SUBSKY: Efficient Computation of Skylines in Subspaces

Yufei Tao
Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
taoyf@cs.cityu.edu.hk

Xiaokui Xiao
Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
xkxiao@cs.cityu.edu.hk

Jian Pei
School of Computing
Simon Fraser University
University Drive, Burnaby, Canada
jpei@cs.sfu.ca

Abstract

Given a set of multi-dimensional points, the skyline contains the best points according to any preference function that is monotone on all axes. In practice, applications that require skyline analysis usually provide numerous candidate attributes, and various users depending on their interests may issue queries regarding different (small) subsets of the dimensions. Formally, given a relation with a large number (e.g., > 10*) of attributes, a query aims at finding the skyline in an arbitrary* subspace *with a low dimensionality (e.g., 2).*

The existing algorithms do not support subspace skyline retrieval efficiently because they (i) require scanning the entire database at least once, or (ii) are optimized for one particular subspace but incur significant overhead for other subspaces. In this paper, we propose a technique SUBSKY *which settles the problem using a single B-tree, and can be implemented in any relational database. The core of* SUBSKY *is a transformation that converts multi-dimensional data to 1D values, and enables several effective pruning heuristics. Extensive experiments with real data confirm that* SUBSKY *outperforms alternative approaches significantly in both efficiency and scalability.*

1 Introduction

Given a set of d-dimensional points, a point p *dominates* another p' if the coordinate of p on each dimension is not larger than that of p', and strictly smaller on at least one dimension. The *skyline* consists of all the points that are not dominated by others. Consider, for example, Figure 1 where $d = 2$, and each point corresponds to a hotel record. The x-dimension represents the *price* of a hotel, and the y-axis captures its *distance* to the beach. Hotel p_1 dominates p_2 because the former is cheaper and closer to the beach, meaning that p_1 is more preferable according to any preference function that is monotone on the two axes [4]. The skyline includes p_1, p_4, and p_5, which offer various trade-offs between *price* and *distance*: p_4 is the nearest to the beach, p_5 is the cheapest, and p_1 may be a good compromise of the two factors.

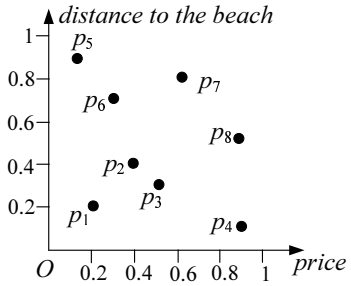


Figure 1. A skyline example

Skyline computation is important in a large number of applications requiring multi-criteria optimization, and has received considerable attention in the database literature [1, 4, 5, 6, 8, 9, 12, 13, 14, 15, 18, 19]. The existing algorithms can be classified in two categories. The first one involves solutions that do not assume any index on the underlying dataset, but they retrieve the skyline by scanning the entire database at least once, resulting in expensive overhead. Methods of the other category incur significantly lower query cost by performing the search on an appropriate structure, e.g., an R-tree [2]. We focus on the second category in this paper.

The motivation of this work is that the previous index-based approaches are optimized for a fixed set of dimensions, and have poor performance for skyline queries targeting different attributes. For instance, assume that, in addition to the dimensions in Figure 1, the database also records the distances of each hotel to several other locations (e.g., the town center, the nearest supermarket, subway station, etc.), the ratings of security, air-quality, traffic-status in the neighborhood, and so on. In other words, the *full* data space can have a high dimensionality, but users may want to retrieve the skyline in any subspace containing a small number of axes.

Obviously, a structure constructed in a particular subspace is useless for queries concerning another. To enable skyline retrieval in all subspaces, a straightforward idea is to create an index that covers all the dimensions (e.g., a high-dimensional R-tree). Unfortunately, the previous algorithms suffer considerable performance degradation when

applied to such a structure (a phenomenon known as "the curse of dimensionality" [3]). Another naive approach is to create multiple low-dimensional structures for different subspaces. For example, assume that the full space has 10 dimensions, and most queries request skylines in subspaces with two attributes. We could create a 2D R-tree for each subspace to achieve the best query efficiency. The number of trees, however, equals $\binom{10}{2}$, and each coordinate of every point must be duplicated 9 times, leading to expensive space consumption and update overhead.

In fact, applications that require skyline analysis usually provide numerous candidate attributes, and various users may issue queries regarding different (small) subsets of the dimensions depending on their interests. Therefore, a truly practical method for skyline retrieval should aim at effectively supporting queries about subspaces (especially those involving few axes), by consuming the smallest amount of space. In this paper, we present a new technique *SUBSKY*, which solves the problem using a transformation that converts each d-dimensional point to a 1D value. The converted values are indexed by a *single* conventional B-tree, which is deployed to find the skyline in any subspace efficiently, based on several interesting observations on the problem characteristics. *SUBSKY* leverages *purely relational technologies*, and hence, can be easily implemented in any commercial database system. Extensive experiments with real datasets confirm that *SUBSKY* is significantly faster than alternative algorithms.

The rest of the paper is organized as follows. Section 2 reviews the previous work related to ours. Section 3 presents the basic *SUBSKY* optimized for uniform data, and Section 4 generalizes the technique to arbitrary data distributions. Section 5 contains an extensive experimental evaluation that demonstrates the efficiency of *SUBSKY*. Section 6 concludes the paper with directions for future work.

2 Related Work

The first skyline algorithm in the database context is *BNL* (block-nested-loop) [4], which compares each point with the others, and reports it as a result only if it is not dominated by any other point. *SFS* [6] (sort-filter-skyline) is based on the same rationale as *BNL*, but improves performance by first sorting the data according to a monotone function. *DC* [4] (divide-and-conquer) divides the data space into several regions, calculates the skyline in each region, and produces the final skyline from the points in the regional skylines. *Bitmap* [18] converts each point p to a bit string, which encodes the number of points having a smaller coordinate than p on every dimension. The skyline is then obtained using only bit operations. *LESS* (linear-elimination-sort for skyline) [9] is an algorithm that has attractive worst-case asymptotical performance. Specifically, when the data distribution is uniform and no two points have the same coordinate on any dimension, *LESS* computes a d-dimensional skyline in expected $O(d \cdot n)$ time, where n is the dataset cardinality.

List 1 (x)	p_5:0.1	p_6:0.3	p_2:0.4	p_7:0.6
List 2 (y)	p_4:0.1	p_1:0.2	p_3:0.3	p_8:0.5

Figure 2. The structure of *Index*

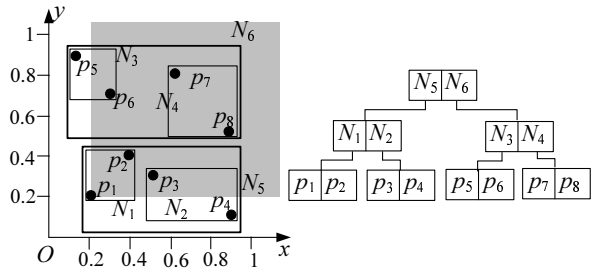


Figure 3. Illustration of *BBS*

As opposed to the above algorithms that must read the whole database at least once, index-based methods need to visit only a fraction of the dataset. *Index* [18] organizes the data into d lists, where the i-th list ($1 \leq i \leq d$) contains points whose coordinates on the i-th axis are the smallest among all the dimensions. Figure 2 shows the $d = 2$ lists for the dataset of Figure 1. For example, p_5 is assigned to List 1 because its x-coordinate 0.1 is smaller than its y-coordinate 0.9. In case a point (e.g., p_1, p_2) has identical coordinates on both dimensions, the list that contains it is decided randomly (assume that p_2 and p_1 are included in Lists 1 and 2, respectively). The entries in List 1 (2) are sorted in ascending order of their x- (y-) coordinates (e.g., entry p_5:0.1 indicates the sorting key 0.1 of p_5).

To compute the skyline, *Index* scans the two lists in a synchronous manner. At the beginning, the algorithm initializes two pointers pt_1 and pt_2 referencing the first entries p_5, p_4 of the two lists, respectively. The referenced entry with a smaller sorting key is processed. Since both p_5 and p_4 have the same sorting key 0.1, *Index* processes the point p_5 from List 1, adds it to the skyline set S_{sky}, and moves pt_1 to the next entry p_6. The next point examined is p_4, which is compared with the existing skyline point p_5, and then inserted in S_{sky} (i.e., it is not dominated by p_5). Similarly, p_1 is also included as a skyline point, after which pt_2 is set to p_3.

Both coordinates of p_1 are smaller than the x-coordinate 0.3 of p_6 (pointed to by pt_1), in which case all the not-yet inspected points p in List 1 can be pruned. To understand this, observe that both coordinates of p are at least 0.3, indicating that p is dominated by p_1. Due to the same reasoning, List 2 can also be eliminated because both coordinates of p_1 are lower than the y-coordinate of p_3. Thus, the algorithm finishes with $\{p_1, p_4, p_5\}$ as the result.

NN (nearest-neighbor) [12] and *BBS* (branch-and-bound

skyline) [14] find the skyline using an R-tree [2]. The difference is that *NN* issues multiple NN queries [17, 11] while *BBS* performs only a single traversal of the tree. It has been proved [14] that *BBS* is I/O optimal, i.e., it accesses fewer disk pages than any algorithm based on R-trees (including *NN*). Hence, the following discussion concentrates on this technique. Figure 3 shows the R-tree for the dataset of Figure 1, together with the minimum bounding rectangles (MBR) of the nodes. *BBS* processes the (leaf/intermediate) entries in ascending order of their *mindist* (minimum distance) to the origin of the data space. At the beginning, the root entries are inserted into a heap H (= $\{N_5, N_6\}$) using their mindist as the sorting key. Then, the algorithm removes the top N_5 of H, accesses its child node, and enheaps all the entries there (H now becomes $\{N_1, N_2, N_6\}$).

Similarly, the next node visited is leaf N_1, where the data points are also added to H (= $\{p_1, p_2, N_2, N_6\}$). Since p_1 tops H, it is taken as the first skyline point, and used for pruning in the subsequent execution. Point p_2 is de-heaped next, but is discarded because it falls in the *dominant region* of p_1 (the shaded area). *BBS* then visits N_2, and inserts only p_4 into H = $\{N_6, p_4\}$ (p_3 is not inserted as it is dominated by p_1). Likewise, accessing N_6 adds only one entry N_3 to H (= $\{N_3, p_4\}$) because N_4 lies completely in the shaded area (i.e., all points in its subtree must be dominated by p_1). The remaining entries processed are N_3 (en-heaping p_5), p_4, p_5, at which point H becomes empty and *BBS* terminates.

Finally, Balke et al. [1] consider skyline retrieval in distributed environments, Lin et al. [13] investigate continuous skyline monitoring on data streams, and Chan et al. [5] study partially-ordered domains. These methods are restricted to their specific scenarios, and cannot be adapted for the problem of this paper. Recently, Pei et al. [15] and Yuan et al. [19] independently propose the *sky-cube*, which consists of the skylines in *all* possible subspaces. These skylines form a lattice similar to the "data cube" [10] of a relation. The topic of this paper differs from skycube in that we aim at computing the skyline of *one* particular subspace, as opposed to all subspaces.

3 The Basic SUBSKY for Uniform Data

In this section, we focus on uniform data, and explain the rationale of *SUBSKY*. Section 3.1 presents an algorithm for computing subspace skylines. Then, Section 3.2 provides theoretical evidence that explains its efficiency. In Section 4, we will generalize *SUBSKY* to arbitrary data distributions.

3.1 The Algorithm

Without loss of generality, we assume a unit d-dimensional (full) space where each axis has domain [0, 1]. We use the term *maximal corner* for the corner A^C of the data space having coordinate 1 on all dimensions. Each data point p is converted to a 1D value $f(p)$ equal to the L_∞ distance between p and A^C:

$$f(p) = \max_{i=1}^{d}(1 - p[i]) \quad (1)$$

where $p[i]$ represents the i-th coordinate of p ($1 \leq i \leq d$). If p_{sky} is a skyline point in the full space, no point p satisfying the following inequality can belong to the skyline:

$$f(p) < \min_{i=1}^{d}(1 - p_{sky}[i]) \quad (2)$$

Figure 4a illustrates a 2D example. Points p satisfying the inequality constitute the shaded square, whose side length equals $\min_{i=1}^{2}(1-p_{sky}[i])$, i.e., the coordinate difference between p_{sky} and A^C on the y-dimension. Obviously, no such p can appear in the skyline because the square is entirely contained in the dominant region of p_{sky}. Consider point p' outside the shaded area (i.e., p' violates Inequality 2). Given *only* $f(p')$ (but not the coordinates of p'), we cannot assert that p_{sky} dominates p' — points whose L_∞ distances to A^C equal $f(p')$ could lie anywhere on the left and bottom edges of the dotted square with side length $f(p')$.

Inequality 2 applies to the original space, while a similar result exists for the skyline of any subspace. Representing a subspace as a set SUB capturing the relevant dimensions (e.g., if the subspace involves the 1st and 3rd axes of a data space, then SUB = {1, 3}), we have:

Lemma 1 *Given a skyline point p_{sky} in a subspace SUB, no point p qualifying the following condition can belong to the skyline of SUB:*

$$f(p) < \min_{i \in SUB}(1 - p_{sky}[i]) \quad (3)$$

For example, assume that the full space has dimensionality 3 whereas the goal is to retrieve the skyline in the first two dimensions (SUB = {1, 2}). If p_{sky} has coordinates (0.05, 0.1, –) (the 3rd coordinate is irrelevant), no point p with $f(p) < 1 - 0.1 = 0.9$ can be in the target skyline. This is correct because the coordinates of p must be at least 0.1 on all dimensions; hence, it is dominated by p_{sky} in the subspace SUB.

Lemma 1 leads to a fast algorithm for computing the skyline of a subspace SUB. Specifically, we access the data points p in descending order of their $f(p)$. Meanwhile, we maintain (i) the current set S_{sky} of skyline points (among the data already examined), and (ii) a value U corresponding to the largest $\min_{i \in SUB}(1-p_{sky}[i])$ (i.e., the right hand side of Inequality 3) for the points $p_{sky} \in S_{sky}$. The algorithm terminates when U is larger than the $f(p)$ of the next p to be processed.

We illustrate the algorithm using the 8 three-dimensional points of Figure 5. The coordinates of each point p_i ($1 \leq$

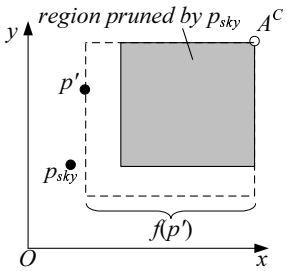


Figure 4. Illustration of Inequality 2

dimension	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8
1 (x)	0.2	0.4	0.5	0.9	0.1	0.3	0.6	0.9
2 (y)	0.2	0.4	0.3	0.1	0.9	0.7	0.8	0.5
3 (z)	0.5	0.9	0.1	0.6	0.3	0.2	0.7	0.6
$f(p_i)$	0.8	0.6	0.9	0.9	0.9	0.8	0.4	0.5

Figure 5. An example dataset

$i \leq 8$) correspond to the numbers in the 2nd-4th rows of the column for p_i (the x- and y-coordinates are the same as in Figure 1). The last row contains the values of $f(p_i)$. Let the query subspace involve the first two dimensions ($SUB = \{1, 2\}$). We process the data in this order: $\{p_3$, p_4, p_5, p_1, p_6, p_2, p_8, $p_7\}$. After examining p_3, the algorithm initializes S_{sky} to $\{p_3\}$, and U to 0.5 (i.e., the smaller between $1 - p_3[1]$ and $1 - p_3[2]$). After p_4, S_{sky} becomes $\{p_3, p_4\}$, but U remains the same. Similarly, p_5 is added to S_{sky} next without affecting U. Processing p_1 inserts it in S_{sky}, removes p_3 from S_{sky} (p_3 is dominated by p_1), and updates U to 0.8. Finally, we inspect p_6, which does not change S_{sky} or U. Since the f-values of the remaining points are smaller than the current $U = 0.8$, the algorithm finishes and reports $\{p_1, p_4, p_5\}$ as the final skyline.

In practice, to support dynamic updates, we can index the $f(p)$ of all points p using a B-tree. To achieve the correct access order in answering a skyline query, the algorithm starts with the leaf node containing the largest f-values. After exhausting all the entries there, it loads the neighboring leaf (using the "neighbor pointer" in the previous node), and repeats the process until the end of execution.

3.2 Analysis

As shown in the experiments, the above solution already significantly outperforms *BBS* in subspace skyline computation, especially when the dimensionality d of the full space is high, and *BBS* suffers serious performance degradation. To explain this, consider a 15D uniform dataset with cardinality 100k, and we want to retrieve the skyline in a subspace SUB containing any two dimensions, as shown in Figure 6. There is a high chance that a skyline point lies very close to the origin in SUB. Specifically, let us examine the square in Figure 6 whose lower-left corner is the origin, and its side length equals some small $\lambda \in [0, 1]$. The

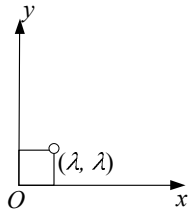


Figure 6. Illustration of the analysis

probability of *not* having any data point in the square equals $(1-\lambda^2)^{100k}$, which is less than 10% for $\lambda = 0.001$. In other words, with an at least 90% chance, we can find a point p_{sky} in the square such that $\min_{i \in SUB}(1 - p_{sky}[i]) \geq 0.999$. In this case, (by Inequality 3) all points p with $f(p) < 0.999$ can be eliminated using the B-tree. The expected percentage of such points in the whole dataset equals the volume of a 15-dimensional square with side length 0.999 (i.e., the square whose opposite corners are the maximal corner of the data space, and the point with coordinate 0.001 on all axes). The volume evaluates to $0.999^{15} = 98.5\%$, that is, we only need to access 1.5% of the dataset!

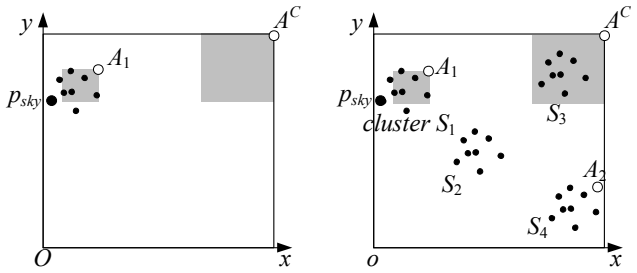
4 The General SUBSKY

In the previous section (where we concentrated on uniform data), $f(p)$ is always computed with respect to one *anchor*, i.e., the maximal corner A^C. In practice where data are usually clustered, the $f(p)$ of various p should be calculated with respect to different anchors to achieve greater pruning power. Section 4.1 elaborates this using concrete examples, and then the subsequent sections develop a new subspace-skyline algorithm based on this idea.

4.1 Pruning with Multiple Anchors

Figure 7a shows a 2D dataset, where the data are clustered near the upper-left corner. If $f(p)$ equals the L_∞ distance between p and the maximal corner, by Lemma 1 a skyline point p_{sky} (in the full space) prunes the right shaded square, which, however, is useless since the square does not cover any data point. On the other hand, let $f(p)$ be the L_∞ distance from p to an alternative anchor A_1. We can eliminate all points p whose $f(p)$ is smaller than $\min_{i=1}^{2}(A_1[i] - p_{sky}[i])$, which equals the coordinate difference of p_{sky} and A_1 on the y-dimension. These points form the left shaded square, which covers a majority of the data, i.e., anchor A_1 offers much stronger pruning power than A^C.

Figure 7b illustrates another 2D example, where the dataset consists of 4 clusters S_1, S_2, ..., S_4. *SUBSKY* would use 3 anchors A_1, A_2, A^C, and convert each data point p to a 1D value $f(p)$ that equals the L_∞ distance between p and one *assigned* anchor *dominated by* p. Assume that A_1 is the anchor for the points of S_1, A_2 for those of S_4, and A^C for the remaining data. Interestingly, a skyline point



(a) A better anchor A_1 (b) Anchors for different clusters

Figure 7. Effects of multiple anchors

creates a pruning region with every anchor it dominates. For example, as in Figure 7a, p_{sky} from S_1 can eliminate two shaded squares obtained with A_1 and A^C, respectively. In Figure 7b, the right square includes the entire S_3, and hence, the data in this cluster can be removed from further consideration.

In general, if the anchor set S_{anc} contains A_1, A_2, ..., A_m, we have the following heuristic for skyline retrieval in any subspace:

Lemma 2 *Let p_{sky} be a skyline point in a subspace SUB, and S'_{anc} the set of anchors whose projections in SUB are dominated by p_{sky}. Then, for each anchor $A \in S'_{anc}$, a data point p assigned to A cannot be in the skyline if*

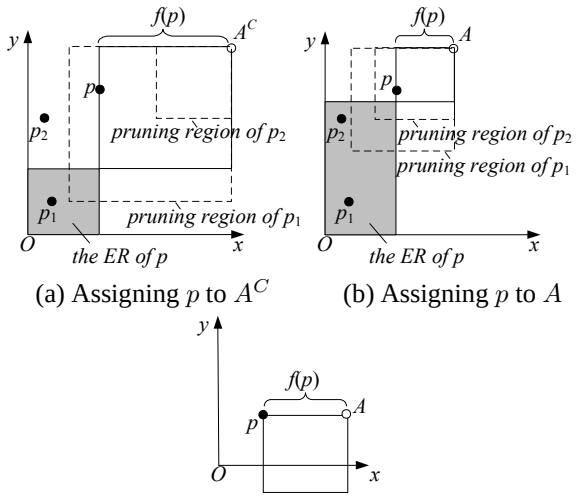
$$f(p) < \min_{i \in SUB}(A[i] - p_{sky}[i]) \quad (4)$$

The lemma degenerates into Lemma 1 for $A = A^C$ (i.e., $A[i] = 1$). As another example, assume $d = 3$, and an anchor A = (0.8, 0.7, 0.1). In the subspace $SUB = \{1, 2\}$, a skyline point p_{sky} = (0.2, 0.2, –) eliminates all points p assigned to A with $f(p) < 0.5$ (= $A[2] - p_{sky}[2]$). Note that the first and second coordinates of p must be larger than 0.3 and 0.2 respectively, indicating that p is dominated by p_{sky} in SUB. In the original 3D space, however, a skyline point p_{sky} = (0.2, 0.2, 0.2) does not produce any pruning region with respect to A, since p_{sky} does not dominate A in this case.

The effectiveness of the heuristic depends on (i) how the data are assigned to anchors, and (ii) how the anchors are selected. We analyze these issues in the next two sections.

4.2 Assigning Points to Anchors

We introduce the concept of *effective region* (ER), which quantifies the benefit of assigning a point to an anchor. Consider Figure 8a where point p is assigned to A^C (the maximal corner). Observe that, in computing the skyline in the 2D space, p can be eliminated with Lemma 2 only if a point is discovered in the shaded square cornered at the origin with side length $1 - f(p)$. The square is the ER of p, which contains the points that (i) dominate p, and (ii) their pruning regions with respect to A^C cover p (the coordinate of



(a) Assigning p to A^C (b) Assigning p to A

(c) ER is empty

Figure 8. The concept of effective region

any point in the ER differs from that of A^C by more than $f(p)$ on all axes). For example, since p_1 (p_2) is inside (outside) the ER, the pruning region of p_1 (p_2) contains (does not contain) p.

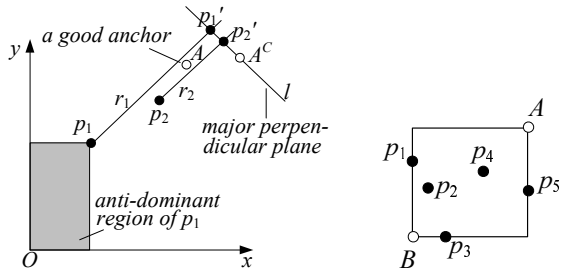
Let us assign p to an alternative anchor A in Figure 8b. The shaded area demonstrates the new ER of p, and it covers both p_1 and p_2 (the dashed rectangles are their pruning regions). This means that p can be eliminated as long as either p_1 or p_2 has been discovered. Compared with assigning p to A^C, the new assignment increases the chance of pruning p.

In general, given a point p and an anchor A dominated by p, *the ER of p (with respect to A) is a d-dimensional rectangle whose opposite corners are the origin and the point having coordinate $A[i] - L_\infty(p, A)$ on the i-th dimension ($1 \leq i \leq d$)*, where $L_\infty(p, A)$ is the L_∞ distance between p and A, and also the value of $f(p)$ if p is assigned to A. The ER is empty if $A[i] < L_\infty(p, A)$ for any dimension i. Figure 8c shows such an example ($d = 2$), where the y-coordinate of A is smaller than $L_\infty(p, A)$, indicating that there does not exist any skyline point that can prune p using Lemma 2.

Hence, we assign p to the anchor that produces the largest ER. Specifically, this is the anchor that is dominated by A and maximizes:

$$\Pi_{i=1}^{d} \max(0, A[i] - L_\infty(p, A)) \quad (5)$$

What is the connection between ER (formulated in the full space) and skyline retrieval in a subspace? To answer this question, we need to interpret Lemma 2 in an alternative manner. Given a skyline point p_{sky} in the query subspace SUB, let us obtain a point p'_{sky} in the original space whose coordinates are equal to those of p_{sky} on the dimensions included in SUB, and 0 on the other axes. Then, a point

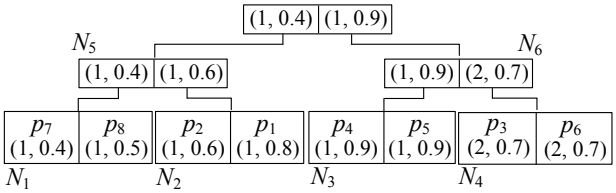


(a) Perfect rays (b) Deciding the anchor

Figure 9. Finding anchors

	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8
ER-vol w.r.t. A_1	8	64	1	1	1	8	216	125
ER-vol w.r.t. A_2	0	-	9	-	-	9	144	75

(a) ER volumes with respect to A_1, A_2 (unit 10^{-3})



(b) The B-tree on the transformed f-values

Figure 10. Illustration of the skyline algorithm

p can be pruned by Lemma 2 (i.e., p does not belong to the skyline of SUB) if and only if p'_{sky} lies in the ER of p. Therefore, enlarging ER also increases the probability of pruning p in computing a subspace skyline.

4.3 Finding the Anchors

An anchor that leads to a large ER for one data point may produce a small ER for another. When we are allowed to keep only m anchors (where m is a small system parameter), how should they be selected in order to maximize the ER volumes of as many points as possible?

Note that the largest ER of a point p corresponds to its *anti-dominant region*, consisting of all the points dominating p. In other words, the maximum value of Formula 5 equals $\Pi_{i=1}^{d} p[i]$, which is achieved when

$$A[i] - L_\infty(p, A) = p[i] \quad (6)$$

on all dimensions $i \in [1, d]$. We call an anchor A satisfying the above equation the *perfect anchor* for p. If p is assigned to A, then p can be pruned with Lemma 2 using *any* skyline point dominating p.

It turns out that each point p has infinite perfect anchors. Let us shoot a ray from p that is in its dominant region, and parallel to the *major diagonal* of the data space (i.e., the diagonal connecting the origin and the maximal corner); every point A on this *perfect ray* is a perfect anchor of p (even including the part of the ray out of the data space). In fact, the coordinate difference between A and p is equivalent on all axes, i.e., $A[i] - p[i] = L_\infty(p, A)$ for any $i \in [1, d]$, leading to the correctness of Equation 6.

In Figure 9a, for example, the perfect ray of point p_1 is r_1, and any anchor on r_1 will result in the ER of p_1 that is the shaded rectangle. Similarly, r_2 is the ray for p_2. Since r_1 and r_2 are very close to each other, if we can keep only a single anchor A, it would lie between the two rays as in Figure 9a — although A is not the perfect anchor of p_1 and p_2, it is a good anchor as it leads to large ERs for both points.

The important implication of the above discussion is that *points with close perfect rays may share the same anchor*. This observation naturally leads to an algorithm for finding anchors based on clustering. Specifically, we first project the data onto the *major perpendicular plane*, which is the d-dimensional plane (passing the maximal corner) perpendicular to the major diagonal of the data space. In Figure 9a ($d = 2$), for instance, the plane is line l, and the projections of p_1 and p_2 are p'_1 and p'_2, respectively. Then, we partition the projected points into m clusters using the k-means algorithm [16, 7], and then formulate an anchor for each cluster. Consider again Figure 7b, where the data in S_2 and S_3 collapse into a single cluster in the major perpendicular plane, which explains why these points were assigned to the same anchor.

It remains to clarify how to decide an anchor A for a cluster S. We aim at guaranteeing that A should produce a non-empty ER for every point $p \in S$ (i.e., $A[i] > L_\infty(p, A)$ on every dimension i, as discussed in Section 4.2); otherwise, p cannot be assigned to A. We illustrate the algorithm using a 2D example but the idea generalizes to arbitrary dimensionality in a straightforward manner.

Assume that S consists of 5 points, and the algorithm examines them in the *original* space (i.e., not their projections), as shown in Figure 9b. We first obtain point B, whose coordinate on each dimension equals the lowest coordinate of the points in S on this axis (note that B necessarily falls inside the data space, and dominates all the points). Then, the algorithm computes the smallest square (or a hyper-square in d-dimensional space) that covers all the points (see Figure 9b). The anchor A for S is the corner of the square opposite to B. The pruning region of B with respect to A encloses the entire square, or equivalently, the ER of every point is not empty since it includes at least B.

4.4 The Data Structure and Query Algorithm

Given a small number m (less than 100 in our experiments), *SUBSKY* first obtains m anchors, by applying the method of Section 4.3 on a random subset of the database. Then, the $f(p)$ of each point p is set to the L_∞ distance between p and its assigned anchor (which maximizes the vol-

Algorithm *subsky* (SUB, $\{A_1, A_2, ... A_m\}$)
/* SUB includes the dimensions relevant to the query subspace; A_1, ..., A_m are the anchors */
1. for j = 1 to m
2. use the B-tree to find the point pt_j with the maximum $f(pt_j)$ among all the points assigned to A_j
3. $pt_j.ER = \Pi_{i=1}^{d}(A_j[i] - L_\infty(pt_j, A_j))$ //Equation 5
4. $S_{sky} = \emptyset$ //the set of skyline points
5. while ($pt_j \neq \emptyset$ for any $j \in [1, m]$)
6. t = the value of j giving the smallest $pt_j.ER$ among all $j \in [1, m]$ such that $pt_j \neq \emptyset$
7. if pt_t is not dominated by any point in S_{sky}
8. remove from S_{sky} the points dominated by pt_t
9. $S_{sky} = S_{sky} \cup \{pt_t\}$
10. for j = 1 to m
11. if $pt_j \neq \emptyset$ and $f(pt_j) < \min_{i \in SUB}(A_j[i] - pt_t[i])$
12. $pt_j = \emptyset$ /* no point assigned to A_j can belong to the skyline (Lemma 2) */
13. pt_t = the point with the next largest $f(p)$ among the data assigned to A_t (this point lies in either the same leaf as the previous pt_t, or a neighboring node)
14. if no more such point exists then $pt_t = \emptyset$
15. else $pt_t.ER = \Pi_{i=1}^{d}(A_t[i] - L_\infty(pt_t, A_t))$
16. return S_{sky}

Figure 11. The query algorithm of *SUBSKY*

ume of ER among the anchors dominated by p[1]). We guarantee the existence of such an anchor by always including the maximal corner in the anchor set.

SUBSKY manages the resulting $f(p)$ with a single B-tree that separates the points assigned to various anchors. We achieve this by indexing a composite key (j, $f(p)$), where $j \in [1, m]$ is the id of the anchor to which p is assigned. Thus, an intermediate entry e of the B-tree has the form ($e.id$, $e.f$), which means that (i) each point p in the subtree of e has been assigned to the j-th anchor with $j \geq e.id$, and (ii) in case $j = e.id$, the value of $f(p)$ is at least $e.f$.

We illustrate the above process using the 3D dataset of Figure 5 and $m = 2$ anchors: the maximal corner A_1, and A_2 = (1, 1, 0.8). The second row of Figure 10a illustrates the ER volume of each data point with respect to A_1, calculated by Equation 5. For instance, the volume 0.125 of p_8 is $\Pi_{i=1}^{3}(A_1[i] - L_\infty(A_1, p_8)) = (1 - 0.5)^3$. Similarly, the third row contains the ER volumes with respect to A_2. A "-" means that the corresponding ER does not exist. For example, the ER of p_2 is undefined because p_2 does not dominate A_2, while there is no ER for p_4 since $A_2[3]$ = 0.8 is smaller than $L_\infty(A_2, p_4)$ = 0.9 (review the formulation of ER in Section 4.2). The white cells of the table indicate each point's ER-volume with respect to its assigned anchor. For example, p_3 is assigned to A_2 since this anchor produces a larger ER than A_1. Figure 10b shows the B-tree indexing the transformed f-values, e.g., the leaf entry p_3:(2, 0.7) in node N_4 captures the fact that $f(p_3)$ equals the L_∞ distance 0.7 between A_2 and p_3.

Assume that we aim at computing the skyline in the subspace SUB = {1, 2}. As the first step, the algorithm identifies, for each anchor, the assigned data point p with the maximum $f(p)$. In Figure 10b, the point for A_1 (A_2) is p_6 (p_5), which is the right-most point assigned to this anchor at the leaf level, and can be easily found by accessing a single path of the B-tree.

Then, the algorithm scans the points assigned to each anchor in descending order of their f-values, i.e., the ordering is {p_5, p_4, p_1, p_2, p_8, p_7} for A_1, and {p_6, p_3} for A_2. For this purpose, we initialize two pointers pt_1 and pt_2 referencing the heads p_5 and p_6 of the two lists, respectively. At each iteration, we process the referenced point with a smaller ER (in case of tie, the next processed point is randomly decided). Continuing the example, since the ER-volume 1 of p_5 is smaller than that 9 of p_6 (implying that p_6 has a larger probability of being pruned by a future skyline point), the algorithm adds p_5 to the skyline set S_{sky}, and advances pt_1 to the next point p_4 in the list of A_1. Similarly, p_4 has a lower ER-volume 1 (than p_6), and is not dominated by p_5; thus, it is also added to S_{sky} (={p_5, p_4}). Pointer pt_1 now reaches p_1 (with ER volume 1), which is included in S_{sky}, too.

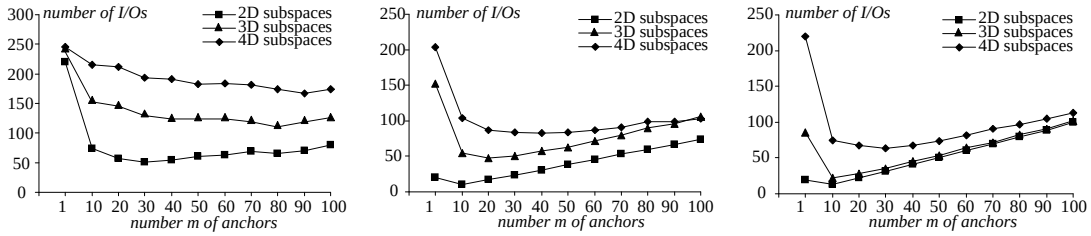
According to Lemma 2, p_1 prunes all the points p assigned to A_1 whose $f(p)$ are smaller than $\min_{i \in SUB}(A_1[i] - p_1[i])$ = 0.8. Since the next point p_2 in the list of A_1 qualifies the condition, none of the remaining data in the list can be a skyline point. Similarly, p_1 also prunes the data p assigned to A_2 satisfying $f(p) < \min_{i \in SUB}(A_2[i] - p[i])$ = 0.8. Thus, the head p_6 in the list of A_2 is also eliminated ($f(p_6) = 0.7 < 0.8$), and no point in the list belongs to the skyline either. Hence, the algorithm terminates with S_{sky} (={p_5, p_4, p_1}). Recall that the basic *SUBSKY* of Section 3.1 needs to inspect additional data p_6 and p_3. Figure 11 formally describes the general algorithm for retrieving a subspace skyline.

4.5 Discussion

We keep the anchor set in memory since it is small (occupying only several k-bytes) and is needed for performing queries and updates. Specifically, to insert/delete a point p, we decide its assigned anchor A as described in Section 4.2, and set $f(p) = L_\infty(p, A)$, after which the insertion/deletion proceeds as in a normal B-tree. Note that the anchor set is never modified after its initial computation. Query efficiency remains unaffected as long as the data distribution does not incur significant changes (we will verify this later with experiments).

For a dynamic dataset, all the data must be retained because a non-skyline point may appear in the skyline after a

[1]The approach is better than assigning p to its closest anchor in the major perpendicular plane, because the ER-volume directly quantifies the benefit of an assignment.



(a) *NBA* (b) *Household* (c) *Color*

Figure 12. *SUBSKY* **performance vs. the number of anchors**

skyline point is deleted. On the other hand, if the dataset is static, points that are not in the skyline of the full space can be discarded since they will not appear in the skyline of any subspace[2]. When d is large, the size of the full-space skyline may still be comparable to the dataset cardinality [8]. Hence, the points (of the skyline) should be managed by a disk-oriented technique (such as *SUBSKY*) to enable efficient retrieval in subspaces.

So far our definition of "dominance" prefers small coordinates on all dimensions, whereas in general a point may be considered dominating another only if its coordinates are larger on some axes. For example, given two attributes *price* and *size* of house records, a reasonable skyline would seek to minimize the *price* but maximize the *size* (i.e., a customer is typically interested in large houses with low prices). Depending on its semantics, a dimension usually has only one "preference direction", e.g., skylines involving *price* (*size*) would most likely prefer the negative (positive) direction of this axis. *SUBSKY* easily supports a positive preference direction by subtracting (from 1) all coordinates on the corresponding dimension (e.g., $1-$ *price*).

5 Experiments

In this section, we experimentally evaluate the efficiency of the proposed techniques. We deploy 3 real datasets *NBA*, *Household*, and *Color*[3]. Specifically, *NBA* contains 17k 13-dimensional points, where each point corresponds to the statistics of a player in 13 categories including the number of points scored, rebounds, assists etc. averaged over the number of minutes played. *Household* consists of 127k 6-dimensional tuples, each of which represents the percentage of an American family's annual income spent on 6 types of expenditures (gas, electricity, etc.). *Color* is a 9-dimensional dataset with a cardinality 68k, and each tuple captures several properties of an image such as brightness, contrast, saturation, and so on. All the values are normalized into the unit range [0, 1].

[2]Strictly speaking, this is correct only if all the data points have distinct coordinates on each dimension. If this is not true, the points that need to be retained include those sharing common coordinates with a point in the full-space skyline. Retrieval of such points is discussed in [15].

[3]These datasets can be downloaded at *http://www.nba.com*, *http://www.ipums.org*, and *http://kdd.ics.uci.edu*, respectively.

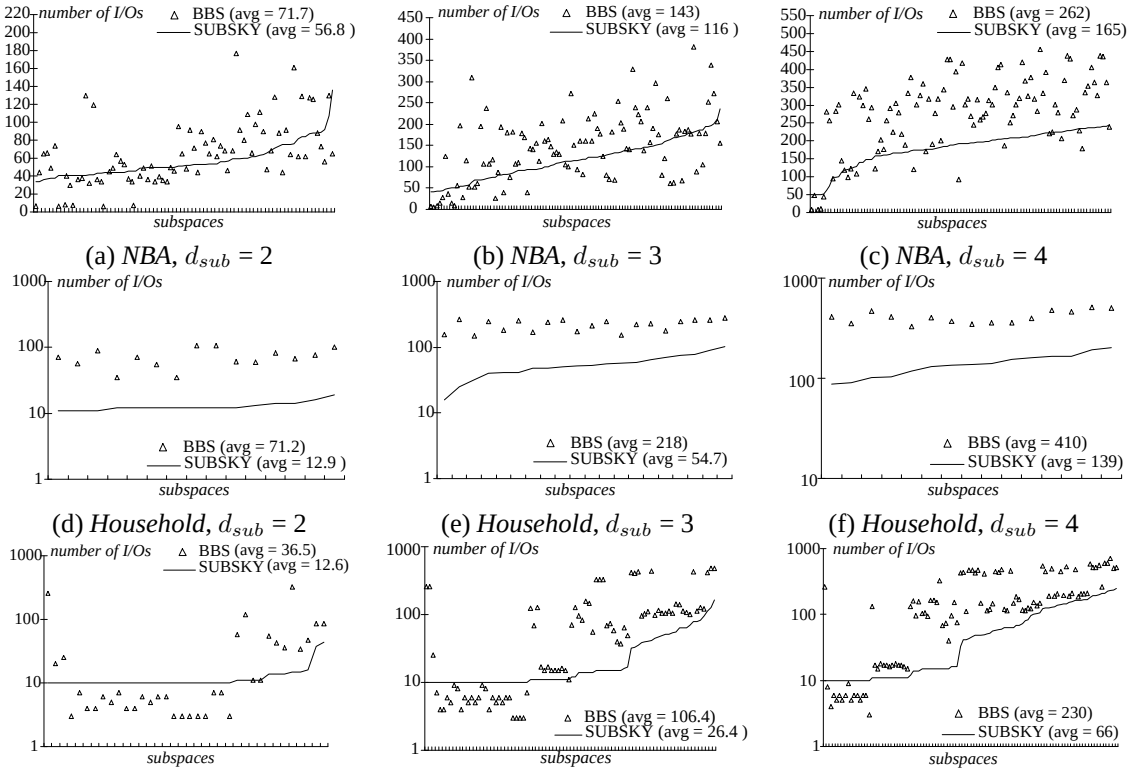
We also generate synthetic data with two distributions. A *uniform* dataset includes random points in a unit space. To create a *clustered* dataset with cardinality N, we first pick 10 cluster centroids randomly. Then, for each centroid, we obtain $N/10$ points whose coordinate on each axis follows a Gaussian distribution with variance 0.05, and a mean equal to the corresponding coordinate of the centroid.

Each *workload* contains 100 queries that request the skylines of 100 random subspaces with the same dimensionality d_{sub}. For example, for *NBA* and $d_{sub} = 3$, each of the $\binom{13}{3}$ three-dimensional subspaces has an equal probability of being queried. If the number of possible subspaces is smaller than 100 (e.g., for *Color*, the number of 3D subspaces equals $\binom{6}{3} = 20$), all the subspaces are inspected. For *NBA* and *Household*, each skyline aims at maximizing the coordinates of the participating dimensions, while queries on the other datasets prefer small coordinates.

We compare *SUBSKY* against *BBS* which is the best existing method for skyline computation (see Section 2). Each dataset is indexed by a B-tree (for *SUBSKY*) and an R-tree (for *BBS*), where the page size is set to 4k bytes in all cases. Each B-tree is constructed with anchors computed (as described in Section 4.3) from a 10% random sample set of the underlying dataset.

Tuning the Number of Anchors. The first set of experiments examines the influence of the number m of anchors on the performance of *SUBSKY*. For each real dataset, we create 11 B-trees by varying m from 1 to 100; then, we use each tree to process a workload, and measure the average (per-query) number of page accesses. Figure 12 plots the cost as a function of m, for workloads with d_{sub} = 2, 3 and 4, respectively. Note that the result for $m = 1$ corresponds to the overhead of the basic *SUBSKY* that uses the maximal corner as the only anchor (Section 3).

As m becomes larger, the query overhead first decreases and then actually increases after m passes certain threshold. The initial decrease confirms the analysis of Section 4 that query efficiency can be improved by using multiple anchors. To explain the performance deterioration, recall that the query algorithm of *SUBSKY* essentially scans m segments of continuous leaf nodes in the B-tree, which require at least m page accesses. For excessively large m, these m accesses constitute a dominant factor in the overall



(a) *NBA*, d_{sub} = 2 (b) *NBA*, d_{sub} = 3 (c) *NBA*, d_{sub} = 4

(d) *Household*, d_{sub} = 2 (e) *Household*, d_{sub} = 3 (f) *Household*, d_{sub} = 4

(g) *Color*, d_{sub} = 2 (h) *Color*, d_{sub} = 3 (i) *Color*, d_{sub} = 4

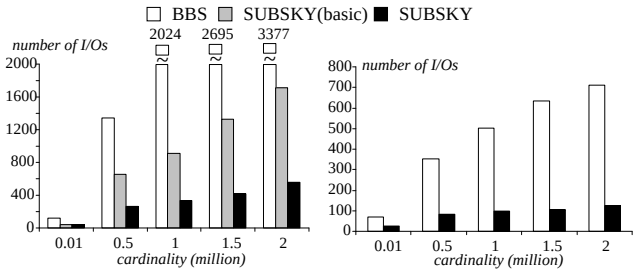
Figure 13. *SUBSKY* **vs.** *BBS* **(real datasets)**

overhead, which thus grows (almost) linearly with m (for *NBA* and $d_{sub} \geq 2$, the linear phenomenon happens after $m \geq 90$).

Even for the same dataset, the optimal m depends on the dimensionality of the subspace queried. Furthermore, the optimal value for a higher d_{sub} is greater than that for a lower d_{sub} (e.g., for *NBA*, the best m equals 30, 80, 90 for d_{sub} = 2, 3, and 4, respectively). This is expected because the number of clusters is larger in a higher-dimensional subspace, as the projection of two clusters onto a lower-dimensional subspace may collapse into a single cluster. In the sequel, we set m to 50, 20, 10 for *NBA*, *Household*, and *Color* which result in the best overall performance for the d_{sub} values tested.

Examination of Individual Subspaces. Figure 13a illustrates the cost of *SUBSKY* and *BBS* for answering each query (in the 2D workload) on the *NBA* dataset. The x-axis represents the subspaces, sorted in ascending order of the corresponding *SUBSKY* overhead. The average cost of each method is shown after its legend (e.g., the per-query overhead of *SUBSKY* equals 71.7 I/Os). In Figures 13b and 13c, we demonstrate a similar comparison for workloads with d_{sub} = 3 and 4, respectively. Figures13d-13i present the results of the same experiments on *Household* and *Color* respectively, except that the y-axes are in logarithmic scale.

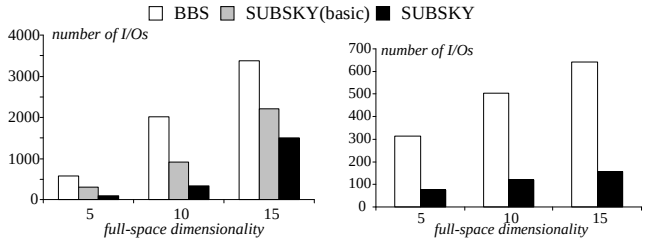
SUBSKY consistently achieves lower average cost than its competitor (with the maximum speedup 5 in Figure 13d). Regarding individual query performance, *SUBSKY* outperforms *BBS* in all queries on *Household*, and most queries on *NBA* and *Color*. The only exception is in Figure 13g, where *BBS* is slightly faster (by less than 5 I/Os) for around 60% of the workload, but significantly slower for the remaining queries, rendering its average overhead 3 times higher than that of *SUBSKY*.



(a) *Uniform* (b) *Clustered*

Figure 14. Cost vs. cardinality (3D subspaces, 10D full-space)

Scalability with the Cardinality and Full-Space Dimensionality. Next, we deploy 10D *uniform* datasets with cardinalities ranging from 10 thousand to 2 million. For each dataset, we decide (for *SUBSKY*) the best number m of anchors that optimize the overall performance for 2-4D subspaces (through a tuning process similar to Figure 12).



(a) *Uniform* (b) *Clustered*

Figure 15. Cost vs. dataset dimensionality (3D subspaces, 1 million cardinality)

Specifically, the best value equals 20 for the dataset containing 10k points, and 70 for the others. Figure 14a compares the average cost of *BBS* and the two versions of *SUBSKY* in executing 3D workloads. Evidently, the proposed methods scale better with cardinality than *BBS*. In particular, for cardinality 2 million, *SUBSKY* outperforms *BBS* by almost an order of magnitude. Figure 14b illustrates the corresponding results for *Clustered* data, confirming similar observations (the m of *SUBSKY* equals 10 for the smallest dataset, and 30 for the other cardinalities). The basic *SUBSKY* is omitted because it targets specifically uniform data.

To examine the influence of the full-space dimensionality d_{full}, we use datasets with cardinality 1 million whose d_{full} varies from 5 to 15. In Figure 15, we measure the cost of each method (in retrieving skylines of 3D subspaces) as a function of d_{full}, for *uniform* and *clustered* distributions. *SUBSKY* again outperforms *BBS* significantly.

6 Conclusions

In practice, skyline queries are usually issued in a large number of subspaces, each of which includes a small subset of the attributes in the underlying relation. In this paper, we develop a new technique *SUBSKY* that supports subspace skyline retrieval with only relational technologies. The core of *SUBSKY* is a transformation that converts multi-dimensional data to 1D values, and permits indexing the dataset with a single conventional B-tree. Extensive experiments verify that *SUBSKY* consistently outperforms the previous solutions in terms of efficiency and scalability.

This work also lays down a foundation for future investigation of several related topics. For instance, certain attributes in the relation may appear in the subspaces of most queries (e.g., a user looking for a good hotel would always be interested in the *price* dimension). In this case, the data structure may be modified to facilitate pruning on these axes. Another interesting issue is to cope with datasets where the data distribution may incur frequent changes. Instead of periodically reconstructing the B-tree, a better approach is to replace only some anchors, and re-organize the data assigned to them. This strategy achieves lower update cost since it avoids accessing the points assigned to the unaffected anchors.

Acknowledgements

Yufei Tao and Xiaokui Xiao were supported by RGC Grant CityU 1163/04E from the HKSAR government. Jian Pei was supported by an NSERC Discovery Grant and NSF Grant IIS-0308001.

References

[1] W.-T. Balke, U. Guntzer, and J. X. Zheng. Efficient distributed skylining for web information systems. In *EDBT*, pages 256–273, 2004.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[3] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree : An index structure for high-dimensional data. In *VLDB*, pages 28–39, 1996.

[4] S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[5] C.-Y. Chan, P.-K. Eng, and K.-L. Tan. Stratified computation of skylines with partially-ordered domains. In *SIGMOD*, pages 203–214, 2005.

[6] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *ICDE*, pages 717–719, 2003.

[7] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis*. Wiley, 1973.

[8] P. Godfrey. Skyline cardinality for relational processing. In *FoIKS*, pages 78–97, 2004.

[9] P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In *VLDB*, pages 229–240, 2005.

[10] 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*, pages 152–159, 1996.

[11] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *TODS*, 24(2):265–318, 1999.

[12] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB*, pages 275–286, 2002.

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

[14] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *TODS*, 30(1):41–82, 2005.

[15] J. Pei, W. Jin, M. Ester, and Y. Tao. Catching the best views of skyline: a semantic approach based on decisive subspaces. In *VLDB*, pages 253–264. VLDB Endowment, 2005.

[16] D. Pelleg and A. W. Moore. X-means: Extending k-means with efficient estimation of the number of clusters. In *International Conference on Machine Learning*, pages 727–734, 2000.

[17] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD*, pages 71–79, 1995.

[18] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *VLDB*, pages 301–310, 2001.

[19] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang. Efficient computation of the skyline cube. In *VLDB*, pages 241–252, 2005.