# Efficient Computation of the Skyline Cube*

Yidong Yuan† Xuemin Lin† Qing Liu† Wei Wang† Jeffrey Xu Yu§ Qing Zhang†

The University of New South Wales†
{yyidong, lxue, qingl, weiw, qzhang}@cse.unsw.edu.au

The Chinese University of Hong Kong§
yu@se.cuhk.edu.hk

## Abstract

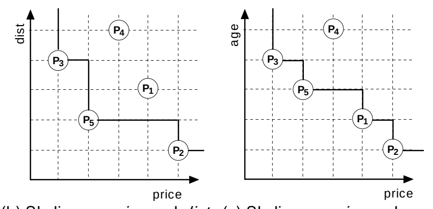
Skyline has been proposed as an important operator for multi-criteria decision making, data mining and visualization, and user-preference queries. In this paper, we consider the problem of efficiently computing a SKYCUBE, which consists of skylines of all possible non-empty subsets of a given set of dimensions. While existing skyline computation algorithms can be immediately extended to computing each skyline query independently, such "shared-nothing" algorithms are inefficient. We develop several computation sharing strategies based on effectively identifying the computation dependencies among multiple related skyline queries. Based on these sharing strategies, two novel algorithms, *Bottom-Up* and *Top-Down* algorithms, are proposed to compute SKYCUBE efficiently. Finally, our extensive performance evaluations confirm the effectiveness of the sharing strategies. It is shown that new algorithms significantly outperform the naïve ones.

## 1 Introduction

The *skyline* operator and its computation have attracted much attention recently. This is mainly due to the importance of skyline results in many applications, such as multi-criteria decision making [6], data mining and visualization [13], and user-preference queries [12]. A skyline query over $d$ dimensions selects the points that are not *dominated* by any other points restricted to those dimensions. Consider a typical skyline query example as follows: a real estate company has a list of properties online, each with *price*, *dist* (distance to city), *age*, and *bedroom number* attributes. Assume there are five properties as listed in Figure 1(a). A user, who is sensitive to *price* and *dist*, issues a skyline query on *price* and *dist* attributes. The result will be $\{P_3, P_5, P_2\}$ as shown in Figure 1(b). $P_4$, for example, is not in the skyline result because it is farther away from city compared with $P_3$ and its price is higher than $P_3$ too; that is, $P_4$ is dominated by $P_3$. The skyline query can greatly help user to narrow down the search range. Given the importance of skyline queries, many efficient skyline computation algorithms have been proposed recently [6, 21, 14, 8, 17, 16].

| | price | dist | age | bedroom num |
|---|---|---|---|---|
| $P_1$ | 4 | 3 | 2 | 2 |
| $P_2$ | 5 | 1 | 1 | 2 |
| $P_3$ | 1 | 4 | 4 | 1 |
| $P_4$ | 3 | 5 | 5 | 1 |
| $P_5$ | 2 | 2 | 3 | 1 |

(a) Properties and Their Value of Each Attribute



(b) Skyline on *price* and *dist* (c) Skyline on *price* and *age*

Figure 1: A Running Example

However, it is not uncommon that different users have different preferences. For example, another user might choose to use a combination of *price* and *age* to issue a skyline query. The result will be $\{P_3, P_5, P_1, P_2\}$, as shown in Figure 1(c). In such a client-server environment, multiple skyline queries with different dimension subsets could be issued simultaneously. In general, there could be $2^d - 1$ different skyline queries (i.e., skyline queries over any of the non-empty subsets of a $d$-dimensional set) for a relation with $d$ dimensions; each of the queries will return different results.

Although previous skyline computation algorithms can be used to compute each of the skyline queries individually, it is likely that the response time and the throughput of such a system might not be satisfactory. Therefore, it is beneficial to share the computation of multiple related skyline queries.

These observations motivate us to study the multiple skyline query processing problem. In particular, we propose to use a complete precomputation-based ap-

*This work was supported by ARC Discovery grant (DP0346004 and DP0345710) and UNSW Goldstar Grant (PS07248).

**Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005**

proach for answering multiple related skyline queries, in order to achieve minimal response time. The basic idea is to precompute the results of *all* possible skyline queries for a given dataset; as a result, no computation is needed to answer any of the queries. Such an approach is reminiscent of the precomputed *data cube* approach in data warehouse environment. Therefore, we propose a novel concept named SKYCUBE, which is a union of skyline results of all the non-empty subsets of $d$-dimensional set. The problem of efficient computing the SKYCUBE will be the focus of this paper.

Both the concept of SKYCUBE and its efficient construction algorithms have many applications. By physically materializing the entire or a selected part of the SKYCUBE according to the query workload, we can achieve minimal query response time and maximal system throughput. On the other hand, even if no precomputation is done, algorithms to efficiently construct the SKYCUBE can be easily extended to compute multiple related skyline queries on-the-fly. In addition, materializing the whole or part of the SKYCUBE enables novel analytical queries, such as "drill down" and "roll up". For example, users primarily concerning with *bedroom number* will get result $\{P_3, P_4, P_5\}$; they can "drill down" to a superset of the current skyline dimensions to find more interesting features of those properties, such as, $P_3$ will be the best choice if their secondary concern is *price*, while $P_5$ will be the choice if they in turn concern *dist*. Such additional information is helpful for the users to make the final decisions.

We note that computing a SKYCUBE over $d$ dimensions is obviously more challenging than computing the skyline over the same or any subset of the dimensions, as the SKYCUBE consists of $2^d - 1$ skyline results. Naïvely computing the $2^d - 1$ skyline results independently can be extremely inefficient. Our experiments have shown that such a naïve algorithm can be two orders of magnitudes slower than our proposed algorithms for the $500k$, 10-dimensional anti-correlated dataset. An immediate intuition is to share computation among the computing of individual skyline results, which is the case for most data cube computation algorithms [20]. However, we note that computing a SKYCUBE is more challenging than computing a data cube in that:

- In data cube computation, the group-by of a set of dimensions (e.g., $AB$) can be computed by the group-by of a superset of dimensions (e.g., $ABC$). However, this does *not* hold for SKYCUBE computation in general. For example, for the running example dataset, the skyline result on *price* and *bedroom number* is $\{P_3\}$; however, the skyline result of *bedroom number* will be $\{P_3, P_4, P_5\}$. This illustrates that non-skyline points (e.g., $P_4, P_5$) sharing the *same* values with a skyline point (e.g., $P_3$) on a subspace can appear as skyline points in that subspace.
- Skyline computation is more computationally expensive than performing a group-by (which is essentially a sorting); the time complexity of the former is $O(n \log^{d-2} n)$ (for $d \geq 4$)) while that of the latter is of $O(n \log n)$ only. Besides, existing skyline computation algorithms are intrinsically more complicated than sorting algorithms. Therefore, it is both important and non-trivial to find good computation sharing strategies.

In this paper, we propose two algorithms to efficiently compute a SKYCUBE, or in general, multiple related skyline results, by sharing as much computation as possible. The first algorithm works in a *breadth-first* and *bottom-up* manner, while the other works in a *depth-first* and *top-down* manner. In both algorithms, we also propose several important optimizations and special measures to deal with the special cases of duplicate values. We show in the experiments that the newly proposed algorithms outperform naïve ones significantly.

Our contributions can be summarized as follows:

- We propose the concept of the SKYCUBE, which is a union of skyline results over all non-empty subsets of the $d$ dimensions. We also propose to extend SQL with a `SKYCUBE BY` keyword. Answering multiple skyline queries using precomputed SKYCUBE can be thought of as a complementary approach to existing approaches where each skyline query is computed individually and on-the-fly.
- We investigate efficient implementation of the SKYCUBE computation. Two algorithms of different styles are developed. They make use of several important computation sharing principles, which are unique in the context of skyline cube computation. We also use several useful optimizations to deal with the hard case where there are points with duplicate values on some dimensions.
- Although the maximization of sharing computation among skyline calculation is a hard problem, we effectively identify several *sharing* strategies, as listed in Table 1. They are useful to other potential SKYCUBE computation algorithms too.
- We perform extensive experimental evaluation of the proposed algorithms. The new algorithms outperform the naïve ones by orders of magnitude.

Table 1: Sharing Strategies for SKYCUBE Computation

| Algorithm | Sharing Strategy | Section |
|---|---|---|
| BUS | share-results, share-sorting | 4 |
| TDS | share-partition-and-merging, share-parent | 5 |

The rest of the paper is organized as follows: Section 2 gives the problem definition and background information of skyline and data cube computation. Section 3 presents the related work. Section 4 describes a breath-first and bottom-up algorithm to compute a SKYCUBE, while Section 5 introduces a depth-first and top-down algorithm. We present experimental study in Section 6. Section 7 concludes the paper.

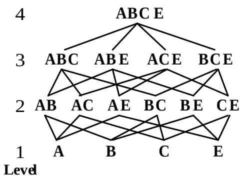
## 2 Preliminaries

### Skyline Query

Given a $d$-dimensional dataset $S$, we use $a_i$ $(1 \leq i \leq d)$ to represent each dimension. $\mathcal{D}$ is the dimension set consisting of all the $d$ dimensions, $\mathcal{D} = \{a_1, ..., a_d\}$. Let $p$ and $q$ be two data points in $S$, we denote the values of $p$ and $q$ on dimension $a_i$ as $p(a_i)$ and $q(a_i)$. For any dimension set $\mathcal{U}$, where $\mathcal{U} \subseteq \mathcal{D}$, $p$ *dominates* $q$ if $\forall a_i \in \mathcal{U}$, $p(a_i) \leq q(a_i)$ and $\exists a_j \in \mathcal{U}$, $p(a_j) < q(a_j)$ $(1 \leq i, j \leq d)$, denoted as $(p \prec q)_{\mathcal{U}}$. The skyline query on $\mathcal{U}$ returns all data points that are not dominated by any other points on $\mathcal{U}$. The result is denoted as $SKY_{\mathcal{U}}(S)$. The data point in the skyline result is called *skyline point*. See the running example in Figure 2, $SKY_{AB}(S) = \{P_2, P_3, P_5\}$.

| | A | B | C | E |
|---|---|---|---|---|
| $P_1$ | 4 | 3 | 2 | 2 |
| $P_2$ | 5 | 1 | 1 | 2 |
| $P_3$ | 1 | 4 | 4 | 1 |
| $P_4$ | 3 | 5 | 5 | 1 |
| $P_5$ | 2 | 2 | 3 | 1 |

(a) Dataset $S$



(b) Lattice Structure of a SKYCUBE

| Cuboid | Skyline |
|---|---|
| ABCE | $\{P_1, P_2, P_3, P_5\}$ |
| ABC | $\{P_1, P_2, P_3, P_5\}$ |
| ABE | $\{P_2, P_3, P_5\}$ |
| ACE | $\{P_1, P_2, P_3, P_5\}$ |
| BCE | $\{P_2, P_5\}$ |
| AB | $\{P_2, P_3, P_5\}$ |
| AC | $\{P_1, P_2, P_3, P_5\}$ |
| AE | $\{P_3\}$ |
| BC | $\{P_2\}$ |
| BE | $\{P_2, P_5\}$ |
| CE | $\{P_2, P_5\}$ |
| A | $\{P_3\}$ |
| B | $\{P_2\}$ |
| C | $\{P_2\}$ |
| E | $\{P_3, P_4, P_5\}$ |

(c) Cuboids

Figure 2: The SKYCUBE of the Running Example

[6] extended SQL's `SELECT` statement by an optional `SKYLINE OF` clause, such that users can specify the skyline dimensions as well as the criteria used to find extreme points on each such dimension (i.e., using one of `MIN`, `MAX` and `DIFF` keywords). For example, the example query expressed in Figure 1(b) can be written as:

```
SELECT  *   FROM Property
SKYLINE OF  price MIN, dist MIN
```

## Skycube

Over a set $S$ of $d$-dimensional points (on dimension set $\mathcal{D}$), there are $2^d - 1$ possible skyline queries on different dimension sets. We term the set of all possible skyline query results as SKYCUBE. In the SKYCUBE, each $SKY_{\mathcal{U}}(S)$ is called cuboid $\mathcal{U}$.

The structure of the SKYCUBE can be visualized in a lattice structure, similar to that of the data cube in the data warehouse (see the sample in Figure 2(b)). From the bottom to the top of the SKYCUBE, we number each level of the cuboid increasingly. For two cuboids $\mathcal{U}$ and $\mathcal{V}$ in the SKYCUBE, if $\mathcal{U} \subset \mathcal{V}$, we call $\mathcal{V}$ ($\mathcal{U}$) *ancestor* (*descendant*) *cuboid*. If their levels differ by one, we also call $\mathcal{V}$ ($\mathcal{U}$) *parent* (*child*) *cuboid*.

We proposed to further extend SQL by incorporating the `SKYCUBE BY` keyword, in a similar spirit of the original `CUBE BY` keyword proposal in [11]. For example, the SKYCUBE of the example property dataset shown in Figure 1(a) can be written as:

```
SELECT *     FROM Property
SKYCUBE BY   price MIN, distance_to_city MIN,
             age MIN, bedroom_number MAX
```

This query will compute the example SKYCUBE as shown in Figure 2(c).

There are several advantages of the proposed `SKYCUBE BY` keyword and the SKYCUBE operator. First, users can calculate the whole SKYCUBE in one concise and semantic-clear query, instead of issuing $2^d - 1$ skyline queries. Second, they provide more optimization opportunities. As will be discussed in the rest of this paper, there are much more efficient algorithms to compute the SKYCUBE (or, in general, multiple related skyline results) than computing each skyline individually.

Therefore, the rest of the paper focus on *how to compute multiple skyline queries efficiently*. The *focus* is on how to share computation of related cuboids as much as possible.

## Distinct Value Condition

We observe that, in general, over a set $S$ of data points on dimension set $\mathcal{D}$, for two dimension sets $\mathcal{U}$ and $\mathcal{V}$ ($\mathcal{U}$, $\mathcal{V} \subseteq \mathcal{D}$), where $\mathcal{U} \subset \mathcal{V}$, there is no containment relationship between $SKY_{\mathcal{U}}(S)$ and $SKY_{\mathcal{V}}(S)$. Consider the example in Figure 2, $SKY_E(S)$ is $\{P_3, P_4, P_5\}$; however, $SKY_{AE}(S)$ is $\{P_3\}$. This challenges us to develop the algorithms to compute multiple skyline queries efficiently. Specifically, Theorem 1 shows the relation between $SKY_{\mathcal{U}}(S)$ and $SKY_{\mathcal{V}}(S)$.

**Theorem 1** *Given a set $S$ of data points on dimension set $\mathcal{D}$, $\mathcal{U}$ and $\mathcal{V}$ are two sub dimension sets ($\mathcal{U}$, $\mathcal{V} \subseteq \mathcal{D}$), where $\mathcal{U} \subset \mathcal{V}$. On dimension set $\mathcal{V}$, each skyline point $q$ in $SKY_{\mathcal{U}}(S)$ is*

- *either dominated by another skyline point $p$ in $SKY_{\mathcal{U}}(S)$;*
- *or a skyline point in $SKY_{\mathcal{V}}(S)$;*

*Proof*: For each skyline point $q$ in $SKY_{\mathcal{U}}(S)$, if there is another data point $p$ such that $p(a_i) = q(a_i)$ $(\forall a_i \in \mathcal{U})$, $p$ may dominate $q$ on dimension set $\mathcal{V}$ if $p$ dominates $q$ on the dimension set $\mathcal{V} - \mathcal{U}$. Obviously, $p$ is a skyline point in $SKY_{\mathcal{U}}(S)$ as well. Otherwise, if such a skyline point $p$ does not exist, $q$ is a skyline point in $SKY_{\mathcal{V}}(S)$ because no other data points can dominate it on dimension set $\mathcal{V}$. ■

Nonetheless, we identify there is a special condition, termed as *distinct value condition*, such that the containment relationship between parent and child cuboids holds, as stated in the following Corollary 1.

**Corollary 1 (Distinct Value Condition)** *Given a set $S$ of data points on dimension set $\mathcal{D}$. For any two data points $p$ and $q$, if $p(a_i) \neq q(a_i)$ $(\forall a_i \in \mathcal{D})$, then for two sub dimension sets $\mathcal{U}$ and $\mathcal{V}$, ($\mathcal{U}$, $\mathcal{V} \subseteq \mathcal{D}$), where $\mathcal{U} \subset \mathcal{V}$, $SKY_{\mathcal{U}}(S) \subseteq SKY_{\mathcal{V}}(S)$.*

In the rest of paper, we assume that user's preference on each dimension is fixed and we use `MIN` for all dimensions in the skyline computation. The following table summarizes the notations used in this paper.

| Notation | Definition |
|---|---|
| $S$ | the dataset |
| $n$ | the number of data points in $S$ |
| $p$, $q$ | data point |
| $\mathcal{D}$ | the dimension set |
| $d$ | the number of dimensions in $\mathcal{D}$ |
| $a_i$ | one dimension $(1 \leq i \leq d)$ |
| $p(a_i)$ | value of $p$ on dimension $a_i$ |
| $\mathcal{U}, \mathcal{V}$ | dimension set, $\mathcal{U}$, $\mathcal{V} \subseteq \mathcal{D}$ |
| $(p \prec q)_{\mathcal{U}}$ | $p$ dominates $q$ on $\mathcal{U}$ |
| $SKY_{\mathcal{U}}(S)$ | skyline of $\mathcal{U}$ in $S$ |

## 3 Related Work

### 3.1 Data Cube

The concept of data cube was first proposed in [11]. Efficiently computing data cubes has been an active research topic. A number of techniques have been reported in the literature [11, 1, 19, 23, 5, 22, 9]. Specifically, several heuristics for computing multiple group-bys (i.e., cuboids) efficiently have been identified, such as *smallest-parent*, *cache-results*, *amortized-scans*, *share-sorts*, and *share-partitions* [20].

### 3.2 Skyline Computation

The problem of finding skyline is a typical type of the multi-objective query processing [2]. It is first investigated in [15] where an $O(n \log^{d-2} n)$ time algorithm for $d \geqslant 4$ and an $O(n \log n)$ time algorithm for $d = 2, 3$ are proposed. An expected linear running time algorithm is presented in [4] if the data distribution on each dimension is independent. [4] also estimates that the expected number of skyline points under the independent distribution assumption is $\mathrm{O}(\ln^{d-1} n)$. In [7], the estimation is improved to $\Theta((\ln^{d-1} n)/(d-1)!)$.

Research in skyline computation in the context of database can be classified into three categories: *nested-loop-based*, *divide-and-conquer-based*, and *index-based*.

*Block-nested-loop* (BNL) algorithm [6] and *Sort-filter-skyline* (SFS) [8] belong to the first category. [6] also presents an algorithm based on the divide-and-conquer technique (DC). As our techniques are developed based on these three algorithms, we dedicate separate parts below to illustrate the details of them.

Index-based techniques are proposed in several work [6, 21, 14, 17]. Index scanning method is proposed in [6]. [21] presents the first *progressive* algorithms, namely *Bitmap* and *Index method*, which can output the skyline without having to scan the whole dataset. A skyline computation algorithm based on the indexed dataset is first developed in [14]. It is based on the nearest neighbour query, which adopts the divide-and-conquer paradigm on the R-tree index. Then [17] proposes a branch and bound algorithm to progressively output skyline points on dataset indexed by R-tree. One of the most important properties of this technique is that it guarantees the minimum I/O costs.

There are some other skyline algorithms studied recently. [3] shows how to make efficient computation through the distributed database system and an approximate computation algorithm is presented in [13].

We note that one of the latest work is from [10], which surveyed the runtime complexities of existing "generic" skyline algorithms and introduced a new generic skyline algorithm (LESS) that has $O(n)$ average case running time.

**BNL and SFS Algorithms**

To compute the skyline, BNL scans the dataset and compares each data point $p$ with a list of candidate skyline points. Initially, the candidate list is empty. If $p$ is dominated by any data point in the list, it is discarded. If $p$ dominates some of data points in the list, it is inserted into the list and all dominated data points are deleted from the list. If $p$ is neither dominated, nor dominates, any data points in the list, it is inserted into the list as a new candidate. After examining all the data points, BNL outputs all the candidates in the list as the skyline result.

In order to reduce the number of pairwise comparisons between data points in BNL, SFS introduces *entropy value* [8]. The entropy value of a data point $p$ on dimension set $\mathcal{U}$ is $E_{\mathcal{U}}(p) = \sum_{\forall a_i \in \mathcal{U}} (\ln p'(a_i) + 1)$, where $p'(a_i)$ is the normalized value of $p(a_i)$. Intuitively, the less an entropy value is, the less likely the data point is dominated by others. Based on this observation, SFS presorts the dataset in non-decreasing order of the entropy value of each data point. Then, SFS examines the data points by this order in the similar way as that of BNL. As the data point with smaller entropy value is examined first, the skyline point could be found earlier. Therefore, the number of comparisons between data points and the non-skyline points in the candidate list, which is unnecessary, is reduced.

**DC Algorithm**

To compute the skyline for a set of data points, DC first divides them into several parts and computes the skyline over each part. Then DC merges these skylines to obtain the final one. Consider the example to compute the skyline of cuboid $AB$ on the dataset in Figure 2(a).

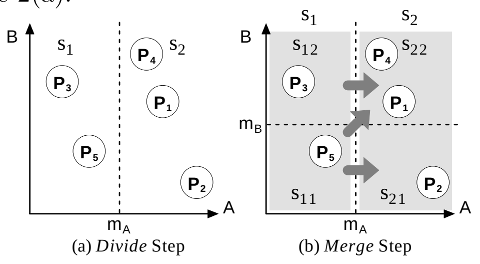


Figure 3: Divide and Conquer Algorithm

- First, DC calculates the median $m_A$ of all data points on dimension $A$ and divides the dataset into two parts, $l_1$ and $l_2$ (*divide* step). $l_1$ contains the data points whose values on dimension $A$ are less than $m_A$. $l_2$ contains all others.
- Then skyline of $l_1$ ($l_2$) is computed. This is done by recursively applying divide step until one point left. In that case, to compute skyline is trivial. The skyline result $s_1$ ($s_2$) is shown in Figure 3(a).

- To obtain the overall skyline, DC eliminates the data points in $s_2$ which are dominated by the ones in $s_1$ (*merge* step). In merge step, the median $m_B$ of data points in $s_1$ on dimension $B$ is calculated. $s_1$ and $s_2$ are further divided into $s_{11}$, $s_{12}$, $s_{21}$, and $s_{22}$ with $m_B$, shown in Figure 3(b). Clearly, the data points in $s_{21}$ have smaller value on dimension $B$ than that of data points in $s_{12}$. Therefore, the data points in $s_{21}$ are not dominated by any one in $s_{12}$. As a result, after further partition, DC only needs to merge $s_{11}$ and $s_{21}$, $s_{12}$ and $s_{22}$, $s_{11}$ and $s_{22}$, respectively (shown as arrows in Figure 3(b)). Each merge applies merge step recursively until one data point is remained in either part, then to merge them is trivial. In this example, $P_1$ and $P_4$ are eliminated from $s_2$ after merge. The final skyline of $AB$ is $\{P_3, P_5, P_2\}$.

### 3.3 The Difficulties of Generalizing Existing Algorithms for Skycube Computation

In this subsection, we briefly outline why naïve generalizations of these algorithms are not likely to be efficient for the SKYCUBE computation problem. For the ease of illustration, we consider computing skylines of a child cuboid $A$ and a parent cuboid $AB$ on a dataset $S$. Our focuses are on the possibility of "sharing" among multiple skyline computation as well as the ability to scale with the number of dimensions $d$.

Firstly, note that, in general, a child cuboid ($SKY_{\mathcal{A}}(S)$) cannot be calculated from a parent cuboid ($SKY_{\mathcal{AB}}(S)$). Therefore, naïve generalization of all the existing algorithms have to calculate each cuboid individually from the source dataset, which leads to huge amount of redundant computation especially when the number of dimension, $d$, is large. In short, the naïve algorithms are not efficient because they *cannot share results* of related computation.

Secondly, all the existing algorithms were designed to compute a single skyline query while having certain pre-conditions, e.g., input is sorted or indexed. Substantial additional preprocessing need to be done to ensure such pre-conditions while computing multiple skylines.

- For example, SFS, the best of the nested-loop-based skyline algorithms, need to sort the source dataset according to the entropy values on the skyline dimensions. It is easy to see that, to compute the skyline of a child cuboid ($A$), a sorting of the whole dataset is still compulsory to obtain the required input even if we sorted the input while computing the skyline on the parent cuboid ($AB$). In general, to compute a SKYCUBE for $d$ dimensions, naïve generalization of SFS algorithm will at least require $2^d - 1$ sorting of the whole dataset $S$.
- For another example, most index-based skyline algorithms require appropriate indexes built on the skyline dimensions. When computing a child cuboid ($A$), the index of the parent cuboid ($AB$) either cannot be used ($B^+$-tree index) or might not be optimal (R-tree index). This requires building $2^d - 1$ indexes beforehand or on-the-fly. In addition, some index-based algorithms do not scale with number of dimensions, either for performance reasons (R-tree index) or space overhead (bitmap index).

In short, the naïve algorithms are not efficient not only because they *cannot share computation*, but also because they incur additional overhead among related computation.

The above analysis also motivates us to find as much data and computation sharing opportunities as possible to efficiently compute multiple skyline queries, which is one of the contributions of this paper.

We note that a similar work is independently developed in [18], where a *Skyey* algorithm is proposed to compute skylines (as well as their *signatures*) of all subspaces of a set of dimensions. *Skyey* algorithm works in top-down manner and takes advantage of sharing sorting among different subspaces.

## 4 Bottom-Up Skycube Computation

In this section, we present our Bottom-Up Skycube algorithm (*BUS*). For the ease of illustration, we first assume that the distinct value condition holds. BUS algorithm takes advantages of two computation sharing strategies: *sharing result* and *sharing sorting*. To save unnecessary pairwise comparisons between points, a filter based heuristic is developed. This heuristic greatly improves the performance of BUS, which is confirmed by our extensive experiment evaluations. Finally we discuss the modifications on BUS when dealing with the case that the distinct value condition does not hold in the dataset.

### 4.1 Bottom-Up Skycube Algorithm (BUS)

The basic idea of BUS algorithm is to compute each cuboid in the SKYCUBE in a level-wise and bottom-up manner. Each cuboid is computed by a nested-loop-based algorithm similar to SFS [8]. Recall that naïve generalization of SFS algorithm for the SKYCUBE computation does not have good performance for two reasons: (1) each cuboid is computed individually; (2) it requires $2^d - 1$ sorting of the original dataset according to the entropy value defined on the dimension set of the cuboids.

In our BUS algorithm, we identify *two computation sharing strategies* which address the first issue; to handle the second issue, we propose a sorting-and-filtering technique that effectively reduces the number of sorting from $2^d - 1$ to $d$ while being able to avoid many dominance tests via the *filtering* function.

#### Sharing Result

According to Corollary 1, we can easily derive that the union of the child cuboids belongs to the parent cuboid. Therefore, during computing a cuboid, the data points, which are in one of its child cuboid, are guaranteed to be skyline points. The advantage of this "result sharing" is two-fold: on one hand, it reduces the size of input to the individual skyline computation process; on the other hand, fewer number of dominance tests are performed because those points do not need to be examined again. We call this strategy sharing result.

In addition, since we compute the SKYCUBE in a level-wise and bottom-up fashion, we can exploit this result sharing on all the child cuboids of the cuboid in question. In other words, we can unite all the child cuboids as the starting point of the computation of the parent cuboid. In terms of implementation, we use efficient bitmap operations so that the union can be done in linear time.

### Sharing Sorting

To avoid the explosion of the number of sorting required by SKYCUBE computation based on the SFS algorithm, we propose to change the sorting criteria as follows: when computing the cuboid $\mathcal{V}$, we accept input sorted on any dimension $a_i$ ($a_i \in \mathcal{V}$) (or in general, any $\mathcal{U} \subseteq \mathcal{V}$). Because of the nested loop nature of the skyline computation algorithms used in our BUS algorithm, changing input sorting order only affects the performance not the correctness. We make such change for two reasons:

- It enables us to reduce the number of sorting required to compute the SKYCUBE from $2^d - 1$ to $d$. We only need to sort the source dataset $d$ times, each on a distinct dimension. It is obvious that such scheme requires the *minimum* number of sorting to compute all the cuboids in the SKYCUBE.
- Such input ordering is *optimal* in the sense that every data point added to the candidate list is guaranteed to be a skyline point. As a result, the length of the candidate list is kept minimum, which saves both memory and computation.

Furthermore, we complement this sorting scheme by a filtering process during the skyline computation, which further reduces the cost of dominance test. We will cover this part shortly in Section 4.2.

In terms of implementation, we use the heuristic such that when computing cuboid $\mathcal{V}$, we always pick the input sorted on the dimension $a_i \in \mathcal{V}$ and the domain of $a_i$ is the largest among all $a_j \in \mathcal{V}$. We note that a similar heuristic is used in the bottom-up data cube computation algorithm too [5].

### BUS Algorithm

Based on the above two sharing strategies, we develop our BUS, which computes the SKYCUBE level by level from bottom to top. The algorithm is listed in Algorithm 1. To compute every cuboid $\mathcal{V}$, BUS examines each data point in the order against the skyline points computed so far. If this data point is a skyline point of child cuboid, it will be inserted into the skyline directly (line 6 − 7 in Algorithm 1). Otherwise it is compared with the current skyline to determine whether it is a new skyline point by calling the function **Evaluate**, i.e., doing a *dominance test* by comparing the $d$ attribute values of two points (line 9).

## 4.2 Optimizing Dominance Test via Filtering

The simplest implementation of the **Evaluate** function (line 9) in Algorithm 1 is to do a dominance test.

**Algorithm 1** **BUS** ($S$)

```
Input:
    $S$: a set of $d$-dimensional data points
Output:
    every cuboid $\mathcal{V}$, $SKY_{\mathcal{V}}(S)$
Description:
 1: sort $S$ on every dimension $a_i$ (in non-decreasing order)
    to form $d$ sorted lists $l_{a_i}$ $(1 \le i \le d)$
 2: for each level from bottom to top of the skycube and
    each cuboid $\mathcal{V}$ in this level do
 3:   $SKY$ = the union of all the child cuboids
 4:   choose a sorted list $l_{a_i}$ $(a_i \in \mathcal{V})$
 5:   for each data point $q$ in $l_{a_i}$ do
 6:     if $q \in SKY$ then
 7:       insert $q$ into $SKY_{\mathcal{V}}(S)$
 8:     else
 9:       Evaluate($q$, $SKY_{\mathcal{V}}(S)$)
```

However, the complexity of this operation is $O(d)$, which might be expensive when $d$ is large. In addition, such operation is called frequently within loops. Therefore, to further optimize the performance of BUS algorithm, we adopt a filtering procedure which can drastically reduce the number of such dominance test. The key idea is we do not need to do the relative expensive dominance test (with complexity $O(d)$) if the two points do not pass an efficient filtering test (with complexity $O(1)$).

We define a *filter function* as a multivariate monotonic non-decreasing function that takes a data point as the parameter; the function value is called the *filter value* of the data point. In BUS algorithm, we use the following filter function defined on dimension set $\mathcal{U}$, because it outperforms other filter functions in our experiments with various parameters.

$$f_{\mathcal{U}}(p) = \sum_{\forall a_i \in \mathcal{U}} p(a_i)$$

For example in Figure 2, $f_{ABCE}(P_1) = 11$ and $f_{ABCE}(P_4) = 14$. From the property of the multivariate monotonic non-decreasing function, it is easy to derive that for two data points $p$ and $q$, if $f_{\mathcal{U}}(p) \le f_{\mathcal{U}}(q)$, then $q$ does not dominate $p$ on dimension set $\mathcal{U}$. In the above example, based on their filter values, without detailed comparison on each dimension, we know that $P_4$ cannot dominate $P_1$ on $ABCE$ because the filter value of $P_1$ is smaller.

Algorithm 2 presents the implementation of **Evaluate** function using the filter based heuristic. The algorithm requires to maintain the candidate list $SP$ such that points inside are in a non-decreasing order of their filter values. When evaluating data point $q$ against skyline point $p$, we first compare their filter values. If $q$'s filter value is smaller than $p$'s, $p$ and all the skyline points after $p$ in the candidate list cannot dominate $q$. Therefore, it is known immediately that $q$ is a new skyline point. Otherwise, $p$ and $q$ are further compared on each dimension to determine whether $q$ is a new skyline point.

Here is an example to illustrate the filter based heuristic. Suppose to compute $SKY_{AB}(S)$ of the dataset in Figure 2(a). The data points are already

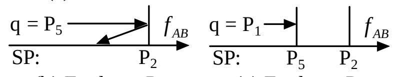
**Algorithm 2** **Evaluate** ($q$, $SP$)

```
Input:
    $q$: a data point to be evaluated
    $SP$: the cuboid $\mathcal{U}$ computed so far
Output:
    insert $q$ into $SP$ if it is a skyline point of $\mathcal{U}$
Description:
1: for each data point $p$ in $SP$ do
2:     if $f_{\mathcal{U}}(q) < f_{\mathcal{U}}(p)$ then
3:         insert $q$ into $SP$; return
4:     else if $(p \prec q)_{\mathcal{U}}$ then
5:         discard $q$; return
6: insert $q$ into $SP$
```

sorted on dimension $B$. The access order and filter value of each data point are given in Figure 4(a). As $P_2$ is skyline point of the child cuboid $B$, it is directly inserted into the skyline. Figure 4(b) shows the case when $P_5$ is evaluated. Because $f_{AB}(P_5) < f_{AB}(P_2)$, $P_5$ is inserted into the skyline immediately without compared with $P_2$ on dimensions $AB$. Then, $P_1$ is evaluated (Figure 4(c)). $P_1$ is first compared with $P_5$. Since $f_{AB}(P_5) < f_{AB}(P_1)$ and $P_5$ dominates $P_1$, $P_1$ is discarded. Similarly, $P_3$ and $P_4$ are evaluated. The skyline of $AB$ is $\{P_5, P_3, P_2\}$.

| Sort on B | $P_2$ | $P_5$ | $P_1$ | $P_3$ | $P_4$ |
|---|---|---|---|---|---|
| $f_{AB}$ | 6 | 4 | 7 | 5 | 8 |

(a) Access Order and Filter Values



(b) Evaluate $P_5$ (c) Evaluate $P_1$

Figure 4: Example of Filter Based Heuristic

### 4.3 BUS in the General Case

If the dataset does not conform to the distinct value condition, we extend BUS as follows.

As stated in Section 2, generally, only part of child cuboid $\mathcal{U}$ belong to parent cuboid $\mathcal{V}$. So, in order to share child cuboid $\mathcal{U}$, now BUS has to examine whether they are still skyline points on dimension set $\mathcal{V}$. According to Theorem 1, to examine data point $q$ in $SKY_{\mathcal{U}}(S)$, we only need to compare $q$ with another data point $p$ in $SKY_{\mathcal{U}}(S)$ which is identical to $q$ on dimension set $\mathcal{U}$. If such a data point $p$ does not exist, $q$ is a skyline point of cuboid $\mathcal{V}$. Otherwise, $q$ and $p$ are compared on dimension set $\mathcal{V} - \mathcal{U}$ to determine which is the skyline point of $\mathcal{V}$. Note that the above examination is trivial because the skyline points are always maintained in non-decreasing order of their filter values. As two skyline points with different filter values are not identical, we examine the skyline points, which have same filter values, with each other only.

When the distinct value condition does not hold, BUS cannot guarantee that only the skyline points are inserted into the candidate list. Consider the example of computing cuboid $CE$ in Figure 2. Assume we have already sorted data points on dimension $E$ and the evaluation order is $P_3 \rightarrow P_4 \rightarrow P_5 \rightarrow P_1 \rightarrow P_2$. $P_3$, which is not a skyline point of cuboid $CE$, is inserted into the candidate list as no data point in the list dominates it. To eliminate such false hit, every time BUS reads a group of data points, which have same values on the sorted dimension, into buffer. Obviously, these data points are not dominated by the ones after them because the former have smaller values on this sorted dimension. Then, the skyline points of these data points are computed as candidates. Since the size of such group of data points is small, to compute the skyline is inexpensive. After that, BUS evaluates these candidates against the skyline computed so far. It is clear that with such extension in BUS, the property that only skyline points are inserted into the candidate list still holds.

## 5 Top-Down Skycube Computation

In this section, we present our Top-Down Skycube algorithm (*TDS*). TDS relies on a novel Shared-Divide-and-Conquer skyline algorithm (*SDC*), which adopts the basic idea of Divide-and-Conquer skyline algorithm (*DC*) [6] while it computes multiple related skyline queries simultaneously with little additional overhead. In TDS, two new computation-sharing strategies: *sharing-partition-and-merging* and *sharing-parent* are developed.

In the following parts, we first introduce SDC algorithm. Then, the detailed TDS algorithm under the distinct value condition is presented. Finally we discuss the necessary modifications on TDS in general case.

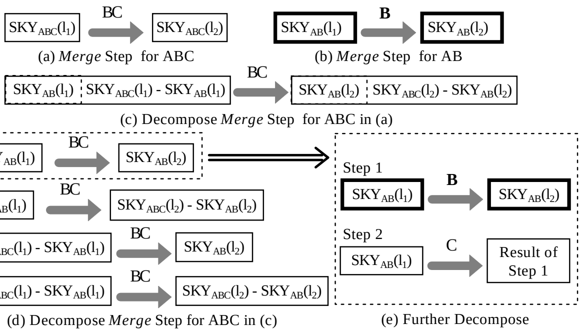
### 5.1 Computation Sharing Opportunities for the DC Algorithm

Although DC is one of the most efficient skyline computation algorithms, it computes skyline for one cuboid only. However, we observe that *both* divide step and merge step can be shared between a parent cuboid and its child cuboid. We call such sharing principles as *sharing-partition-and-merging* collectively.

As described in details in Section 3.2, the DC algorithm always chooses the median point with respect to the same dimension to divide the input set in the divide phase. We call such a dimension *partitioning dimension* (e.g., $A$ is the example shown in Figure 3(a)). Obviously, to compute skyline for child cuboid $\mathcal{U}$ and parent cuboid $\mathcal{V}$, if we divide the dataset using the same partitioning dimension, the partition results in both computations are the same. Hence, if we compute $SKY_{\mathcal{U}}(S)$ and $SKY_{\mathcal{V}}(S)$ simultaneously using the DC algorithm, the divide step can be shared.

In merge step, DC merges the skylines of each part to obtain the final skyline. We observe that in this step the computation for the parent cuboid $\mathcal{V}$ can be shared by its child cuboid $\mathcal{U}$. Therefore, the merge step for $\mathcal{U}$ is saved. We use the example illustrated in Figure 5 to give an intuition of such sharing. We consider computing two cuboids: $ABC$ and $AB$. After the divide step with the partitioning dimension $A$, the dataset is divided into two parts $l_1$ and $l_2$. Then for cuboid $ABC$ ($AB$), the skyline of each part is calculated as $SKY_{ABC}(l_1)$ and $SKY_{ABC}(l_2)$ ($SKY_{AB}(l_1)$

and $SKY_{AB}(l_2)$). After that, DC eliminates the points in $SKY_{ABC}(l_2)$ ($SKY_{AB}(l_2)$) which are dominated by the ones of $SKY_{ABC}(l_1)$ ($SKY_{AB}(l_1)$) on dimension set $BC$ ($B$), as shown in Figure 5(a) ((b)). According to Corollary 1, it is easy to derive that the skyline of $l_1$ ($l_2$) computed for the parent cuboid $ABC$ contains that for the child cuboid $AB$. So, we decompose the skylines of $l_i$ ($i = 1, 2$) on $ABC$, $SKY_{ABC}(l_i)$, into two parts as $SKY_{AB}(l_i)$ and $SKY_{ABC}(l_i) - SKY_{AB}(l_i)$ (Figure 5(c)). Now we use the two parts on the left side to merge those on the right side separately which brings us to Figure 5(d). The first part in Figure 5(d) can be further decomposed into two steps (Figure 5(e)). In step 1, $SKY_{AB}(l_1)$ and $SKY_{AB}(l_2)$ are merged on dimension set $B$ only. Then, we merge $SKY_{AB}(l_1)$ and the result of step 1 on dimension set $C$. Note that here the step 1 is just the merge step for $AB$ (Figure 5(b)). As a result, it is clear that the merge step for the parent cuboid is shared by its child cuboid.



SKY_ABC(l1)
BC
SKY_ABC(l2)
(a) Merge Step for ABC
SKY_AB(l1)
B
SKY_AB(l2)
(b) Merge Step for AB
SKY_AB(l1) | SKY_ABC(l1) - SKY_AB(l1)
BC
SKY_AB(l2) | SKY_ABC(l2) - SKY_AB(l2)
(c) Decompose Merge Step for ABC in (a)
Step 1
Step 2
C
Result of Step 1
(d) Decompose Merge Step for ABC in (c)
(e) Further Decompose

Figure 5: Share Merging

From the above observation, it is clear that divide step and merge step in DC can be shared by the two cuboids as long as the dimension sets of one cuboid contains that of another (i.e., ancestor/descendant or parent/child relationship in the cube lattice). This motivates our SDC algorithm.

## 5.2 Shared-Divide-and-Conquer Algorithm (SDC)

The basic idea of Shared-Divide-and-Conquer Algorithm (SDC) is to compute a number of "related" cuboids at a time based on the aforementioned sharing principles. More specifically, SDC can compute a set of cuboids on a *path* in the cube lattice at a time. We call the set of dimensions associated with the cuboids *path of dimension sets*, or simply *path* when there is no ambiguity. Without loss of generality, we can always represent a path in a canonical form by (1) arranging the dimension sets in ascending order of their level values; and (2) two adjacent dimension sets share the common prefix. For instance, the canonical form of the path $\{ABCE, ABE, AB\}$ is $\langle AB, ABE, ABEC\rangle$.

### Skylist

The key of SDC algorithm is a novel data structure, *skylist*, that concisely represents skylines for cuboids belonging to a path. Given a path $c$, a skylist consists of a number of **elements**, each of which stores the skyline points for the corresponding cuboid in $c$. Recall Corollary 1, for two cuboids $\mathcal{U}$ and $\mathcal{V}$, $SKY_{\mathcal{U}}(S) \subseteq SKY_{\mathcal{V}}(S)$ if $\mathcal{U} \subset \mathcal{V}$. A skylist organizes the skyline points of each cuboid in $c$ in the following accumulative way: the first element stores the skyline points of the first cuboid; and the $i$-th element stores the difference between the skyline points of the $i$-th cuboid and the $(i-1)$-th one. For example, for a path $c = \langle A, AB, ABC\rangle$ on the dataset in Figure 2, the skylist is $\langle\{P_3\}, \{P_2, P_5\}, \{P_1\}\rangle$.

The skylist data structure can benefit our SDC algorithm in the following ways. First, skylist stores the skyline points for a path in a compact way which saves much storage space. Second, as storing the data points accumulatively, skylist enables our SDC algorithm to decompose divide step and merge step during the computation such that they can be shared by all cuboids in the path.

Here are some basic operations on the skylist data structure, which are used in the SDC algorithm.

**split($l$, $v$, $a_i$)**: Given a skylist $l$ and a value $v$ on dimension $a_i$, split operation divides $l$ into two skylists $s_1$ and $s_2$. For each data point $p$ in $l$, if $p(a_i) < v$, it is moved to the corresponding element in $s_1$. Otherwise, $p$ is moved to the corresponding element in $s_2$.

**union($s_1$, $s_2$)**: Unite the data points in the corresponding elements of the two skylists $s_1$ and $s_2$ to build a new skylist.

**filter($c$, $s_1$, $s_2$)**: Given a path $c$ and two skylists $s_1$ and $s_2$, filter operation updates $s_2$ such that the remaining points in $s_2$ is still skyline points on the corresponding dimension set even if $s_1$ and $s_2$ are united. Implementation-wise, for each data point in every (the $i$-th) element of $s_2$, if it is dominated by any data points in $s_1$ on the dimension set corresponding to that element, it is moved to the next element of $s_2$.

For example, consider the skylists of the path $\langle A, AB\rangle$ on the top left ($\{P_3\}$) and the top right ($\{P_1, P_4\}$) datasets in Figure 3(b). Two skylists are constructed, $s_{12} = \langle\{P_3\}, \{\}\rangle$ and $s_{22} = \langle\{P_4\}, \{P_1\}\rangle$. To filter $s_{22}$ by $s_{12}$, we first examine $P_4$ against $P_3$ on dimension set $A$. As $P_4$ is dominated by $P_3$ on $A$, it is moved to next element. Then the second element of $s_{22}$ is examined. Because $P_3$ dominates $P_4$ but not $P_1$ on $AB$, $P_4$ is discarded. So after filter, $s_{22} = \langle\{\}, \{P_1\}\rangle$.

### SDC Algorithm

To answer skyline queries on a path, SDC processes the data points in the divide-and-conquer manner similar to DC. The detailed SDC is presented in Algorithm 3. To compute skylines on a path $c$, an initial skylist $l$ corresponding to $c$ is constructed, where all data points are stored in the first element of $l$.

We use the following example to further illustrate SDC. Consider the skyline queries on a path $c = \langle A, AB, ABC\rangle$, over the dataset in Figure 2. Initially, a skylist $l$ is constructed according to $c$ and all data points are stored in the first element of $l$ as shown in Figure 6(a). After recursive split on dimension $A$, $l$ is split into 5 skylists, each of which contains one data point only (Figure 6(b)). Then we merge the skylist $l_3$

**Algorithm 3** **SDC** ($c$, $l$)

```
Input:
    $c$: a path, the last dimension set in this path is $\mathcal{V}$
    $l$: a skylist corresponding to $c$ stores the data points
Output:
    a skylist corresponding to $c$ stores the skyline result
Description:
 1: if $|l| = 1$ then /* $|l|$: the number of points in $l$ */
 2:     return $l$
 3: else
 4:     $a_1$ = the first dimension of $\mathcal{V}$
 5:     $m_{a_1}$ = the median of $l$ on dimension $a_1$
 6:     $(l_1, l_2)$ = split($l$, $m_{a_1}$, $a_1$)
 7:     $s_1$ = SDC($c$, $l_1$)
 8:     $s_2$ = SDC($c$, $l_2$)
 9:     $s_3$ = SDC_Merge($c$, $s_1$, $s_2$, 2)
10:     return union($s_1$, $s_3$)
```

and $l_4$ first. As both of them contain one data point only, in this merge we directly call filter operation on them. In the filter process, $P_5$ is compared against $P_3$. Since $P_3$ dominates $P_5$ on dimension set $A$ and does not on dimension set $AB$, $P_5$ is moved to the second element. Then, we union them and the merge result $s_1$ is shown in Figure 6(c). Similarly, $l_6$, $l_7$, and $l_8$ are merged and the result is $s_2$. In order to merge $s_1$ and $s_2$ to obtain the final skylines, we split $s_1$ and $s_2$ with the median of data points in $s_1$ on dimension $B$. Figure 6(d) shows the split results. After that, three filter operations are processed. The filter results $r_1$ and $r_3$ are shown in Figure 6(e). Finally, we union $s_1$ and the merge results to the final skylist. To retrieve the skyline of the $i$-th cuboid in $c$, we output all data points in all the $j$-th ($j \leq i$) elements in the final skylist. For example, $SKY_{AB}(S) = \{P_3, P_2, P_5\}$.

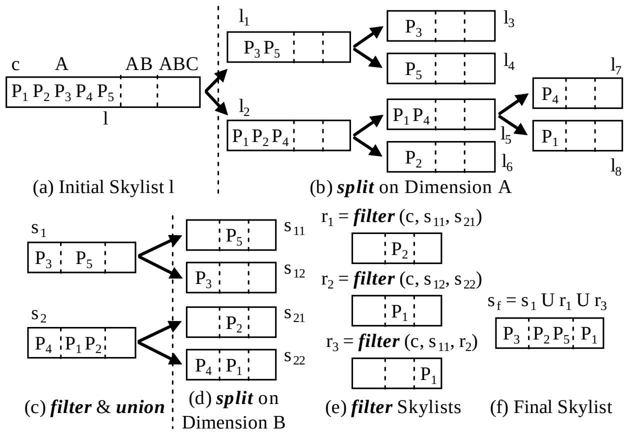


Figure 6: An Example of SDC Algorithm

## 5.3 Top-Down Skyline Algorithm (TDS)

In this subsection, we present our TDS algorithm which employs SDC algorithm to compute SKYCUBE in top-down manner. As analyzed in Section 5.2, SDC can compute skyline queries on a path simultaneously. Thus, in order to compute SKYCUBE using SDC efficiently, we need to find a minimal set of paths such that every node in the cube lattice is covered by one path while the total number of such paths is minimized. In TDS, we apply the path finding algorithm in [19] to find such minimal set of paths. In [19], it is proved that on $d$-dimensional data space the size of the minimal set of such paths is $\binom{d}{\lceil d/2 \rceil}$. Figure 7(a) shows the minimal set of paths for 4-dimensional dataset.

**Algorithm 4** **SDC_Merge** ($c$, $s_1$, $s_2$, $i$)

```
Input:
    $c$: a path, the last dimension set in this path is $\mathcal{V}$
    $s_1$, $s_2$: two skylists
    $i$: the $i$-th dimension of $\mathcal{V}$ is used to split $s_1$ and $s_2$
Output:
    the revised $s_2$ that eliminates the data point in each
    element which is dominated by ones in $s_1$ on the cor-
    responding cuboid in $c$
Description:
 1: if $|s_1| = 1$ or $|s_2| = 1$ or $i = |\mathcal{V}|$ then /* $|\mathcal{V}|$: the
    number of dimensions in $\mathcal{V}$ */
 2:     return filter($c$, $s_1$, $s_2$)
 3: else
 4:     $a_i$ = the $i$-th dimension of $\mathcal{V}$
 5:     $v$ = the median on dimension $a_i$
 6:     $(s_{11}, s_{12})$ = split($s_1$, $v$, $a_i$)
 7:     $(s_{21}, s_{22})$ = split($s_2$, $v$, $a_i$)
 8:     $r_1$ = SDC_Merge($c$, $s_{11}$, $s_{21}$, $i$)
 9:     $r_2$ = SDC_Merge($c$, $s_{12}$, $s_{22}$, $i$)
10:     $r_3$ = SDC_Merge($c$, $s_{11}$, $r_2$, $i+1$)
11:     return union($r_1$, $r_3$)
```

After finding the minimal set of paths, TDS calls SDC to compute cuboids on each of the paths. Note that here only a few SDC calls need to compute the top cuboid of the path from the whole dataset. According to Corollary 1, for the path in which the last dimension set is $\mathcal{U}$, the SDC procedure may compute the cuboid $\mathcal{U}$ from the cuboid $\mathcal{V}$, if $\mathcal{U} \subseteq \mathcal{V}$ and cuboid $\mathcal{V}$ has been computed. This leads to an important saving of computation because (1) the whole dataset might be much larger than the cuboid $\mathcal{V}$; and (2) DC algorithm is more sensitive to the size of the input, as its time complexity is $O(n \log^{d-2} n)$. We call this sharing strategy *sharing parent.*

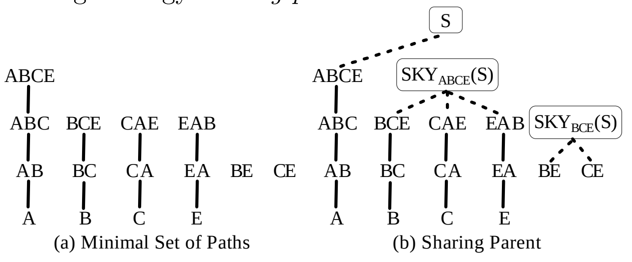


Figure 7: An Example of TDS Algorithm

Now we describe the detailed TDS algorithm. To compute the SKYCUBE, TDS first finds the minimal set of paths that cover all the cuboids. Then SDC algorithm is applied to compute the cuboids for each path. To take advantages of the sharing parent strategy, among the un-computed paths, we always pick from the remaining paths the one whose last dimension set has the highest level value. If there does not exist a suitable ancestor cuboid to start the computation of the current path, SDC will choose to start from the whole dataset. Finally, we retrieve the skylines for each cuboid from the skylist returned by SDC algorithm. Figure 7(b) presents an example. We compute skylines for the paths in the left to right order in the figure. For each path, the dataset from which SDC starts the computation is shown in the rounded rectangle. Take the path $\langle E, EA, EAB\rangle$ for example, SDC computes the skylines from the result of cuboid $ABCE$, $SKY_{ABCE}(S)$.

### 5.4 TDS in the General Case

Now we address some necessary modifications in TDS to support the general case in which the distinct value condition does not hold.

In the general case, there is no containment relation between a parent cuboid and its child cuboid. Therefore, due to the sharing parent strategy, some skyline points may be missed in the result. For example, in Figure 2 if we compute skylines for the path $\langle E, EA, EAB\rangle$ from the skyline of cuboid $ABCE$, which is $\{P_1, P_2, P_3, P_5\}$, we can never get $P_4$. However, $P_4$ is a skyline point of cuboid $E$. Based on Theorem 1, these "missing" data points (e.g., $P_4$) should have same values on some dimensions as some skyline points of the parent cuboid. Now for the skyline queries on a path, SDC computes from the dataset contains not only the parent cuboid but also the above missing data points. To find out the missing data points, for each skyline point $p$ in the parent cuboid on each dimension $a_i$, we collect all the data point $q$ such that $q(a_i) = p(a_i)$. In order to speed up such collection, we presort the whole dataset on each dimension.

Recall that under the distinct value condition, for the skyline queries on a path $c$, SDC returns a skylist. To retrieve skyline for the $i$-th cuboid in $c$, the data points in all the $j$-th ($j \leq i$) elements in the skylist are output. However, according to Theorem 1, in the general case, the data points in the $j$-th ($j < i$) element may not be the skyline points of the $i$-th cuboid in $c$. As a result, we cannot output them directly. In order to determine whether these data points are still the skyline points of the $i$-th cuboid in $c$ efficiently, we take advantages of their filter values, which is the same as the ones in BUS algorithm. That is, the data points in each element are maintained in non-decreasing order of their filter values. Then, the same method of sharing result in the general case of BUS algorithm is adopted to retrieve the skyline result.

## 6 Experimental Evaluation

In this section, we present the comprehensive performance evaluations of our techniques. As mentioned earlier, there is no existing technique specifically designed to support efficient SKYCUBE computation. In our performance study, we implement some skyline algorithms (e.g., BNL, SFS, and DC) to compute each skyline query independently and use them as benchmark algorithms to evaluate our techniques. Below are the algorithms that have been evaluated.

| | |
|---|---|
| **BUS** | Our Bottom-Up Skyline algorithm. |
| **BNLS** | The algorithm computes each skyline query by BNL algorithm [6]. |
| **SFSS** | The algorithm computes each skyline query by SFS algorithm [8]. |
| **TDS** | Our Top-Down Skyline algorithm. |
| **DCS** | The algorithm computes each skyline query by DC algorithm [6]. |

Similar to BUS, BNLS and SFSS compute the SKYCUBE in the bottom-up manner. We also apply the sharing result strategy to both of them to share the child cuboid for computing the parent cuboid. DCS computes the SKYCUBE in the top-down manner. The sharing parent strategy is applied to DCS as well, which enables DCS to compute each cuboid from one of its parent cuboids instead of the whole dataset.

In the following parts, we first study the overall performance of BUS and TDS algorithms, which includes the sensitivity and scalability against the data distribution, dimensionality, and cardinality. Then, we evaluate efficiency of the filter based heuristic to BUS and the sharing parent strategy to TDS, respectively. Finally, the effect of the number of duplicate values per dimension on our techniques is studied.

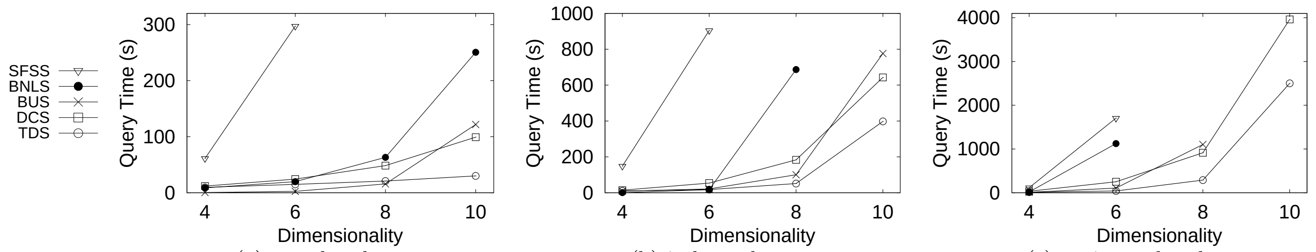
As we do not have the real datasets, we employ the three most popular synthetic benchmark datasets, *correlated*, *independent*, and *anti-correlated* [6], with dimensionality $d$ in the range $[4, 10]$ and cardinality $n$ in the range $[100k, 500k]$ in our experiment study. For each experiment, we evaluate the query times of different algorithms for SKYCUBE computation on these datasets. All the experiments have been carried out on a Pentium 4 PC with a 2.8GHz processor and 1GB main memory.

### 6.1 Effect of Dimensionality

We first study the effect of dimensionality on our techniques. Correlated, independent, and anti-correlated datasets with dimensionality $d$ between 4 to 10 and cardinality $n = 500k$ are used in this experiment. Figure 8 shows the query times of all the algorithms.

From the results, it is clear that although SFSS and BNLS adopt the sharing result strategy, their performances are worse than that of BUS in all datasets. This is because both SFSS and BNLS insert some non-skyline points into the candidate list, which leads to more unnecessary pairwise comparison. Due to $\mathrm{O}(2^{\mathrm{d}})$ presorting overhead, SFSS has the worst performance among all algorithms. We do not plot the results of SFSS and BNLS in high dimensional datasets because their query times are too large. For instance, in anti-correlated dataset with $d = 10$, BNLS takes $265,000$ seconds while TDS consumes $2,500$ seconds only, which outperforms the former by 2 orders of magnitude. Due to their bad performance, we do not evaluate SFSS and BNLS in the following experiments.

In lower dimensional datasets, BUS is faster than DCS and TDS. However, with the growth of skyline size (e.g., in high dimensional dataset or in anti-correlated dataset), more pairwise comparison between data points and skyline points are performed



(a) correlated (b) independent (c) anti-correlated

Figure 8: Effect of Dimensionality ($n = 500k$)

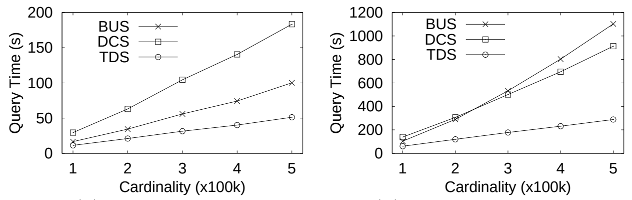
in BUS. Therefore, BUS performs badly in these datasets. As TDS employs SDC algorithm, which can compute cuboids of a path with the same cost as that of DC to compute the last cuboid in the path, TDS outperforms DCS in all datasets. The difference between their query times increases in high dimensional dataset. TDS is always at least 1.5 times faster than DCS in the dataset with dimensionality $d = 10$.

## 6.2 Effect of Cardinality

In order to evaluate the effect of cardinality on our techniques, we use the datasets with the dimensionality $d = 8$ and vary the cardinality $n$ from $100k$ to $500k$. As the skyline in correlated dataset contains few data points, we evaluate the performance of BUS, DCS, and TDS in independent and anti-correlated datasets only. Their query times are shown in Figure 9.

Although both DCS and TDS take advantages of the sharing parent strategy, DCS is slower than BUS in independent datasets while TDS outperforms BUS. It indicates that simply adopting the sharing parent strategy only in DCS can not improve it significantly as DCS computes each skyline independently.

It is shown that TDS is the winner among these algorithms. In the low cardinality datasets ($d \leq 2$), TDS is 2 times faster than others. Furthermore, with the growth of cardinality, TDS query time increases slightly while others increase rapidly. For example, in anti-correlated datasets, the ratio of TDS query time with the dataset $n = 500k$ to that with $n = 100k$ is 4.8, while BUS's and DCS's are 10.7 and 6.6, respectively. Compared with other algorithms, it is seen that TDS not only has better performance but also has better scalability.



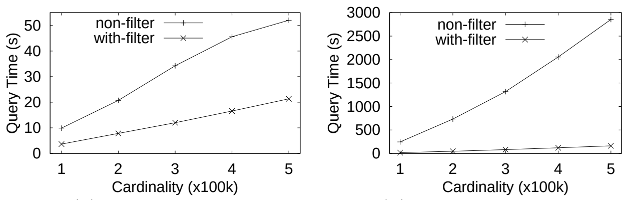
(a) independent (b) anti-coordinated

Figure 9: Effect of Cardinality ($d = 8$)

## 6.3 Efficiency of Filter Based Heuristic

Now we study the efficiency of filter based heuristic to our BUS algorithm. We implement our BUS in two ways. One (*with-filter*) adopts filter based heuristic, the other (*non-filter*) does not, which evaluates data points in the same way as that of BNL. We evaluate them in the datasets with dimensionality $d = 6$ and vary cardinality $n$ from $100k$ to $500k$. Figure 10 shows the query time in each dataset.

From the results, it is seen that with-filter outperforms non-filter in all datasets. As analyzed in section 4.2, filter based heuristic can reduce the pairwise comparison between data points and the skyline points, which is the most expensive computation in BUS. Thus, filter based heuristic improves BUS. In independent datasets with-filter is 40% faster than non-filter. With the growth of cardinality, the query time of with-filter remains fairly consistent while that of non-filter increases significantly. In anti-correlated dataset with cardinality $n = 500k$, non-filter is 27 times slower than with-filter.



(a) independent (b) anti-correlated

Figure 10: Efficiency of Filter Based Heuristic ($n = 6$)

## 6.4 Efficiency of Sharing Parent Strategy

In this subsection, we evaluate the efficiency of sharing parent strategy to TDS algorithm. For two implementations (*sharing-parent* and *non-sharing-parent*) of TDS, we study their performance in correlated, independent, and anti-correlated datasets with dimensionality $d = 6$ and cardinality $n = 200k, 400k$. Figure 11 reports the query times on these datasets. The number above the bars inside of the figure is the ratio of non-sharing-parent query time to that of sharing-parent.

It is clear that sharing parent strategy improves TDS in all datasets. Sharing parent strategy enable TDS to compute the cuboids of one path from its parent cuboid while non-sharing-parent does this from the whole dataset. Generally, the parent cuboid is much smaller than the whole dataset. Furthermore by the query time ratios, it is seen that sharing-parent becomes more efficient in the larger dataset.

## 6.5 Effect of Duplicate Values

In the final experiment, we study the effect of number of duplicate values per dimension on our techniques. We define the duplicate ratio $\alpha$ as on each

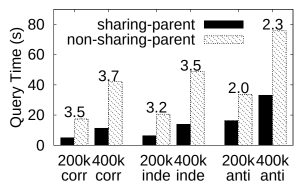
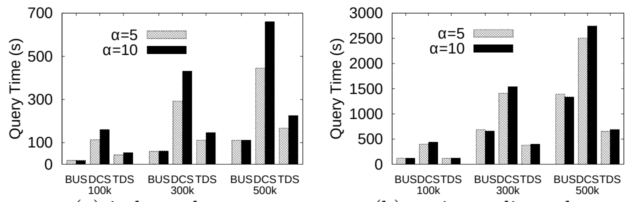


Figure 11: Efficiency of Sharing Parent ($d = 6$)

dimension $a_i$ for every data point $p$ there are $\alpha - 1$ number of other data points with the same coordinate as $p$'s. In this experiment, independent and anti-correlated datasets with dimensionality $d = 8$, cardinality $n = 100k, 300k, 500k$, and duplicate ratio $\alpha = 5, 10$ are generated. Note that compared with the datasets used in the above experiments, these datasets contain more duplicate values on each dimension. Figure 12 shows the query times in these datasets.

In independent dataset, all algorithms take more query time in higher duplicate ratio datasets, because the size of each cuboid is larger. Among these algorithms, the effect of duplicate values on BUS is the least. Although larger size of cuboid in higher duplicate ratio dataset causes more pairwise comparison in BUS, it also makes sharing result strategy, which is adopted in BUS, more efficient. As a result, BUS has similar performance in these two datasets with different duplicate ratios. On the contrary, in higher duplicate ratio dataset, larger size of cuboid decreases the efficiency of sharing parent strategy because the difference between the size of whole dataset and that of parent cuboid is smaller. Therefore, DCS, which employs sharing parent strategy only, is affected by duplicate values most.

In anti-correlated datasets, it is interesting that BUS is a little bit faster in higher duplicate ratio dataset than that in lower one. It confirms that sharing result strategy is more efficient when each cuboid's size is larger. However, with the growth of the dataset, the number of pairwise comparison between data points and skyline points in BUS increases rapidly, which slows BUS down. Similar to the trends in the above experiments (Figure 8, 9), TDS performs best among these algorithms. In the datasets with cardinality $n = 500k$ and duplicate ratio $\alpha = 10$, TDS takes 690 seconds which is 4 times faster than DCS. Moreover, it is clear that the difference between TDS query times on two duplicate ratio databases is smaller than that of DCS.



(a) independent (b) anti-coordinated

Figure 12: Effect of Duplicate Values ($d = 8$)

## 7 Conclusions

In this paper, we consider the application where multiple related skyline queries need to be frequently processed. We propose to adapt the data cube concept into the skyline computation problem and propose the concept of the SKYCUBE. Intuitively, a SKYCUBE is the union of skyline results of all non-empty combination of $d$ dimensions on a dataset. We focus on the issue of efficient computation of the SKYCUBE and propose two novel algorithms: the bottom-up algorithm (BUS) and the top-down algorithm (TDS). Their efficiencies mainly come from a set of computation sharing strategies we identified for multiple skyline computation. We demonstrated the effectiveness of the computation sharing strategies and the superior performance of the proposed algorithms in our extensive experimental evaluation under various settings.

As one of our future work, we plan to explore the issue of disk-based SKYCUBE computation for very large datasets. We are looking at techniques to convert in-memory skyline computation algorithms to be disk-based [6]. Another possible direction for future work is to investigate the materialized view selection problem for the SKYCUBE under various constraints.

## References

[1] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *VLDB 1996*.

[2] W.-T. Balke and U. Güntzer. Multi-objective query processing for database systems. In *VLDB 2004*.

[3] W.-T. Balke, U. Güntzer, and J. X. Zheng. Efficient distributed skylining for web information systems. In *EDBT 2004*.

[4] 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. *JACM*, 25(4):536–543, 1978.

[5] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *SIGMOD 1999*.

[6] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE 2001*.

[7] C. Buchta. On the average number of maxima in a set of vectors. *Information Processing Letters*, 33(2):63–65, 1989.

[8] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *ICDE 2003*.

[9] Y. Feng, D. Agrawal, A. E. Abbadi, and A. Metwally. Range cube: Efficient cube computation by exploiting data correlation. In *ICDE 2004*.

[10] P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In *VLDB 2005*.

[11] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *ICDE 1996*.

[12] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A system for the efficient execution of multi-parametric ranked queries. In *SIGMOD 2001*.

[13] W. Jin, J. Han, and M. Ester. Mining thick skylines over large databases. In *PKDD 2004*.

[14] D. Kossmann, F. Ramsak, and S. Rost. Shooting starts in the sky: An online algorithm for skyline queries. In *VLDB 2002*.

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

[16] X. Lin, Y. Yuan, W. Wang, and H. Lu. Stabbing the sky: Efficient skyline computation over sliding windows. In *ICDE 2005*.

[17] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD 2003*.

[18] J. Pei, W. Jin, M. Ester, and Y. Tao. Catching the best views of skyline: A semantic approach based on decisive subspaces. In *VLDB 2005*.

[19] K. A. Ross and D. Srivastava. Fast computation of sparse datacubes. In *VLDB 1997*.

[20] S. Sarawagi, R. Agrawal, and A. Gupta. On computing the data cube. Technical report, IBM Almaden Research Center, 1996.

[21] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *VLDB 2001*.

[22] D. Xin, J. Han, X. Li, and B. W. Wah. Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. In *VLDB 2003*.

[23] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *SIGMOD 1997*.