DeltaSky: Optimal Maintenance of Skyline Deletions without Exclusive Dominance Region Generation *

Ping Wu, Divyakant Agrawal, Ömer Eğecioğlu, Amr El Abbadi
Department of Computer Science
University of California at Santa Barbara
{pingwu, agrawal, omer, amr}@cs.ucsb.edu

Abstract

*This paper addresses the problem of efficient maintenance of a materialized skyline view in response to skyline removals. While there has been significant progress on skyline query computation, an equally important but largely unanswered issue is on the incremental maintenance for skyline deletions. Previous work suggested the use of the so called exclusive dominance region (*EDR*) to achieve optimal I/O performance for deletion maintenance. However, the shape of an* EDR *becomes extremely complex in higher dimensions, and algorithms for its computation have not been developed. We derive a systematic way to decompose a d-dimensional* EDR *into a collection of hyper-rectangles. We show that the number of such hyper-rectangles is $O(m^d)$, where m is the current skyline result size. We then propose a novel algorithm `DeltaSky` which determines whether an intermediate R-tree MBR intersects with the* EDR *without explicitly calculating the* EDR *itself. This reduces the worse case complexity of the* EDR *intersection check from $O(m^d)$ to $O(md)$. Thus `DeltaSky` helps the branch and bound skyline algorithm achieve I/O optimality for deletion maintenance by finding only the newly appeared skyline points after the deletion. We discuss implementation issues and show that `DeltaSky` can be efficiently implemented using one extra B-Tree. Moreover, we propose two optimization techniques which further reduce the average cost in practice. Extensive experiments demonstrate that `DeltaSky` achieves orders of magnitude performance gain over alternative solutions.*

1 Introduction

Recently there has been significant amount of interest in supporting skyline queries in database systems. Skyline queries provide a viable database solution for applications where users do not have a clear query formulation but want to optimize multiple attributes of interest. It distinguishes itself as an effective alternative to the exact match query paradigm. For instance, a user who wants to find all the "interesting" deals in a used car database may be interested in both low price and low mileage. Skyline query returns all the objects that are not dominated by any other objects in the database. An object *A dominates B*, if *A* is no worse than *B* on all attributes, and strictly better on at least one attribute. The answers to the skyline query are exactly those points that can be the best according to some monotonic ranking functions.

As the skyline query is gaining popularity, we believe that it will appear more frequently in common database workloads. In general, a skyline query is expensive to evaluate especially for large and complex datasets. Materializing a precomputed query result set as views can help reduce the query processing time significantly and has been used in different contexts for many years [15]. We believe that this technique remains promising for skyline queries as well.

The question that naturally arises then is how to design efficient algorithms to maintain the skyline results in the presence of database updates. Recomputing the entire skyline query from scratch each time a skyline update occurs is obviously not an efficient solution. In particular, the entire recomputation itself may be too expensive when the dataset is large and complex; the underlying dataset may be physically hosted at a remote site so that shipping the entire new result set for each single skyline update can be slow; the workload may be update-intensive where the result set may undergo slight changes over time.

Although the research community has made remarkable progress in accelerating the skyline computation in different contexts, efficiently maintaining a materialized skyline result set is a relatively unexplored territory. In this paper, we focus on skyline result maintenance due to deletion. The reason is two-fold. First, maintaining insertions is much easier and can be handled by existing techniques described in [22]. More importantly, we believe that skyline points are more likely to be removed from the database due to the fact that they correspond to exactly those "interesting" objects which may potentially attract a large group of users, and hence may easily be "consumed". In the case of a skyline deletion, the base table needs to be accessed since some points that are not in the

*This work is supported in part by IBM Ph.D. Fellowship, NSF grants IIS 02-23022 and CF 04-23336.

current skyline view may be "promoted" to the view when the original skyline points that dominate them get deleted. Borrowing the terminology from the traditional view maintenance literature, a skyline view is self-maintainable for insertion but not for deletion. In fact, a skyline query can be seen as the generalized form of the MAX query over a partially-ordered database (dominance relationship) and MAX query has been known to be not self-maintainable with respect to deletion [5] [13].

In previous view maintenance literature, auxiliary data has often been used to make views runtime self maintainable [25]. The papers [20] and [30] suggest the use of *work areas* to achieve runtime self maintenance for MIN, MAX and top-k queries. However, unlike those queries that operate on a totally-ordered dataset, a skyline query operates on a partially-ordered database. Therefore, to make a skyline view run-time self maintainable, the size of the work areas can become much larger because all the "top-tier" points in the partial order graph have to be materialized and maintained as well. This paper does not explore further along this direction. Instead, we target at a more fundamental issue of optimizing the "refill" query performance, *i.e.* to quickly find only the new skyline points after a skyline deletion occurs. The problem of making a skyline view runtime self maintainable is complementary to this paper and is our on-going work.

In [22], Papadias *et al.* suggest that optimal deletion maintenance can be achieved by issuing constrained skyline queries over the so called "Exclusive Dominance Region" (*EDR*), which is the part of the data space only dominated by the deleted skyline point. However, while a 2-d *EDR* is always one single rectangle, it becomes quite complex in high dimensional cases, and computing the d-dimensional *EDR* for a given deleted skyline point has remained an *unsolved* problem. Our first contribution in this paper is to derive a systematic way of representing a d-dimensional *EDR* as the union of a collection of regular hyper-rectangles. This algorithm runs in time $O(m^d)$ to compute the *EDR*, where m represents the cardinality of the current skyline result set. While incorporating the *EDR* representation into the *BBS* algorithm [21] can guarantee optimal I/O performance, we find it too expensive in practice because: 1) it is too expensive to compute *EDR* when the skyline number m is large; 2) in general the number of generated hyper-rectangles is big so that the intersection check with *EDR* also becomes slow. Our second contribution is a novel $O(md)$ algorithm `DeltaSky` for the *EDR* intersection check which works *without* explicitly calculating the representation of *EDR* but nevertheless achieves the same I/O optimality.

We discuss the implementation issues for `DeltaSky` and show that it has certain nice locality properties allowing further optimization. Our third contribution is two optimization techniques to exploit such inherent locality, thereby significantly reducing its computation cost in practice. We conduct extensive experiments which show that `DeltaSky` achieves several orders of magnitude performance gain over alternative solutions.

The rest of the paper is organized as follows. Section 2 presents a review of related work. We discuss preliminary notions in Section 3. In Section 4, we derive the decomposition of the exclusive domain regions. We present `DeltaSky` algorithm and its optimization variants in Section 5. We show the experimental results in Section 6 and Section 7 concludes the paper.

2 Related Work

In the context of database systems, previous work can be divided into two groups based on the requirement for data preprocessing: non-index-based (e.g. *Block-Nested Loops*, *Divide and Conquer* [6], *Sort Filter Skyline* [10] and `LESS` [11]) and index-based methods (e.g. *Bitmap*, *Index* [26], *NN* [17], and *BBS* [21]). As the name indicates, non-index-based methods do not require any offline preprocessing of the data set, and thus are quite "generic" in the sense that they can easily be pipelined with other relational operators (e.g. join). On the other hand, index-based methods are usually more efficient, among which *BBS* is considered the state-of-the-art and we shall review *BBS* algorithm in detail later.

Skyline operator is applied beyond its original context. For example, Chan *et al.*extend skyline to databases with partially-ordered categorical attribute domains [7]. [31] and [23] study the notion of skyline views and subspace skyline computation methods. Tao *et al.* further propose `SUBSKY` which answers skyline queries in arbitrary subsets of the attributes [27]. [16] investigate the support of skyline queries in MANETs. [3] studied the efficient computation of skyline queries over distributed Web sources. [9] provides several skyline cost estimators to support its plan enumeration and optimization. [18] and [19] consider continuous skyline queries over data streams. [8] proposes approximate algorithms to solve the "too many" answer problem in high dimensional skylines.

Papadias *et al.* first introduce the notion of exclusive dominance regions (*EDR*) for optimal skyline deletion maintenance [22]. This idea is not fully developed in [22] and it is still unknown how to compute an *EDR* beyond the 2-dimensional cases. Consequently, it is not even clear whether fully computing a *d*-dimensional *EDR* is a viable solution in the first place. As we discover in this paper: these problems are technically non-trivial and *EDR* computation is not necessary to achieve I/O optimality for deletion maintenance. Xia *et al.*[29] present efficient update algorithms for compressed skyline Cubes.

Materialized view maintenance is a well-understood problem in data warehousing and there exists a large body of work on this topic. Interested readers are referred to [14] for a survey. Closely relating to the maintenance of skyline results, previous work includes the maintenance of aggregation views for SQL MAX, MIN queries [12] [2] [20] and the top-k query [30]. No previous result is available on the view

maintenance problem for MAX query on partially-ordered datasets in general and skyline queries in particular.

3 Preliminaries

3.1 Skyline Queries

For a given d-dimensional data set DB, a global skyline query computes a subset S consisting of objects that are not "dominated" by any other object inside DB. We say object $O_i(O_i^1, O_i^2, ..., O_i^d)$ dominates $O_j(O_j^1, O_j^2, ..., O_j^d)$ if O_i is no worse than O_j on any dimension and is better on at least one dimension. Without loss of generality, we assume that users prefer the minimum value on all dimensions. Fig. 1(a) illustrates this on a 2-d used car data set with x-axis being the mileage and y-axis representing the price. The global skyline consists of 3 records a,b,c.

The generalized form of the global skyline query is the so called *constrained skyline query* [21] where users are interested in finding the skyline points among a subset of DB satisfying multiple hard constraints. For example, a user may only be interested in the "interesting" records within the price range from 7K to 11K dollars and with mileage between 40K and 90K miles (the constrained region depicted by the box $aehf$). Fig. 1(a) shows that there are exactly two skyline points d_1, d_2 in this constrained region.

3.2 Branch and Bound Skyline(BBS) Algorithm

Algorithm 1 BBS

```
1:  R: R-Tree index on the database
2:  Q: constrained query region
3:  BBS(R, Q)
4:  Procedure
5:  initialize the skyline result set S to empty;
6:  insert all entries in the root node of R into heap H;
7:  while H is not empty do
8:     pop top entry e
9:     if e is not dominated by any point in S then
10:       if e is an intermediate node then
11:          for every child entry e_i in e do
12:             if e_i intersects with Q then
13:                if e_i is not dominated by any point in S then
14:                   insert e_i into H
15:                end if
16:             end if
17:          end for
18:       else
19:          insert e_i into S
20:       end if
21:    end if
22: end while
23: End Procedure
```

Since our proposed algorithm `DeltaSky` operates within the framework of *BBS* [21], we review the *BBS* algorithm in this section. Algorithm 1 describes the general form of the *BBS* algorithm which progressively outputs all skyline points in a constrained query region Q (It behaves the same as the original *BBS* algorithm if Q is set to be the entire data space).

BBS assumes the presence of an R-tree index over the target dataset. It maintains an in-memory priority queue when traversing down the R-Tree. A R-Tree node is expanded only when: 1) it intersects with the query region (`line 12`); 2) it is not dominated by current skyline results (`line 13`). Since its priority queue is ordered according to each node's *MinDist*, *BBS* has the nice property that each computed data point is guaranteed to be in the final skyline and it only accesses the R-tree nodes that may contain skyline points, achieving optimal I/O performance.

Note that our proposed algorithm leverages the framework of *BBS* to find the skyline points within the exclusive domain region *EDR* (the definition of *EDR* to be given in Section 3) of a deleted skyline point.

3.3 Problem Description

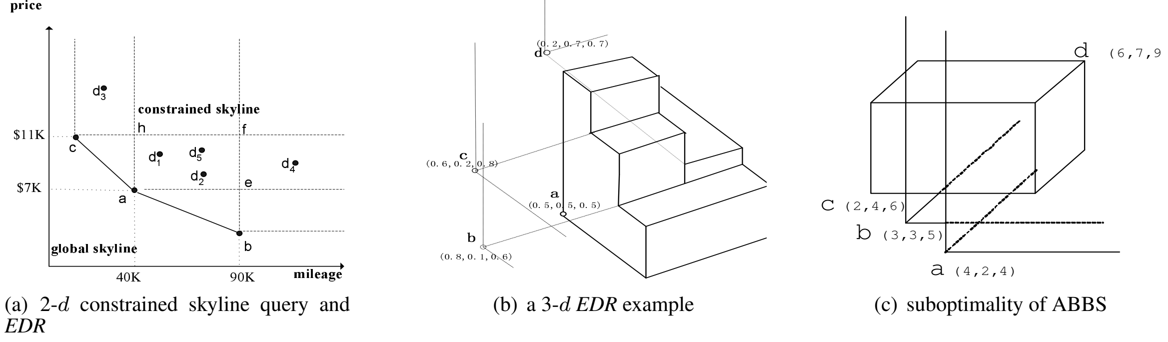
Materializing pre-computed view to reduce the response time of complex queries is a common practice in data warehousing. We believe that skyline queries can benefit from materialization so that the system responds to the user without rerunning the query from scratch once again. Skyline views can be easily added by further extending the SQL CREATE VIEW statement [24].

The main challenge now is to develop efficient mechanisms to maintain the view up-to-date as the source data is modified. Assume that we have a skyline view S on database DB and the operations on DB consist of insertions and deletions. As discussed in [22], insertions can be maintained based on the information from S itself. On the other hand, deletion maintenance is much more complicated. In Fig. 1(a), both d_1 and d_2 become the skyline after point a is deleted. To achieve optimal I/O performance in deletion maintenance, we want to only access the R-Tree nodes which may contain these new skyline points due to the deletion. The central issue to this optimality lies in the exclusive dominance region of the deleted skyline point, which we defined next.

For a given skyline point $S_i \in S$, we define its dominance region $DR(S_i)$ as the whole data space that is dominated by S_i and its exclusive dominance region $EDR(S_i)$ is the data space that is only dominated by S_i. For example, in Fig. 1(a), the exclusive dominance region of skyline point a is depicted by rectangle $ahfe$. Intuitively, $EDR(S_i)$ defines the smallest data space that may contain the new skyline points after deletion of S_i. And the constrained skyline results within $EDR(S_i)$ are exactly those points that must be added to the new skyline set after S_i is deleted, because those points are *exclusively* dominated by S_i. We establish the following lemma based on this intuition.

Lemma 1. **(Minimality of *EDR*)** *Let S denote the original skyline, $S'(S_i)$ denote the new skyline view after the deletion of skyline point S_i ($S_i \in S$). Let $\triangle S$ denotes the skyline points in the constrained region $EDR(S_i)$, then $S'(S_i) - S = \Delta S$. There does not exist a subregion $EDR'(S_i) \subset EDR(S_i)$, s.t., the constrained skyline points in $EDR'(S_i)$ equals to $S'(S_i) - S$, for all possible datasets.* $\Box$

For example, in Fig. 1(a), d_1 and d_2 are the skyline points in constrained region $EDR(a)$ and they are "promoted" to the skyline view once a is removed. Consequently, we can run *BBS* with *EDR* as its constrained region to find the exact $\triangle S$ with optimal I/O performance. The key issue now is computing the *EDR* of a given skyline point S_i. While 2-d *EDR* is always a single rectangle, its shape becomes extremely complex in higher dimensional cases. Fig.1(b) shows a "toy" 3-d *EDR* example. There are four skyline points a, b, c, d, and assume a is to be deleted. After we "chop" away from $DR(a)$ all the overlapping pieces with other dominance regions, we have a non-regular shape which constitutes the $EDR(a)$. As we can see, $EDR(a)$ is composed of a number of non-regular rectangles. Note that in



(a) 2-d constrained skyline query and *EDR*

(b) a 3-d *EDR* example

(c) suboptimality of ABBS

Figure 1. Exclusive Dominance Region Examples

practice, the number of skyline points is usually much bigger and the dimensionality greater, therefore the corresponding *EDRs* becomes much more complicated.

3.4 Simple Solutions

We start with the discussion of two simple approaches. The most straightforward method for skyline deletion maintenance is to recompute from scratch by *BBS*. Clearly, this approach is overly simplistic because a considerable portion of the data space is not affected by the deleted point at all. This *BBS*-based recomputation can be easily optimized for the purpose of deletion maintenance. For each MBR e_i extracted from the heap, the optimized version does two checks. First, it checks if this MBR intersects with the dominance region of the deleted skyline point S_i. Second, it checks whether e_i has already been dominated by existing skyline points other than S_i. It only accesses the corresponding disk page when both checks satisfied. Since this BBS extension adapts to the location of the deleted point, we call this method $ABBS$(Adaptive Branch-and-Bound-Search). The first intersection check of $ABBS$ takes constant time. For the second check, an extra R-tree can be used to index the points in the current skyline set, then a range query is formed by the lower right corner point of e_i and the origin point. If there is any skyline point that falls inside this range, $ABBS$ prunes this node since it has already been dominated by some current skyline point.

While $ABBS$ effectively avoids visiting a large number of R-tree nodes that are otherwise accessed by recomputation, it still involves unnecessary I/Os because some MBR that satisfies both checks may not intersect with the $EDR(S_i)$. We give a simple $3d$ example in Fig. 1(c) with only two skyline points a and b. Assume that point a is the point to be deleted, the MBR cd will be visited by $ABBS$ but does not intersect with the $EDR(a)$. Such "false positives" tend to happen more frequently in higher dimensional large data set. As a result, $ABBS$ considerably (and consistently) underperforms the optimal solution both in terms of I/O and response time.

4 Understanding d-dimensional *EDR*s

In this section, we address the problem of representing a high dimensional *EDR* with a collection of hyper-rectangles. We first describe the basic idea and then provide formal derivations of the *EDR* representation by transforming its corresponding Boolean function. Finally, we describe and analyze decomposition algorithms.

4.1 Boolean Functions

A basic Boolean function f_v is defined as: $f_v : [0,1] \rightarrow \{0,1\}; f_v(x) = 1, \text{if } x \geq v; 0, \text{if } x < v;$. We use $\overline{f_v}$ to denote the function whose values are the Boolean negation of the values of f_v.

Starting with the basic Boolean function, we can construct more complex ones to represent more useful concepts. For instance, let us consider a d-dimensional query box R_{ab} where $a = (a^1, a^2, \cdots, a^d)$ and $b = (b^1, b^2, \cdots, b^d)$ are the lower-left and upper-right corner points, respectively. We can write the following function to test whether a given point $o = (o^1, o^2, \cdots, o^d)$ is in R_{ab}.

$$F_{R_{ab}}(o) = f_{a^1}(o^1) f_{a^2}(o^2) \cdots f_{a^d}(o^d)$$
$$\overline{f_{b^1}}(o^1)\overline{f_{b^2}}(o^2) \cdots \overline{f_{b^d}}(o^d) = \prod_{i=1}^{d} \left(f_{a^i}(o^i) \overline{f_{b^i}}(o^i) \right) \quad (4.1)$$

This function ensures that a point o is in R_{ab} by testing that in any dimension d_i, value o^i belongs to the interval [a^i,b^i). The following lemma gives a useful property of the basic Boolean function.

Lemma 2. *Given n numbers $v^1, v^2, \cdots v^n$, among which v_{max} is the maximum and v_{min} is the minimum, then $f_{v^1} f_{v^2} \cdots f_{v^n} \equiv f_{v_{\max}}$ and $\overline{f_{v^1} f_{v^2} \cdots f_{v^n}} \equiv \overline{f_{v_{\min}}}$.* □

4.2 Representing EDRs with Hyper-rectangles

Assume the current skyline set S consists of m points, *i.e.* $S = \{S_1, S_2, \cdots S_m\}$. Each skyline point S_h is a d-dimensional point and is written as $S_h = (S_h^1, S_h^2, \cdots S_h^d)$. We use the following Boolean function $F_{EDR(S_i)}$ to represent the exclusive dominance region $EDR(S_i)$ of a deleted skyline point S_i.

$$\begin{aligned} F_{EDR(S_i)}(o) &= f_{s_i^1}(o^1) f_{s_i^2}(o^2) \cdots f_{s_i^d}(o^d) \\ &\quad (1 - f_{s_1^1}(o^1) f_{s_1^2}(o^2) \cdots f_{s_1^d}(o^d)) \\ &\quad \cdots\cdots\cdots\cdots\cdots\cdots \\ &\quad (1 - f_{s_m^1}(o^1) f_{s_m^2}(o^2) \cdots f_{s_m^d}(o^d)) \\ &= \prod_{l=1}^{d} f_{s_i^l}(o^l) \prod_{h=1, h \neq i}^{m} \left(1 - \prod_{j=1}^{d} f_{s_h^j}(o^j) \right) \end{aligned} \quad (4.2)$$

The first term $\prod_{l=1}^{d} f_{s_i^l}(o^l)$ enforces that object o belongs to S_i's dominance region and the remaining terms require

that o does not belong to the dominance regions of any skyline points other than S_i (*exclusive terms*). In each exclusive term, we notice that 1 (always true) can be written as the conjunction of all possible conditions, therefore we rewrite the exclusive terms as follows:

$$\left(1 - f_{s_k^1}(o^1) f_{s_k^2}(o^2) \cdots f_{s_k^d}(o^d)\right) = \\ = \sum_{\exists j, M_k^j = \overline{f_{s_k^j}}(o^j)} \prod_{j=1}^{d} M_k^j, \qquad M_k^j = f_{s_k^j}(o^j) \text{ or } \overline{f_{s_k^j}}(o^j) \tag{4.3}$$

For clarity of the derivation, in Equation 4.3 we introduce M_k^j to uniformly refer to $\overline{f_{s_k^j}}(o^j)$ and $f_{s_k^j}(o^j)$. And we say that M_k^j is "positive" in places where it takes value $f_{s_k^j}(o^j)$ or "negative" where it takes value $\overline{f_{s_k^j}}(o^j)$. So combining Equation 4.2 and 4.3, we get:

$$F_{EDR(S_i)}(o) = \prod_{l=1}^{d} f_{s_i^l}(o^l) \prod_{h=1, h \neq i}^{m} \left(\sum_{\exists j, M_h^j = \overline{f_{s_h^j}}(o^j)} \prod_{j=1}^{d} M_h^j \right) \tag{4.4}$$

Recall that our goal is to represent a high dimensional *EDR* as the union of a number of hyper-rectangles. Therefore, our target representation of $F_{EDR(S_i)}(o)$ must be of the following form: $\sum \prod \left(f_{v_1}(x) \overline{f_{v_2}}(x)\right)$ (recall Equation 4.1). To this end, we further transform Equation 4.4:

$$F_{EDR(S_i)}(o) = \\ \sum_{\forall h, \exists j, M_h^j = \overline{f_{s_h^j}}(o^j)} \left(\prod_{l=1}^{d} f_{s_i^l}(o^l) \right) \left(\prod_{j=1}^{d} \left(\prod_{h=1, h \neq i}^{m} M_h^j \right) \right) \\ = \sum_{M_i^j = f_{s_i^j}(o^j) \text{ and } \forall h \neq i, \exists j, M_h^j = \overline{f_{s_h^j}}(o^j)} \left(\prod_{j=1}^{d} \prod_{h=1}^{m} M_h^j \right) \tag{4.5}$$

For each term in Equation 4.5, on any given dimension $d_j (1 \leq j \leq d)$, we have m Boolean functions: $M_h^j (1 \leq h \leq m)$, where M_h^j corresponds to the skyline point S_h's projection S_h^j on dimension d_j. For each given term $\prod_{j=1}^{d} \prod_{h=1}^{m} M_h^j$ in Equation 4.5, let S_+^j denotes a set containing all the S_h's whose corresponding M_h^j's are "positive" and S_-^j denotes the set containing all the S_h's whose M_h^j's are negative. Let S_{+max}^j be the maximum projection value from S_+^j and S_{-min}^j be the minimum projection value from S_-^j. According to Lemma 2, we have $\prod_{h=1}^{m} M_h^j = f_{s_{+\max}^j}(o^j) \overline{f_{s_{-\min}^j}}(o^j)$ and therefore we finally get:

$$F_{EDR(S_i)}(o) = \\ \sum_{\bigcup_{j=1}^{d} S_-^j = S - \{S_i\}} \left(\prod_{j=1}^{d} \left(f_{s_{+\max}^j}(o^j) \overline{f_{s_{-\min}^j}}(o^j) \right) \right) \tag{4.6}$$

The above equation defines a set of disjoint hyper-rectangles whose union is equivalent to the d-dimensional $EDR(S_i)$. At this point, there are two points worth noting. First, for each skyline point $S_h (h \neq i)$, there should be at least one negative $M_h^j (1 \leq j \leq d)$. This is equivalent to saying that the union of all negative terms should contain all S_h's except for S_i. This is captured by $\bigcup_{j=1}^{d} S_-^j = S - \{S_i\}$ in Equation 4.6. Second, the number of rectangles in Equation 4.6 is exponential, or $O(2^{md})$. It is easy to see that in Equation 4.3, each "exclusive term" has been transformed into the summation of $2^d - 1$ terms. Because there are $m - 1$ such exclusive terms in Equation 4.2, the total number of hyper-rectangles appearing in Equation 4.6 is easily calculated as $(2^d - 1)^{m-1}$.

4.3 Reducing from $O(2^{md})$ to $O(m^d)$

The bound $O(2^{md})$ is overly generous because not every term in Equation 4.6 corresponds to a valid hyper-rectangle. In particular, we observe that every valid rectangle must satisfy certain properties as we show below.

Lemma 3. *Any valid rectangle in Equation 4.6,* $S_i^j \leq S_{+max}^j < S_{-min}^j, \quad (1 \leq j \leq d)$.

Proof: Please refer to [24] for all proofs hereafter.

Lemma 3 implies that the interval $\left[S_{+max}^j, S_{min}^j\right]$ of a valid rectangle should be "tight" in the sense that there cannot be any projection value within this interval because any skyline projection should belong to either S_+^j or S_-^j.

Theorem 1. *Let* $S^j = \{S_1^j, S_2^j, \cdots, S_m^j\}$ *be the projections of all skyline points on dimension* d_j. *Let* $\tilde{S}^j = \{\tilde{S}_1^j, \tilde{S}_2^j, \cdots, \tilde{S}_m^j\}$ *be a non-descending permutation of* S^j, *i.e.* $\tilde{S}_1^j \leq \tilde{S}_2^j \leq \cdots \leq \tilde{S}_m^j$. *Let* $\tilde{S}_{\tilde{i}}^j = S_i^j$, *where* S_i *is the point to be deleted. For any rectangle* $R_{ab} = \prod_{j=1}^{d} \left(f_{a^j}(o^j) \overline{f_{b^j}}(o^j)\right)$ *in Equation 4.6,* R_{ab} *represents a valid rectangle, i.f.f : 1)* $\forall j, S_i^j \leq a^j < b^j$;
2) $\forall j, \exists k (\tilde{i} \leq k \leq m - 1), s.t. \left[a^j, b^j\right] = \left[\tilde{S}_k^j, \tilde{S}_{k+1}^j\right]$; *or* $\left[a^j, b^j\right] = \left[\tilde{S}_m^j, +\infty\right]$;
Furthermore, $R_{ab} \subset EDR(S_i)$ *i.f.f.*
3) it must satisfy the original requirement in Equation 4.6: $\bigcup_{j=1}^{d} S_-^j = S - \{S_i\}$, *where* S_-^j *refers to the sublist of* $\tilde{S}_j$ *greater than* a^j. □

We explain Theorem 1 with Fig. 2. Fig. 2 shows d sorted lists $\tilde{S}^1, \tilde{S}^2, \cdots, \tilde{S}^d$. The jth list $\tilde{S}^j$ contains the projection values of all the skyline points on dimension d_j in ascending order. Assume that S_i is the skyline point to be deleted. Its projection values together define a cut (the dark dotted line in Fig 2). Condition (1) requires that any valid interval $\left[a^j, b^j\right]$ on dimension d_j must be non-empty, *i.e.* a^j should be strictly less than b^j, and always be "under" the S_i^j, *i.e.* $a^j \geq S_i^j$. For example, in $\tilde{S}^1$, $\left[a^1, b^1\right]$ is below S_i^1. On the other hand, in

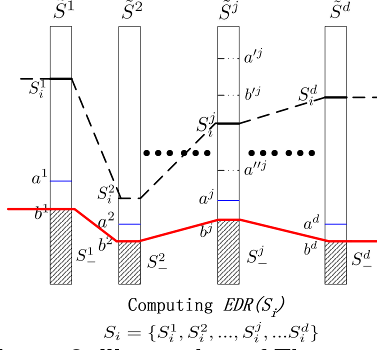


Figure 2. Illustration of Theorem 1

$\tilde{S}^j$, $\left[a'^j, b'^j\right]$ is not a valid interval, because it lies "above" the S_i^j. In addition, Condition (2) requires that any valid interval from Equation 4.6 $\left[a^j, b^j\right]$ should be consecutive in $\tilde{S}^j$. For example, in $\tilde{S}^j$, interval $\left[a''^j, b^j\right]$ is not valid, since a^j falls in between.

However, not all the cross products of these valid intervals belong to $EDR(S_i)$. Condition (3) further requires that the union of all the sublist below a^j should contain all skyline points except S_i. In Fig. 2, all intervals of hyper-rectangle R_{ab} forms a "cut" (denoted by the red solid line), $\bigcup_{j=1}^{d} S_-^j$ corresponds to the union of all the sublists "below" the cut. Intuitvely, condition (3) enforces that any skyline point other than S_i should appear below this "cut" at least once.

According to Condition (2), there are at most m valid intervals on each dimension d_j because a^j and b^j must be adjacent values in a non-descending permutation of the projection values. Since there are d dimensions in total, the number of valid hyper-rectangles in Equation 4.6 is bounded by m^d.

4.4 Algorithms for Query Box Enumeration

In [24], we discuss the implementation of different *EDR* algorithms and possible optimizations. We also implemented all the *EDR* computation algorithms discussed above. As we will show next, this enumeration process can be avoided and the same intersection check can be performed without explicitly computing an *EDR*. Therefore, we do not pursue the direction of tightening the *EDR* computation bounds any longer. Nevertheless, our main result in Theorem 1 provides a solid foundation for all later algorithms.

5 DeltaSky: Deletion Maintenance without EDR computation

5.1 Disadvantages of the "Generate-and-Compute" Approach

The deletion maintenance method developed so far can be summarized by two steps. In the first step, we calculate $EDR(S_i)$ by representing it with a collection of query boxes (*Generate* step). In step two, we input these boxes as the constrained region Q to *BBS* and *BBS* runs in the same way as before except that the intersection check returns true if an R-Tree region intersects with any of these generated boxes (*Compute* step). Due to the minimality property of $EDR(S_i)$ (Lemma 1) and the I/O optimality of BBS [21], this "Generate-and-Compute" approach guarantees I/O optimality for deletion maintenance and outputs only the new skyline points. However, it suffers from at least two types of overhead. First, as we have shown in Section 4.3, the *Generate* part takes $O(m^d)$, where the number of skyline points m is usually large in any real world databases. Second, each intersection check between an intermediate R-Tree entry with $EDR(S_i)$ also becomes $O(m^d)$ in the worst case (the R-Tree region only overlaps with the last box!) Although one can accelerate this intersection check by using some temporary in-memory structures to index the boxes after the *Generate* step, the *EDR* computation overhead is hard to improve.

In this section, we answer a fundamental question: can we avoid the $Generate$ step altogether and perform the intersection check without computing the entire *EDR*? Fortunately, the answer to this question is positive! The idea is to transform the problem of intersection check between $EDR(S_i)$ and intermediate R-Tree MBR into an equivalent problem: given an R-Tree MBR R_{xy}, does there exist any query box R_{ab} ($R_{ab} \subset EDR(S_i)$) according to Theorem 1, *s.t.* R_{ab} intersects with R_{xy}? We show that this problem can be answered in $O(md)$. In addition, it has certain nice locality properties that provide opportunities for further optimization to achieve almost constant cost in practice. Our final algorithm `DeltaSky` achieves both optimal I/O performance and superior response time at the same time.

5.2 Algorithm Description

For two d-dimensional regions to intersect with each other, their projections on every dimension must overlap. Let R_{xy} denotes the minimum bounding rectangle (MBR) of a given R-tree entry e_i. Point $x = (x^1, x^2, \cdots, x^d)$ and $y = (y^1, y^2, \cdots, y^d)$ denote the lower left and the upper right points of R_{xy}, respectively. Let $\tilde{S}^1, \tilde{S}^2, \cdots, \tilde{S}^d$ be d sorted lists, where $\tilde{S}^j$ contains the projections of all m current skyline points on dimension d_j. Assume that we want to delete skyline point $S_i = (S_i^1, S_i^1, \cdots, S_i^d)$. In each dimension d_j, there can be 5 possible cases as shown in Fig. 3 assuming that $x^j \neq S_i^j \neq y^j$ (equality cases can be handled similarly). In case 1, $x^j < y^j < S_i^j$, no valid interval can overlap with $[x^j, y^j]$, since Theorem 1 requires that any valid interval must lie "below" S_i^j. In case 2,3, and 4, the high end point of a valid interval must fall below x^j in order to overlap with $[x^j, y^j]$. In case 5, the only interval that can overlap with $[x^j, y^j]$ is $[\tilde{S}_m^j, +\infty]$.

Now we introduce the notion of Minimum Cut and Maximum Coverage in the following two definitions.

Definition 1. (**Minimum Cut**) *For each dimension d_j, we define MC_j, as the least projection value in $\tilde{S}_j$ that is greater than both S_i^j and x^j. The Minimum Cut $MC = \{MC_1, MC_2, ..., MC_d\}$.* □

Definition 2. (**Maximum Coverage**) *For each dimension d_j, we define MCV_j, as the set of skyline points S_k whose projection value, S_k^j, is greater or equal to MC_j,* i.e. *$MCV_j = \{S_k | S_k^j \geq MC_j\}$. The Maximum Coverage $MCV = \bigcup_{j=1}^{d} MCV_j$.* □

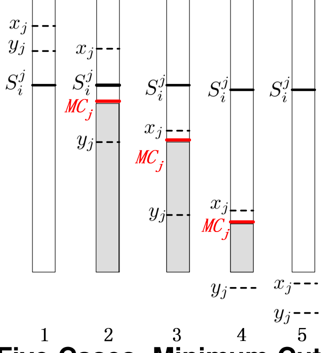


Figure 3. Five Cases, Minimum Cut and Maximum Coverage

We illustrate the above definitions in Fig. 3. The red lines in case 2,3 and 4 represent the positions of MC_j in various cases and the grey area below MC_j, stands for the corresponding MCV_j. Note that for the extreme case like case 5, MC_j, is defined as $+\infty$, since x^j is greater than the greatest value in $\tilde{S}_j$ and hence the corresponding MCV_j is empty.

Now we establish some useful properties of the minimum cut and the maximum coverage.

Lemma 4. (Minimality of MC) *For a given R-Tree entry e_i whose MBR is denoted as R_{xy} and a valid hyper-rectangle R_{ab} generated according to Theorem 1, R_{ab} intersects R_{xy}, then $\forall j\,(1 \leq j \leq d), b^j \geq MC_j$.*$\square$

In particular, in the extreme case like case 5, $[a^j, b^j] = [\tilde{S}^j_m, +\infty]$, where $\tilde{S}^j_m$ is the mth (greatest) projection value in list $\tilde{S}^j_m$.

Lemma 5. (Maximality of MCV) *For a given R-Tree entry e_i whose MBR is denoted as R_{xy}, and a valid hyper-rectangle R_{ab} generated according to Theorem 1, R_{ab} intersects R_{xy}, then the union of all $\tilde{S}^j_{-}\,(1 \leq j \leq d)$, is a subset of MCV, where $\tilde{S}^j_{-}$, denotes the sublist of $\tilde{S}^j$ "below" a^j.*$\square$

Theorem 2. *Given an R-Tree entry e_i whose MBR is denoted as R_{xy}, the exclusive dominance region $EDR(S_i)$ of point S_i intersects with R_{xy}, i.f.f. $\forall j, 1 \leq j \leq d, y^j > S_i^j$ and the corresponding $MCV \equiv \bigcup_{j=1}^{d} MCV_j = S - \{S_i\}$*$\square$.

According to Theorem 2, we can decide if an intermediate R-Tree MBR intersects with $EDR(S_i)$ by only computing the corresponding MCV and check if it contains all the distinct skyline point IDs expect S_i. The `DeltaSky` algorithm (Shown in Algorithm 2) implements this idea. `DeltaSky` takes as input the corner points of R_{xy} and the skyline point to delete S_i. If `DeltaSky` finds R_{xy} impossible to overlap with $EDR(S_i)$ on any dimension (case 1), it directly returns false without looking into the remaining dimensions any more `(line 7-11)`. It skips the computation of MCV_j on the remaining dimensions if the MCV set already contains m-1 distinct items `(line 14-15)`. By doing so, it avoids unnecessary scans of the remaining sublists. It scans the sublist "below" the Minimum Cut MC_j on $\tilde{S}_j$ and adds the corresponding skyline IDs into MCV `(line 18-19)`. After visiting all the d lists, it has computed the Maximum Cut MCV for the current R-tree MBR R_{xy} and can determine if R_{xy} intersects with $EDR(S_i)$ or not according to Theorem 2 `(line 22-24)`.

`DeltaSky` works within the framework of ***BBS*** as the intersection check procedure. During the deletion maintenance process, for each subentry e_i resulting from the expansion of an intermediate R-tree entry, ***BBS*** calls `DeltaSky` to check whether e_i's MBR intersects with $EDR(S_i)$. It is important to note that once this intersection check returns true, the following dominance check *only* needs to consider the newly appeared skyline points in ΔS, since any MBR intersecting with $EDR(S_i)$ cannot be dominated by any existing skyline points in $S - S_i$.

Algorithm 2 DeltaSky

```
1:  S_i: the skyline point to delete
2:  S_i^j: S_i's projection value on dimension d_j
3:  R_xy: an intermediate R-tree MBR
4:  S~^j: a sorted list of all the skyline projection values on dimension d_j
5:  DeltaSky(R_xy, S_i)
6:  Procedure
7:  for every dimension d_j, 1 ≤ j ≤ d do
8:    if y^j < S_i^j then
9:      return false;
10:   end if
11: end for
12: initialize the Maximum Coverage set MCV to ∅;
13: for every dimension d_j, 1 ≤ j ≤ d do
14:   if MCV.size()== m-1 then
15:     continue;
16:   end if
17:   MC_j = least projection value in S~^j greater than max(S_i^j, y^j);
18:   for every skyline point S_k whose projection value S_k^j ≥ MC_j do
19:     MCV = MCV ∪ {S_k};
20:   end for
21: end for
22: if MCV.size()==m-1 then
23:   return true;
24: end if
25: return false;
26: End Procedure
```

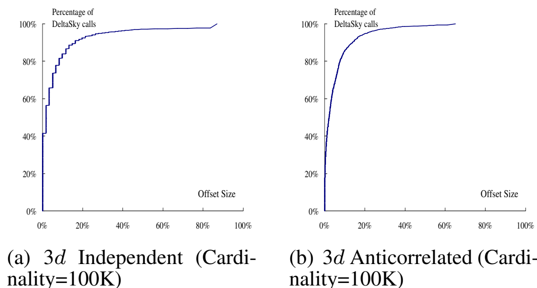
In our implementation, we store each list $\tilde{S}^j$ in a clustered B-Tree index with projection value as the search key and skyline tuple ID as the value. To compute the MCV_j from each list, we first locate the position of MC_j among the B-Tree leaf nodes via a B-Tree lookup and then perform a B-Tree scan down the list at the leaf level to get the skyline point IDs "below" MC_j. A hashtable is used to record the distinct skyline point IDs in MCV. The computation of the maximum coverage MCV involves the union of d lists, which amounts to d B-tree scans. Moreover, we can also use one single B-Tree to store all the d lists and separate them by adding DimensionID to the search key.

In most applications, the skyline view size is reasonable, thus normally these B-Trees can reside in main-memory to improve performance. However, we do not adopt an in-memory implementation for $\tilde{S}^j$, because 1) there can be lots of skyline views defined over the same database on different attribute sets, each of which needs to be maintained; 2) for high-dimensional skylines, the size of a single view can be big (order of Megabytes); 3) in any real-world database server, there can be concurrent queries running to compete for memory resources. Therefore it is unsafe to assume that all lists in `DeltaSky` can be held in memory.

5.3 Optimizations

5.3.1 Exploiting the Locality Property of DeltaSky

Recall that MCV contains all the skyline point IDs that falls "below" the Minimum Cut MC, where MC is determined by the deleted point S_i and the lower left point x of e_i's MBR. We hypothesize that there exists certain locality property in both the Minimum Cut value and the Minimum Coverage set across adjacent `DeltaSky` calls. The reason is that S_i always stays the same during the maintenance process. Furthermore, *BBS* visits intermediate R-tree entries according to some order (in the ascending order of their $MinHist$ values). Hence it is unwise to recompute the Maximum Coverage for each intersection check as the same part of each list $\tilde{S}^j$ may be repeatedly scanned.



(a) $3d$ Independent (Cardinality=100K)

(b) $3d$ Anticorrelated (Cardinality=100K)

Figure 4. Locality Property of DeltaSky

We conduct a case study to verify this hypothesis. In our case study, both anti-correlated and independent datasets of cardinality 100K are used, and our workload consists of 100 deletions of randomly picked skyline points. `DeltaSky` is invoked around 25000 times in both workloads. For each call, we record the position of its Minimum Cut on a fixed dimension and compute the offset between every consecutive call. Fig 4 plots the cumulative density function of the offset size distribution. The y-axis represents the percentage of `DeltaSky` calls, and x-axis depicts the corresponding offset size in terms of the percentage of the list size. As we can see, the offset size between consecutive `DeltaSky` invocations is typically very small. Specifically, on both datasets, almost half of the `DeltaSky` invocations have the same Minimum Cut position as its previous invocation (offset equals to *zero*) and 90 percent has offset less than 10 percent of the list size!

This locality property motivates us to share the MCV computation across invocations. Specifically, each `DeltaSky` call "inherits" the Maximum Coverage set MCV_{old} from its preceding invocation. It computes the MCV_{new} for the current R-tree entry based on MCV_{old} by only scanning the offsets. We will show in our experiments that this achieves considerable performance gain especially when the skyline result size is big and a complete scan down each list is expensive.

5.3.2 Pushing Dominance Checks

Another optimization is to push the dominance check ahead of the intersection check. In the original *BBS* algorithm, the intersection check takes constant cost and the cost of the dominance check is at least in the order of $\log |S|$ (if an extra R-tree is used to index the Skyline Set). However, in our case, an intersection check has worst case complexity of $O(|S|d)$ and a dominance check costs at most O($|\Delta S| * d$), where $|\Delta S|$ is the number of newly appeared skyline points after deletion. As we will show that ΔS is less than 1 on average, therefore, it is unwise to invoke `DeltaSky` for an "unpromising" R-tree MBR which is later dominated and discarded. This technique filters out considerable number of intermediate R-tree regions and thus saves unnecessary intersection checks.

5.4 Algorithm Analysis

Based on the minimality of *EDR* (Lemma 1) and the I/O optimality of *BBS* [21], we establish the optimality of our approach in the following theorem.

Theorem 3. *The number of nodes accessed by* BBS *with DeltaSky as intersection check is optimal for deletion maintenance.* □

According to the analysis from the original *BBS* paper [21], both the number of node accesses and heap size of *BBS* are proportional to the size of the skyline result. These analytical results also apply to the deletion maintenance in our case. The performance advantage of `DeltaSky` over *BBS* re-computation can thus be approximated by $|\Delta S|/|S|$, where ΔS only consists of the new skyline points after the deletion and S is the original skyline set.

Intuitively, the expected number of skyline points in a *EDR* should be less than 1. The reason is that the total number of the "second tier" skyline points are expected to be slightly less than the current skyline size, among which most are dominated by more than one skyline point. Therefore, the number of second-tier skyline which are exclusively dominated by one single skyline point should be less than 1 on average. In a longer version [24], we empirically measure the size of ΔS in our experiments and find it typically *very* small (less than 1 on average) compared to $|S|$. As such, the benefit of using `DeltaSky` is expected to be significant for deletion maintenance.

Besides being I/O optimal, the optimized version of `DeltaSky` scans only the offsets of consecutive Minimum Cuts, which typically involves constant cost on average. For example, for the deletion workloads in Fig. 4(a) and 4(b), on average `DeltaSky` only scans less than 4 percent of the lists, i.e. with the average cost of $0.04md$. Moreover, we would like to reiterate here that the dominance check when using `DeltaSky` is performed against the skyline points in ΔS instead of S as in the original BBS algorithm.

6 Experiment Results

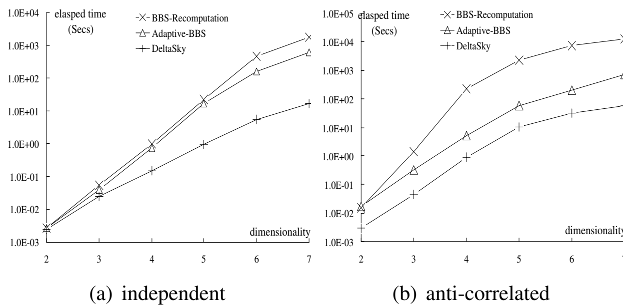
6.1 Experiment Setup

We have implemented all the methods described in the paper. We use the B-Tree index provided in Berkeley DB storage manager and the R*-tree index [4] from Spatial Index Library [1]. The page size of both indices equals to 4K bytes and R-Tree node capacity is set to 100. We tested the correctness of all the implementations by comparing the results against an $O(n^2)$ naive implementation (each point compared against every other points to find the skyline).

We borrowed the data generator from the authors of the original skyline paper [6] which has been used extensively in previous literature. Following the common methodology, both independent and anti-correlated datasets are used for

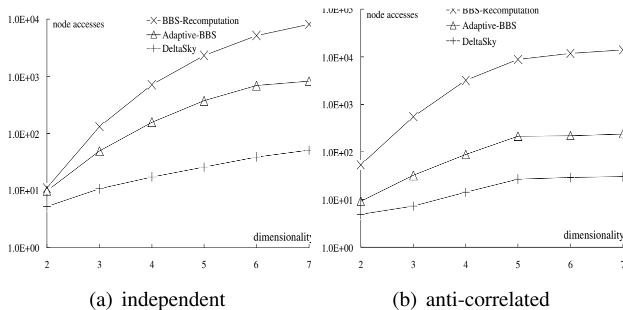
the evaluation with dimensionality from 2 to 7 and cardinality from 100K to 5M, covering the scale of most real world scenarios[1]. We conduct head-to-head comparison between DeltaSky, ***BBS*** re-computation and the adaptive version of ***BBS*** ($ABBS$ discussed in Section 3.4) under various settings. For each experiment setting, we evaluate DeltaSky and $ABBS$ 100 times by deleting 100 randomly selected points in the skyline view and report the average of the results of both algorithms. Please note that different skyline points chosen for deletion leads to distinct performance impact on both DeltaSky and $ABBS$. Basically, the ***EDR*** size varies considerably across different skyline points. In addition to the average performance, we also report some detailed comparison result for certain settings [24]. By default, the results are from a Linux box with an Intel Pentium IV 2GMHz processor and 1 GB of RAM.

6.2 Experiment 1: The Impact of Dimensionality



(a) independent (b) anti-correlated

Figure 5. Effects of Dimensionality on Maintenance Time (Cardinality=1M)

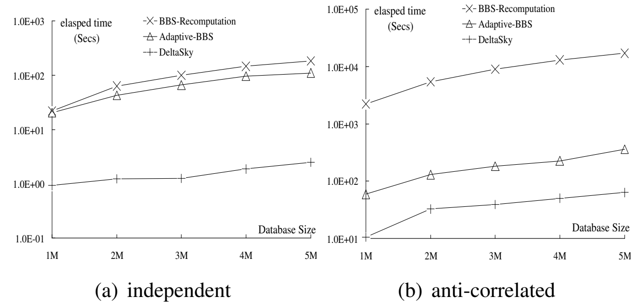


(a) independent (b) anti-correlated

Figure 6. Effects of Dimensionality on I/O (Cardinality=1M)

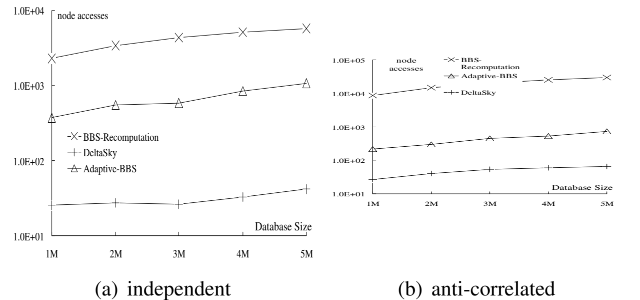
In this experiment, we study the impact of dimensionality on the performance of deletion maintenance. We fix the cardinality of the datasets at 1M and report the results by varying the dimensionality from 2 to 7. Fig. 5 illustrates the maintenance time as a function of the dimensionality for both independent (Fig. 5(a)) and anti-correlated (Fig. 5(b)) datasets. Fig. 6 illustrates the corresponding I/O costs in term of the number of R-Tree node accesses. For each point on the graph, we measure both DeltaSky and $ABBS$ with 100 different skyline deletions and report the average. As we can see, DeltaSky is the clear winner in *any* settings. For instance, Fig. 5(b) shows that on a $6d$ anti-correlated dataset, deletion maintenance of most skyline points can be completed within 10 seconds by DeltaSky (with an average of 30 seconds), in contrast the ***BBS***-recomputation take more than hours and $ABBS$ takes more than 3 minute on average! The performance of all methods degrades with the growth of the dimensionality, which is consistent with earlier studies [26][21]. The increase of DeltaSky's processing time in Fig. 5 is due to: 1) more projection lists to scan for the Maximum Coverage computation, 2) the increase of the skyline result size in high dimensions leads to larger projection lists, 3) R-Tree's inability to isolate relevant data in high dimensions. On the other hand, Fig. 6 shows that the I/O performance of DeltaSky is very stable. The reason is that DeltaSky guarantees the I/O optimality for any dimensions and the number of new skyline points ($|\Delta S|$) is usually small even for high dimensional cases. Besides these average values, we also report in [24] detailed comparision under some setting to show clearly that DeltaSky consistently beats $ABBS$ on *every single* deletion in both settings.

6.3 Experiment 2: The Impact of Cardinality



(a) independent (b) anti-correlated

Figure 7. Effects of Cardinality on Maintenance Time(Dimensionality=5)



(a) independent (b) anti-correlated

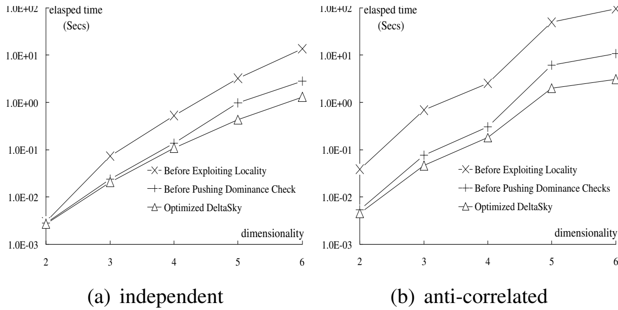
Figure 8. Effects of Cardinality on I/O (Dimensionality=5)

Fig. 7 and Fig. 8 report the impact of cardinality on the performance of all three methods. We fix the dimensionality at 5 and vary the datasets' size from 1 million to 5 million records. The impact of cardinality is shown to be small on both methods, which is demonstrated by the "flat" lines in Fig. 7 and Fig. 8. This also agrees with the results from prior studies. As we can see, DeltaSky consistently outperforms alternative approaches by orders of magnitude. In particular, for a $5d$ anti-correlated dataset containing 5 million records,

[1]The same set of experiments have also been run over the correlated data set. Correlated data set is the least challenging case for skyline computation since a point that is good at one dimension tends to be "dominating" on other dimensions as well which is not a good reflection for most practical scenarios of optimizing "conflicting" goals. In fact, all methods return quickly even in high dimensions and DeltaSky consistently outperforms alternatives. Like [21], we do not plot them here.

on average `DeltaSky` finds new skyline points in less than 1 minute and most deletion maintenance cases are actually completed within seconds. In comparison, $ABBS$ on average spends more than 5 minutes on each deletion maintenance. The advantage of `DeltaSky` becomes even greater on independent datasets (50 times faster than $ABBS$ on 5M dataset). Again, the detailed comparison in [24] further confirms that `DeltaSky` is the winner on *every* deletion instance.

6.4 Experiment 3: Effectiveness of Optimization Techniques



(a) independent (b) anti-correlated

Figure 9. Effectiveness of Optimization Techniques (Cardinality=100K)

This set of experiments is conducted on a Linux box with 800MHZ Pentium III processor and 1G memory. We evaluates the effectiveness of the optimization techniques proposed in Section 5.3. We use both anti-correlated and independent datasets with cardinality 100K and vary the dimension number from 2 to 6. Fig 9 compares the average maintenance time of the fully optimized `DeltaSky` algorithm against its un-optimized versions. Clearly, both optimization techniques are effective. Specifically, exploitation of the locality property is shown to be crucial to `DeltaSky`. By sharing the maximum coverage computation across different invocations, `DeltaSky` achieves order of magnitude performance gain on both independent and anti-correlated datasets. This result is consistent with our previous case study in which on average less than 4 percent of the list is scanned per intersection check. On the other hand, the advantage of pushing dominance check becomes clearer with the increase of the dimensionality in both Fig 9(a) and 9(b). The reason is that in high dimensions, as the number of intermediate R-tree nodes increases, dominance checks prune away more "unpromising" nodes and thus more likely to avoid unnecessary intersection checks.

7 Conclusion

In this paper, we have addressed the important problem of efficiently maintaining a materialized skyline result set in response to skyline deletions. Our solution `DeltaSky` guarantees I/O optimality and can be easily implemented. Central to this problem is the notion of the exclusive dominance region(*EDR*). We devised a principled way to represent a d-dimensional *EDR* with a collection of hyper-rectangles based on the derivation of its boolean function. We show that the time complexity of d-dimensional *EDR* computation is $O(m^d)$ where m refers to the number of skyline points in the dataset. We then introduce a novel algorithm `DeltaSky` which performs the *EDR* intersection check without explicit *EDR* calculation, and reduces the time complexity to $O(md)$. We discuss the implementation issues and devise optimization techniques that effectively reduce the runtime complexity to almost constant in practice. By integrating `DeltaSky` into the existing *BBS* framework, we achieve both optimal I/O and superior response time for skyline deletion maintenance. Experimental results show that our method outperforms the *BBS*-based alternative methods by orders of magnitude in terms of both R-tree node access and the maintenance time. As on-going work, we are investigating the usage of auxiliary data for skyline view maintenance and the efficient maintenance of skyline cubes for batch updates.

References

[1] Spatial index library. `http://u-foria.org/marioh/spatialindex/`.

[2] M. O. Akinde, O. G. Jensen, and M. H. Bohlen. Minimizing detail data in data warehouses. In *Proc. of EDBT*, 1998.

[3] W. Balke, U. Guntzer, and X.Zheng. Efficient distributed skylining for web information systems. In *Proc. of EDBT*, 2004.

[4] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proc. of SIGMOD*, 1990.

[5] J. A. Blakeley, N. Coburn, and P. A. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM TODS*, 14(3), 1989.

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

[7] C.-Y. Chan, P.-K. Eng, and K.-L. Tan. Stratified computation of skylines with partially-ordered domains. In *Proc. of SIGMOD*, 2005.

[8] C.-Y. Chan, H. Jagadish, K.-L. Tan, A. K. Tung, and Z. Zhang. On high dimensional skylines. In *Proc. of EDBT*, 2006.

[9] S. Chaudhuri, N. Dalvi, and K. Raghav. Robust cardinality and cost estimation for skyline operator. In *Proc. of ICDE*, 2006.

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

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

[12] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *Proc. of VLDB*, 1995.

[13] A. Gupta, H. V. Jagadish, and I. S. Mumick. Data integration using self-maintainable views. In *Proc. of EDBT*, 1996.

[14] A. Gupta and I. S. Mumick. *Materialized Views: Techniques, Implementations and Applications*. MIT Press, 1999.

[15] A. Halevy. Answering queries using views: a survey. *VLDB Journal*, 10(4), 2001.

[16] Z. Huang, C. S. Jensen, H. Lu, and B. C. Ooi. Skyline queries against mobile lightweight devices in manets. In *Proc. of ICDE*, 2006.

[17] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: an online algorithm for skyline queries. In *Proc. of VLDB*, 2002.

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

[19] M. Morse, J. Patel, and W. Grosky. Efficient continuous skyline computation. In *Proc. of ICDE*, 2006.

[20] T. Palpanas, R. Sidle, R. Cochrane, and H. Pirahesh. Incremental maintenance of non-distributive aggregate functions. In *Proc. of VLDB*, 2002.

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

[22] D. Papadias, Y. Tao, F. Greg, and B. Seeger. Progressive skyline computation in database systems. *ACM TODS*, 30(1), 2005.

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

[24] P.Wu, D. Agrawal, O. Eğecioğlu, and A. E. Abbadi. Deltasky: Optimal maintenance of skyline deletions without exclusive dominance region generation. In *UCSB Tech Report*, 2006. `http://www.cs.ucsb.edu/~pingwu/deltasky.pdf`.

[25] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. 1996.

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

[27] Y. Tao, X. Xiao, and J. Pei. Subsky: Efficient computation of skylines in subspaces. In *Proc. of ICDE*, 2006.

[28] P. Wu, C. Zhang, Y. Feng, B. Y.Zhao, D. Agrawal, and A. E. Abbadi. Parallelizing skyline queries for scalable distribution. In *Proc. of EDBT*, 2006.

[29] T. Xia and D. Zhang. Refreshing the sky: The compressed skycube with efficient support for frequent updates. In *Proc. of SIGMOD*, 2006.

[30] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient maintenance of materialized top-k views. In *Proc. of ICDE*, 2003.

[31] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang. Efficient computation of the skyline cube. In *Proc. of VLDB*, 2005.