To appear in IEEE TKDE

Title: Efficient Skyline and Top-k Retrieval in Subspaces
Keywords: Skyline, Top-k, Subspace, B-tree

Contact Author:
Yufei Tao (*taoyf@cse.cuhk.edu.hk*)
Department of Computer Science and Engineering
Chinese University of Hong Kong
Sha Tin, Hong Kong
Tel: +852-26098437 Fax: +852-26035024

Efficient Skyline and Top-k Retrieval in Subspaces

Yufei Tao[†] Xiaokui Xiao[†] Jian Pei[‡]

[†]Department of Computer Science and Engineering
Chinese University of Hong Kong
Sha Tin, Hong Kong
{*taoyf, xkxiao*}*@cse.cuhk.edu.hk*

[‡]School of Computing
Simon Fraser University
Burnaby, BC Canada V5A 1S6
jpei@cs.sfu.ca

Abstract

Skyline and top-k queries are two popular operations for preference retrieval. In practice, applications that require these operations usually provide numerous candidate attributes, whereas, depending on their interests, users may issue queries regarding different subsets of the dimensions. The existing algorithms are inadequate for subspace skyline/top-k search because they have at least one of the following defects: they (i) require scanning the entire database at least once; (ii) are optimized for one subspace but incur significant overhead for other subspaces; (iii) demand expensive maintenance cost or space consumption.

In this paper, we propose a technique, *SUBSKY*, which settles both types of queries using purely relational technologies. The core of *SUBSKY* is a transformation that converts multidimensional data to 1D values. These values are indexed by a simple B-tree, which allows us to answer subspace queries by accessing a fraction of the database. *SUBSKY* entails low maintenance overhead, which equals the cost of updating a traditional B-tree. Extensive experiments with real data confirm that our technique outperforms alternative solutions significantly in both efficiency and scalability.

1 Introduction

A multidimensional point p *dominates* another p', if the coordinate of p on each axis does not exceed that of p', and is strictly smaller on at least one dimension. Given a set of points, the *skyline* consists of all the points that are not dominated by others. Figure 1 shows a dataset with dimensionality d = 2. 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. The skyline includes p_1, p_4, and p_5, which offer various tradeoffs 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.

The notion of skyline is generalized to "skyband" in [22]. Specifically, the k-*skyband* of a dataset includes all the points that are dominated by less than k points. For instance, the 2-skyband of the dataset in Figure 1 contains all the objects except p_7 and p_8. Clearly, the skyline is the 1-skyband. In general, for any k and k' satisfying $k' < k$, the k'-skyband is a subset of the k-skyband.

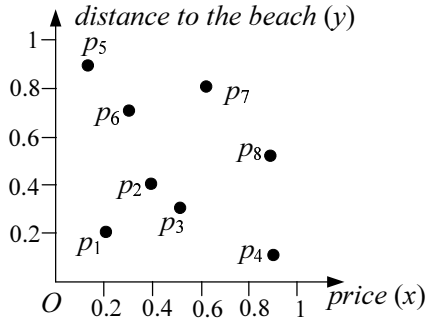


Figure 1: A dataset with hotel records

Skylines have been extensively studied in the literature, due to their close relationship to preference search. A preference is usually formulated through a monotone[1] *preference function* g, which returns a *score* $g(p)$ for every point p. Given such a function, a *top-k query* retrieves the k points in a dataset with the lowest scores. For example, for $g(p) = 3p[x] + p[y]$, the top-1 hotel in Figure 1 is p_1 (score 0.8). Regardless of the choice of g, the top-1 object always lies in the skyline. Furthermore, every skyline point is the top-1 object for a certain function (i.e., a skyline does not contain any redundant point for top-1 search [5]). Similarly, the k-skyband contains *all and only* the objects retrieved by top-k queries.

The motivation of this work is that, in practice, a skyline/preference search application typically provides numerous candidate attributes, whereas a user chooses only a small number of them in her/his query. Assume that, in addition to the dimensions in Figure 1, the database also stores 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. It is unlikely that a customer would consider all the attributes in selecting her/his hotel. Instead, s/he would take into account only some of them, i.e., a *subspace* of the universe. Alternative customers may have different concerns. Therefore, the system must be prepared to perform skyline/top-k retrieval in a variety of subspaces. Unfortunately, this observation has been ignored by the previous research. As discussed later, the existing skyline/top-k algorithms are optimized for the whole universe, but entail expensive cost for subspace queries.

This paper presents the first study on indexes for efficient skyline and top-k computation in arbitrary subspaces. We develop *SUBSKY*, a novel technique that settles both problems using purely

[1]Monotonicity means $g(p) > g(p')$ for two arbitrary points p and p', which share the same coordinates on $d-1$ dimensions, and p has a larger coordinate on the remaining dimension.

relational technologies, and hence, can be incorporated into a conventional database system immediately[2]. The core of *SUBSKY* is a transformation that converts each multidimensional point to a 1D value. The converted values are indexed by a B-tree, which can be used to handle all types (i.e., skyline, skyband, top-k) of queries effectively. In the presence of tuple insertions/deletions, the tree can be maintained at the same cost of updating a traditional B-tree. Extensive experiments confirm that the proposed solutions significantly outperform the state-of-the-art skyline/top-k algorithms.

The rest of the paper is organized as follows. Section 2 reviews the previous work related to ours. Section 3 adapts the existing algorithms for subspace skyline/top-k processing, and elaborates their deficiencies. Section 4 presents the basic *SUBSKY* optimized for skyline search on uniform data, and Section 5 generalizes the technique to arbitrary data distributions. Section 6 discusses skyband and top-k processing. Section 7 contains an experimental evaluation that demonstrates the efficiency of *SUBSKY*. Section 8 concludes the paper with directions for future work.

2 Related Work

Section 2.1 surveys the algorithms for computing skylines in the whole universe. Then, Section 2.2 discusses the "sky-cube" that is highly relevant to subspace skylines. Finally, Section 2.3 reviews the previous work on top-k search.

2.1 Skyline Retrieval in the Universe

The existing algorithms can be classified in two categories. The first one involves solutions that do not assume any preprocessing on the underlying dataset, but they retrieve the skyline by scanning the entire database at least once. The second category reduces query cost by utilizing an index structure. In the sequel, we survey both categories, focusing on the second one, since it also involves our solutions.

Algorithms Requiring No Preprocessing. The first skyline algorithm in the database context is *BNL* (block-nested-loop) [5], which simply inspects all pairs of points, and returns an object if it is not dominated by any other object. *SFS* [10] (sort-filter-skyline) is based on the same rationale, but improves the performance by sorting the data according to a monotone function. The performance

[2]Including a non-relational method into a commercial DBMS is difficult, because it requires fixing complex issues related to concurrency control, recovery, etc.

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

(a) The sorted lists used by *Index*

List 1 (x)	p_5:0.2	p_1:0.2	p_6:0.3	p_2:0.4	p_3:0.5	p_7:0.6	p_8:0.9	p_9:0.9
List 2 (y)	p_4:0.1	p_1:0.2	p_3:0.3	p_2:0.4	p_8:0.5	p_6:0.7	p_7:0.8	p_5:0.9

(b) The sorted lists used by *TA*

Figure 2: Illustration of algorithms leveraging sorted lists

of *BNL* and *SFS* is analyzed in [25]. *D&C* [5] (divide-and-conquer) divides the universe into several regions, calculates the skyline in each region, and produces the final skyline from the regional skylines. When the entire dataset fits in memory, this algorithm produces the skyline in $O(n \log^{d-2} n + n \log n)$ time, where n is the dataset cardinality and d its dimensionality. *Bitmap* [26] 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) [14] is an algorithm that has good worst-case asymptotical performance. Specifically, when the data distribution is uniform and no two points have the same coordinate on any dimension, *LESS* computes the skyline in $O(d \cdot n)$ time in expectation.

Algorithms Based on Sorted Lists. *Index* [26] organizes the dataset into d lists. The i-th list ($1 \leq i \leq d$) contains points p with the following property: $p[i] = \min_{j=1}^{d} p[j]$, where $p[i]$ is the i-th coordinate of p. Figure 2a 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 has identical coordinates on both dimensions, the list containing it is decided arbitrarily (in Figure 2a, p_2 and p_1 are randomly assigned to 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 pointers ptr_1 and ptr_2 referencing the first entries p_5, p_4, respectively. Then, at each step, *Index* processes the referenced entry with a smaller sorting key. Since both p_5 and p_4 have the same key 0.1, *Index* randomly picks one for processing. Assume that p_5 is selected; it is added to the skyline S_{sky}, after which ptr_1 is moved to p_6. As p_4 has a smaller key (than p_6), it is the second point processed. p_4 is not dominated by any point in S_{sky}, and hence, is inserted in S_{sky}. Pointer ptr_2 is then shifted to p_1. Similarly, p_1 is processed next, and included in the skyline,

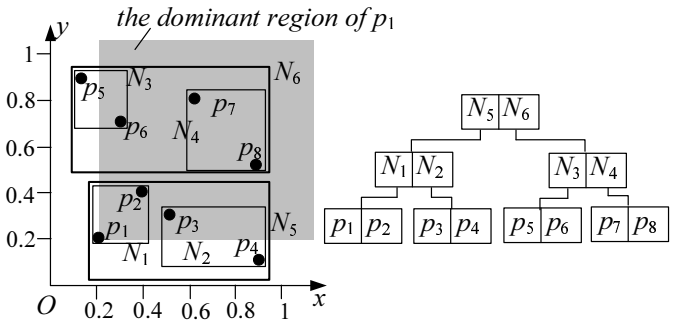


Figure 3: Illustration of *BBS*

after which ptr_2 is set to p_3. At this stage, S_{sky} = $\{p_1, p_4, p_5\}$.

Both coordinates of p_1 are smaller than the x-coordinate 0.3 of p_6 (referenced by ptr_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 is also eliminated because both coordinates of p_1 are lower than the y-coordinate of p_3 (referenced by ptr_2). The algorithm finishes with $\{p_1, p_4, p_5\}$ as the result.

Borzsonyi et al. [5] develop an algorithm *TA*[3] that deploys a different set of sorted lists. For a d-dimensional dataset, the i-th list ($1 \leq i \leq d$) enumerates all the objects in ascending order of their i-th coordinates. Figure 2b demonstrates the two lists for the dataset in Figure 1. *TA* scans the d lists synchronously, and stops as soon as the same object has been encountered in all lists. For instance, assume that *TA* accesses the two lists in Figure 2b in a round-robin manner. It terminates the scanning after seeing p_1 in both lists. At this moment, it has retrieved p_5, p_4 and p_1. Clearly, if a point p has not been fetched so far, p must be dominated by p_1, and thus, can be safely removed from further consideration. On the other hand, p_5, p_4 and p_1 may or may not be in the skyline. To verify this, TA obtains the y-coordinate of p_5 (notice that the scanning discovered only its x-coordinate), and x-coordinate of p_4. Then, it computes the skyline from $\{p_5, p_4, p_1\}$, which is returned as the final skyline.

Algorithms Based on R-trees. *NN* (nearest-neighbor) [18] and *BBS* (branch-and-bound skyline) [22] find the skyline using an R-tree [2]. The difference is that *NN* issues multiple NN queries [15] while *BBS* performs only a single traversal of the tree. It has been proved [22] that *BBS* is I/O optimal, i.e., it accesses the least number of disk pages among all algorithms based on R-trees

[3]This algorithm is called *B-tree* in [5]. We refer to it as *TA* to emphasize its connection to Fagin's threshold algorithm [12].

(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 universe. At the beginning, the root entries are inserted into a min-heap H (= $\{N_5, N_6\}$) using their mindist as the sorting key. Then, the algorithm removes the top element N_5 of H, accesses its child node, and en-heaps 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 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. Following the same rationale, the remaining entries processed are N_3 (en-heaping p_5), p_4, p_5, at which point H becomes empty and *BBS* terminates.

Retrieval of Skyline Variants. Balke et al. [1] consider skyline computation in distributed environments. Also in the distributed framework, the work of Huang et al. [17] studies skyline search on mobile objects. Lin et al. [19] investigate continuous skyline monitoring on data streams. In [6], skylines are extended to partially-ordered domains. Chan et al. [8, 7] propose various approaches to improve the usefulness of skylines in high-dimensional spaces. The above methods, however, are restricted to their specific scenarios, and cannot be adapted to the problem of this paper.

2.2 The Sky-Cube

Pei et al. [23] and Yuan et al. [31] independently propose the *sky-cube*, which consists of the skylines in *all* possible subspaces. In the sequel, we explain this concept, assuming that no two points have the same coordinate on any axis (see [23, 31] for a general discussion overcoming the assumption).

Suppose that the universe has $d = 3$ dimensions x, y, and z. Figure 4 shows the 7 possible non-empty subspaces. All the points in the skyline of a subspace belong to the skyline of a subspace containing additional dimensions. For instance, the skyline of subspace xy is a subset of the skyline

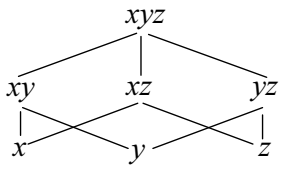


Figure 4: The lattice of skycube

in the universe, represented by an edge between xy and xyz in Figure 4. Specially, if a subspace involves only a single dimension, its skyline consists of the point having the smallest coordinate on this axis.

The skycube can be computed in a top-down manner. First, we retrieve the skyline of the universe. Then, a child skyline can be found by applying a conventional algorithm on a parent skyline (instead of the original database). For example, the skyline of xyz can produce those of xy, xz, and yz, while the skyline of x can be obtained from that of either xy or xz. To reduce cost, several heuristics are proposed in [23, 31] to avoid the common computation in different subspaces. Xia and Zhang [29] explain how to dynamically maintain a skycube, after the underlying database has been updated.

2.3 Top-k Search in the Universe

There is a bulk of research on distributed top-k processing (see [20] and the references therein). In that scenario, the data on each dimension is stored at a different server; the goal is to find the top-k objects with the least network communication. Our work falls in the category of centralized top-k search, where all the dimensions are retained at the same server, and the objective is to minimize queries' CPU and I/O cost. Next, we concentrate on this category.

Chang et al. [9] develop ONION, which answers only top-k queries with *linear* preference functions. Hristids and Papakonstantinou [16] propose the PREFER system, which supports a broader class of preference functions, but requires duplicating the database several times. Yi et al. [30] suggest a similar approach with lower maintenance cost. Tsaparas et al. [28] present a technique that can handle arbitrary preference functions. This technique, however, is limited to two dimensions, and supports only top-k queries whose k does not exceed a certain constant. The state-of-the-art solution [27, 28] is based on "best-first traversal" [15] on an R-tree. It enables top-k queries with any k and monotone preference function, on data of arbitrary dimensionality.

3 Extending the Previous Algorithms to Subspaces

Without loss of generality, we assume a d-dimensional universe where each axis has domain [0, 1]. Given a point p, $p[i]$ denotes its coordinate on the i-th dimension ($1 \leq i \leq d$). *BNL*, *SFS*, *D&C*, *Bitmap*, and *LESS* (in general, any algorithm that does not demand preprocessing) can be trivially extended to compute the skyline in a subspace, by ignoring the irrelevant coordinates of each point. However, they entail expensive cost by scanning the entire dataset multiple times.

Index can be adapted for subspace skyline retrieval, by re-constructing the underlying data structure for every query. Assume, for example, $d = 10$. As mentioned in Section 2.1, the preprocessing of *Index* creates ten sorted lists L_1, ..., and L_{10}, where L_i ($1 \leq i \leq 10$) includes all points p satisfying $p[i] = \min\{p[1], ..., p[10]\}$. Consider a query that aims at finding the skyline in the first two dimensions. To apply *Index*, we must organize the database into two sorted lists L'_1 and L'_2, where L'_i ($1 \leq i \leq 2$) contains all points p such that $p[i] = \min\{p[1], p[2]\}$. Notice that, the pre-computed L_1, ..., L_{10} provide little help for deriving L'_1 and L'_2. The most efficient way to calculate L'_1 and L'_2 would ignore the pre-computed lists, scan the database once to assign each object to L'_1 or L'_2, and then perform two external sorts to obtain the proper order in the two lists.

TA is directly applicable to subspace computation, by operating on only the sorted lists corresponding to the axes of the target subspace. Unfortunately, this technique is inappropriate for dynamic datasets, because its sorted lists are costly to maintain. Recall that, every object has an entry in each of the d sorted lists. Since a list is organized with a B-tree [5], a single tuple insertion/deletion requires modifying d B-trees, which is prohibitively expensive for large d.

NN and *BBS* can find a subspace skyline by ignoring the extents of an MBR along the irrelevant dimensions. It suffices to discuss only *BBS* since *NN* is always slower. Imagine the rectangles in Figure 3 as the projections, in the 2D subspace demonstrated, of the MBRs in a d-dimensional R-tree for any $d > 2$. *BBS* retrieves the skyline of the subspace in exactly the same way as described in Section 2.1. The performance of *BBS*, however, severely degrades when the dimensionality d increases, due to two reasons. First, as d grows, MBRs become considerably larger, which significantly decreases the probability that an MBR falls in the dominant regions of the skyline points (recall that this is the pruning condition of *BBS*).

The other (less obvious) reason is the emergence of the "random grouping" phenomenon, after

d exceeds a certain threshold. Consider, for example, a 15D uniform dataset with 100k points. Given a node capacity of 100 entries, the R-tree would contain approximately 1000 leaf nodes. Since $1000 \approx 2^{10}$, each leaf MBR has been split once along roughly 10 dimensions, while the remaining 5 axes are not considered at all in the R-tree construction [3, 4]. Assume that we use the R-tree to retrieve the skyline in a 2D subspace including any two of those 5 dimensions. The expected performance is as poor as deploying a pathological 2D R-tree, which groups the points (in *that* 2D subspace) into leaf nodes in a completely random manner, regardless of their spatial proximity.

Finally, the skycube algorithm discussed in Section 2.2 is not suitable for retrieving the skyline in *one* subspace, because it performs unnecessary work by fetching skylines in many non-requested subspaces. An alternative approach is to pre-compute the entire skycube. In that case, although the skyline of any subspace can be obtained immediately, the skycube occupies significant space, and incurs expensive update cost (whenever the original database is updated).

In summary, the existing approaches are inadequate for subspace skyline search because they have at least one of the following defects: they (i) require scanning the entire database at least once; (ii) are optimized for one subspace but incur significant overhead for other subspaces; (iii) demand expensive maintenance cost or space consumption. In the following sections, we remedy all defects by proposing a new technique.

4 The Basic SUBSKY

We use the term *maximal corner* for the corner A^C of the universe 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:

$$f(p) = L_\infty(p, A^C) = \max_{i=1}^{d}(1 - p[i]). \quad (1)$$

Point p dominates all points p' satisfying the following inequality:

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

Figure 5 illustrates a 2D example. Points p' obeying the inequality constitute the shaded square, whose side length equals $\min_{i=1}^{2}(1-p[i])$. Obviously, no such p' can appear in the skyline because the square is entirely contained in the dominant region of p.

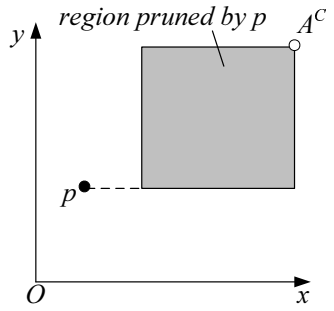


Figure 5: Illustration of Inequality 2

Inequality 2 applies to the whole universe, while a similar result exists in 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 the universe, then SUB = {1, 3}), we have:

Property 1. *Given an arbitrary point p, no point p' qualifying the following condition can belong to the skyline of SUB:*

$$f(p') < \min_{i \in SUB}(1 - p[i]) \tag{3}$$

For example, assume $d = 3$, and that the goal is to retrieve the skyline in SUB = {1, 2}. If p has coordinates (0.05, 0.1, –) (the 3rd coordinate is irrelevant), no point p' with $f(p') < \min(1 - 0.05, 1 - 0.1) = 0.9$ can be in the target skyline. This is correct because the coordinates of p' must be at least 0.1 on all dimensions; hence, p' is dominated by p in SUB.

Property 1 leads to an algorithm, referred to as the *basic SUBSKY*, for computing the skyline in 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 equal to the largest $\min_{i \in SUB}(1 - p[i])$ of the objects $p \in S_{sky}$. The algorithm terminates when U exceeds the $f(p)$ of the next p to be processed. The pseudocode of the basic *SUBSKY* is given in Figure 6.

We illustrate the basic *SUBSKY* using the 8 three-dimensional points in Figure 7 (the x- and y-coordinates are the same as in Figure 1), where the last row indicates the f-value of each point. The query subspace SUB is $\{1, 2\}$. Objects are processed in this order: $\{p_3, p_4, p_5, p_1, p_6, p_2, p_8, p_7\}$. After examining p_3, *SUBSKY* initializes S_{sky} as $\{p_3\}$, and U as $\min_{i \in SUB}(1 - p_3[i]) = 0.5$. After the second point p_4 is inspected, S_{sky} becomes $\{p_3, p_4\}$, since p_4 is not dominated by p_3; U remains 0.5, because $0.5 > \min_{i \in SUB}(1 - p_4[i]) = 0.1$. Similarly, next p_5 is added to S_{sky} without

Algorithm BASIC-SUBSKY (SUB)
/* SUB includes the dimensions relevant to the query subspace */
1. DB = the given set of data points sorted in descending order of their f-values
2. U = 0; $S_{sky} = \emptyset$
3. p = the first point in DB
4. while $p \neq \emptyset$ and $U \leq f(p)$ do
5. if p is not dominated by any point in DB
6. add p to S_{sky}
7. remove from S_{sky} the points dominated by p
8. U = the maximum $\min_{i \in SUB}(1 - p'[i])$ of all $p' \in S_{sky}$
9. set p to the next point in DB
10. return S_{sky}

Figure 6: The basic *SUBSKY* algorithm

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 7: An example dataset

affecting U. To handle the 4th point p_1, we insert it in S_{sky}, but remove p_3 from S_{sky}, as p_3 is dominated by p_1. Moreover, U is increased to $\min_{i \in SUB}(1 - p_1[i]) = 0.8$. The algorithm proceeds to inspect p_6, which does not change S_{sky} and 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.

As shown in the experiments, despite its simplicity, the basic *SUBSKY* is highly efficient in computing subspace skylines, when the data distribution is uniform. Next we provide the intuition, under the same settings as used in Section 3 to illustrate the defects of *BBS*. Consider a 15D uniform dataset with cardinality 100k. Assume that we want to retrieve the skyline in a subspace SUB containing any two dimensions, as shown in Figure 8. There is a high chance that a skyline

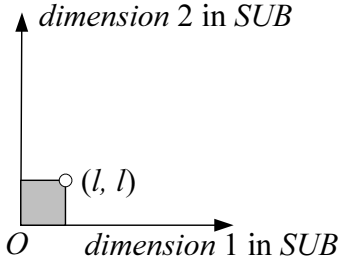


Figure 8: Illustration of the analysis

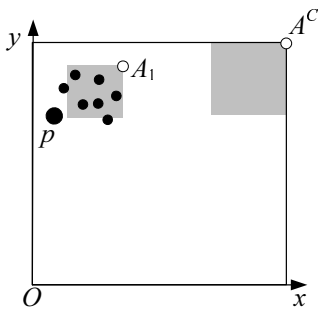


Figure 9: Pruning effects of different anchors

point lies very close to the origin in SUB. Specifically, let us examine the square in Figure 8 whose lower-left corner is the origin, and its side length equals some small $\lambda \in [0, 1]$. The probability of *not* having any object in the square equals $(1 - \lambda^2)^{100000}$, which is less than 10% for $\lambda = 0.001$. In other words, with at least 90% chance, we can find a point p in the square such that $\min_{i \in SUB}(1-p[i]) \geq 0.999$. In this case, (by Property 1) all points p' with $f(p') < 0.999$ are eliminated by *SUBSKY*. The expected percentage of such points in the whole dataset equals the volume of a 15-dimensional square with side length 0.999. The volume evaluates to $0.999^{15} = 98.5\%$, that is, we only need to access 1.5% of the dataset!

5 The General SUBSKY

In the basic *SUBSKY*, $f(p)$ is always computed using one *anchor*, i.e., the maximal corner A^C. This works fine for uniform data. In practice where data is clustered, however, the $f(p)$ of various p should be calculated with respect to different anchors to achieve greater pruning power.

To illustrate this, Figure 9 shows a 2D dataset, where all objects gather around the upper-left corner. In the basic *SUBSKY*, (by Property 1) point p prunes the right shaded square, which is useless since the square does not cover any object. Alternatively, let us compute $f(p)$ as the L_∞ distance from p to another anchor A_1. As a direct corollary of Property 1, we can eliminate all points p' whose $f(p')$ is smaller than $\min_{i=1}^{2}(A_1[i]-p[i])$. These points form the left shaded square, which encloses a significant portion of the dataset. Namely, A_1 offers stronger pruning power than A^C.

Based on this idea, in Sections 5.1-5.4, we develop the general version of *SUBSKY*. Finally, Section 5.5 discusses issues related to updates and other "preference directions".

5.1 Pruning with Multiple Anchors

Given a dataset, we will compute a set S_{anc} of anchors A_1, A_2, ..., A_m. Then, every data point p is converted to a 1D value as follows.

Definition 1. *Each data point p is converted to a value $f(p) = L_\infty(p, A)$, where A belongs to S_{anc}, and is called the* **assigned anchor** *of p.*

Next, we formalize our pruning heuristic based on multiple anchors.

Property 2. *Let p be an arbitrary object, and S'_{anc} the set of anchors whose projections in supspace SUB are dominated by p. Then, for each anchor $A \in S'_{anc}$, an object p' assigned to A cannot be in the skyline if*

$$f(p') < \min_{i \in SUB} (A[i] - p[i]) \tag{4}$$

The above result degenerates to Property 1 when $A = A^C$. To explain the case where $A \neq A^C$, consider $d = 3$, and A = (0.8, 0.7, 0.1). In subspace SUB = {1, 2}, an object p = (0.2, 0.2, –) eliminates all points p' assigned to A with $f(p') < \min(A[1] - p[1], A[2] - p[2]) = 0.5$. 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 in SUB.

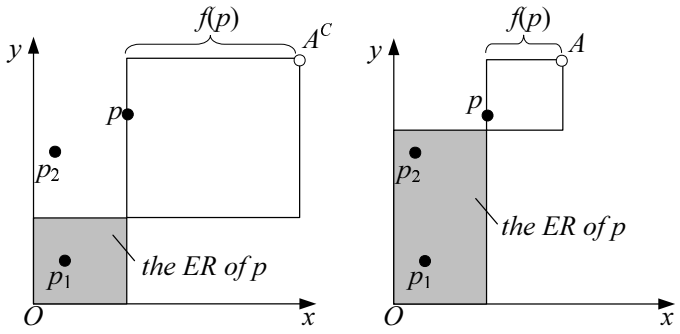
The pruning effectiveness of Property 2 depends on (i) how data is assigned to anchors, and (ii) how the anchors are selected. We analyze these issues in the next two subsections, respectively.

5.2 Assigning Points to Anchors

We introduce the concept of *effective region*.

Definition 2. *Given a data point p and an anchor A dominated by p, the* **effective region** *(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 ($1 \leq i \leq d$). The ER does not exist, if $A[i] < L_\infty(p, A)$ for any $i \in [1, d]$.*

ERs are closely related to the benefit of assigning a point to an anchor. To understand this, consider Figure 10a where d = 2, and point p is assigned to A^C. As a result, in finding the skyline in the



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

Figure 10: The concept of effective region

universe, p can be eliminated with Property 2, if and only if another point is discovered in the shaded square, which is the ER of p with respect to A^C. For example, since p_1 (p_2) is inside (outside) the ER, we can (cannot) eliminate p after encountering p_1 (p_2). Let us assign p to an alternative anchor A in Figure 10b. The shaded area demonstrates the new ER of p, which covers both p_1 and p_2. 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.

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

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

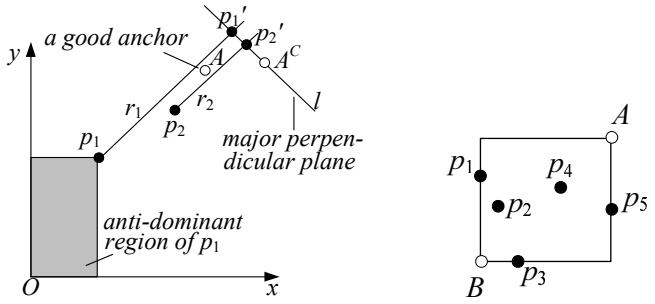
5.3 Finding the Anchors

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 integer, set as a system parameter), how should they be selected in order to maximize the ER volumes of as many points as possible?

Notice that the largest ER of a point p corresponds to its *anti-dominant region*, consisting of all the points in the universe dominating p. These points form a rectangle that has the origin and p as its opposite corners. In other words, the maximum value of Formula 5 equals $\Pi_{i=1}^{d} p[i]$, which is achieved when

$$A[i] - L_\infty(p, A) = p[i] \tag{6}$$

holds on all dimensions $i \in [1, d]$. We refer to an anchor A satisfying the above equation as a *perfect anchor* for p.



(a) Perfect rays (b) Deciding the anchor

Figure 11: Finding anchors

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. This is because the coordinate difference between A and p is equivalent on all axes, i.e., $A[i] - p[i] = L_\infty(p, A)$ for any $i \in [1, d]$, thus establishing Equation 6.

In Figure 11a, 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 (i.e., the anti-dominant region of p_1). 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 11a. 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 all objects onto the *major perpendicular plane*, i.e., the d-dimensional plane that passes the maximal corner, and is perpendicular to the major diagonal of the universe. In Figure 11a ($d = 2$), 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 [11, 24], and formulate an anchor for each cluster.

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 suggested in Definition 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 objects p_1, p_2, ..., p_5. The algorithm examines them in the *original* universe (i.e., not in the major perpendicular plane), as shown in Figure 11b. 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, we compute the smallest square that covers all the points in S (see Figure 11b). The anchor A for S is the corner of the square opposite to B.

5.4 The Data Structure and Query Algorithm

We are ready to clarify the details of our *SUBSKY* technique. Given a small number m (less than 100 in our experiments), *SUBSKY* first obtains m anchors, by applying the method in Section 5.3 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 volume of ER among the anchors dominated by p). 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 7 and $m = 2$ anchors: the maximal corner A_1 (= A_C), and A_2 = (1, 1, 0.8). The second row of Figure 13a illustrates the ER volume of each data point with respect to A_1, calculated by Equation 5. For instance, the volume 125 ($\times 10^{-3}$) of p_8 is derived from $\Pi_{i=1}^{3}(A_1[i] - L_\infty(A_1, p_8))$ = $(1 - 0.5)^3$. 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 Definition 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 13b 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.

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 a B-tree to find the point ptr_j with the maximum $f(ptr_j)$ among all the points assigned to A_j
3. $ptr_j.ER = \prod_{i=1}^{d}(A_j[i] - L_\infty(ptr_j, A_j))$ //Equation 5
4. $S_{sky} = \emptyset$ //the set of skyline points
5. while ($ptr_j \neq \emptyset$ for any $j \in [1, m]$)
6. t = the value of j giving the smallest $ptr_j.ER$ among all $j \in [1, m]$ such that $ptr_j \neq \emptyset$
7. if ptr_t is not dominated by any point in S_{sky}
8. remove from S_{sky} the points dominated by ptr_t
9. $S_{sky} = S_{sky} \cup \{ptr_t\}$
10. for j = 1 to m, and $j \neq t$
11. if $ptr_j \neq \emptyset$ and $f(ptr_j) < \min_{i \in SUB}(A_j[i] - ptr_t[i])$
12. $ptr_j = \emptyset$ /* no point assigned to A_j can belong to the skyline (Property 2) */
13. ptr_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 ptr_t, or a neighboring node)
14. if $ptr_t \neq \emptyset$
15. for every point $p_{sky} \in S_{sky}$
16. if $f(ptr_t) < \min_{i \in SUB}(A_t[i] - p_{sky}[i])$
17. $ptr_t = \emptyset$ /* no point assigned to A_t can belong to the skyline (Property 2) */
18. $ptr_t.ER = \prod_{i=1}^{d}(A_t[i] - L_\infty(ptr_t, A_t))$
19. return S_{sky}

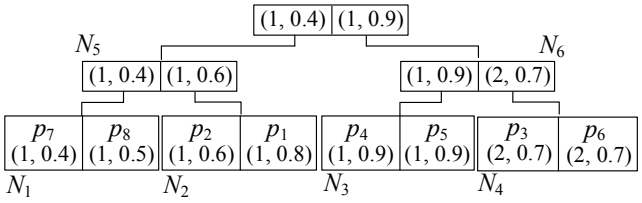
Figure 12: The algorithm of finding a subspace skyline

Figure 12 formally describes the query algorithm of *SUBSKY*. At a high level, *SUBSKY* divides the dataset into m lists, such that the i-th ($1 \leq i \leq m$) list contains all the points assigned to anchor A_i, sorted in descending order of their f-values. Given a query subspace SUB, the algorithm scans the m lists in a synchronous manner. Initially, pointers ptr_1, ptr_2, ..., ptr_m are positioned at the first elements of the m lists, respectively. The subsequent execution runs in iterations, until all the pointers have become $\emptyset$. In each iteration, *SUBSKY* processes the point p that has the smallest ER, among all the points currently referenced by the m pointers. Specifically, it first updates the skyline set S_{sky} (i.e., whenever necessary, add p to S_{sky} and remove the points from S_{sky} dominated by p). Then, the algorithm checks whether a list can be eliminated with p, according to Property 2. Once a list is pruned, its corresponding pointer is set to $\emptyset$. Afterwards, the pointer referencing p is advanced to the next point, and another iteration starts.

As an example, assume that we want to compute 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 13b, the point for A_1 (A_2) is p_6 (p_5), which is the right-most point assigned to this

	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 13: Illustration of the skyline algorithm

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. Initially, ptr_1 and ptr_2 reference 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 a 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 ptr_1 to the next point p_4 in the list of A_1. Since p_4 has a lower ER-volume (than p_6 pointed to by ptr_2) and is not dominated by p_5, it is also added to S_{sky} (=$\{p_5, p_4\}$). Pointer ptr_1 now reaches p_1, which is processed next, and is included in S_{sky}, too.

According to Property 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 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} =$\{p_5, p_4, p_1\}$.

5.5 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 5.2, and set $f(p)$ = $L_\infty(p, A)$, after which the insertion/deletion proceeds as in a normal B-tree. 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.

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

skyline after a skyline point is deleted. On the other hand, if the dataset is static, points that are not in the skyline of the whole universe can be discarded since, as mentioned in Section 2.1, they will not appear in the skyline of any subspace[4]. When d is large, the size of the full-space skyline may still be comparable to the dataset cardinality [13]. 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 attributes *price* and *size* of houses, 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 converting it to a negative direction, which can be achieved by subtracting (from 1) all coordinates on the corresponding dimension (e.g., $1-$ *price*).

It is worth mentioning that, sometimes a dimension may have two preference directions. For example, consider the attribute *nearest-subway-station-distance* of properties. People, who travel with the subway frequently, may prefer to minimize this attribute. Others, who are seeking quiet neighborhoods, may prefer to maximize it. In this case, two instances of *SUBSKY* may be maintained, each supporting one preference direction.

6 Extensions of SUBSKY

In the sequel, we show that SUBSKY can be adapted to perform other types of search in subspaces efficiently. Section 6.1 first elaborates this for skyband queries, and then Section 6.2 discusses top-k processing.

[4]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 [23].

6.1 Subspace Skyband Retrieval

As mentioned in Section 1, the k-*skyband* of a dataset consists of all the data points that are dominated by less than k other points. *SUBSKY* can be easily modified to find the k-skyband in any subspace SUB. In terms of theoretical reasoning, the modification lies in Property 2: a point p' cannot appear in the k-skyband, if there are k points p_1, ..., p_k, such that each p_j ($1 \leq j \leq k$) satisfies Inequality 4, replacing p with p_j. This observation implies that a subspace k-skyband can be extracted using the B-tree deployed by *SUBSKY*, in a way similar to finding a subspace skyline. Intuitively, the only difference is that, here, a sorted list can be eliminated, only after its remaining points are guaranteed to be dominated by k points already seen. Based on this idea, Figure 14 demonstrates the *SUB-SKYBAND* algorithm.

Next, we illustrate the algorithm by using the index in Figure 13b to extract the 2-skyband in $SUB = \{1, 2\}$ of the dataset in Figure 7. *SUB-SKYBAND* scans two sorted lists $\{p_5, p_4, p_1, p_2, p_8, p_7\}$ and $\{p_6, p_3\}$ in a synchronous manner. Recall that the points in the first (second) list are assigned to anchor A_1 (A_2) and sorted in descending order of their f-values. Following the process discussed in Section 5.4, we access, in this order, p_5, p_4, and p_1 of the first list, after which ptr_1 and ptr_2 are referencing p_2 and p_6 respectively, and $S_{sky} = \{p_5, p_4, p_1\}$ (which should be interpreted as the 2-skyband set now). As explained in Section 5.4, p_1 definitely dominates all the un-inspected points in both lists — the reason for terminating the skyline search in the example in Section 5.4. To obtain the complete 2-skyband, we continue to process p_2, because its ER has a smaller volume than that of p_6. Since p_2 is dominated by only a single point p_1 in S_{sky}, it is added to S_{sky}; then, ptr_1 is moved to p_8.

p_2 must dominate all the un-examined points p in the first list, which can be understood by combining Property 2 with the fact: $f(p) \leq f(p_8) = 0.5 < \min_{i \in SUB}(A_1[i] - p_2[i]) = 0.6$. Now that we have found two points (p_1 and p_2) that dominate the un-inspected part of the first list, the list is eliminated from further consideration.

The discovery of p_2 does not prune the second list. Hence, p_6 is examined, and added to S_{sky} because it is dominated by only one point p_1 in S_{sky}. Pointer ptr_1 now references p_3, which is also checked, and included in the 2-skyband. Since the second list has been exhausted, the algorithm finishes, reporting $S_{sky} = \{p_5, p_4, p_1, p_2, p_6, p_3\}$ as the final result.

Algorithm SUB-SKYBAND (SUB, k, $\{A_1, A_2, ... A_m\}$)
/* The meanings of SUB, A_1, ..., and A_m follow those in Figure 12; k is the parameter in "k-skyband" */

1. for j = 1 to m
2. use a B-tree to find the point ptr_j with the maximum $f(ptr_j)$ among all the points assigned to A_j
3. $ptr_j.ER = \prod_{i=1}^{d}(A_j[i] - L_\infty(ptr_j, A_j))$ //Equation 5
4. $S_{sky} = \emptyset$ //the set of points in the k-skyband in SUB
5. for j = 1 to m
6. $S_{prune}[j] = \emptyset$ //$S_{prune}[j]$ is the set of points that dominate the un-inspected points assigned to A_j
7. while ($ptr_j \neq \emptyset$ for any $j \in [1, m]$)
8. t = the value of j giving the smallest $ptr_j.ER$ among all $j \in [1, m]$ such that $ptr_j \neq \emptyset$
9. if ptr_t is dominated by less than k points in S_{sky}
10. for every point $p \in S_{sky}$ dominated by ptr_t
11. $p.cnt++$ //$p.cnt$ is the number of data points found dominating p
12. if $p.cnt = k$ then remove p from S_{sky}
13. $S_{sky} = S_{sky} \cup \{ptr_t\}$; $ptr_t.cnt = 0$
14. for j = 1 to m, and $j \neq t$
15. if $ptr_j \neq \emptyset$ and $f(ptr_j) < \min_{i \in SUB}(A_j[i] - ptr_t[i])$
16. $S_{prune}[j] = S_{prune}[j] \cup \{ptr_t\}$
17. if $|S_{prune}[j]| = k$ then $ptr_j = \emptyset$ /* no point assigned to A_j can belong to the k-skyband */
18. ptr_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 ptr_t, or a neighboring node)
19. if $ptr_t \neq \emptyset$
20. for every point $p_{sky} \in (S_{sky} - S_{prune}[j])$
21. if $f(ptr_t) < \min_{i \in SUB}(A_t[i] - p_{sky}[i])$
22. $S_{prune}[t] = S_{prune}[t] \cup \{p_{sky}\}$
23. if $|S_{prune}[t]| = k$ then $ptr_t = \emptyset$ /* no point assigned to A_t can belong to the k-skyband */
24. $ptr_t.ER = \prod_{i=1}^{d}(A_t[i] - L_\infty(ptr_t, A_t))$
25. return S_{sky}

Figure 14: The algorithm of finding a subspace skyband

6.2 Subspace Top-k Retrieval

Given a monotone preference function g (concerning a subspace SUB), a top-k query returns the k data points with the lowest scores. Since, in any SUB, any top-k result is always included in the corresponding k-skyband, an obvious solution to answering the query is to extract the k-skyband, compute the scores of the retrieved points, and report the k ones with the lowest scores. This approach, however, may perform considerable unnecessary work, if the k-skyband is sizable. Here, we propose a faster algorithm, which employs exactly the same B-tree used by our *SUBSKY* methodology, and is applicable to any monotone preference function g. The algorithm requires the notion of "ER max-corner", defined as follows:

Definition 3. *Given a point p and an anchor A dominated by p, let R be the ER of p with respect*

Algorithm SUB-TOPK (SUB, k, g $\{A_1, A_2, ... A_m\}$)
/* The meanings of SUB, A_1, ..., and A_m follow those in Figure 12; k is the parameter in "top-k"; g is the preference function of the top-k query */
1. for j = 1 to m
2. use a B-tree to find the point ptr_j with the maximum $f(ptr_j)$ among all the points assigned to A_j
3. $S_{top} = \emptyset$ //the top-k set
4. while ($ptr_j \neq \emptyset$ for any $j \in [1, m]$)
5. t = the value of j giving the smallest $g(ptr_j)$ among all $j \in [1, m]$ such that $ptr_j \neq \emptyset$
6. if $g(ptr_t)$ is smaller than the score of some point in S_{top}
7. remove from S_{top} the point with the largest score
8. $S_{top} = S_{top} \cup \{ptr_t\}$
9. for j = 1 to m, and $j \neq t$
10. if $ptr_j \neq \emptyset$
11. ptr_{j_C} is the ER max-corner of ptr_j
12. if $g(ptr_{j_C})$ is larger than the scores of all points in S_{top}
13. $ptr_j = \emptyset$ /* no point assigned to A_j can belong to the top-k set */
14. ptr_t = the point with the next largest $f(p)$ among the data assigned to A_t
15. if $ptr_t \neq \emptyset$
16. ptr_{t_C} is the ER max-corner of ptr_t
17. if $g(ptr_{t_C})$ is larger than the scores of all points in S_{top}
18. $ptr_t = \emptyset$ /* no point assigned to A_t can belong to the top-k set */
19. return S_{top}

Figure 15: The algorithm of answering a subspace top-k query

to A (formulated in Definition 2). Then, the **ER max-corner** *of p* **with respect to** *A is the corner of R opposite to the origin (which is also a corner of R).*

For example, in Figure 10a, the ER max-corner of p is the upper-right corner of the shaded region. Such corners have two important properties:

Property 3. *A data point p' cannot be in the result of a top-k query (which specifies a preference function g concerning subspace SUB), if there exist k points p_1, ..., p_k such that, for all $j \in [1, k]$,*

$$g(p_j) < g(p'_C) \tag{7}$$

where p'_C is the ER max-corner of p' with respect to its assigned anchor.

Property 4. *Let p_1 and p_2 be two points assigned to the same anchor A. If $f(p_1) \geq f(p_2)$, then $g(p_{1_C}) \leq g(p_{2_C})$, where g is any monotone preference function, and p_{1_C} and p_{2_C} are the ER max-corners of p_1 and p_2 with respect to A, respectively.*

Figure 7 formally describes the proposed algorithm *SUB-TOPK* for subspace top-k retrieval. As with subspace skyline/skyband search, *SUB-TOPK* leverages m lists, where the i-th ($1 \leq i \leq m$)

list juxtaposes the points assigned to the i-th anchor in descending order of their f-values. Given a query subspace SUB, the algorithm again uses m pointers ptr_1, ptr_2, ..., ptr_m to scan the m lists synchronously, and maintains the set S_{top} of k objects that have the smallest scores among all the objects scanned so far. In each iteration, *SUB-TOPK* processes the referenced point p with the smallest score, and attempts to prune a list according to Properties 3 and 4.

In the sequel, we explain the algorithm using the dataset in Figure 13, assuming a top-2 query in $SUB = \{1, 2\}$ with $g(p) = 3p[1] + p[2]$. *SUB-TOPK* examines two sorted lists $\{p_5, p_4, p_1, p_2, p_8, p_7\}$ and $\{p_6, p_3\}$. At the beginning, pointers ptr_1 and ptr_2 reference the top elements of the two lists, respectively. At each step, we process the referenced point with a smaller score. Since $g(p_5) = 1.2 < g(p_6) = 1.6$, p_5 is added to S_{top}, and ptr_1 is moved to the next element p_4.

As $g(p_6) < g(p_4) = 2.8$, the algorithm adds p_6 to S_{top} (which becomes $\{p_5, p_6\}$, sorted in ascending order of their scores), and shifts pointer ptr_2 nto p_3. Similarly, p_3 is the third point inspected, but is discarded, because $g(p_3) = 1.8$ is larger than the score of the current top-2 object p_6. The second list has been exhausted; hence, the subsequent execution focuses on the first list. We continue to process p_5 and p_4, which are also ignored, due to the same reason for discarding p_3. Next, p_1 is examined, and included in S_{top}, whereas p_6 is removed from S_{top}, because it has a higher score than p_1 and p_5.

The algorithm terminates here with $S_{top} = \{p_1, p_5\}$ as the final result. To explain this, notice that the element p_2 referenced by ptr_1 now has an ER max-corner p_{2_C} = (0.4, 0.4, 0.4). The score $g(p_{2_C})$ = 1.6 of p_{2_C} is greater than those (0.8 and 1.2) of p_1 and p_5. Hence, by Property 3, p_2 cannot belong to the top-2 result. Furthermore, let p be any point in the first list that has not been examined, and p_C the ER max-corner of p. As $f(p_2) \geq f(p)$, according to Property 4, $g(p_C)$ must be at least $g(p_{2_C})$, and hence, larger than the scores of both p_1 and p_5. Therefore, p cannot be in the top-2 result, either. The complete algorithm is formally described in Figure 15.

7 Experiments

In this section, we experimentally evaluate the efficiency of the proposed techniques. We deploy three real datasets *NBA*, *Household*, and *Color*[5]. Specifically, *NBA* contains 17k 8-dimensional

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

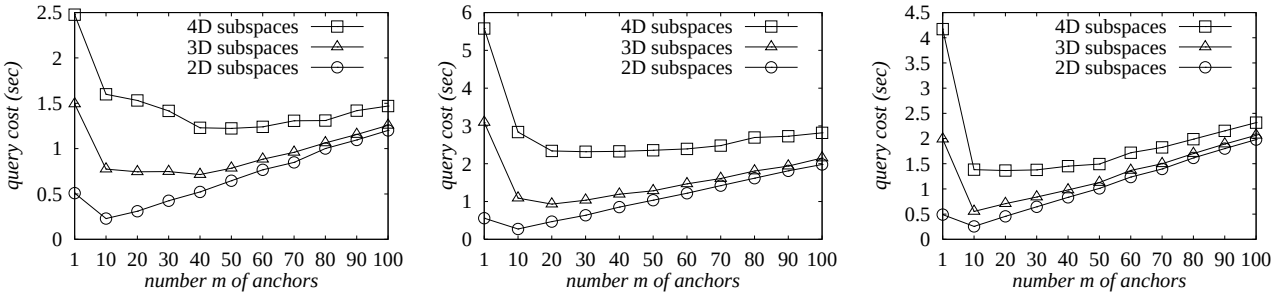
points, where each point corresponds to the statistics of a player in 8 categories. These categories include the numbers of points scored, rebounds, assists, steals, blocks, field goals attempted, free throws, and three-point shots, all averaged over the number of minutes played. *Household* consists of 127k 6-dimensional tuples, each of which represents the percentage of an American family's annual income spent on 6 types of expenditure: gas, electricity, water, heating, insurance, and property tax. *Color* is a 9-dimensional dataset with a cardinality 68k, and a tuple captures several properties of an image. Specifically, each image is encoded in the HSV space, and those 9 dimensions record the mean, standard deviation, and skewness of all the pixels in the H, S, and V channels, respectively. All the values are normalized into the unit range [0, 1].

We also generate synthetic data with four distributions: *uniform*, *correlated*, *anti-correlated*, and *clustered*. The first three distributions are commonly adopted in the literature for evaluating skyline algorithms; we refer our readers to [5] for their generalization. To create a *clustered* dataset with cardinality N, we first pick 10 cluster centroids randomly. Then, for each centroid, we obtain $N/10$ points, such that the coordinate of a point on each axis follows a Gaussian distribution with standard deviation 0.05, and a mean equal to the corresponding coordinate of the centroid.

7.1 Efficiency of Subspace Skyline Retrieval

We compare *SUBSKY* against the adapted versions of *BBS*, *SFS*, and *TA* discussed in Section 3. To apply *SUBSKY* (*BBS*), we build a B- (R-) tree on each dataset. Each B-tree is constructed with anchors computed (as elaborated in Section 5.3) from a 10% random sample set of the employed dataset. For *TA*, we create d sorted lists as described in Section 2.1. Recall that, *TA* executes in two phases. The first phase extracts the ids of a set of candidate objects, i.e., the ids scanned until the same id is encountered in all lists. Then, the second step retrieves the concrete coordinates of each candidate. To optimize the second phase, we employ a B-tree to index the underlying dataset on the ids; thus, the phase can be completed via a single traversal of the tree, which visits only the nodes on the paths from the root to the leaves containing at least a candidate id. Our approach incurs much lower cost than the traditional implementation [5, 26], where the second phase invokes a blocked nested loop. *SFS* is implemented in the same way as presented in [10]. The page size is set to 4k bytes in all cases.

A *workload* contains as many queries as the number of subspaces with the same dimensionality



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

Figure 16: *SUBSKY* performance vs. the number of anchors

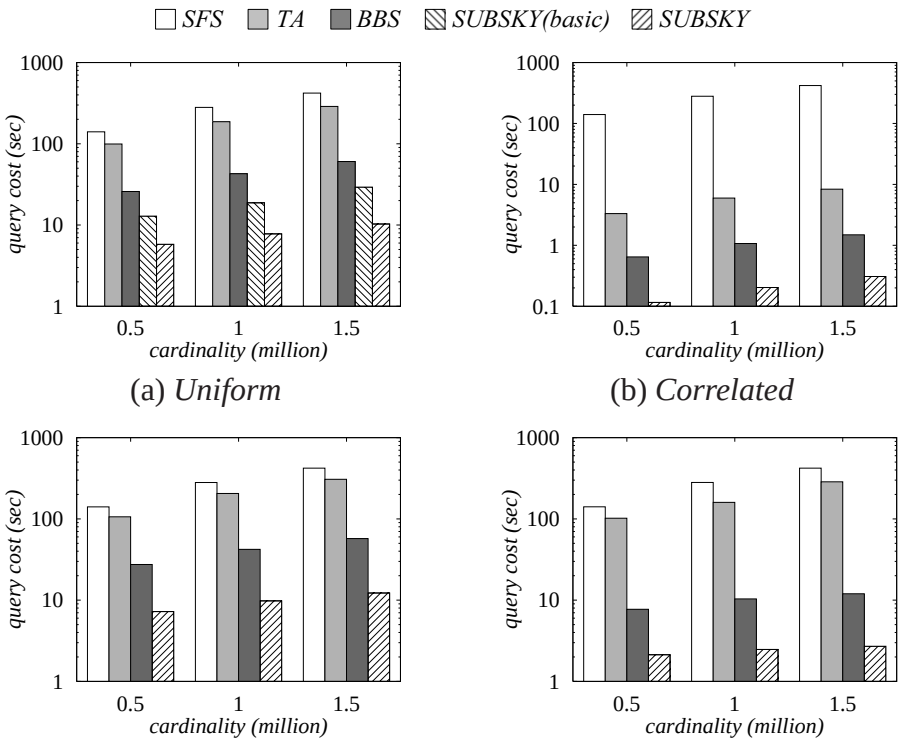
d_{sub} in the underlying dataset, where d_{sub} is a parameter of the workload. For example, for *NBA* and $d_{sub} = 3$, there are $\binom{8}{3} = 56$ three-dimensional subspaces, and hence, the corresponding workload includes 56 queries. For *NBA* and *Household*, each skyline aims at maximizing the coordinates of the participating dimensions, while queries on the other datasets prefer small coordinates.

We measure query cost as the total overhead, which includes both the CPU and I/O time. In particular, I/O cost involves 20ms for each random access, and 4ms for each sequential access. All the experiments are performed on a machine with a Pentium IV CPU at 3GHz and 1 Giga bytes memory.

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 cost per query. Figure 16 plots the cost as a function of m, for workloads with d_{sub} = 2, 3 and 4, respectively. Note that the results for $m = 1$ correspond to the overhead of the basic *SUBSKY* that uses the maximal corner as the only anchor (Section 4).

As m becomes larger, the query overhead first decreases and then actually increases after m passes a certain threshold. The initial decrease confirms the analysis of Section 5 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 a B-tree, which requires at least m page accesses. For excessively large m, these m accesses constitute a dominant factor in the overall overhead, which thus grows (almost) linearly with m.

Even for the same dataset, the optimal m is greater, when the dimensionality d_{sub} of the query



(a) *Uniform* (b) *Correlated*

(c) *Anti-correlated* (d) *Clustered*

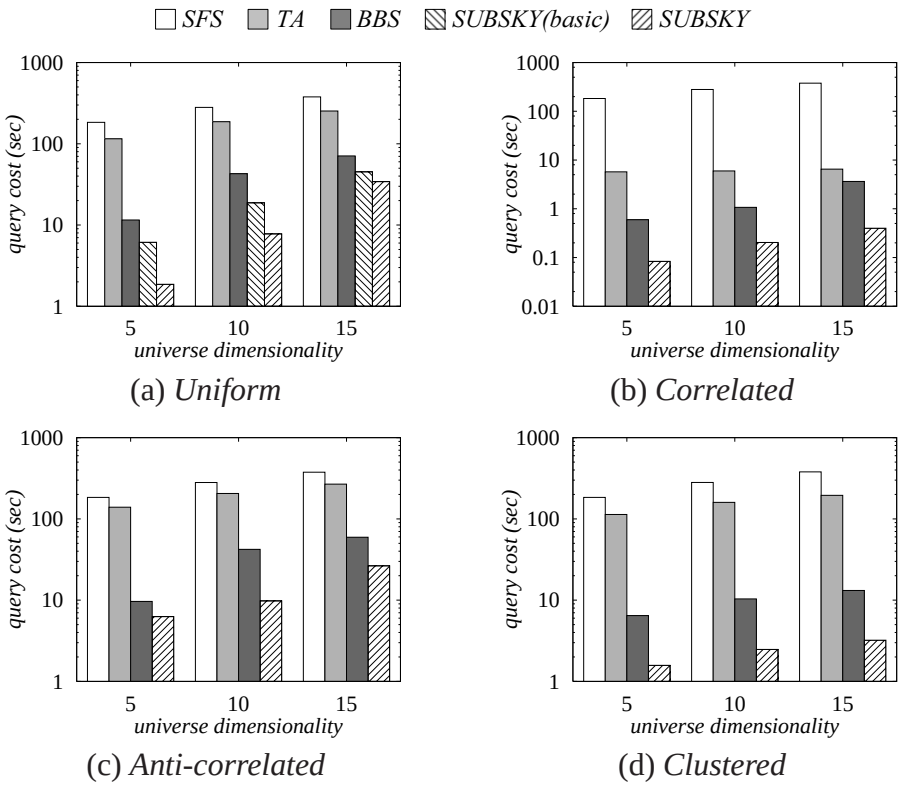
Figure 17: Cost of skyline search vs. cardinality ($d_{sub} = 3$, d = 10)

subspace is higher. For example, for *NBA*, the best m equals 10, 40, 50 for d_{sub} = 2, 3, and 4, respectively. In the sequel, we set m to 10 for real datasets, since this value offers the best overall performance.

Through a similar tuning process, we use m = 70, 1, 100, and 30 for all the *uniform*, *correlated*, *anti-correlated*, and *clustered* datasets, respectively.

Scalability with the Cardinality and Universe Dimensionality. In the next experiment, we deploy 10D *uniform* datasets with cardinalities ranging from 0.5 to 1.5 million. Deploying a 3D workload, Figure 17a compares the average cost (of all queries in a workload) of *BBS*, *SFS*, *TA*, the basic and general *SUBSKY*.

The proposed techniques significantly outperform their competitors. In particular, the two *SUBSKY* methods are faster than *SFS* and *TA* by a factor of an order of magnitude. Furthermore, the general *SUBSKY* is also nearly 10 times faster than *BBS*. In Figures 17b-17d, we present the results of the same experiments on synthetic datasets of the other distributions, confirming the above observations. The basic *SUBSKY* is omitted because it targets uniform data specifically.



(a) *Uniform* (b) *Correlated*

(c) *Anti-correlated* (d) *Clustered*

Figure 18: Cost of skyline search vs. universe dimensionality ($d_{sub} = 3$, 1 million cardinality)

To examine the influence of the universe dimensionality d, we utilize datasets with cardinality 1 million, whose d varies from 5 to 15. In Figure 18, again leveraging 3D workloads, we measure the average cost of alternative methods as a function of d, for the four types of synthetic distributions, respectively. *SUBSKY* consistently outperforms the other approaches significantly.

It is worth mentioning that, all algorithms are I/O-bounded, such that the CPU cost accounts for at most 2% of the total running time of any query. In the rest experiments, we omit *SFS* and *TA*, because they are not comparable with *BBS* and *SUBSKY*. Furthermore, we will use the general *SUBSKY* as the representative of our technique.

Characteristics of *SUBSKY*. We proceed to study several intrinsic properties of the proposed technique. For this purpose, we focus on *uniform* datasets, so that we can explain the observed behavior without worrying about the complex influences caused by the irregularity in the data distribution.

First, we examine the percentage of a database (universe dimensionality 10) that must be inspected by *SUBSKY*. The 2nd (3rd, 4th) row of Table 1a shows the percentage for answering a 2D (3D, 4D,

cardinality	10k	500k	1m	1.5m	2m
$d_{sub} = 2$	1.4%	1.4%	0.90%	0.61%	0.49%
$d_{sub} = 3$	3.7%	3.6%	3.5%	3.1%	2.5%
$d_{sub} = 4$	17%	15%	13%	12%	10%

(a) Percentage vs. dataset cardinality (d = 10)

d	5	10	15
$d_{sub} = 2$	0.63%	0.90%	0.90%
$d_{sub} = 3$	0.77%	3.5%	10.8%
$d_{sub} = 4$	0.65%	15%	28%

(b) Percentage vs. universe dimensionality d (cardinality 1 million)

Table 1: The percentage of a database accessed by *SUBSKY*

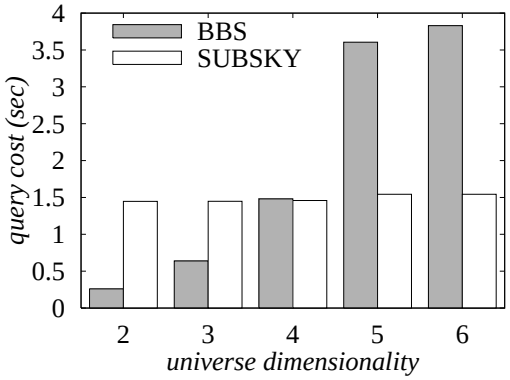
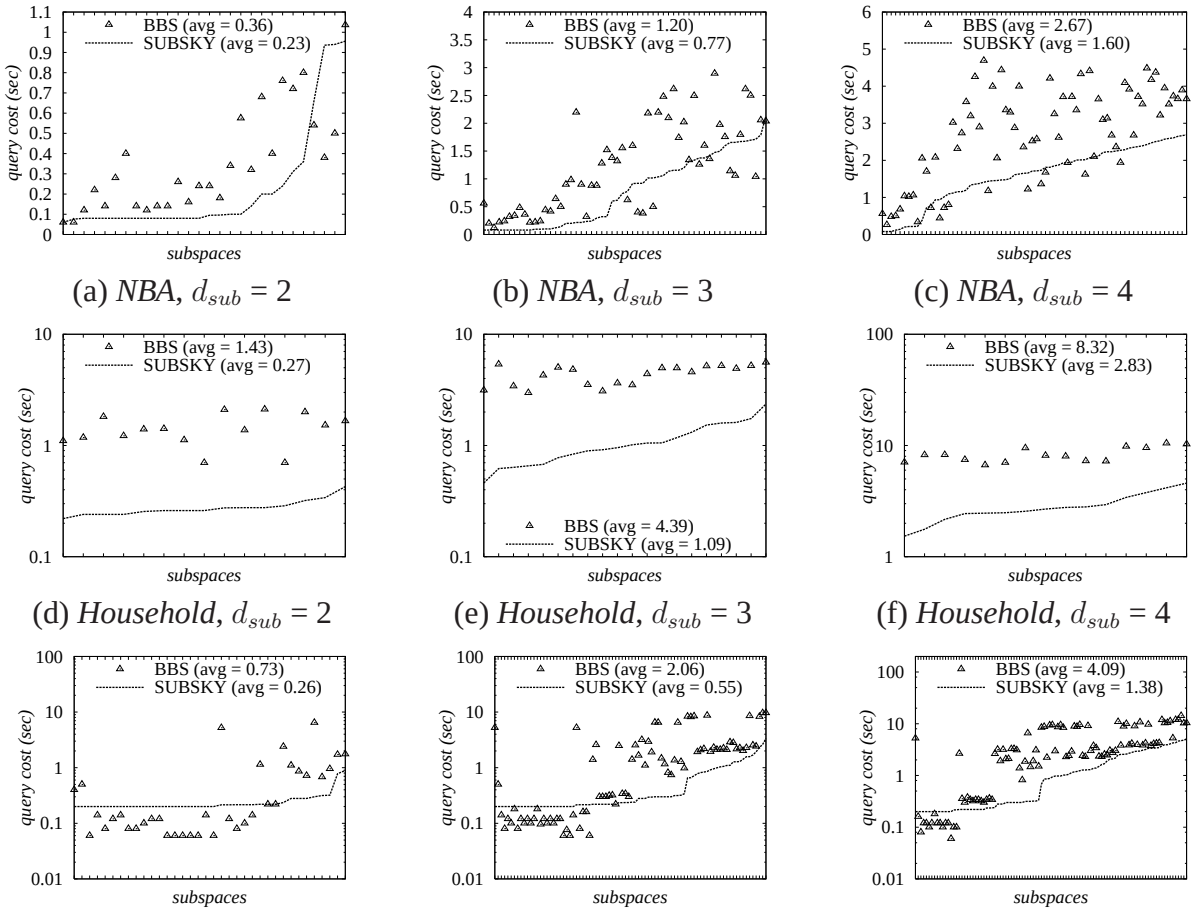


Figure 19: The break-even d between *BBS* and *SUBSKY* ($d_{sub} = 2$, 1 million cardinality)

respectively) workload, as the dataset cardinality grows from 10k to 2 million. For the same d_{sub}, the percentage is actually lower for a more sizable dataset. To explain this, consider the object whose L_∞ distance to the origin is the smallest. Let λ denote that distance. This object prunes all the data points p satisfying $f(p) < 1-\lambda$, where $f(p)$ is the L_∞ distance between p and the maximal corner. When the dataset is larger, λ is smaller; therefore, a higher percentage of the dataset can be pruned. Note that the above phenomenon does not contradict the results in Figure 17. As the cardinality grows, the actual number of objects inspected by *SUBSKY* still increases, even though the percentage is reduced.

Table 1b demonstrates the percentages for performing 2D, 3D, and 4D workloads, with respect to various universe dimensionalities d. As expected, the percentage increases with d, confirming the intuition that subspace skyline retrieval is more difficult in a higher-dimensional universe. The two tables also indicate that, given the same cardinality and d, the percentage grows with d_{sub}. This is reasonable, because our heuristics are less effective in subspaces with more dimensions.



(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 20: Cost of skyline search vs. subspace dimensionality

Second, we investigate the "break-even" universe dimensionality where *BBS* and *SUBSKY* switch their relative superiority. As analyzed in Section 3, *BBS* is expensive only if the universe dimensionality d is sufficiently high (so that the structure of the underlying R-tree degrades significantly). If d is small, *BBS* would be faster than *SUBSKY*, due to the information loss in the dimension-reduction transformation adopted by *SUBSKY*.

To capture the break-even point, we fix d_{sub} to 2 and cardinality to 1 million, but measure the cost of the two methods by gradually raising d. The results are demonstrated in Figure 19. The overhead of *SUBSKY* is not significantly affected when d distributes in the tested range, whereas the cost of *BBS* escalates quickly. As expected, for small d, *BBS* entails cheaper computation time, but *SUBSKY* starts being the better method at $d = 5$.

Examination of Individual Subspaces. Figure 20a illustrates the cost of *SUBSKY* and *BBS* for

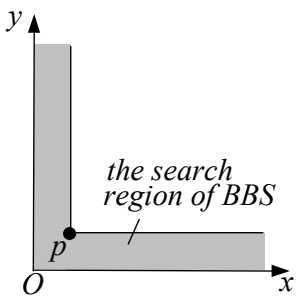


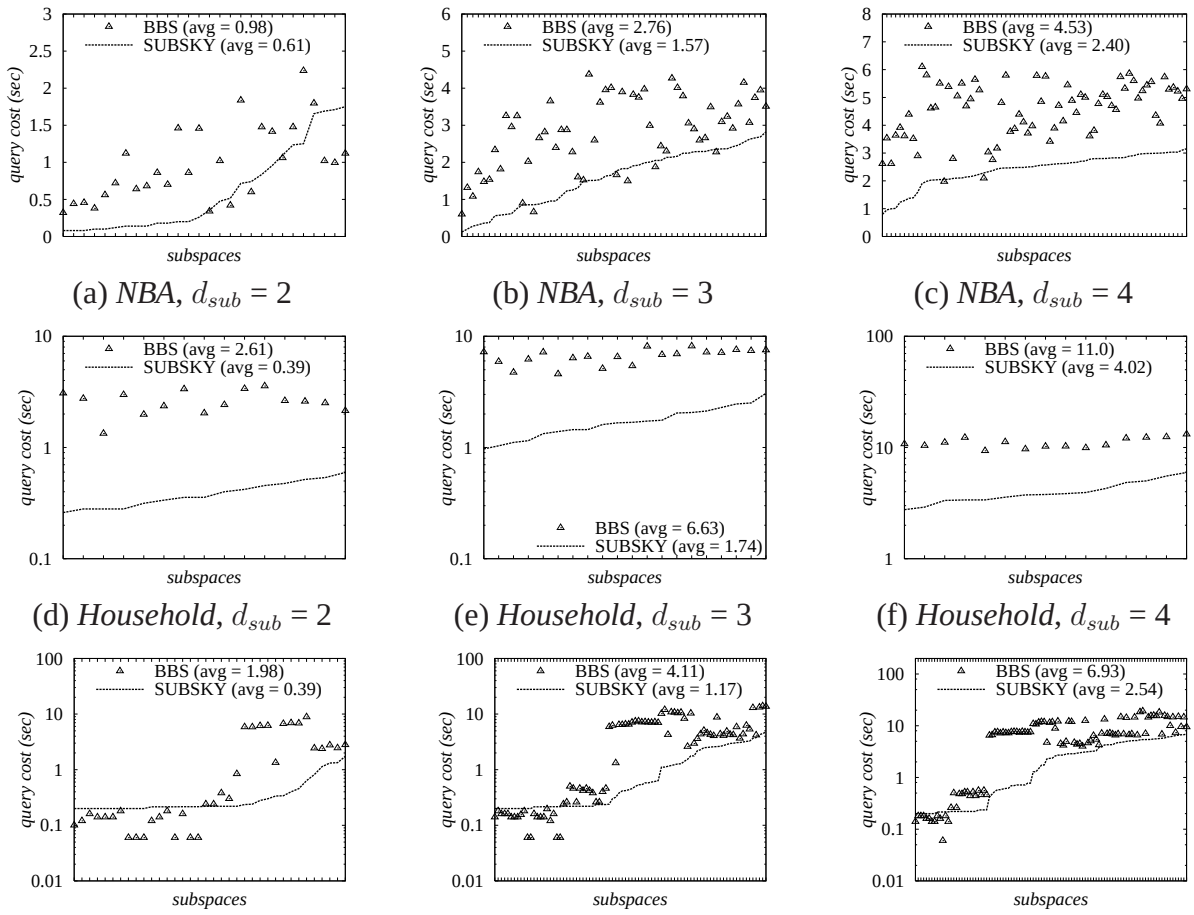
Figure 21: The best case for *BBS*

answering each query in a 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 0.23 seconds). In Figures 20b and 20c, we demonstrate a similar comparison for workloads with d_{sub} = 3 and 4, respectively. Figures20d-20i 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 20d). 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 20g, where *BBS* is slightly faster for around 60% of the workload, but significantly slower for the remaining queries, rendering its average overhead nearly 3 times higher than that of *SUBSKY*.

Why can *BBS* sometimes be so efficient even when the structure of the R-tree incurs serious deterioration caused by the high dimensionality of the dataset? To answer this question, Figure 21a shows an extreme case where the skyline includes a single point p (i.e., p dominates all the other points). *BBS* accesses only the nodes whose MBRs intersect the shaded region. No matter how the leaf nodes of the R-tree are obtained, there is only one leaf (i.e., the one that contains p) whose MBR intersects the region (recall that there exists a data point on each edge of an MBR). The same analysis also applies to nodes of higher levels, i.e., *BBS* needs to access only a single path of the R-tree.

In general, given a "bad" R-tree, *BBS* may still have satisfactory performance if the skyline points are close to the origin. However, when the condition is violated, the efficiency of this technique drops considerably due to the reasons discussed in Section 3. *SUBSKY*, on the other hand, is able to find a skyline that contains numerous points far-away from the origin with much lower overhead.



(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

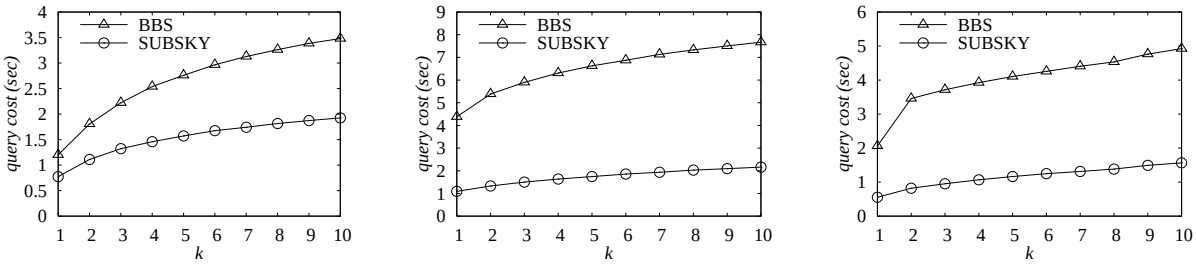
(g) *Color*, d_{sub} = 2 (h) *Color*, d_{sub} = 3 (i) *Color*, d_{sub} = 4

Figure 22: Cost of k-skyband search vs. subspace dimensionality (k = 5)

7.2 Efficiency of Skyband and Top-k Retrieval

Having demonstrated the superiority of *SUBSKY* in answering skyline queries, we proceed to evaluate the efficiency of our techniques for subspace k-skyband and top-k search. Specifically, for k-skyband (or top-k) queries, we compare *SUBSKY* against the extended *BBS* in [21] (or *BF*, standing for the best-first algorithm mentioned in Section 2.3). Each query workload is created in the same way as described in Section 7.1, except that here it contains another parameter k. For all datasets examined in the sequel, the indexes employed are exactly the same as those used in the skyline experiments.

Figures 22a-22i demonstrate the results of 5-skyband retrieval under the settings identical to those in Figures 20a-20i, respectively. The characteristics of *SUBSKY* and *BBS* are similar to those in skyline search. Fixing d_{sub} to 3, Figure 23 compares the average query cost in a workload of the



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

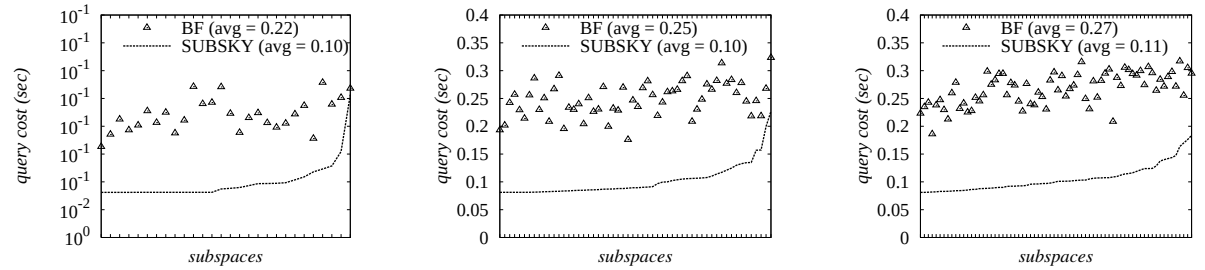
Figure 23: Cost of k-skyband search vs. k ($d_{sub} = 3$)

two methods, by varying k from 1 to 10. Figures 24 and 25 repeat these experiments with respect to top-k queries. Clearly, *SUBSKY* outperforms its competitor considerably in all cases.

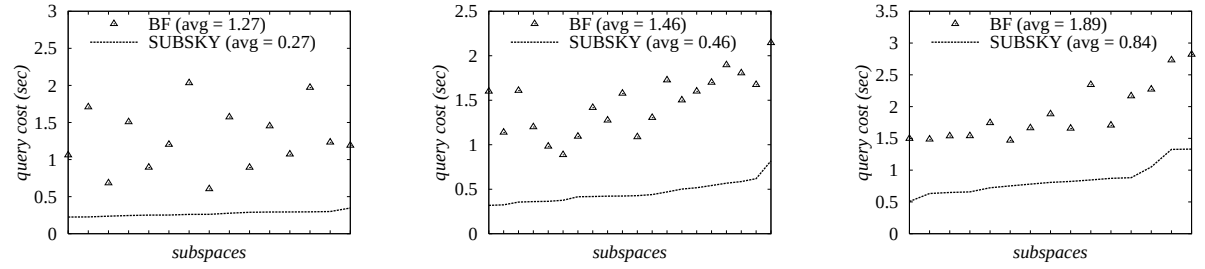
8 Conclusions

In practice, skyline and top-k 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/top-k retrieval with only relational technologies. The core of *SUBSKY* is a transformation that converts multidimensional 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.

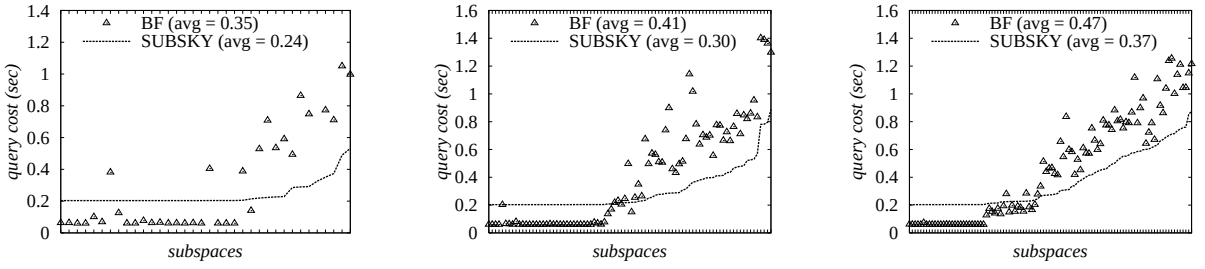
This work also lays down a foundation for future investigation of several related topics. For instance, certain attributes in the relation may appear in the subspaces of most queries (e.g., a user looking for a good hotel would always be interested in the *price* dimension). In this case, the data structure 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.



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

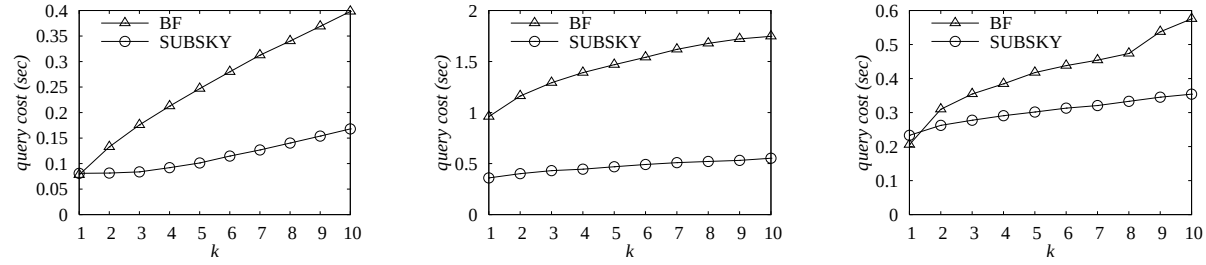


(d) *Household*, d_{sub} = 2 (e) *Household*, d_{sub} = 3 (f) *Household*, d_{sub} = 4



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

Figure 24: Cost of top-k search vs. subspace dimensionality (k = 5)



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

Figure 25: Cost of topk-k search vs. k ($d_{sub} = 3$)

Acknowledgements

Yufei Tao and Xiaokui Xiao were supported by CERG Grant CUHK 1202/06 from the Research Grant Council of the HKSAR government. Jian Pei was supported by NSERC Discovery Grants Program, NSERC Collaborative Research and Development Grants Program, and IBM Faculty

award.

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] C. Bohm. A cost model for query processing in high dimensional data spaces. *TODS*, 25(2):129–178, 2000.

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

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

[7] C.-Y. Chan, H. Jagadish, K.-L. Tan, A. Tung, and Z. Zhang. On high dimensional skylines. In *EDBT*, pages 478–495, 2006.

[8] C.-Y. Chan, H. V. Jagadish, K.-L. Tan, A. K. H. Tung, and Z. Zhang. Finding k-dominant skylines in high dimensional space. In *SIGMOD*, pages 503–514, 2006.

[9] Y.-C. Chang, L. D. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith. The onion technique: Indexing for linear optimization queries. In *SIGMOD*, pages 391–402, 2000.

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

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

[12] R. Fagin. Combining fuzzy information from multiple systems (extended abstract). In *PODS*, pages 216–226, 1996.

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

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

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

[16] V. Hristidis and Y. Papakonstantinou. Algorithms and applications for answering ranked queries using ranked views. *The VLDB Journal*, 13(1):49–70, 2004.

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

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

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

[20] S. Michel, P. Triantafillou, and G. Weikum. Klee: a framework for distributed top-k query algorithms. pages 637–648, 2005.

[21] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD*, pages 467–478, 2003.

[22] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *TODS*, 30(1):41–82, 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 *VLDB*, pages 253–264. VLDB Endowment, 2005.

[24] 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.

[25] R. K. Surajit Chaudhuri, Nilesh Dalvi. Robust cardinality and cost estimation for skyline operator. In *ICDE*, 2006.

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

[27] Y. Tao, V. Hristidis, D. Papadias, and Y. Papakonstantinou. Branch-and-bound processing of ranked queries. *To appear in Information Systems*.

[28] P. Tsaparas, T. Palpanas, Y. Kotidis, N. Koudas, and D. Srivastava. Ranked join indices. In *ICDE*, pages 277–288, 2003.

[29] T. Xia and D. Zhang. Refreshing the sky: the compressed skycube with efficient support for frequent updates. In *SIGMOD*, pages 491–502, 2006.

[30] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient maintenance of materialized top-k views. In *ICDE*, pages 189–200, 2003.

[31] 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.