Subset Approach to Efficient Skyline Computation

Dominique H. Li
LIFAT Laboratory, University of Tours, France
dominique.li@univ-tours.fr

ABSTRACT

Skyline query processing is essential to the database community. Many algorithms have been designed to perform efficient skyline computation, which can be generally categorized into sorting-based and partitioning-based by considering the different mechanisms to reduce the dominance tests. Sorting-based skyline algorithms first sort all points with respect to a monotone score function, for instance the sum of all values of a point, then the dominance tests can be bounded by the score function; partitioning-based algorithms create partitions from the dataset so that the dominance tests can be limited in partitions. On the other hand, the incomparability between points has been considered as an important property, that is, if two points are incomparable, then any dominance test between them is unnecessary. In fact, the state-of-the-art skyline algorithms effectively reduce the dominance tests by taking the incomparability into account. In this paper, we present a subset-based approach that allows to integrate subspace-based incomparability to existing sorting-based skyline algorithms and can therefore significantly reduce the total number of dominance tests in large multidimensional datasets. Our theoretical and experimental studies show that the proposed subset approach boosts existing sorting-based skyline algorithms and makes them comparable to the state-of-the-art algorithms and even faster with uniform independent data.

1 INTRODUCTION

Given a set of multidimensional points, the *skyline operator* [4] returns the *skyline* that is the set of all non-dominated points. A point p_i is said *non-dominated* if there is no any other point p_j such that p_j is better than p_i in all dimensions with respect to a user defined preference order. Figure 1 shows a skyline example that is very commonly used in the literature: assume a set of hotels where we want to select the ones with minimized price (Y axis) and distance from the beach (X axis), then the hotels $\{a, c, e, h, m\}$ form the skyline because no other hotels can be better than them on both of the price and the distance from the beach. The *skyline computation problem* has received intensive attention from the database community.

The simplest way to compute the skyline is nested loop-based pairwise comparison: for each point p_i in the dataset, compare p_i with each other point p_j, if p_i dominates p_j, then drop p_j; if p_j dominates p_i, then drop the point p_i and break the nested loop; otherwise, continue the nested loop while keeping both p_i and p_j. Such a nested loop procedure finally outputs the set of all non-dominated points in $O(dN^2)$ time where d is the number of dimensions in each point and N is the number of points in the dataset, if we consider $O(d)$ time for testing the dominance relation (*dominance test*) between two points of d dimensions.

The dominance tests are the major cost of skyline computation. In order to efficiently resolve the skyline computation problem, many algorithms have been designed and developed based on the reduction of dominance testes, which can be categorized into two classes [14]: sorting-based (such as `BNL` [4], `Index` [25], `SFS` [7, 8], `LESS` [9, 10], `SaLSa` [1, 2], `BSkyTree-S` [13, 14], and `SDI` [18]) and partitioning-based (such as `D&C` [4], `NN` [11], `BBS` [20, 21], `LS` [19], `OSPS` [27], `ZSearch` [16], and `BSkyTree-P` [13, 14]). Besides, indexing techniques are also applied to skyline algorithms, such as `Index`, `BBS`, `ZINC` [17], and `SDI`, with different mechanisms.

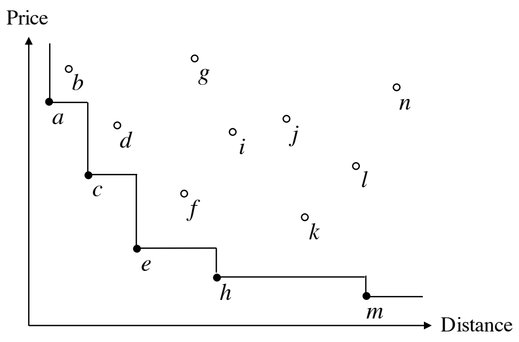


Figure 1: An example of skyline.

Godfrey et al. [9, 10] and Sheng and Tao [24] theoretically analyzed the time complexity of existing skyline algorithms and concluded that in the worst case, sorting-based algorithms finish in $O(dN^2)$ time, however partitioning-based algorithms can finish in $O(N \log^{d-2} N)$ time for any dimensionality $d > 2$. Godfrey et al. [9, 10] also gave a time complexity analysis for the average case under the *uniform independence* and *component independence* conditions. Essentially, the efficiency of a skyline algorithm heavily relies on how dominance tests are reduced. As the state-of-the-art skyline algorithms, `BSkyTree-S` and `BSkyTree-P` use a pivot point selection schema to map data points to incomparable regions, each can be considered as an optimized algorithm of sorting-based and partitioning-based categories.

Basically, given a d-dimensional space, a *subspace* [22] is a subset of all d dimensions, which is extended studied with several skyline problems such as *subspace skyline* [15, 23, 26] and *skycube* [3, 23]. Let $\mathcal{P}$ be the dataset, $p \in \mathcal{P}$ be a skyline point, and $q \in \mathcal{P}$ be any point such that $p \neq q$, then, if p does not dominate q, there must exist a subspace where in each dimension the value of q is better than the value of p, that is, q dominates p in this subspace. We call such a subspace a *dominating subspace*. If all points dominated by the skyline point p have been pruned, each point that remains in the dataset must possess a dominating subspace where it dominates p. Indeed, the above skyline point p, also called a *pivot point*, is used by the `BSkyTree-S` and `BSkyTree-P` algorithms to partition points into incomparable regions in order to reduce the dominance tests.

In this paper, we present a novel subspace-based skyline indexing approach to manage the incomparability between testing points and skyline points. Instead of partitioning the dataset, our method indexes the skyline points by dominating subspaces to reduce the dominance tests. We first show that the dominating subspace of a point can be merged with respect to multiple pivot points, called *maximum dominating subspace* with respect to a given number of pivot points, then, we show that if a point q_1 dominates a point q_2, q_1's maximum dominating subspaces with respect to each pivot point must be a superset of q_2's maximum dominating subspaces. Based on the properties presented in this paper, the sketch of the application of our method can be described as following:

(1) Find multiple pivot points to generate the maximum dominating subspace for each non-pruned point.
(2) Run a skyline algorithm with the following new actions:
  (a) Once a skyline point is determined, put it to the proposed skyline index structure with respect to its maximum dominating subspace.
  (b) While testing a point with the current skyline, get only the set of comparable points from the skyline index structure with respect to the maximum dominating subspace.
(3) Return the skyline.

Therefore, our proposed method does not concern concrete skyline computation algorithms because it is designed as a component like a container that allows to store (as *put* function) the skyline points and to retrieve (as a *get* function) a minimum number of skyline points to compare with a testing point. The paradigm of our method fits best sorting-based skyline algorithms that can progressively output the skyline points, however partitioning-based skyline algorithms cannot benefit much from our method because the data have already been partitioned and the dominance tests are limited in partitions, the double partitioning of data and skyline brings additional costs. Therefore, the advantage of our method is to boost existing skyline algorithms. Several sorting-based skyline algorithms such as SFS, SaLSa, and SDI do not depend on any particular data structures (for instance, the lattice and SkyTree required by BSkyTree-S and BSkyTree-P, the ZB-tree required by ZINC, the R-tree required by BBS, and the B$^+$-tree required by Index, etc.), which can be easily implemented in any programming languages including Python and PHP, so to boost such algorithms has realistic interests to data industries.

The main result presented in this paper is that the skyline indexing can be efficiently resolved by a **subset query**. The *subset query problem* is defined as: given a set $\mathcal{X}$ of distinct subsets of a universe D, for any set $Q \subseteq D$, return the set $\{Q' \in \mathcal{X} \mid Q \subset Q'\}$ that contains all supersets of Q. The set Q is called the *query set*. Indeed, given a testing point, its maximum dominating subspace can be considered as a query set, the task is to return all skyline points of which the maximum dominating subspaces are the supersets of the query set, hence, the required dominance tests can be limited in returned skyline points. In order to make our method efficient, we reversed the subset query problem, that is, to return the set $\{Q' \in \mathcal{X} \mid Q' \subset Q\}$. The subset query data structure and algorithms proposed in this paper is hash map based (supported by the most of programming languages), which can add a skyline point in linear time with respect to the dimensionality d and can retrieve a set of skyline points with respect to a given subspace in $O((\frac{d}{2})^2)$ time in the average case for any $d > 2$. Particularly, in the case of $d = 2$, subset query is a binary problem so the usefulness of our proposed method is very limited.

Our experimental results based on SFS, SaLSa, and SDI show the effectiveness and efficiency of our method. We note that the data types considered in skyline computation is generally categorized as [4] *anti-correlated* (AC), *correlated* (CO), and *uniform independent* (UI) with respect to the characteristics of real data, however, in literature, there is no single algorithm be the best on all these three types of data. For instance, BSkyTree-S performs much better than BSkyTree-P on CO data, however BSkyTree-P is the best algorithm on AC and UI data. In our experiments, the boosted SFS, SaLSa, or SDI perform better than BSkyTree-P on UI data.

The rest of this paper is organized as follows. Section 2 reviews previous skyline algorithms. Section 3 defines preliminary concepts required by the formalization. We show in Section 4 that the maximum dominating subspace of each point can be merged from multiple pivot points and propose a subspace union algorithm. In Section 5, we present our subset approach to index the skyline, with store and query algorithms. Section 6 reports our experimental evaluation, which shows the performance of our method in boosting skyline algorithms. Finally, we conclude in Section 7.

2 RELATED WORK

In this section, we review mainstream skyline algorithms by categorizing them into sorting-based and partitioning-based classes. We also discuss the use of indexing techniques in existing skyline algorithms.

In general, typical sorting-based skyline algorithms such as SFS [7, 8] presort all points by a monotone sorting function f such that for any two points p_i and p_j, we have $f(p_i) < f(p_j) \Rightarrow p_j \nprec p_i$ (we denote by $p_i \prec p_j$ that p_i dominates p_j and by $p_i \nprec p_j$ that p_i does not dominate p_j). If we consider the minima for the skyline, it is clear that the point $\arg\min_{p \in \mathcal{P}}(f(p))$ in the dataset $\mathcal{P}$ is immediately the first skyline point, then, by following the order defined by the sorting function f, if a point is not dominated by all skyline points, it is a new skyline point and can be added to the skyline.

The sorting function used in sorting-based algorithms is heuristic that heavily affects the total number of dominance tests. LESS [9, 10] extends SFS with external sort-merge routines. SaLSa [1, 2] studied different sorting functions with the notion of a stop point that allows to terminate the algorithm by outputting the exact skyline without testing all points. The use of stop point in SaLSa can effectively prune points that are not necessary to be tested. Index [25] builds a B$^+$-tree data structure to sort and index each dimension value of all points in order to prune irrelevant points and to retrieve skyline points by comparing their min/max values. BSkyTree-S first selects a pivot point, then maps any data point to a binary vector w.r.t. the pivot point, so existing sorting-based algorithms can be improved by bypassing dominance tests between incomparable points.

SDI [18] is a sort-and-scan skyline algorithm that integrates the designs of Index, SFS, and SaLSa. In the sort phase, SDI sorts all indexed data point IDs in each dimension, with respect to the value of each point in each dimension (*dimension value*), then each point at the top position of each sorted index is immediately a skyline point; in the scan phase, SDI uses the breadth-first strategy to traverse among dimensions. In any dimension, it is necessary to test by progressive depth-first traversal whether all known skyline points in this dimension (called *dimension skyline*) dominate the current testing point: if the current point is not dominated, then it is a new skyline point and the algorithm switches to the dimension that possesses the least number of skyline points; otherwise, the current point is marked as dominated and it continues to test the next point.

In SDI, a testing point can be skipped if it has already been marked as dominated on another dimension or be added to the dimension skyline without test; in the case of duplicate dimension values, that is, the same value in the same dimension for different tuples, SFS-like local dominance tests will be performed among concerned points. Furthermore, any data point can be used as the stop point in SDI, that is, if the progressive depth-first traversal has passed such a point in each dimension, then the algorithm can be safely stopped. In practice, the data point having the minimum Euclidean distance is the most efficient stop point, and by distributing the dominance tests to each dimension, SDI can effectively reduce the total number of dominance tests.

Different from sorting-based algorithm, partitioning-based algorithms divide points to different groups, called *partitions* (or *regions*), the dominance tests are limited among the points in the same partition.

As a representative divide-and-conquer method, D&C [4] is designed from the algorithm introduced by Kung et al. [12] in order to work with external memory. Basically, D&C partitions the dataset into as fewer as possible blocks to fit in the main memory, then the *local skyline* will be computed from each block, and recursively the *true skyline* can be computed from local skyline by applying the basic divide-and-conquer algorithm. The study of Godfrey et al. [9, 10] shows that the average performance of divide-and-conquer methods deteriorates with increasing the dimensionality of data. The study of Sheng and Tao [24] shows that the algorithm presented in [12] requires $O(d^2 N \log^{(d-2)} N)$ time however their result can effectively finish in $O(N \log^{(d-2)} N)$ time.

NN [11] partitions the dataset by the nearest neighbor of the query point, then the first nearest neighbor is immediately a skyline point that allows to prune all dominated points; then, the second nearest neighbor can be found in all points that are dominated by the first nearest neighbor in one or several dimensions (a *region*), and finally all skyline points can be directly output by recursive calls. The effect of NN is similar to SFS and SaLSa if we use Euclidean distance as the sorting function. BBS [20, 21] improves the design of NN by using R-tree to index points that can efficiently prune non-skyline points with dominance tests inside the region. Different techniques have been proposed to perform region-level pruning of non-skyline points, for instance, LS [19] uses a lattice structure, OSPS [27] uses point-based space partition, ZSearch [16] uses Z-order index.

BSkyTree-P first selects a pivot point, then recursively partitions a specific region into 2^d disjoint sub-regions in a divide-and-conquer manner. Existing partitioning-based skyline algorithms can be improved by the schema of BSkyTree-P.

On the other hand, indexing techniques are also widely used in skyline computation to access sorted or partitioned data. For instance, Index uses B$^+$-tree to store the sorted dataset; BBS uses R-tree to partition and indexing the dataset; BSkyTree-S and BSkyTree-P use SkyTree to access indexed dataset; SDI uses indexes to access the dataset from sorted dimension values. In particular, ZINC uses ZB-tree to index the dataset, which can perform skyline computation in both totally ordered and partially ordered data attribute domains. In this paper, we focus on the skyline computation problem in the totally ordered domain.

In our method presented in this paper, we do not index any data points but only the skyline points, in order to minimize the skyline points required by dominance tests, which makes our method being a generic component of skyline algorithms.

3 PRELIMINARY CONCEPTS

We consider a dataset $\mathcal{P}$ of N d-dimensional points, where we call N the *cardinality* and d the *dimensionality* of the dataset. Let p be a point, we denote $p[i]$ the *dimension value* of p in the dimension i, where $1 \leq i \leq d$. We consider the *preference order* as a total order in each dimension of points for the skyline.

Without loss of generality, the preference order can be defined as the relation $<$ on the values in each dimension. Given two points p and q, $p[i]$ is *better than* $q[i]$ if $p[i] < q[i]$; $p[i]$ is *equal to* $q[i]$ if $p[i] = q[i]$; and $q[i]$ is *not worse* than $p[i]$ if $p[i] \leq q[i]$. To simplify the formal description, in the rest of this paper, any *dimension* refers to an integer value in the range $[1, d]$.

Definition 3.1. A point p *dominates* a point q, denoted by $p \prec q$, if and only if in each dimension i we have $p[i] \leq q[i]$, and in at least one dimension k, $1 \leq k \leq d$, we have $p[k] < q[k]$. ■

Given two points p and q, we denote $p \nprec q$ that p does not dominate q; we denote $p \nsim q \iff (p \nprec q) \wedge (q \nprec p)$ that p and q are *incomparable*. We extend $\{\prec, \nprec, \preceq, \nsim\}$ to the set of points:

- $p \prec X$ (or $p \preceq X$) denotes $\forall q \in X, p \prec q$ (or $p \preceq q$);
- $X \prec p$ (or $X \preceq p$) denotes $\exists q \in X, q \prec p$ (or $q \preceq p$);
- $X \nprec p$ (or $X \npreceq p$) denotes $\nexists q \in X, q \prec p$ (or $q \preceq p$);
- $p \nsim X$ and $X \nsim p$ denote $\forall q \in X, p \nsim q$.

Definition 3.2. Given a dataset $\mathcal{P}$, a point $p \in \mathcal{P}$ is a *skyline point* if and only if $\nexists q \in \mathcal{P}$ such that $q \prec p$. The *skyline* of $\mathcal{P}$ is the complete set of skyline points $\{p \in \mathcal{P} \mid \nexists q \in \mathcal{P}, q \prec p\}$. ■

The *skyline computation* problem is to compute the complete set of skyline points from a multidimensional dataset with respect to a user defined preference order $\prec$.

Definition 3.3. Given a d-dimensional dataset $\mathcal{P}$, the set $D = \{1, 2, \ldots, d\}$ is the *space* of $\mathcal{P}$. Any subset $D' \subseteq D$ is a *subspace* of $\mathcal{P}$. ■

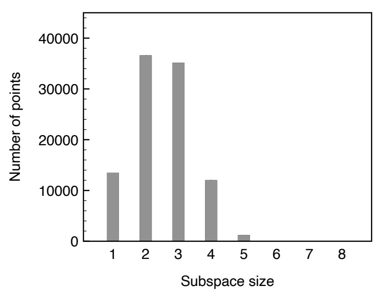
Definition 3.4. Let p and q be two points. Let $D_{p<q}$ denote the subspace such that $\forall i \in D_{p<q} \Rightarrow p[i] < q[i]$ and $\forall i \notin D_{p<q} \Rightarrow q[i] \leq p[i]$, then $D_{p<q}$ is the *dominating subspace* of p with respect to q. ■

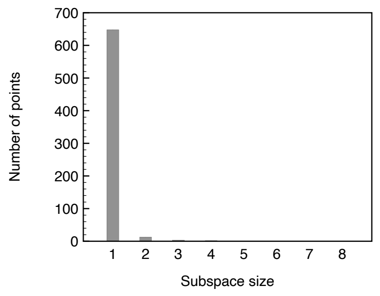
According to Definition 3.4, given a dataset $\mathcal{P}$, let D be the space of $\mathcal{P}$ and $p, q \in \mathcal{P}$ be two points, we have:

- $D_{p<q} = \emptyset \Rightarrow q \prec p$ or $p = q$;
- $D_{p<q} = D \Rightarrow p \prec q$.

Lemma 3.5. *Given a dataset $\mathcal{P}$, let $p \in \mathcal{P}$ be a skyline point and $q_1, q_2 \in \mathcal{P}$, $q_1, q_2 \neq p$ be two arbitrary points such that $p \nprec q_1$, $p \nprec q_2$, and $q_1 \neq q_2$. $q_1 \nsim q_2$ if $D_{q_1<p} \nsubseteq D_{q_2<p}$ and $D_{q_2<p} \nsubseteq D_{q_1<p}$.* ■

Proof. p is a skyline point, so $D_{q_1<p} \neq D$ and $D_{q_2<p} \neq D$, that is, $D_{p<q_1} \neq D$ and $D_{p<q_2} \neq D$, which impose that $p \nprec q_1$ and $p \nprec q_2$. We have that $p \nprec q_1 \iff \exists i, q_1[i] < p[i]$ and $p \nprec q_2 \iff \exists i, q_2[i] < p[i]$, hence, $D_{q_1<p} \nsubseteq D_{q_2<p}$ implies that there is at least one dimension i such that $q_1[i] < p[i] \leq q_2[i]$ and $D_{q_2<p} \nsubseteq D_{q_1<p}$ implies that there is at least one dimension j such that $q_2[j] < p[j] \leq q_1[j]$. Thus, $q_1 \nprec q_2$ and $q_2 \nprec q_1$, that is, $q_1 \nsim q_2$. □


(a) 8-D 100K AC dataset.


(b) 8-D 100K CO dataset.

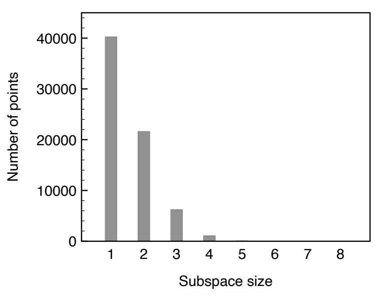

(c) 8-D 100K UI dataset.

Figure 2: Distribution of points with respect to subspace size where the pivot point is the skyline point with the minimal Euclidean distance to the zero point.

More simply, Lemma 3.5 can be rewritten as:

$$|D_{q_1<p} \cap D_{q_2<p}| < min(|D_{q_1<p}|, |D_{q_2<p}|) \Rightarrow q_1 \nsim q_2.$$

LEMMA 3.6. *Given a dataset $\mathcal{P}$, let $p \in \mathcal{P}$ be a skyline point and $q_1, q_2 \in \mathcal{P}$, $q_1, q_2 \neq p$ be two arbitrary points such that $p \nprec q_1$, $p \nprec q_2$, and $q_1 \neq q_2$. $D_{q_1<p} \nsupseteq D_{q_2<p} \Rightarrow q_1 \nprec q_2$.* ■

PROOF. We have the same context as Lemma 3.5. If $D_{q_2<p} \nsupseteq D_{q_1<p}$, according to the proof of Lemma 3.5, there exists at least on dimension i such that $i \in D_{q_1<p}$ and $i \notin D_{q_2<p}$, that is, $q_2[i] < p[i] \leq q_1[i]$. Thus, $D_{q_1<p} \nsupseteq D_{q_2<p} \Rightarrow q_1 \nprec q_2$. $D_{q_1<p} \supseteq D_{q_2<p}$ is a necessity of $q_1 \prec q_2$. □

Lemma 3.6 shows an inevitable constraint to partition points with respect to Lemma 3.5: if a point p is a skyline point in the set of points determined by the subspace D_x, then p must be compared with all skyline points in the set of points determined by any subspace $D_y \supset D_x$. Since a d-dimensional space contains $2^d - 2$ subspaces without $\emptyset$ neither the full space in our context, obviously, the dominance tests can be effectively reduced if all points can be distributed to as many incomparable subspaces as possible.

4 SUBSPACE UNION

In this section, we resolve the unbalanced point distribution problem. According to Lemma 3.5, given a dataset $\mathcal{P}$ in space D, if a skyline point $p \in \mathcal{P}$ is compared with each other point $q \in \mathcal{P}$, then every non-pruned point q can be attributed a dominating subspace $D_{q<p}$, where $D_{q<p} \neq \emptyset$ and $D_{q<p} \neq D$.

Figure 2 shows the distribution of non-pruned points in AC, CO, and UI synthetic datasets[1] with 100K points of 8 dimensions, where the pivot point is the skyline point with the minimal Euclidean distance to the zero point. Due to page length limit, it is difficult to list the number of points for all $2^8 - 2$ subspaces, we show in Figure 2 the number of points with respect to subspace size. We see that the distribution of points is unbalanced, most of them are in small-size zones, far away from the 2^d level. Although recursive calls can be applied to each subspace to find more incomparable subspaces with respect to Lemma 3.5, Lemma 3.6 limits the immediate output of skyline points.

We propose a subspace union method to distribute points to as many subspaces as possible , where the term *as many as possible* is controlled by a given threshold that finally affects the number of pivots points. Given a point, the dominating subspace can be merged from multiple pivot points, that is, a set of skyline points since all pivot points are skyline points, and we call such a merged subspace a *maximum dominating subspace*, where the term *maximum* means the maximum number of dimensions where the given point dominates the pivot points.

Definition 4.1. Let S be a set of skyline points in a dataset $\mathcal{P}$ of space D. For a point $q \in \mathcal{P}$, the union of dominating subspaces $D_{q<S} = \bigcup_{p \in S} D_{q<p}$, where $D_{q<S} \subseteq D$, is the *maximum dominating subspace* of q. ■

With the above definition, we have the following extensions of Lemma 3.5 and Lemma 3.6, which are the bases of our results.

LEMMA 4.2. *Given a dataset $\mathcal{P}$, let S be a set of skyline points of $\mathcal{P}$ and $q_1, q_2 \in \mathcal{P}$ be two arbitrary points such that $q_1, q_2 \notin S$, $S \nprec q_1$, $S \nprec q_2$, and $q_1 \neq q_2$. $q_1 \nsim q_2$ if $D_{q_1<S} \nsubseteq D_{q_2<S}$ and $D_{q_2<S} \nsubseteq D_{q_1<S}$.* ■

PROOF. The proof is as the proof of Lemma 3.5. If $D_{q_1<S} \nsubseteq D_{q_2<S}$ and $D_{q_2<S} \nsubseteq D_{q_1<S}$, there must be at least one skyline point $p \in S$ on at least one dimension i where $q_1[i] < p[i] \leq q_2[i]$ or $q_2[i] < p[i] \leq q_1[i]$. Thus, $q_1 \nsim q_2$. □

LEMMA 4.3. *Given a dataset $\mathcal{P}$, let S be a set of skyline points of $\mathcal{P}$ and $q_1, q_2 \in \mathcal{P}$ be two arbitrary points such that $q_1, q_2 \notin S$, $S \nprec q_1$, $S \nprec q_2$, and $q_1 \neq q_2$. $D_{q_1<S} \nsupseteq D_{q_2<S} \Rightarrow q_1 \nprec q_2$.* ■

PROOF. If $q_1 \prec q_2$, then $\forall i \in D, q_1[i] \leq q_2[i]$. If $D_{q_1<S} \nsupseteq D_{q_2<S}$, then there exists at least one skyline point $p \in S$ on at least one dimension i where $q_2[i] < p[i] \leq q_1[i]$. Thus, $q_1 \nprec q_2$. □

In practice, it is difficult to determine the number of pivot points, an optimal value depends on many factors including the number of all skyline points, which should be considered as unknown. Too few pivot points cannot distribute all points to a large number of subspaces but too many pivot points will clearly slow down our method. In general, we use a sorting-based process to select pivot points and merge dominating subspaces, the maximum dominating subspace is constructed in iteration. Each skyline point can assign a dominating subspace to a point $q \in \mathcal{P}$ and all dominated points will be pruned.

In each iteration, we determine the change of point number of each subspace size, that is, the number of points within the same subspace with the same size, which is limited by $d - 1$ instead of all $2^d - 2$ subspaces. Let $\mathcal{P}$ be the dataset and D_q be the maximum dominating subspace of a point $q \in \mathcal{P}$, then, for each point q, the subspace D_q may be changed by current pivot point p as $D_q \cup D_{q<p}$. Hence, we propose a heuristic measure, the *stability*

[1]All concerned synthetic datasets are generated by Skyline Benchmark Data Generator from http://pgfoundry.org/projects/randdataset.

threshold, denoted by σ, to stop merging dominating subspaces. The stability threshold is the number of subspace sizes that do not change while iterating, which means that no new pivot points are necessary to continue to change the maximum dominating subspaces.

The following Algorithm 1 merges dominating subspaces of each non-pruned point in a dataset. The algorithm stops while the stability threshold is reached.

Algorithm 1: Merge

```
   Input: The dataset $\mathcal{P}$ and the stability threshold $\sigma$
   Output: The initial skyline $S$ of $\mathcal{P}$ and $\mathcal{P}$ with
           non-pruned points
 1 Score each point $q \in \mathcal{P}$ by Euclidean distance to the zero
     point
 2 Initialize the maximum dominating subspace $D_q = \emptyset$ for
     each point $q \in \mathcal{P}$
 3 $S \leftarrow \emptyset$
 4 $\sigma' \leftarrow 0$
 5 while $\sigma' < \sigma$ do
 6     if $\mathcal{P} = \emptyset$ then
 7         return $\mathcal{S}, \mathcal{P}$
 8     $p \leftarrow$ the point with the minimal score (which is a
         skyline point)
 9     $\mathcal{S} \leftarrow \mathcal{S} \cup \{p\}$
10     $\mathcal{P} \leftarrow \mathcal{P} \setminus \{p\}$
11     foreach $q \in \mathcal{P}$ do
12         Compute dominating subspace $D_{q<p}$
13         if $D_{q<p} = \emptyset$ then
14             if $q = p$ then
15                 $\mathcal{S} \leftarrow \mathcal{S} \cup \{q\}$
16             $\mathcal{P} \leftarrow \mathcal{P} \setminus \{q\}$
17             continue
18         $D_q \leftarrow D_q \cup D_{q<p}$
19     $\sigma' \leftarrow$ compute the stability of point distribution
20 return $\mathcal{S}, \mathcal{P}$
```

First, in our algorithm we score each point in the dataset $\mathcal{P}$ by its Euclidean distance to the zero point (line 1). In this step, the sorting of all points is not necessary, which requires $O(N \log N)$ time: assume that the stability threshold σ can be satisfied by k skyline points, the search of minimal score (line 9) can be done in $O(kN)$ time in the worst case, where $k \ll N$. The algorithm runs in the iterative loop from line 5 to line 24 with respect to the stability measure $\sigma' < \sigma$. In each iteration, any time if all points have been pruned, then the algorithm returns the skyline S and the empty dataset $\mathcal{P}$ (line 7), so that the whole computation can be terminated. With pruning dominated points in the dataset, the point p having the minimal score is immediately a skyline point (line 9 and 10) and can be pruned from the dataset (line 11). Then, the point p will be compared with each point $q \in \mathcal{P}$ in the dataset (line 12 to line 22) by computing the dominating subspace $D_{q<p}$ (line 13). If $S_{q<p} = \emptyset$, then $p \nprec q$ or $p = q$ (here we denote by $p = q$ that $\forall i \in D, p[i] = q[i]$), in any case, q will be pruned from $\mathcal{P}$ (line 18) and q will be added to the skyline if $p = q$ (line 15 to line 17); otherwise, we merge the maximum dominating subspace $D_{q<p}$ of q (line 21). At the end of each iteration, the stability measure σ' will be updated (line 23). Finally, the algorithm returns the skyline S and the dataset $\mathcal{P}$ that contains non-pruned points only (line 25), that is $\forall q \in \mathcal{P}, S \nprec q$.

In summary, Algorithm 1 is designed to merge the maximum dominating subspace of each point in order to distribute the points in the dataset to as much as possible subspaces, the stability threshold σ is set to control the algorithm. Because each merge procedure of the maximum dominating subspace requires the dominance tests against all non-pruned points in the dataset, the value of stability threshold is sensitive to the performance of our method. For small datasets, the selection of stability threshold is less important because any skyline algorithm can finish in short time; for large datasets, the stability threshold can be tested from a random sample of the dataset. We also note that any measure can be applied to stop dominating subspace merging.

5 SUBSET QUERY FOR SKYLINE INDEXING

In the previous section, we show a maximum dominating space can be assigned to each point, this principle can be used to partition and index skyline points in order to reduce the dominance tests. Lemma 4.3 shows that $D_{q_1<S} \nsupseteq D_{q_2<S}$ is necessary to determine whether $q_1 \prec q_2$ while q_1 and q_2 have been attributed a subspace generated from a set of skyline points S.

Lemma 5.1. *Given a dataset $\mathcal{P}$, let S be a set of skyline points of $\mathcal{P}$ and $D_{q<S}$ be the superposed dominating subspace of a point $q \in \mathcal{P}$, where all points dominated by S have been pruned. Let p be a skyline point in $\mathcal{P}$, then, the dominance tests to determine whether a point $q \in \mathcal{P}$ is a skyline point can be done only with each skyline point p such that $D_{p<S} \supseteq D_{q<S}$, in the condition of presorting.* ■

Proof. In the condition of presorting like [1, 2, 7, 8, 18], it is enough to compare a point with all known skyline points to determine whether the testing point is in the skyline. The rest of the proof is as the proof of Lemma 4.3, if q_1 in Lemma 4.3 is a skyline point. □

Our result shows that subspace based partitioning of skyline points can significantly reduce the dominance tests: a testing point is necessary to compare only with the skyline points with the maximum dominating subspace specified in Lemma 5.1. Therefore, if the skyline points can be stored and retrieved by a generic container, then this container can be used to improve any skyline algorithm by reducing the total number of dominance tests. So we have the following problem statements.

Problem 1. Design a data structure with which:

(1) *each skyline point can be stored/indexed and partitioned by its maximum dominating subspace;*
(2) *given a subspace D_q of a testing point q, all skyline points with any subspace $D' \supseteq D_q$ can be efficiently returned.*

Let $D_q^{\neg}$ denote the reversed maximum dominating subspace with respect to D of a point q, then the above problem is to find all subsets of $D_q^{\neg}$ in order to retrieve associated skyline points.

Problem 2. Given a set $\mathcal{X}$ of m subsets of a universe D, for any query set $Q \subseteq D$, return the set $\{Q' \in \mathcal{X} \mid Q' \subset Q\}$.

Many studies have been addressed to the subset query problem. The recent result [5] shows that the subset query can be accomplished in $O(\frac{dM}{c})$ time. In our context, where d is the the dimensionality, M is the number of maximum dominating subspaces, and $c \leq M$ is a constant. We propose a very simple data structure to resolve our reversed subset query problem in $O(\frac{d}{2})$ average time for adding a point and in $O((\frac{d}{2})^2)$ average time for query, note that $d \ll M$.

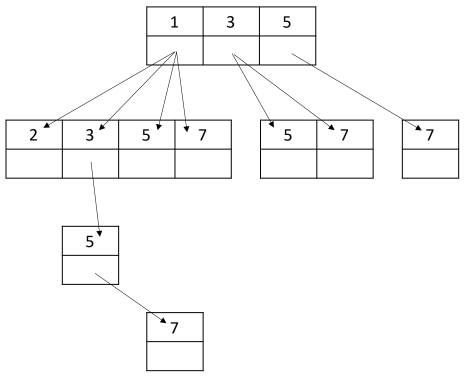


Figure 3: A map based data structure for subset query.

Figure 3 shows our data structure, which is a prefix tree like structure based on maps, where each value of the key-value pair of a map consists in two parts: a set of skyline points (IDs, references, or pointers, for instance) and a set of sub-maps (references or pointers, for instance). Each key-value pair is considered as a *node*. In considering the access cost, hash map is the best choice for constant insertion and retrieval of nodes, hence, the following description and analysis are all based on hash map. Indeed, any map implementation can be used to construct the proposed data structure, for instance, a sorted map, but in this case, the access will be no longer in constant time but in log time. The index size is the map-based prefix tree that contains all dimensions in concerned subspaces plus the total number of skyline points represented by IDs, references, or pointers.

In the example illustrated in Figure 3, the reversed maximum dominating subspaces are organized by the index of dimensions, where in this example we have the following subspaces

$$\{\{1,2\},\{1,3,5,7\},\{1,5\},\{1,7\},\{3,5\},\{3,7\},\{5,7\}\}$$

in the tree. Each node consists of the index of subspace and the set of all points (any point can be represented by a pointer, a reference, or its ID, a real copy is not necessary) assigned with this subspace. Given a query set $\{1,3,5\}$, we first locate the node 1, the retrieve all points associated with this node, and then, from the node 1, we locate the node 3 with retrieving all associated points, and the node 5. In order to find all subsets of $\{1,3,5\}$, the node 3 and the node 5 at the first level (root node) should also be accessed.

We propose the following two subset query algorithms to store and retrieve skyline points. We suppose that each point in the dataset has been attributed a maximum dominating subspace computed from the Algorithm 1, represented by a bit set (which is supported by many programming languages) or a binary vector of size d.

Algorithm 2 is quite simple. With respect to the data structure shown in Figure 3, we let the initial node be the root node (line 1). For each dimension in the reversed subspace $D\neg_q$ (the index of the *true* bit with respect to a bit set or of the value 1 with respect to a binary vector), we get the last node by a simple loop (line 2 to line 4). We define that the $get(i)$ method (line 3) of a node returns the sub-node with the index value i, which can

Algorithm 2: Store

```
  Input: A point $q$ with a subspace $D_q$
1 $node \leftarrow root$
2 foreach $i \in D_q^\neg$ do
3     $node \leftarrow node.get(i)$
4 $node.put(q)$
```

finish in constant time while using a hash map to implement the data structure, or in $O(\log d)$ time if a sorted map is used. If the sub-node with the index value i does not exist, the $get(i)$ method create the sub-node. Finally, the point q is added to the last node (line 5) by the method $put(q)$.

Lemma 5.2. *Algorithm 2 finishes in $O(1)$ time in the best case, in $O(d-1)$ time in the worst case, and in $O(\frac{d}{2})$ time in the average case.* ■

Proof. In the best case, $|D_q^\neg| = 1$, the algorithm finishes immediately. In the worst case, $|D_q^\neg| = d-1$ because q is not dominated by any initial skyline points, therefore the algorithm requires $d-1$ retrievals of sub-node, which requires $O(d-1)$ time with hash map based data structure. In the average case, the mean size of subspaces can be computed by dividing the sum of the total size of all subspaces (which is known as $d2^{d-1}$) by the total number of subspaces (which is 2^d), we have $\frac{d2^{d-1}}{2^d} = \frac{d}{2}$. Hence, this average time complexity is $O(\frac{d}{2})$. □

Lemma 5.2 also shows that in the case of $d = 2$, Algorithm 2 finishes always in $O(1)$, however, there will be only two independent nodes at the top level to store skyline points, therefore, our method can contribute very limited performance improvement.

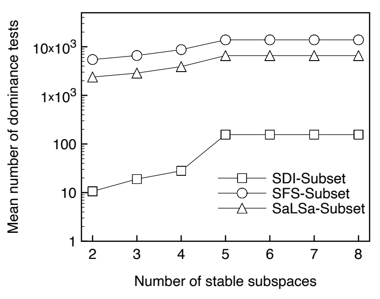
Algorithm 3: Query

```
  Input: A subspace $D_q$
  Output: A set $S$ of points such that $\forall p \in S, D_p \supseteq D_q$
1 $S \leftarrow root.points$
2 foreach $node \in root.nodes$ do
3     if $node.index \in D_q^\neg$ then
4         $query(D_q^\neg, node, S)$
5 return $S$
```

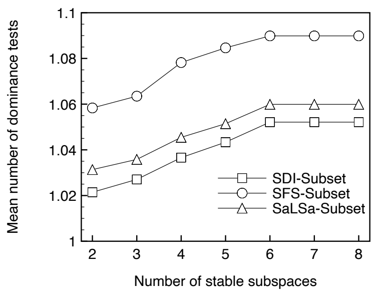
Algorithm 4: Recursive query

```
  Input: A subspace $D_q$, a node, and a set $S$ of points such
         that $\forall p \in S, D_p \supseteq D_q$
1 $S \leftarrow S \cup node.points$
2 foreach $node \in node.nodes$ do
3     if $node.index \in D_q^\neg$ then
4         $query(D_q^\neg, node, S)$
```

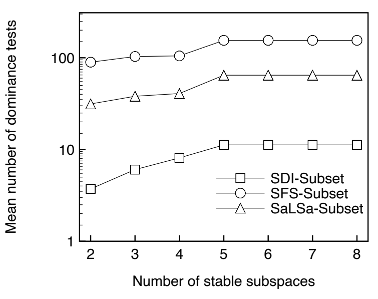
Algorithm 3 presents a recursive retrieval of partitioned skyline points with respect to a given subspace D_q, all points distributed to any super set of D_q will be returned. Algorithm 4 corresponds to the *query* method in the line 4 of Algorithm 3. The skyline point set S is updated by each call of the method



(a) 8-D 100K AC dataset.

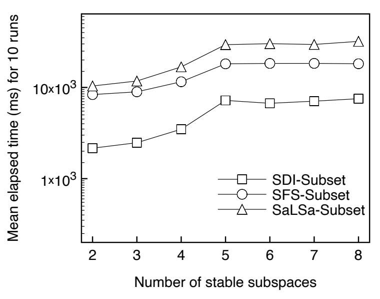


(b) 8-D 100K CO dataset.

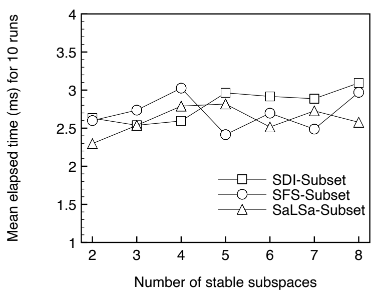


(c) 8-D 100K UI dataset.

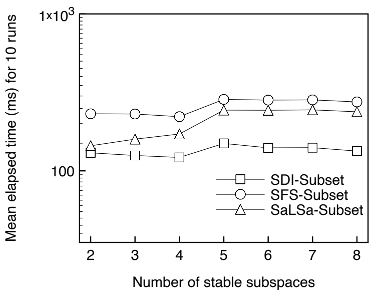
Figure 4: Mean dominance test numbers with respect to stable subspaces on 8-D 100K datasets.



(a) 8-D 100K AC dataset.



(b) 8-D 100K CO dataset.



(c) 8-D 100K UI dataset.

Figure 5: Elapsed processor time (ms) for 10 runs with respect to stable subspaces 8-D 100K datasets.

query (Algorithm 4), which is initialized by all points distributed to the root node (line 1 of Algorithm 3).

Lemma 5.3. *Given a subspace D_q, Algorithm 3 returns the set S such that $\forall p \in S, D_p \supseteq D_q$ in $O(1)$ time in the best case, in*

$$O(\frac{(d-1)(d-2)}{2})$$

time in the worst case, and in

$$O(\frac{(d/2)(d/2-1)}{2})$$

time in the average case. The average time complexity of Algorithm 3 can be considered as $O((\frac{d}{2})^2)$. ■

Proof. Let $|D_q^-| = n$, then $\frac{n(n-1)}{2}$ tests must be done with respect to the data structure shown in Figure 3 to retrieve all subsets of D_q^-, that is, all super sets of D_q. In the best case, $n = 1$, 1 test returns the subset of D_q^-; in the worst case, $n = d - 1$, hence $O(\frac{(d-1)(d-2)}{2})$ time is required. In the average case, as shown in the proof of Lemma 5.2, $n = \frac{d}{2}$, therefore, Algorithm 3 terminates in $O(\frac{(d/2)(d/2-1)}{2})$ time, which can be considered in $O((\frac{d}{2})^2)$ time complexity. □

We note that the dimensionality d of the dataset is much less than the cardinality N of the dataset, the $O((\frac{d}{2})^2)$ average subset query time can be ignored in comparison with $O(dN^2)$ time or $O(N \log^{(d-2)} N)$ time. However, our method is not suitable to directly partition points in a dataset because $2^d - 2$ subspaces can be generated in the worst case and in this case $O((\frac{d}{2})^2 N)$ time is required to retrieve comparable points. Hence, we propose our method to partition skyline points only, and the dominance tests can be performed in general ways. Therefore, the best application of our result is to boost sorting-based skyline algorithms.

6 EXPERIMENTAL RESULTS

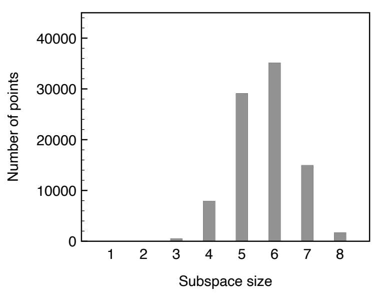
In this section, we report the experimental results of our method. We applied our method to SFS, SaLSa, and SDI algorithms without changing their original designs, the main function of our method is to store and to retrieve skyline points. The code is implemented in C++ with the C++11 standard[2] and tested on AMD Epyc 7702 2GHz CPU with 512 GB RAM.

The evaluation metrics are based on the *mean dominance test number* [14] and the *elapsed processor time*, where the mean dominance test number is defined as the ratio of the total number of dominance tests on the total number of points. All elapsed processor time results, in milliseconds, are based on the mean time of 10 runs, all data have been loaded into the main memory before counting.

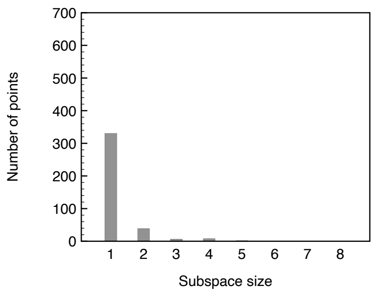
6.1 Effect of Stability Threshold

First, we study the effect of the stability threshold σ, where $1 < \sigma \leq d$ (d is the dimensionality of dataset). Note that it is meaningless to set $\sigma = 1$ because the objective of our method is to balance the distribution of points among subspaces. Figure 4 and Figure 5 show that low-value stability thresholds can effectively reduce the mean number of dominance tests however there

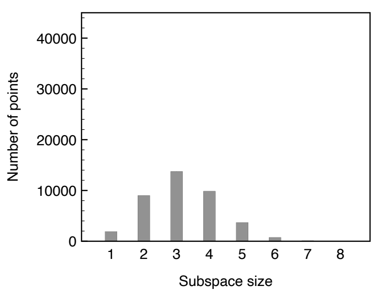
[2]The source code of our method is available at https://github.com/dominiquehli/skyline-subset. We thank Jongwuk Lee for the source code of BSkyTree-S and BSkyTree-P.



(a) 8-D 100K AC dataset.



(b) 8-D 100K CO dataset.



(c) 8-D 100K UI dataset.

Figure 6: Distribution of points with respect to subspace size where the stability threshold is set to 3.

Dimensionality	2-D	4-D	6-D	8-D	10-D	12-D	16-D	20-D	24-D	
AC datasets	55	4806	40423	95898	139770	166641	190360	197142	199048	
CO datasets	3	20	57	148	252	1303	4473	9582	20891	
UI datasets	14	382	3275	13046	37379	77200	154827	190501	198742	
Cardinality	100K	200K	300K	400K	500K	600K	700K	800K	900K	1M
AC datasets	55969	95898	131632	164204	193488	221317	247651	273151	297131	320138
CO datasets	135	148	189	203	219	238	260	259	202	208
UI datasets	55969	95898	131632	164204	193488	221317	247651	273151	297131	320138

Table 1: Skyline size of synthetic datasets.

is no exact corresponding changes in elapsed processor time on CO and UI (with SDI-Subset) datasets. Note that the format of Y axis of AC and UI datasets is logarithmic and that of CO is linear. The main reason is that AC data require a huge number of dominance tests so that the time variance among different dominance tests can be statistically neutralized, however CO and UI data require much less dominance tests so the different dominance test time makes sense. Indeed, in comparison with Figure 2, Figure 6 can show that while $\sigma = 3$, the most of points in AC data are distributed in high subspaces but that in CO and UI data are distributed in low subspaces, which requires less skyline index accesses and the index is much smaller. We have the similar results on other AC/CO/UI datasets with different dimensionality and cardinality, where the fastest σ for SDI-Subset is around $d/3$. Therefore, in the reported performance evaluations, the stability threshold σ is set to rounded $d/3$.

6.2 Effect of Data Type

We now report the improvements of SFS, SaLSa, and SDI algorithms with the boost of our method, where the state-of-the-art algorithms BSkyTree-S and BSkyTree-P are used as the baseline. Two groups of tests have been conducted in order to study the effect of the data dimensionality and the effect of the data cardinality on synthetic AC, CO, and UI datasets. In the first group, the cardinality of all datasets is fixed to 2×10^5 (200K) points, the dimensionality varies from 2-D to 24-D; in the second group, the dimensionality of all datasets is fixed to 8-D, the cardinality varies from 10^5 (100K) to 10^6 (1M) points. Table 1 lists the skyline size of all synthetic datasets.

Because the scale of obtained values is huge, our results are presented as numbers listed in Table 2 — Table 13, where the algorithms with -Subset suffix are boosted by our method. Furthermore, we introduce the *performance gains* metric into the presented tables, which is calculated as the ration of any value obtained without boosting on the value boosted by our method. If there is no performance gain, we mark it as "–".

From Table 2 to Table 5, we study the the performance of our method on AC data. It is not surprising that BSkyTree-P is the absolute winner on AC data, and the boost of our method is very limited. We can preview this result from the previous analysis. However, in high-dimensional data, for instance 20-D and 24-D datasets, our method can boost SFS, SaLSa, and SDI up to 30 to 40 times. The reason is that the 200K points have been distributed till to 2^{20} and 2^{24} subspaces, that is also why SDI-Subset wins BSkyTree-P in 16-D and 24-D.

The effectiveness of our method on CO data is studied from Table 6 to Table 9. Except SaLSa and SDI, all other methods require at least on scan of the full dataset. We can see that there is almost no performance gain in many datasets, as already shown in Figure 6: it is difficult to distribute CO data. We also see that SDI is the winner in mean dominance test numbers because of its early stop mechanism, as SaLSa. However SDI needs much more time to finish the computation in comparison with BSkyTree-S because it must first build the dimension index of the full dataset.

Table 10 to Table 13 prove the performance of our method. We note the in Table 10, our method cannot reduce any mean dominance test numbers on the datasets of which the dimensionality is less than 6 in comparison with SaLSa and SDI because of the same reason described with CO data. On 6-D dataset, SaLSa-Subset is the winner, lightly faster then SDI-Subset. However, from 6-D data, the boosted methods SaLSa-Subset and SDI-Subset run faster than BSkyTree-P. From 8-D datasets[3], SDI-Subset becomes the fastest algorithm on UI data.

Finally, our experimental results show that BSkyTree-P is the best choice for AC data, BSkyTree-S is always the winner on CO data and low-dimensional data (for instance, $d < 6$), and the boosted methods, particularly SDI-Subset, is best for UI

[3]The 16-D dataset is exceptional. We have tested different 16-D UI datasets from 100K to 1M points, all results are very similar to that of AC datasets.

Dimensionality	2-D	4-D	6-D	8-D	10-D	12-D	16-D	20-D	24-D
SFS	3.66836	141.496	4568.9	23648.6	49406.2	69797.2	90707.7	97192	99059.0
SFS-Subset	3.66836	301.28	2010.01	4884.64	10568.8	15764.2	11408.7	2059.39	2182.93
Performance Gain	—	—	× 2.27	× 4.84	× 4.67	× 4.43	× 7.95	× 47.19	× 45.38
SaLSa	**1.05371**	41.3559	1780.08	10883.6	25035.1	37825.5	53188.3	58872	61164.7
SaLSa-Subset	3.66836	210.186	702.753	1871.68	4751.66	7716.71	6154.24	1195.77	1264.72
Performance Gain	—	—	× 2.53	× 5.81	× 5.27	× 4.90	× 8.64	× 49.23	× 48.36
SDI	4.1517	57.3677	611.713	1775.54	2908.51	3544.81	3975.69	3748.68	3538.47
SDI-Subset	3.66836	42.8875	147.725	260.142	601.158	911.765	655.527	**132.627**	**125.826**
Performance Gain	× 1.13	× 1.34	× 4.14	× 6.83	× 4.84	× 3.89	× 6.06	× 28.26	× 28.12
BSkyTree-S	2.77378	62.2657	1037.81	2642.78	3251.29	2770.63	1570.63	1375.05	803.409
BSkyTree-P	2.96938	**20.8456**	**70.0527**	**153.205**	**240.783**	**263.435**	**382.818**	359.256	466.324

Table 2: Mean dominance test numbers on synthetic 200K AC dataset with respect to the dimensionality from 2-D to 24-D.

Dimensionality	2-D	4-D	6-D	8-D	10-D	12-D	16-D	20-D	24-D
SFS	32.6347	316.748	11519.5	79646.1	200778	290403	438638	513578	495081.0
SFS-Subset	14.4623	794.527	6220.12	18914.7	35340.6	55567.1	39347.6	20817.5	16630.1
Performance Gain	× 2.26	—	× 1.85	× 4.21	× 5.68	× 5.23	× 11.15	× 24.67	× 29.77
SaLSa	31.942	149.111	5897.42	39627.7	134451	174043	294857	338326	410082.0
SaLSa-Subset	17.4618	785.225	7097.27	25528.3	50362.1	93603	98041.8	25432.1	17157.5
Performance Gain	× 1.83	—	—	× 1.55	× 2.67	× 1.86	× 3.01	× 13.30	× 23.90
SDI	48.2699	427.674	7512.17	27096.3	42502.4	52725.2	60998.5	57874.9	52973.4
SDI-Subset	15.7008	538.322	1579.89	5197.69	10514.9	14936.4	12372.4	**12074.1**	**9632.24**
Performance Gain	× 3.07	—	× 4.75	× 5.21	× 4.04	× 3.53	× 4.93	× 4.79	× 5.50
BSkyTree-S	**8.2063**	382.894	12472.8	36874.9	59113.1	58579	52648.1	51648.3	40451.4
BSkyTree-P	8.7188	**82.9531**	**696.73**	**2139.81**	**4062.37**	**5389.26**	**10650**	17095.5	29731.9

Table 3: Elapsed processor time (ms) on synthetic 200K AC dataset with respect to the dimensionality from 2-D to 24-D.

Cardinality	100K	200K	300K	400K	500K	600K	700K	800K	900K	1M
SFS	16094.3	23648.6	29744	34746.4	38630.9	42150.5	45269.6	48205	50733.8	53040.0
SFS-Subset	5448.8	4884.64	6506.3	9785.86	12398	12136.6	14463.9	15638.7	16222.5	18461.7
Performance Gain	× 2.95	× 4.84	× 4.57	× 3.55	× 3.12	× 3.47	× 3.13	× 3.08	× 3.13	× 2.87
SaLSa	7686.93	10883.6	13400.6	15422.7	16974.9	18335.3	19523.3	20675.1	21621.6	22470.2
SaLSa-Subset	2380.6	1871.68	2471.55	3748.81	4780.51	4588.65	5453.09	5776.88	5950.9	6818.11
Performance Gain	× 3.23	× 5.81	× 5.42	× 4.11	× 3.55	× 4	× 3.58	× 3.58	× 3.63	× 3.30
SDI	1197.53	1775.54	2285.43	2715.21	3057.96	3651.34	3679.31	3955.05	4206.04	4439.36
SDI-Subset	380.506	260.142	347.751	592.091	773.557	706.018	991.316	1005.2	1111.23	1323.31
Performance Gain	× 3.15	× 6.83	× 6.57	× 4.59	× 3.95	× 5.17	× 3.71	× 3.93	× 3.79	× 3.35
BSkyTree-S	1840.34	2642.78	3294.33	3825.6	4226.94	4585.38	4924.65	5224.82	5495.08	5746.91
BSkyTree-P	**138.385**	**153.205**	**165.693**	**172.064**	**179.472**	**184.118**	**185.679**	**189.311**	**191.758**	**191.702**

Table 4: Mean dominance test numbers on synthetic 8-D AC dataset with respect to the cardinality from 100K to 1M.

Cardinality	100K	200K	300K	400K	500K	600K	700K	800K	900K	1M
SFS	19460.6	79646.1	217338	397901	443378	569681	1606980	2139990	2393550	2710520.0
SFS-Subset	5626.8	18914.7	28654.9	54100.5	96181.3	115770	166344	223040	269236	374428.0
Performance Gain	× 3.46	× 4.21	× 7.58	× 7.35	× 4.61	× 4.92	× 9.66	× 9.59	× 8.89	× 7.24
SaLSa	17124.8	39627.7	217466	420102	572934	433723	931318	1159820	1112750	1204340.0
SaLSa-Subset	7106.61	25528.3	28876.9	60517.9	106613	130710	174473	230938	283176	398255.0
Performance Gain	× 2.41	× 1.55	× 7.53	× 6.94	× 5.37	× 3.32	× 5.34	× 5.02	× 3.93	× 3.02
SDI	5348.27	27096.3	44927.6	84776.9	131919	199842	248619	316241	386424	459770.0
SDI-Subset	1443.94	5197.69	5854.34	11830	20698.5	23743.9	37005.5	43266.8	70195.4	94795.2
Performance Gain	× 3.70	× 5.21	× 7.67	× 7.17	× 6.37	× 8.42	× 6.72	× 7.31	× 5.50	× 4.85
BSkyTree-S	8754.06	36874.9	65591.6	113072	169633	232899	302621	379215	450423	503851.0
BSkyTree-P	**855.191**	**2139.81**	**3231.9**	**4597**	**6280.92**	**7493.29**	**9186.88**	**10759.3**	**12452.9**	**13986.8**

Table 5: Elapsed processor time (ms) on synthetic 8-D AC dataset with respect to the cardinality from 100K to 1M.

Dimensionality	2-D	4-D	6-D	8-D	10-D	12-D	16-D	20-D	24-D
SFS	1	1.00095	1.01652	1.09495	1.21073	6.14055	53.3143	233.428	1095.44
SFS-Subset	1	1.00035	1.01202	1.04543	1.06489	2.51762	9.72743	18.7451	45.6721
Performance Gain	—	× 1	× 1	× 1.05	× 1.14	× 2.44	× 5.48	× 12.45	× 23.98
SaLSa	0.000015	0.00032	0.01269	0.054905	0.09861	1.74134	18.7257	91.0814	474.549
SaLSa-Subset	1	1.00005	1.00721	1.02232	1.03373	1.46619	3.07304	7.96234	20.3894
Performance Gain	—	—	—	—	—	× 1.19	× 6.09	× 11.44	× 23.27
SDI	**0**	**0.00014**	**0.003715**	**0.02436**	**0.03205**	**1.08491**	8.20914	21.5959	52.8217
SDI-Subset	1	1.00004	1.00556	1.01909	1.0234	1.30619	**1.86312**	**3.4836**	**6.7105**
Performance Gain	—	—	—	—	—	—	× 4.41	× 6.20	× 7.87
BSkyTree-S	1.00011	1.00383	1.07056	1.98896	8.24004	3.38524	13.6984	19.9785	33.2862
BSkyTree-P	1.0001	1.00379	1.07219	1.98422	8.22813	3.14621	13.0169	17.1449	33.0747

Table 6: Mean dominance test numbers on synthetic 200K CO dataset with respect to the dimensionality from 2-D to 24-D.

Dimensionality	2-D	4-D	6-D	8-D	10-D	12-D	16-D	20-D	24-D
SFS	20.4586	29.0367	38.259	40.8126	40.8972	58.4573	204.267	783.707	3411.88
SFS-Subset	2.0127	2.838	5.11	8.6771	7.2286	37.1798	331.794	1179.47	4011.02
Performance Gain	× 10.16	× 10.23	× 7.49	× 4.70	× 5.66	× 1.57	—	—	—
SaLSa	25.8268	27.3071	29.3963	30.6528	30.7115	40.746	97.8189	353.89	1760.31
SaLSa-Subset	2.109	3.6676	4.2471	6.4295	7.0722	20.3191	140.599	474.649	1540.02
Performance Gain	× 12.25	× 7.45	× 6.92	× 4.77	× 4.34	× 2.01	—	—	× 1.14
SDI	37.8009	91.3422	135.347	178.336	221.828	276.807	440.341	659.926	1064.64
SDI-Subset	2.4486	3.9579	3.9521	6.3952	7.6565	20.9283	**95.6926**	**266.429**	**892.259**
Performance Gain	× 15.44	× 23.08	× 34.25	× 27.89	× 28.97	× 13.23	× 4.60	× 2.48	× 1.19
BSkyTree-S	**0.0732**	**0.1962**	**0.2866**	**0.8575**	**2.0497**	**13.768**	135.577	281.453	933.818
BSkyTree-P	0.8813	1.4264	2.1908	6.1156	11.9934	23.1115	124.2	330.976	1209.79

Table 7: Elapsed processor time (ms) on synthetic 200K CO dataset with respect to the dimensionality from 2-D to 24-D.

Cardinality	100K	200K	300K	400K	500K	600K	700K	800K	900K	1M
SFS	1.15714	1.09495	1.11023	1.0878	1.09627	1.10625	1.1115	1.10019	1.0526	1.05208
SFS-Subset	1.05833	1.04543	1.04546	1.03052	1.03156	1.0359	1.03737	1.03598	1.01502	1.0148
Performance Gain	× 1.09	× 1.05	× 1.06	× 1.06	× 1.06	× 1.07	× 1.07	× 1.06	× 1.04	× 1.04
SaLSa	0.09037	0.0549	0.0568	0.042585	0.041912	0.0379	0.0393	0.03806	0.01542	0.015499
SaLSa-Subset	1.03135	1.02232	1.02243	1.01315	1.01379	1.01574	1.01626	1.01845	1.00788	1.00828
Performance Gain	—	—	—	—	—	—	—	—	—	—
SDI	**0.03404**	**0.02436**	**0.03979**	**0.02133**	**0.02357**	**0.02051**	**0.0209586**	**0.022**	**0.004756**	**0.004795**
SDI-Subset	1.02565	1.01909	1.01635	1.00973	1.00999	1.01229	1.01224	1.01482	1.00575	1.0057
Performance Gain	—	—	—	—	—	—	—	—	—	—
BSkyTree-S	2.37996	1.98896	1.19064	1.74626	1.85352	1.15658	1.7801	1.95128	1.02078	1.10242
BSkyTree-P	2.35791	1.98422	1.19132	1.74511	1.8546	1.15933	1.78536	1.95586	1.02092	1.10346

Table 8: Mean dominance test numbers on synthetic 8-D CO dataset with respect to the cardinality from 100K to 1M.

Cardinality	100K	200K	300K	400K	500K	600K	700K	800K	900K	1M
SFS	10.4544	40.8126	47.7369	66.9709	106.479	122.469	122.189	139.056	155.7	171.368
SFS-Subset	2.2779	8.6771	13.5451	13.7252	31.7945	21.079	21.1029	22.3662	42.2938	32.2714
Performance Gain	× 4.59	× 4.70	× 3.52	× 4.88	× 3.35	× 5.81	× 5.79	× 6.22	× 3.68	× 5.31
SaLSa	14.084	30.6528	44.4031	59.58	74.613	91.1816	103.349	119.546	134.693	146.871
SaLSa-Subset	1.935	6.4295	7.4033	8.454	10.7574	14.6004	15.0803	17.5672	20.1752	21.2981
Performance Gain	× 7.28	× 4.77	× 6	× 7.05	× 6.94	× 6.25	× 6.85	× 6.81	× 6.68	× 6.90
SDI	79.3928	178.336	256.307	343.177	442.941	536.764	628.812	724.978	810.822	913.288
SDI-Subset	2.8573	6.3952	10.4126	11.9223	23.9418	17.4361	20.745	25.2528	34.5934	31.4668
Performance Gain	× 27.79	× 27.89	× 24.62	× 28.78	× 18.50	× 30.78	× 30.31	× 28.71	× 23.44	× 29.02
BSkyTree-S	**0.7982**	**0.8575**	**1.5527**	**2.0601**	**3.6962**	**3.2064**	**3.434**	**3.8724**	**2.0915**	**2.0029**
BSkyTree-P	3.9025	6.1156	6.1724	13.3343	16.8883	14.1379	22.9712	28.8244	16.5354	21.6284

Table 9: Elapsed processor time (ms) on synthetic 8-D CO dataset with respect to the cardinality from 100K to 1M.

Dimensionality	2-D	4-D	6-D	8-D	10-D	12-D	16-D	20-D	24-D
SFS	1.00995	2.07385	38.3914	459.212	3568.61	15015.3	60021.5	90753.8	98751.0
SFS-Subset	1.00995	1.59669	17.9581	106.088	710.948	3082.02	26693.7	11347	2034.05
Performance Gain	—	× 1.30	× 2.14	× 4.33	× 5.02	× 4.87	× 2.25	× 8	× 48.55
SaLSa	0.00975	**0.532165**	12.3735	167.818	1430.21	6809.19	32449.3	53197.5	58806.3
SaLSa-Subset	1.00995	1.39582	7.60971	43.9786	273.812	1364.84	14261.8	6486.06	1194.34
Performance Gain	—	—	× 1.63	× 3.82	× 5.22	× 4.99	× 2.28	× 8.20	× 49.24
SDI	**0.00602**	0.54185	12.7038	75.1358	371.175	1305.16	2884.92	3712.77	3484.04
SDI-Subset	1.00995	1.11015	**3.02379**	**10.2973**	**47.508**	185.514	1512.53	645.27	**114.729**
Performance Gain	—	—	× 4.20	× 7.30	× 7.81	× 7.04	× 1.91	× 5.75	× 30.37
BSkyTree-S	1.01232	7.0461	31.7958	133.137	332.591	717.365	1137.52	936.944	558.125
BSkyTree-P	1.01279	7.21108	25.7719	79.2428	93.519	**158.027**	**529.538**	**582.166**	432.831

Table 10: Mean dominance test numbers on synthetic 200K UI dataset with respect to the dimensionality from 2-D to 24-D.

Dimensionality	2-D	4-D	6-D	8-D	10-D	12-D	16-D	20-D	24-D
SFS	23.1289	40.7084	120.397	1268.57	11673.8	68903.9	313820	472481	559174.0
SFS-Subset	3.3954	12.6331	99.9347	696.141	2832.31	16612.6	122205	46537.1	13353.8
Performance Gain	× 6.81	× 3.22	× 1.20	× 1.82	× 4.12	× 4.15	× 2.57	× 10.15	× 41.87
SaLSa	20.8913	25.4035	72.0775	608.382	5749.55	32369	235476	321931	355698.0
SaLSa-Subset	3.0395	9.6911	**59.2101**	403.433	2367.69	18146.9	249288	100806	17245.0
Performance Gain	× 6.87	× 2.62	× 1.22	× 1.51	× 2.43	× 1.78	—	× 3.19	× 20.63
SDI	35.8351	82.9728	265.237	988.093	5087.94	23951.7	52147.6	75153.3	60557.6
SDI-Subset	2.3276	11.2018	83.3118	**337.974**	**755.396**	**2447.71**	31524	**12808.6**	**9872.25**
Performance Gain	× 15.40	× 7.41	× 3.18	× 2.92	× 6.74	× 9.79	× 1.65	× 5.87	× 6.13
BSkyTree-S	**0.1235**	**7.6218**	104.15	896.382	5651.38	15539.5	42210.9	38429	32554.3
BSkyTree-P	1.155	24.1729	95.2192	434.162	1146.56	2879.16	**13397.5**	27845.5	32266.7

Table 11: Elapsed processor time (ms) on synthetic 200K UI dataset with respect to the dimensionality from 2-D to 24-D.

Cardinality	100K	200K	300K	400K	500K	600K	700K	800K	900K	1M
SFS	460.313	459.212	499.667	531.194	522.63	507.032	522.836	528.471	520.552	507.664
SFS-Subset	89.5718	106.088	129.523	122.864	120.545	114.264	141.636	144.664	139.262	139.63
Performance Gain	× 5.14	× 4.33	× 3.86	× 4.32	× 4.34	× 4.44	× 3.69	× 3.65	× 3.74	× 3.64
SaLSa	175.827	167.818	176.903	185.034	179.431	171.196	176.839	178.822	174.434	167.308
SaLSa-Subset	31.4026	43.9786	38.6649	39.2604	36.4398	34.823	38.6292	39.7039	37.9282	37.959
Performance Gain	× 5.60	× 3.82	× 4.58	× 4.71	× 4.92	× 4.92	× 4.58	× 4.50	× 4.60	× 4.41
SDI	70.879	75.1358	86.6211	96.2773	97.7019	101.154	103.341	105.916	105.649	105.536
SDI-Subset	**8.84229**	**10.2973**	**9.57091**	**9.85854**	**9.60977**	**9.08958**	**9.54601**	**9.87711**	**8.79965**	**8.82683**
Performance Gain	× 8.02	× 7.30	× 9.05	× 9.77	× 10.17	× 11.13	× 10.83	× 10.72	× 12.01	× 11.96
BSkyTree-S	140.838	133.137	133.913	131.735	118.279	111.169	88.9518	88.8225	162.526	100.47
BSkyTree-P	85.5219	79.2428	79.191	72.7893	61.2489	52.4952	32.0429	36.6836	99.3828	38.4963

Table 12: Mean dominance test numbers on synthetic 8-D UI dataset with respect to the cardinality from 100K to 1M.

Cardinality	100K	200K	300K	400K	500K	600K	700K	800K	900K	1M
SFS	569.967	1268.57	2519.03	3879.07	4406.86	5569.82	6462.62	9452.43	7414.07	8824.39
SFS-Subset	213.963	696.141	1213.75	1679.07	2010.92	2396.66	3437.86	4159.97	4627.68	4730.24
Performance Gain	× 2.66	× 1.82	× 2.08	× 2.31	× 2.19	× 2.32	× 1.88	× 2.27	× 1.60	× 1.87
SaLSa	320.66	608.382	1001.58	1423.56	1707.41	1943.35	2388.99	2749.06	2987.54	3272.88
SaLSa-Subset	139.294	403.433	585.847	799.217	877.945	1069.97	1547.82	1848.41	1622.37	1709.46
Performance Gain	× 2.30	× 1.51	× 1.71	× 1.78	× 1.94	× 1.82	× 1.54	× 1.49	× 1.84	× 1.91
SDI	361.305	988.093	1263.2	1879.79	2403.84	2938.16	3639.35	4111.78	4371.11	4802.71
SDI-Subset	**99.4546**	**337.974**	**386.365**	**481.068**	**550.036**	**546.609**	**898.925**	**1049.89**	**1082.65**	**1220.89**
Performance Gain	× 3.63	× 2.92	× 3.27	× 3.91	× 4.37	× 5.38	× 4.05	× 3.92	× 4.04	× 3.93
BSkyTree-S	380.996	896.382	1157.3	1599.01	1944.32	2368.51	2818.7	3104.79	3777.96	4034.15
BSkyTree-P	210.467	434.162	600.422	790.853	916.168	1012.09	1041.63	1189.88	2039.03	1428.89

Table 13: Elapsed processor time (ms) on synthetic 8-D UI dataset with respect to the cardinality from 100K to 1M.

data. Besides, the usefulness of our method is also limited in low-dimensionality domains, such as 2-D to 4-D datasets. The main reason is that the number of subspaces is not enough to boost SFS, SaLSa, and SDI: there is no subspace in 2-D dataset and only $2^4 - 2 = 14$ subspaces can be generated to distribute points. In the presented experiments, any tested algorithm can finish the skyline computation of a 4-D UI dataset with 2×10^5 (200K) points in millisecond-level, so our method can not give additional performance gain. However, on larger dataset, for instance, on a 4-D UI dataset with 10^6 (1M) points, our extended experiments show that all boosted methods, SFS-Subset, SaLSa-Subset, and SDI-Subset perform better than BSkyTree-S and BSkyTree-P, where the skyline contains 423 points, as shown in Figure 14 (DT: Mean dominance test numbers; RT: Elapsed processor time).

Method	DT	RT
SFS	1.38777	193.684 ms
SFS-Subset	1.39038	74.731 ms
Performance Gain	–	×2.59
SaLSa	0.298219	169.865 ms
SaLSa-Subset	1.17633	49.637 ms
Performance Gain	–	×3.42
SDI	0.388056	574.287 ms
SDI-Subset	1.03643	64.737 ms
Performance Gain	–	×8.87
BSkyTree-S	8.96764	111.378 ms
BSkyTree-P	9.06151	120.388 ms

Table 14: Results on 4-D UI dataset with 1M points.

Method	DT	RT	σ
SFS	149.09	29.167 ms	
SFS-Subset	104.749	23.84 ms	2
Performance Gain	×1.42	×1.22	
SaLSa	115.763	26.005 ms	
SaLSa-Subset	74.4728	24.236 ms	2
Performance Gain	×1.55	×1.07	
SDI	17.817	24.488 ms	
SDI-Subset	18.5333	22.376 ms	2
Performance Gain	–	×1.04	
BSkyTree-S	54.5101	26.769 ms	
BSkyTree-P	39.7169	17.681 ms	

Table 16: The NBA dataset.

Method	DT	RT	σ
SFS	10793.9	43279.4 ms	
SFS-Subset	9530.47	43231.2 ms ms	3
Performance Gain	×1.42	1.0001 = –	
SaLSa	3080.55	14349.4 ms	
SaLSa-Subset	2630	26169.5 ms	3
Performance Gain	×1.17	–	
SDI	537.066	10653.4 ms	
SDI-Subset	525.12	10617.3 ms	3
Performance Gain	×1.02	×1.04	
BSkyTree-S	3116.63	43495.9 ms	
BSkyTree-P	561.723	11694.2 ms	

Table 17: The WEATHER dataset.

6.3 Results on Real Datasets

We also tested our method on the three real world datasets HOUSE (6-D, 127,931 points, 5,774 skyline points), NBA (8-D, 17,264 points, 1,796 skyline points), and WEATHER (15-D, 566,268 points, 26,713 skyline points) [6] as listed from Table 15 to Table 17 (DT: Mean dominance test numbers; RT: Elapsed processor time).

Method	DT	RT	σ
SFS	173.446	298.147 ms	
SFS-Subset	135.977	245.348 ms	4
Performance Gain	×1.28	×1.22	
SaLSa	94.4504	211.248 ms	
SaLSa-Subset	73.2253	249.887 ms	4
Performance Gain	×1.29	–	
SDI	21.6086	214.724 ms	
SDI-Subset	18.9641	111.7 ms	4
Performance Gain	×1.14	×1.92	
BSkyTree-S	60.3785	287.097 ms	
BSkyTree-P	13.1706	56.219 ms	

Table 15: The HOUSE dataset.

Our experimental results show that our method can boost SFS, SaLSa, and SDI, where all stability threshold σ values have been manually adjusted. We note and analyzed the relatively limited (< 2) effectiveness on these three datasets. HOUSE is an AC type dataset, we have already shown the limits of our method with such a data type. NBA is a small dataset with only 17,264 points dataset, SaLSa cannot be boosted at all in terms of elapsed processor time because the I/O of our proposed skyline index requires additional processor time. WEATHER dataset consists of 15 dimensions, so SDI works well because it was designed for high-dimensionality domains. Besides, there are a large number of duplicate values in several dimensions in the WEATHER dataset, and this factor causes that there may be a lot of skyline points in one single node of our proposed skyline index, which will affect the performance of the skyline index. However, both SDI and SDI-Subset perform better then BSkyTree-P.

7 CONCLUSION

In this paper, we present a subset approach to efficient skyline computation, which is designed to boost sorting-based skyline algorithms. We proposed a subspace union method to assign all points a maximum dominating subspace, with which the dominance tests between skyline points and testing points are only necessary between comparable subspaces. In order to efficiently retrieve skyline points for dominance tests with respect to maximum dominating subspaces, we proposed a subset query method to index the skyline. Our theoretical analysis and experimental results show that the skyline computation can be boosted by our subspace union and subset query method. The perspective of the work presented in this paper includes: (1) extending the proposed method to AC and CO data; (2) developing a cost model to improve the stability threshold in order to find the best number of pivot points; (3) adapting the proposed method to updating data such as data streams.

ACKNOWLEDGMENT

The author benefited from the use of the cluster at the *Centre de Calcul Scientifique en Région Centre-Val de Loire*, France.

REFERENCES

[1] Ilaria Bartolini, Paolo Ciaccia, and Marco Patella. Salsa: Computing the skyline without scanning the whole sky. In *CIKM*, pages 405–414, 2006.

[2] Ilaria Bartolini, Paolo Ciaccia, and Marco Patella. Efficient sort-based skyline evaluation. *ACM Transactions on Database Systems (TODS)*, 33(4):1–49, 2008.

[3] Kenneth S Bøgh, Sean Chester, Darius Šidlauskas, and Ira Assent. Hashcube: A data structure for space-and query-efficient skycube compression. In *CIKM*, pages 1767–1770, 2014.

[4] Stephan Borzsony, Donald Kossmann, and Konrad Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[5] Moses Charikar, Piotr Indyk, and Rina Panigrahy. New algorithms for subset query, partial match, orthogonal range searching, and related problems. In *International Colloquium on Automata, Languages, and Programming*, pages 451–462. Springer, 2002.

[6] Sean Chester, Darius Šidlauskas, Ira Assent, and Kenneth S Bøgh. Scalable parallelization of skyline computation for multi-core processors. In *ICDE*, pages 1083–1094, 2015.

[7] Jan Chomicki, Parke Godfrey, Jarek Gryz, and Dongming Liang. Skyline with presorting. In *ICDE*, volume 3, pages 717–719, 2003.

[8] Jan Chomicki, Parke Godfrey, Jarek Gryz, and Dongming Liang. Skyline with presorting: Theory and optimizations. In *Intelligent Information Processing and Web Mining*, pages 595–604. Springer, 2005.

[9] Parke Godfrey, Ryan Shipley, and Jarek Gryz. Maximal vector computation in large data sets. In *VLDB*, volume 5, pages 229–240, 2005.

[10] Parke Godfrey, Ryan Shipley, and Jarek Gryz. Algorithms and analyses for maximal vector computation. *The VLDB Journal*, 16(1):5–28, 2007.

[11] Donald Kossmann, Frank Ramsak, and Steffen Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB*, pages 275–286, 2002.

[12] Hsiang-Tsung Kung, Fabrizio Luccio, and Franco P Preparata. On finding the maxima of a set of vectors. *Journal of the ACM (JACM)*, 22(4):469–476, 1975.

[13] Jongwuk Lee and Seung-won Hwang. Bskytree: scalable skyline computation using a balanced pivot selection. In *EDBT*, pages 195–206, 2010.

[14] Jongwuk Lee and Seung-won Hwang. Scalable skyline computation using a balanced pivot selection technique. *Information Systems*, 39:1–21, 2014.

[15] Jongwuk Lee and Seung-won Hwang. Toward efficient multidimensional subspace skyline computation. *The VLDB Journal*, 23(1):129–145, 2014.

[16] Ken CK Lee, Wang-Chien Lee, Baihua Zheng, Huajing Li, and Yuan Tian. Z-sky: an efficient skyline query processing framework based on z-order. *The VLDB Journal*, 19(3):333–362, 2010.

[17] Bin Liu and Chee-Yong Chan. Zinc: Efficient indexing for skyline computation. *Proceedings of the VLDB Endowment*, 4(3):197–207, 2010.

[18] Rui Liu and Dominique Li. Efficient skyline computation in high-dimensionality domains. In *EDBT*, pages 459–462, 2020.

[19] Michael Morse, Jignesh M Patel, and Hosagrahar V Jagadish. Efficient skyline computation over low-cardinality domains. In *VLDB*, pages 267–278, 2007.

[20] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD*, pages 467–478, 2003.

[21] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. Progressive skyline computation in database systems. *ACM Transactions on Database Systems (TODS)*, 30(1):41–82, 2005.

[22] Jian Pei, Wen Jin, Martin Ester, and Yufei Tao. Catching the best views of skyline: A semantic approach based on decisive subspaces. In *VLDB*, pages 253–264, 2005.

[23] Jian Pei, Yidong Yuan, Xuemin Lin, Wen Jin, Martin Ester, Qing Liu, Wei Wang, Yufei Tao, Jeffrey Xu Yu, and Qing Zhang. Towards multidimensional subspace skyline analysis. *ACM Transactions on Database Systems (TODS)*, 31(4):1335–1381, 2006.

[24] Cheng Sheng and Yufei Tao. Worst-case i/o-efficient skyline algorithms. *ACM Transactions on Database Systems (TODS)*, 37(4):1–22, 2012.

[25] Kian-Lee Tan, Pin-Kwang Eng, Beng Chin Ooi, et al. Efficient progressive skyline computation. In *VLDB*, volume 1, pages 301–310, 2001.

[26] Yufei Tao, Xiaokui Xiao, and Jian Pei. Subsky: Efficient computation of skylines in subspaces. In *ICDE*, pages 65–65. IEEE, 2006.

[27] Shiming Zhang, Nikos Mamoulis, and David W Cheung. Scalable skyline computation using object-based space partitioning. In *SIGMOD*, pages 483–494, 2009.