{D} - U$, t can not be in the skyline of any subspace V' , $V' \cap (\mathcal{D} - U) \neq \emptyset$. \square

Let us see an example using the dataset in Figure 3. Assume t_9 is not in the table and we are about to insert t_9 into the table. We compare t_9 with the full-space skyline objects in the array shown in Figure 7. Since the first object t_5

¹This can be decided by searching the array which contains the objects not in $sky(\mathcal{D})$.

dominates t_9 , we stop searching and check $mss(t_5)$ to determine $mss(t_9)$. The minimum subspaces of t_9 are $\langle u_1, u_2 \rangle$ and $\langle u_1, u_3 \rangle$, as t_9 overlaps with t_5 on dimensions u_1, u_2 and u_3 . Therefore, we link t_9 with those two cuboids.

- **Case 2:** $t_{new} \in sky(\mathcal{D})$.

In this case, the introduction of the new full-space skyline t_{new} may affect the existing objects in CSC. For example, if we insert a new object $t_{10} = \langle 1, 3, 1, 3 \rangle$ into the table in Figure 3, the previous full-space skyline object t_1 will be dominated by the new object, and t_1 becomes a false positive in the cuboid $\langle u_1, u_3 \rangle$. Thus t_1 should be removed from the cuboid $\langle u_1, u_3 \rangle$ in CSC.

In general, *Case 2* is also handled via two steps. First, object t_{new} is compared to existing skyline objects t_e to determine the status of t_{new} . Since $t_{new} \in sky(\mathcal{D})$, none of the existing skyline objects will dominate t_{new} , otherwise we go to *Case 1*. During the comparison routine, we eliminate false positives using t_{new} . There are two sub-cases in eliminating the false positives in CSC.

1. *Object t_e is dominated by t_{new} on \mathcal{D} .* We remove any subspace $U \in mss(t_e)$, on which t_e is dominated by t_{new} . If $mss(t_e) = \emptyset$, t_e is removed from CSC. Otherwise, if t_e is previously in $sky(\mathcal{D})$, t_e is moved from the full-space skyline array to the other array.
2. *Object t_e is dominated by t_{new} not on \mathcal{D} , but on some subspace $U \in mss(t_e)$.* We need to compute new minimum subspaces V of t_e such that $V \supset U$. Let T be the set of objects in cuboid U , sharing the same values on U with t_e . Theorem 5 guarantees that we compare t_e *only* with the objects in $T \cup \{t_{new}\}$ to search for the new minimum subspace $V \supset U$.²

THEOREM 5. *Given a subspace U and a set of objects T that share the same values on U , assume $\forall t \in T$, $U \in mss(t)$. For all $V \supset U$ and $t \in T$, either $t \in sky(V)$, or t is dominated by some object $\in T$ but not by any object $\notin T$.*

PROOF. Since $U \in mss(t)$, by Definition 1, $t \in sky(U)$. Suppose t is dominated by an object $t' \notin T$ on V . Since t' does not share the same values with t on U and $V \supset U$, t' also dominates t on U . This contradicts with the fact that $t \in sky(U)$. \square

In the second step, we need to compute $mss(t_{new})$, so that t_{new} is inserted into the CSC properly. In contrast to finding the minimum subspaces of an object not in $sky(\mathcal{D})$, which is discussed in *Case 1*, computing the minimum subspaces of a full-space skyline object is *not* trivial for the following reasons. First, the $mss(t_{new})$ are determined by all existing full-space skyline objects, not just one object. Second, subspaces in $mss(t_{new})$ are not restricted to the existing cuboids. A straightforward way is to compare t_{new} with all existing full-space skyline objects on every subspaces. In Section 4.3, we provide an efficient algorithm, which utilizes the comparison information obtained in the first step and reports the minimum subspaces without examining any false positive subspaces.

²In fact, we do this once for all objects in T and only the full-space skyline objects in T are compared, as the non-full-space skyline objects in T can be matched with the full-space skyline object afterward.

We use a running example to show how *Case 2* works. Suppose a new object $t_{10} = \langle 1, 3, 1, 3 \rangle$ is inserted into the table in Figure 3. Since object t_5 is dominated by t_{10} on $\langle u_1, u_3 \rangle$, we retrieve t_5 and t_9 which share the same values on u_1 and u_3 . Since t_5 already has minimum subspaces of $\langle u_4 \rangle$ and $\langle u_1, u_2 \rangle$, there is no new minimum subspace containing $\langle u_1, u_3 \rangle$ but not $\langle u_4 \rangle$ or $\langle u_1, u_2 \rangle$. Object t_1 is strictly dominated by t_{10} , and is removed from CSC. The updated CSC after the insertion of t_{10} is shown in Figure 10.

Minimum Subspaces	
t_1	$\langle u_1, u_3 \rangle$
t_5	$\langle u_4 \rangle, \langle u_1, u_2 \rangle, \langle u_1, u_3 \rangle$
t_6	$\langle u_2 \rangle, \langle u_3 \rangle$
t_7	$\langle u_1 \rangle, \langle u_4 \rangle$
t_{10}	$\langle u_1 \rangle, \langle u_3 \rangle$
t_4	$\langle u_4 \rangle$
t_9	$\langle u_1, u_2 \rangle, \langle u_1, u_3 \rangle$

(a) Minimum subspaces.

Cuboid	Skyline
u_1	t_7, t_{10}
u_2	t_6
u_3	t_6, t_{10}
u_4	t_5, t_7, t_4
u_1, u_2	t_5, t_9
u_1, u_3	t_1, t_5, t_9

(b) Cuboids of the compressed skycube.

Figure 10: The updated CSC after insertion of t_{10} .

4.2 Handling Updates of Objects in $sky(\mathcal{D})$

In this section, we consider the updates of full-space skyline objects. Assume $t \in sky(\mathcal{D})$ is updated to t_{new} . Because the full-space skyline is changed, new skyline objects may be retrieved from the disk and inserted into CSC.

If t_{new} dominates t , it is easy to see that disk accesses are not necessary, since every object dominated by t is still dominated by t_{new} . We handle such update by deleting t and inserting t_{new} into CSC as in *Case 2* in Section 4.1.

If t_{new} does not dominate t , it is possible that some objects that were dominated by t may become skyline objects. Disk accesses are needed to retrieve those objects. First, we handle t_{new} as in Section 4.1. Then to retrieve new objects after deleting t from CSC, straightforwardly, we can utilize SFS [9] to compute the new full skyline objects appearing in CSC. A modification is that we also keep objects that overlap with the new full-space skyline objects on some subspace, since they may appear in subspace skylines. However, this method is inefficient, because *all* objects are retrieved and compared with existing full-space skyline objects.

We observe that objects which may become a skyline object must be *exclusively* dominated by t and not by any other full-space skyline object. We define the region that contains such objects as *Exclusive Region (ER)* of t . An example of the two-dimensional ER is shown in Figure 11. The shadowed region is the ER of a full-space skyline object t . The ER can be computed on the fly by comparing with other objects in $sky(\mathcal{D})$ once. The ER is used to immediately filter the objects that do not qualify as candidates.

In practice, objects are usually stored in B+-tree(s) on the attribute(s). We retrieve leaf pages of the B+-tree that intersect with the range of the ER on the indexed attribute. Any object *not* in the ER of t is immediately discarded. Any object in the ER is only compared with other objects also in ER. Compared to the straightforward method, the number of comparisons is greatly reduced. After the retrieval of objects from the disk, we insert them into CSC in the same way described in Section 4.1.

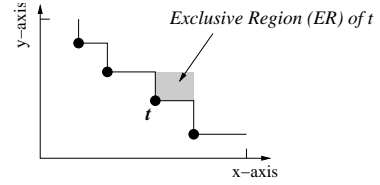


Figure 11: Two-dimensional *Exclusive Region* of t .

4.3 Identifying the Minimum Subspaces of a Full-space Skyline Object

Given a new object t in full-space skyline, we need to identify $mss(t)$ in order to insert t into CSC. As we discussed in Section 4.1, subspaces in $mss(t)$ are not restricted to the existing cuboids in CSC. Straightforwardly, the minimum subspaces of a full-space skyline object t can be computed by visiting all possible cuboids and comparing t with their skylines. This method is obviously inefficient as it examines too many cuboids that are not in $mss(t)$. In this section, we propose a novel solution which efficiently reports the minimum subspaces of a full-space skyline object without examine any cuboids. We first define the concept of the *non-dominant dimensions*, then propose Theorem 6 which is the basis of our solution.

Definition 3. Given an object t and a full-space skyline object t_i , the *non-dominant dimensions* of t with regard to t_i is a dimension set $U_i(t)$, such that:

1. For any dimension u_i , if $t(u_i) < t_i(u_i)$, $u_i \in U_i(t)$.
2. For any subset of dimensions $V = \{u_j \mid t(u_j) = t_i(u_j)\}$, if $V \in mss(t_i)$, $V \subseteq U_i(t)$.

Intuitively, $U_i(t)$ contains those dimensions on which t is *not* dominated by (i.e. has smaller values than) t_i . However, when t and t_i share the same values on some dimensions, the meaning of “dominate” is ambiguous. To solve this, the second condition in Definition 3 defines that only when t overlaps with t_i on some minimum subspace of t_i , the overlapping dimensions are the non-dominant dimensions. For example, let $mss(t_i)$ be $\langle u_1, u_2 \rangle$ and $\langle u_1, u_3 \rangle$. If t shares the same values with t_i on u_1, u_2 and u_3 , all of them are in $U_i(t)$. However, if t shares the same values with t_i only on u_2 and u_3 , none of the u_2 and u_3 will be in $U_i(t)$.

Given a new full-space skyline object t , for each existing object t_i in $sky(\mathcal{D})$, we can compute $U_i(t)$. The minimum subspaces of t are closely related to the set of non-dominant dimensions $\{U_i(t) \mid t_i \in sky(\mathcal{D})\}$, as described by Lemma 4 and Theorem 6.

LEMMA 4. Given an object t , a subspace V , and a set of non-dominant dimensions $\{U_i(t) \mid t_i \in sky(\mathcal{D})\}$, if $t \in sky(V)$, then $\forall U_i(t), V \cap U_i(t) \neq \emptyset$.

PROOF. We divide the objects in $sky(\mathcal{D})$ into two parts. For any object $t_j \in sky(\mathcal{D})$ that shares the same values on V with t , since $t \in sky(V)$, t_j is also in $sky(V)$. By Definition 1, $\exists V' \subseteq V, V' \in mss(t_j)$. By Definition 3, $V' \subseteq U_j(t)$. Thus, $V \cap U_j(t) \neq \emptyset$. For all other objects t_k , since t is not dominated on V by t_k , $\exists u \in V$ such that $t(u) < t_k(u)$. By Definition 3, $u \in U_k(t)$. Thus, $V \cap U_k(t) \neq \emptyset$. In conclusion, $\forall U_i(t), V \cap U_i(t) \neq \emptyset$. \square

THEOREM 6. (SHARING AND MINIMAL CONDITIONS)

Given an object t , a subspace V and a set of non-dominant dimensions $\{U_i(t) \mid t_i \in \text{sky}(\mathcal{D})\}$, if both the conditions hold: (1) $\forall U_i(t), V \cap U_i(t) \neq \emptyset$, and (2) $\nexists V' \subset V$ such that $\forall U_i(t), V' \cap U_i(t) \neq \emptyset$, then $V \in \text{mss}(t)$.

PROOF. We first prove $t \in \text{sky}(V)$. Suppose $t \notin \text{sky}(V)$. Let $t_j \in \text{sky}(D)$ be an object dominating t on V and let V' be $V \cap U_j(t)$. Since $V' \neq \emptyset$, by Definition 3, t share the same values with t_j on V' and $V' \in \text{mss}(t_j)$, which means t is in $\text{sky}(V')$ (Lemma 3). Since $t_j \in \text{sky}(V')$, by Lemma 4, $\forall U_i(t), V' \cap U_i(t) \neq \emptyset$. It contradicts with the assumption that there is no such V' .

Now we prove $\nexists V' \subset U, t \in \text{sky}(V')$. Suppose $\exists V' \subset U, t \in \text{sky}(V')$. Similarly, by Lemma 4, $\forall i, V' \cap U_i \neq \emptyset$. It contradicts with the assumption that there is no such V' .

In conclusion, by Definition 1, $V \in \text{mss}(t)$. \square

According to Theorem 6, the minimum subspaces of $t \in \text{sky}(\mathcal{D})$ can be computed as follows. We obtain the set $\{U_i(t) \mid t_i \in \text{sky}(\mathcal{D})\}$ in the previous steps of our update scheme. Then the minimum subspaces are the minimal subsets satisfying the sharing condition in Theorem 6. Formally, our problem is reduced to the following problem:

Given a list \mathcal{U} of itemsets, enumerate all minimal itemsets, each of which shares at least one item with every itemset $U_i \in \mathcal{U}$.

For example, let \mathcal{U} be $\{u_1u_2, u_1u_3, u_2u_3, u_2u_4\}$. The minimal itemsets that share at least one item with every itemset are $u_1u_2, u_1u_3u_4$ and u_2u_3 .

This problem is challenging because a naive solution, which enumerates all possibilities by choosing one item from every itemset in \mathcal{U} , will examine an exponential number of combinations. In particular, assuming each itemset U_i in \mathcal{U} contains m_i items, there are $\prod_{1 \leq i \leq |\mathcal{U}|} m_i$ number of combinations. However, in practice the result size is much smaller because of item overlap. As in the previous example, there are in total $2^4 = 16$ combinations, while the size of the results (minimal itemsets) is only 3.

Our solution is a recursive algorithm enumerating *only* the minimal itemsets. The idea is that we dynamically choose an object u and recursively enumerate the minimal itemsets in two cases based on whether they contain u or not. In a special case, if an itemset only contains u , all minimal itemsets must contain u . If we choose u to be contained in the minimal itemsets, all given itemsets that contain u can be omitted, since the sharing condition is satisfied for these itemsets. However, the straightforward implementation of this idea may still enumerate false positives. We introduce a *filter* to further prevent the false positives. We will discuss the idea of the filter in our example. Figure 12 shows the complete algorithm *FindMinimum*.

We use an example to illustrate our algorithm. Suppose the list of itemsets \mathcal{U} is $\{u_1u_2, u_1u_3, u_2u_3, u_2u_4\}$, we invoke the function with $\mathcal{U}, M = \emptyset$ and a filter $F = \text{false}$. The minimal itemsets are divided into two cases: those not containing u_1 (Case 1) and those containing u_1 (Case 2).

To enumerate the minimal itemsets not containing u_1 , we just remove u_1 from all the itemsets in \mathcal{U} (Step 4). The algorithm is recursively invoked on the transformed list.

To enumerate the minimal itemsets containing u_1 , itemsets u_1u_2 and u_1u_3 are removed (Step 8.1), since they con-

Algorithm *FindMinimum* (\mathcal{U}, M, F)

Input: A list \mathcal{U} of itemsets, a set M of chosen items, and a logic formula F as a filter.

Action: report M as a minimal itemset.

1. If $\mathcal{U} = \emptyset$, report non-empty M and **return**.
2. If $\exists U \in \mathcal{U}, |U| = 1$, // *Special case*.
 - 2.1 For every such itemset $U, |U| = 1$
 - (1) Add the only item $u \in U$ to M and remove U .
 - (2) Set $F = F(u = \text{true})$.
 - 2.2 Remove all itemsets U' ($U' \cap M \neq \emptyset$) from \mathcal{U} .
 - 2.3 If $F \neq \text{true}$, invoke **FindMinimum** (\mathcal{U}, M, F).
 - 2.4 **return**.
3. Let U be an itemset with minimum number of items. Choose an arbitrary item $u_i \in U$. There are two cases:
 - /* Case 1: find the itemsets not containing u_i . */
 - 4. Let \mathcal{V} be a list of non-empty itemsets derived from \mathcal{U} by removing all occurrences of u_i .
 - 5. Let $F' = F(u_i = \text{false})$ and $M' = M$.
 - 6. If $F' \neq \text{true}$, invoke **FindMinimum** (\mathcal{V}, M', F').
 - /* Case 2: find the itemsets containing u_i . */
 - 7. Let $F_{\text{new}} = \text{true}$.
 - 8. For every itemset $U \in \mathcal{U}$ such that $u_i \in U$,
 - 8.1 $\mathcal{U} = \mathcal{U} - \{U\}$. // Remove U from \mathcal{U} .
 - 8.2 Let C be a clause by connecting all other items ($\neq u_i$) in U with \vee . // Connect items using OR.
 - 8.3 $F_{\text{new}} = F_{\text{new}} \wedge C$. // Connect clauses using AND.
 - 9. Let $F' = F(u_i = \text{true}) \vee F_{\text{new}}$ and $M' = M \cup \{u_i\}$.
 - 10. If $F' \neq \text{true}$, invoke **FindMinimum** (\mathcal{U}, M', F').

Figure 12: Algorithm *FindMinimum*.

tain item u_1 . Furthermore, items ($\neq u_1$) in those pruned itemsets can be used as a *filter*. For example, if u_1 appears in the minimal itemset M , u_2 (in u_1u_2) and u_3 (in u_1u_3) can not all appear in M . Otherwise M is not minimal, since by removing u_1 , M still satisfies the sharing condition. In general, the filter is represented as a logic formula F in the following way. For all U_i which contains u_1 , we connect other items in the same U_i with operator \vee . Then we connect those clauses with \wedge (Step 8.2 and 8.3). Therefore, the filter F of u_1 is $u_2 \wedge u_3$. Notice that we never invoke any recursive call with F being true. After choosing u_1 , the algorithm is recursively invoked on the rest of the itemsets. In each call, a new filter in the recursive call is connected with the old filter using \vee .

Figure 13 illustrates the complete process of running our algorithm on the example. For clarity, the function name is omitted in every recursive call except the initial invocation.

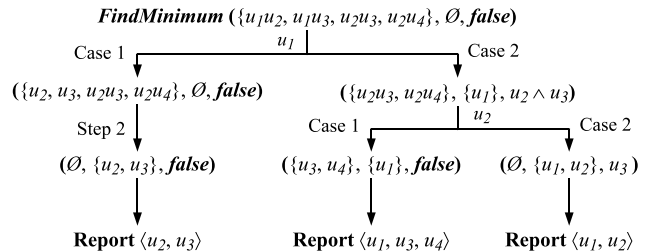


Figure 13: An example of Algorithm *FindMinimum*.

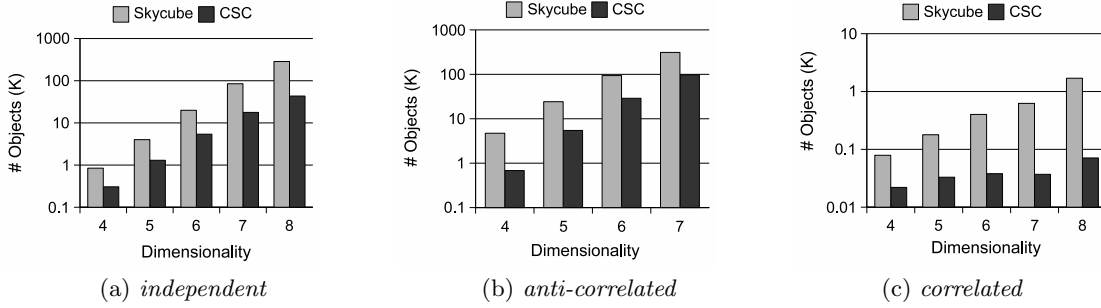


Figure 14: Comparing storage by varying dimensionality.

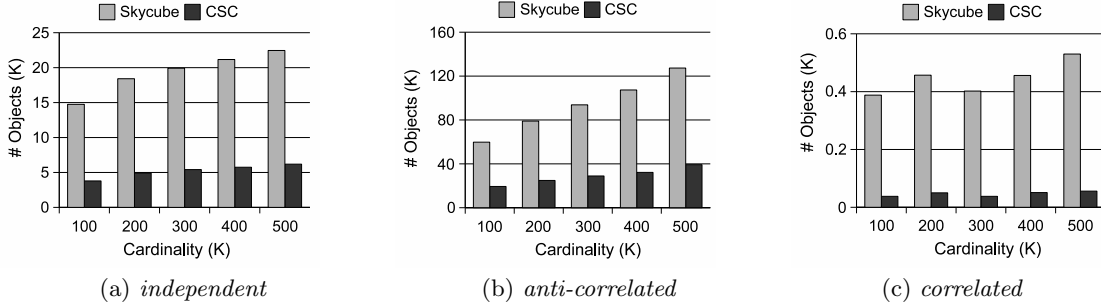


Figure 15: Comparing storage by varying cardinality.

Finally, we point out that in the Case 1 of u_2 , F is set to *false* because of $u_2 = \text{false}$. And in Case 2 of u_2 , the filter F is computed as follows: $F = (u_2 \wedge u_3) \vee (u_3 \wedge u_4) = (\text{true} \wedge u_3) \vee (u_3 \wedge u_4) = u_3 \vee (u_3 \wedge u_4) = u_3$.

5. EXPERIMENTAL EVALUATION

In this section, we present the extensive experimental results on the performance of CSC. We compare CSC with the original skycube in [21]. All our experiments were implemented in Java, running on a PC with P4 2.66-GHz processor and 1GB main memory. Our datasets are three most popular synthetic benchmark datasets, *independent*, *anti-correlated* and *correlated*, used for evaluating the skyline queries. Details of the data generator can be found in [7]. In particular, each dataset has 20% of the objects each of which overlaps with another object on a random subspace. The dimensionality d of the datasets varies in the range [4, 8], and the cardinality varies in the range [100K, 500K]. The default dimensionality and cardinality are 6 and 300K, respectively.

The skycube is constructed using the top-down skyline algorithm (TDS). As shown in [21], TDS outperforms the bottom-up skyline algorithm (BUS) in most cases, and is the current best algorithm to compute the skycube. Since TDS is an in-memory algorithm, for a fair comparison, all our experiments were performed in memory, and the CPU time is reported. Our previous analyses in the disk-based scenario are also applicable in the in-memory scenario, since accessing the whole dataset is much more costly than accessing only the skyline objects. In the rest of this section, we first show the storage advantage of CSC over the skycube, followed by the query performance of CSC. Then we compare the update processing on CSC and on the skycube.

5.1 Storage Comparison

We first compare the storage of our CSC and the Skycube, showing how much of the storage CSC can save. We construct CSC from the skycube, by removing redundant objects based on the concept of the minimum subspace. The storage size is computed by summing up the number of all objects stored in each cuboid.

Figure 14 shows the storage comparison by varying dimensionality. Notice that we use the *logarithmic* scale to reflect the exponential effect of the dimensionality. For *independent* and *anti-correlated*, the number of the skyline objects in both CSC and Skycube increases with the increase of the dimensionality. CSC is better than Skycube in up to an order of magnitude. Since the skyline objects in *anti-correlated* increase dramatically with the increase of the dimensionality, we do not show the evaluation of *anti-correlated* on dimensionality $d = 8$ throughout all experiments. For *correlated* in Figure 14(c), CSC shows stability with the increase of the dimensionality. This is because the number of distinct skyline objects in correlated data does not increase exponentially in dimensionality. The effect of duplicates elimination is most obvious for correlated data, as CSC is smaller than Skycube in more than one order of magnitude.

Figure 15 shows the storage comparison by fixing the dimension to 6 and varying the cardinality. Again, due to less number of duplicates, CSC is less affected by cardinality than Skycube, and is smaller than Skycube for at least 70% in size for all data distribution. The advantage in the storage size of CSC is the basis of efficient and scalable processing of updates. We will show the experimental evidence in Section 5.3.

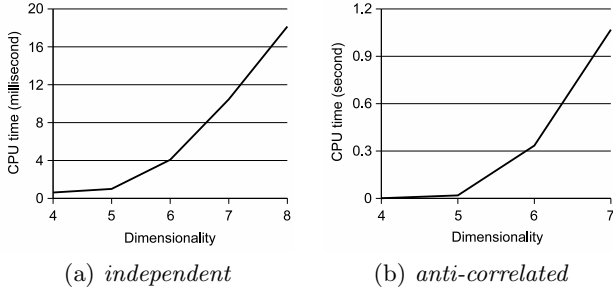


Figure 16: Effect of dimensionality on query.

5.2 Query Performance

Both the Skycube and the CSC can very efficiently support skyline queries for arbitrary combinations of dimensions, by pre-computing some information. Therefore our major interest is to compare the update performance of the CSC against the Skycube (in Section 5.3). Nevertheless, in this section we experimentally verify that the query time for the CSC is indeed small, ranging from a fraction of a millisecond to a fraction of a second.

We compute the average query processing time of 100 randomly selected subspace skyline queries. The query is performed without any query buffer. That is, for each query, we compute the results from CSC. Since *correlated* contains very few skyline objects (less than 100 objects in all cases), the size of CSC is small. Our preliminary experimental results show the query time on *correlated* is very small, only a fraction of a *millisecond*. Therefore, we show the query performance of CSC on *independent* and *anti-correlated* only.

Figure 16 shows the query performance of CSC by varying the dimensionality, and Figure 17 shows that of CSC by varying the cardinality of the objects. Since querying on the Skycube is just fetching the cuboids and does not involve any computation, we omit its query time in our figures. In general, query processing of CSC is very fast (less than a second in most cases). This is due to the *local-comparison* property discussed in Section 3.2. Since the number of objects in each cuboid in CSC is greatly reduced, as shown in previous experiments, the number of comparisons within a cuboid is small, which leads to efficient query processing. Note that we use “millisecond” for *independent* and “second” for *anti-correlated*. The query response time on *anti-correlated* is larger than that on *independent*, because of the much larger size of CSC on *anti-correlated*.

5.3 Update Performance

In this section, we compare the update support of CSC and Skycube. The experiments are divided into two parts to show how good the object-aware update scheme is and how efficiently the CSC executes the object-aware update scheme. An object is updated in one dimension at a time as follows. We randomly choose a dimension of the object and randomly generate a value for that dimension. Again, we show the results on *independent* and *anti-correlated*.

To see the effect of the object-aware update scheme, we randomly choose an object to update from all objects. As a result, not all updated objects may affect CSC/Skycube. We call such scenario as “general update”. We implement the object-aware scheme on CSC and the naive update scheme

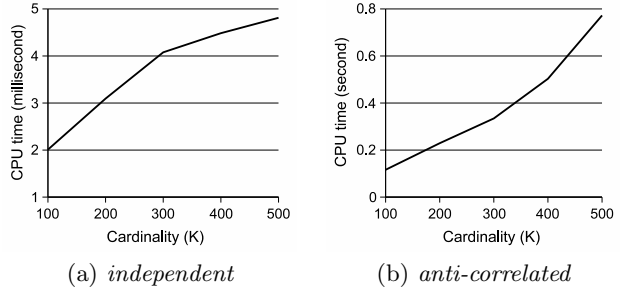


Figure 17: Effect of cardinality on query.

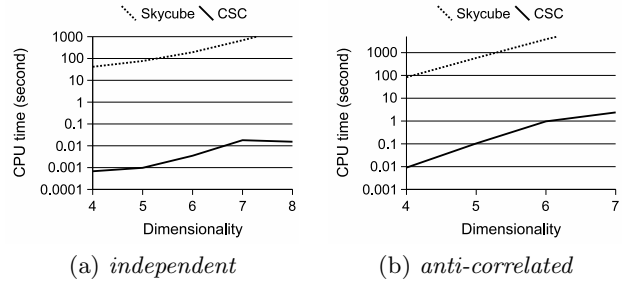


Figure 18: Comparing general update by varying dimensionality.

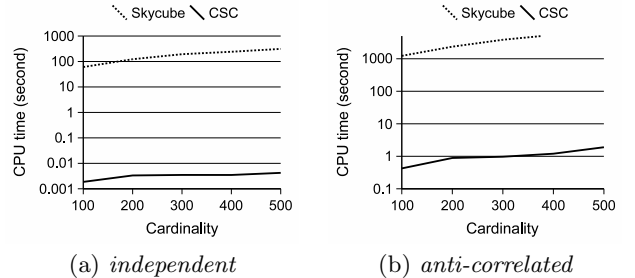


Figure 19: Comparing general update by varying cardinality.

on Skycube. Upon each update of an object, the skycube is computed from scratch. Figure 18 shows the general update by varying the dimensionality, and Figure 19 shows that by varying the cardinality. As expected, CSC outperforms Skycube by several orders of magnitude. This is because our object-aware scheme updates CSC incrementally and avoids many unnecessary computations when an object’s update does not affect the CSC structure. As we see in the figures, recomputing the skycube is extremely inefficient, and the cost increases dramatically with the increase of the dimensionality and cardinality of objects.

In the next set of experiments, we randomly choose a full-space skyline object to update, such that each update changes the CSC/Skycube structure. We call this scenario as “skyline update”. By eliminating those false updates, which does not trigger any computation on CSC, we show how efficiently our CSC executes the object-aware scheme.

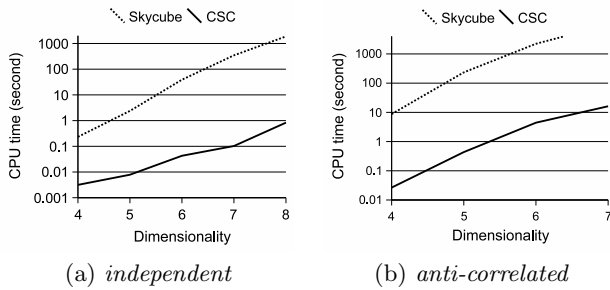


Figure 20: Comparing skyline update by varying dimensionality.

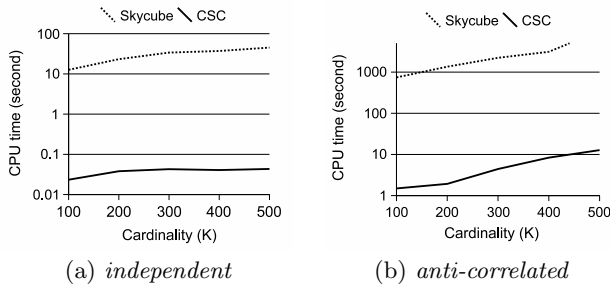


Figure 21: Comparing skyline update by varying cardinality.

For a fair comparison, the skycube is not computed from scratch upon each update. Instead, we first retrieve the candidate objects from the dataset if necessary, and then compute the skycube using existing objects in the skycube plus the new ones. Figure 20 shows the skyline update by varying the dimensionality and Figure 21 shows that by varying the cardinality of the objects. Even though Skycube is now computed from a much smaller dataset, CSC still outperforms Skycube by more than one order of magnitude. This is due to the expensive cost of cuboid computation in Skycube. In our CSC, existing cuboids are *not* recomputed. Instead, we only recompute the minimum subspaces of the affected objects. Since CSC is a concise structure and maintains much smaller number of objects in each cuboid than Skycube, the amount of computation is greatly reduced.

6. CONCLUSIONS

This paper is the first work addressing the update support of the skycube. The skycube precomputes all subspace skylines and provides fast query response at query time. However, in a dynamic environment where objects are updated frequently, updating the complete skycube becomes extremely inefficient. To support object updates while efficiently answering any subspace skyline query without accessing the whole dataset, we propose a new structure, called the *compressed skycube* (CSC), and an incremental and scalable update scheme. CSC concisely preserve the essential information of all subspace skyline, without comprising the query efficiency. Our update scheme for CSC is object-aware, such that updates of different objects trigger different amount of computation. Our extensive experiments shows

the CSC with object-aware update scheme outperforms the skycube in update support by several orders of magnitude. Besides, the CSC utilizes about 10% disk space compared with the Skycube, and can efficiently support queries.

7. REFERENCES

- [1] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Sarawagi. On the Computation of Multidimensional Aggregates. In *VLDB*, pages 506–521, 1996.
- [2] W.-T. Balke and U. Gütntzer. Multi-objective Query Processing for Database Systems. In *VLDB*, pages 936–947, 2004.
- [3] W.-T. Balke, U. Gütntzer, and J. X. Zheng. Efficient Distributed Skylining for Web Information Systems. In *EDBT*, pages 256–273, 2004.
- [4] O. Barndorff-Nielsen and M. Sobel. On the Distribution of the Number of Admissible Points in a Vector Random Sample. *Theory of Probability and its Application*, 11(2):249–269, 1966.
- [5] J. L. Bentley, K. L. Clarkson, and D. B. Levine. Fast Linear Expected-time Algorithms for Computing Maxima and Convex Hulls. In *Proc. of Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 179–187, 1990.
- [6] J. L. Bentley, H. T. Kung, M. Schkolnick, and C. D. Thompson. On the Average Number of Maxima in a Set of Vectors and Applications. *Journal of ACM*, 25(4):536–543, 1978.
- [7] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *ICDE*, pages 421–430, 2001.
- [8] C. Y. Chan, P.-K. Eng, and K.-L. Tan. Stratified Computation of Skylines with Partially-Ordered Domains. In *SIGMOD*, pages 203–214, 2005.
- [9] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with Presorting. In *ICDE*, pages 717–816, 2003.
- [10] P. Godfrey, R. Shipley, and J. Gryz. Maximal Vector Computation in Large Data Sets. In *VLDB*, pages 229–240, 2005.
- [11] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A System for the Efficient Execution of Multi-parametric Ranked Queries. In *SIGMOD*, pages 259–270, 2001.
- [12] W. Jin, J. Han, and M. Ester. Mining Thick Skylines over Large Databases. In *European Conf. on Principles of Data Mining and Knowledge Discovery (PKDD)*, pages 255–266, 2004.
- [13] D. Kossmann, F. Ramsak, and S. Rost. Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In *VLDB*, pages 275–286, 2002.
- [14] H. T. Kung, F. Luccio, and F. P. Preparata. On Finding the Maxima of a Set of Vectors. *Journal of ACM*, 22(4):469–476, 1975.
- [15] X. Lin, Y. Yuan, W. Wang, and H. Lu. Stabbing the Sky: Efficient Skyline Computation over Sliding Windows. In *ICDE*, pages 502–513, 2005.
- [16] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An Optimal and Progressive Algorithm for Skyline Queries. In *SIGMOD*, pages 467–478, 2003.
- [17] J. Pei, W. Jin, M. Ester, and Y. Tao. Catching the Best Views of Skyline: A Semantic Approach Based on Decisive Subspaces. In *VLDB*, pages 253–264, 2005.
- [18] F. Preparata and M. Shamos. *Computational Geometry: An Introduction*. Springer Verlag, 1985.
- [19] K.-L. Tan, P. K. Eng, and B. C. Ooi. Efficient Progressive Skyline Computation. In *VLDB*, pages 301–310, 2001.
- [20] Y. Tao, X. Xiao, and J. Pei. SUBSKY: Efficient Computation of Skylines in Subspaces. In *ICDE*, 2006.
- [21] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang. Efficient Computation of the Skyline Cube. In *VLDB*, pages 241–252, 2005.