2-2015

On Processing Reverse k-skyband and Ranked Reverse Skyline Queries

Yunjun GAO
Zhejiang University

Qing LIU
Zhejiang University

Baihua ZHENG
Singapore Management University, bhzheng@smu.edu.sg

Mou LI
Zhejiang University

Gang CHEN
Zhejiang University

See next page for additional authors

Citation

GAO, Yunjun; LIU, Qing; ZHENG, Baihua; LI, Mou; CHEN, Gang; and LI, Qing. On Processing Reverse k-skyband and Ranked Reverse Skyline Queries. (2015). *Information Sciences*. 293, 11-34. Research Collection School Of Computing and Information Systems.
Available at: https://ink.library.smu.edu.sg/sis_research/2467

Author
Yunjun GAO, Qing LIU, Baihua ZHENG, Mou LI, Gang CHEN, and Qing LI

On processing reverse k-skyband and ranked reverse skyline queries[^#]

Yunjun Gao[a*], Qing Liu[a], Baihua Zheng[b], Li Mou[a], Gang Chen[a], Qing Li[c]

a. College of Computer Science, Zhejiang University, 38 Zheda Road, Hangzhou 310027, China
b. School of Information Systems, Singapore Management University, Singapore
c. Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Kowloon, Hong Kong

Published in Information Sciences, 2015 February, 293, 11-34.
DOI: 10.1016/j.ins.2014.08.052

Abstract: In this paper, for the first time, we identify and solve the problem of efficient *reverse k-skyband* (R*k*SB) query processing. Given a set *P* of multi-dimensional points and a query point *q*, an R*k*SB query returns all the points in *P* whose dynamic *k*-skyband contains *q*. We formalize R*k*SB retrieval, and then propose five algorithms for computing the R*k*SB of an *arbitrary* query point efficiently. Our methods utilize a conventional data-partitioning index (e.g., R-tree) on the dataset, and employ *pre-computation*, *reuse* and *pruning* techniques to boost the query efficiency. In addition, we extend our solutions to tackle an interesting variant of reverse skyline queries, namely, *ranked reverse skyline* (RRS) query where, given a data set *P*, a parameter *K*, and a preference function *f*, the goal is to find the *K* reverse skyline points that have the *minimal score* according to the user-specified function *f*. Extensive experiments using both real and synthetic data sets demonstrate the effectiveness of our proposed pruning heuristics and the performance of our proposed algorithms under a variety of experimental settings.

Keywords: Skyline, reverse k-skyband, ranked reverse skyline, query processing, algorithm

1. Introduction

The skyline operator is important for many multi-criteria decision making applications. Given a set *P* of multi-dimensional points, a *traditional/static* skyline query returns all the points in *P* that are not dominated by any other point. A point p dominates another point p' if p is not worse than p' in all dimensions, and p is strictly better than p' in at least one dimension. Fig. 1(a) shows a classical example of the static skyline over a hotel dataset $S_h = \{p_1, p_2, \ldots, p_{15}\}$ in a 2-dimensional (2D) space, where the x-axis represents the room *price* of each hotel and the y-axis captures the *distance* from each hotel to the

^ A preliminary version of this work has been published in the Proceedings of the 17th International Conference on Database Systems for Advanced Applications (DASFAA 2012). Substantial new technical materials have been added to this journal submission. Specifically, the paper extends the DASFAA 2012 paper by mainly including (i) a comprehensive overview of the related work (Section 2), (ii) improved RkSB query algorithms by employing the reuse mechanism and the novel pruning heuristic (Section 3), (iii) a new variant of reverse skyline queries, namely, RRS query, and its corresponding search algorithms (Section 4), and (iv) an enhanced experimental evaluation that incorporates the new type of queries (Section 5).

Notes: (i) This manuscript is the authors' original work and has not been published nor has it been submitted simultaneously elsewhere, except for the preliminary version (i.e., [26]) mentioned previously; (ii) the main differences between the conference version and this submission are stated above; and (iii) all authors have checked the manuscript and have agreed to the submission.

* Corresponding author. E-mail addresses: gaoyj@zju.edu.cn (Y. Gao), liuq@zju.edu.cn (Q. Liu), bhzheng@smu.edu.sg (B. Zheng), moul@zju.edu.cn (L. Mou), cg@zju.edu.cn (G. Chen), itqli@cityu.edu.hk (Q. Li).

beach. Hotel p_1 dominates hotel p_2 due to its cheaper price and shorter distance to the beach. Since hotels p_1, p_4, p_5, and p_{14} are not dominated by others, they constitute the static skyline of the dataset S_h. The skyline query can be applied to the multi-criteria decision-making or some queries with respect to users' preferences.

As shown in the above example, the objects' attributes considered in the traditional skyline query are *static*. Different from the traditional skyline query, a dynamic skyline query takes into account the attributes of every point that are calculated w.r.t. user-defined functions or query points dynamically. Specifically, given a query point q, each point p is mapped to a new point p' $\langle |p[1]-q[1]|, |p[2]-q[2]|, \ldots, |p[d]-q[d]| \rangle$ with d representing the dimensionality of the dataset. Assume that a dynamic skyline query is issued at p_5 $\langle 3,2 \rangle$, Fig. 1(b) illustrates all the new points p'_i. For example, p_1 $\langle 1,9 \rangle$ is mapped to p'_1 $\langle 5,9 \rangle$. The dynamic skyline is to retrieve all the points in P that are not dynamically dominated w.r.t. a given query point q. Back to Fig. 1(b), p_1 is dominated by p_3 w.r.t. p_5, and the dynamic skyline query returns p_2, p_3, p_4, and p_{15} as the result w.r.t. p_5.

Based on the dynamic skyline query, reverse skyline is firstly introduced in [9]. In general, if q belongs to the dynamic skyline of a certain point p, p is said to be a reverse skyline point of q. A reverse skyline query finds all the points in P that have a given query point q as a member of their dynamic skylines. For instance, Fig. 1(c) depicts the reverse skyline w.r.t. q, i.e., p_3, p_{10}, p_{12}, and p_{13}. The reverse skyline operator is useful for many applications such as business location planning [9,39] and environmental monitoring [23,37]. As an example, suppose Apple wants to identify the potential customers of its new iPad mini before its formal launch. The senior manager can take the iPad mini as a query point q and the potential customer preferences (e.g., display, screen size, memory size, price, etc.) as a dataset P, and then perform the reverse skyline query for marketing analysis.

Although the reverse skyline query is helpful, it may not well match users' requests especially when the result returned by reverse skyline retrieval is very small. Back to the Apple iPad mini example. If the potential customer base of iPad mini is very small, Apple might need to adjust certain settings in order to ensure a large user base. However, it might not be easy to change the location of q in the data space such that more points actually include q in their dynamic skylines. Moreover, under some circumstances, the location of q is fixed, and thus can not be changed. As an alternative, in this paper, we propose a new operator, namely, *reverse k-skyband* (R*k*SB) query. Given a multi-dimensional data set P and a query point q, an R*k*SB query retrieves all the points in P whose *dynamic k-skyband* contains q. The formal definition is to be presented in Section 3. Fig. 1(d) is an example of the reverse 1-skyband of q, including p_3, p_4, p_5, p_9, p_{10}, p_{12}, and p_{13}. Compared with the reverse skyline shown in Fig. 1(c), the result set of reverse 1-skyband is larger. Hence, it can provide users more useful information. In addition to the R*k*SB query, we propose another variant of reverse skyline queries, i.e., *ranked reverse skyline* (RRS) query. In particular, given a dataset P, a parameter K, and a preference function f, an RRS query returns the K reverse skyline points that have the minimal scores according to the input function f. Take Fig. 1(c) as an example again, and suppose $K = 2$ and $f = L_1$-norm. Since $f(p_3) = 12$, $f(p_{12}) = 9$, and $f(p_{13}) = 13.8$, the result set of RRS retrieval w.r.t. q is $\langle p_{12}, 9 \rangle$, $\langle p_3, 12 \rangle$ in this order. Here is the example application of the RRS query. When buying a new computer, different users have different preferences. For instance, user A may think the capacity of memory and hard disk is most important, while user B may rank the speed of the processor as the most important factor. The traditional reverse skyline query assumes that all the attributes are equally important, which cannot reflect the users' real preferences. RRS retrieval can customize the preference function in order to address this deficiency.

Actually, the R*k*SB query is the generalization of the reverse skyline query. As discussed in Section 2, the R*k*SB query is different from the reverse skyline query and its variants based on their definitions and query processing techniques. Thus, the existing methods specifically designed for the reverse skyline query and its variations are not (directly) applicable to tackle the R*k*SB query efficiently. Therefore, in this paper, we propose five algorithms to support efficient processing of the R*k*SB query, viz., *Branch-bound-based RkSB algorithm* (BR*k*SB), *Pre-computation-based RkSB algorithm* (PR*k*SB), *Optimized PRkSB* (OPR*k*SB), *Reused-based RkSB algorithm* (RR*k*SB), and *Global-skyband-based RkSB algorithm* (GR*k*SB). Our approaches utilize a conventional data-partitioning index (e.g., R-tree [3]) on the dataset, and employ *offline pre-computation*, *reuse*, and *pruning* techniques to improve the query efficiency. Specifically, our algorithms employ a filter-refinement framework, develop efficient pruning heuristics using dynamic k-skyband and global k-skyband to boost the query performance, and adopt the reuse technique to avoid the repeated traversal of the R-tree. Furthermore, these techniques can be easily extended to support RRS search.

In brief, the key contributions of this paper are summarized as follows.

- We formalize the R*k*SB query, an interesting variant of reverse skyline queries, and identify its characteristics. To the best of our knowledge, this work is the first attempt on this problem.
- We develop several algorithms, i.e., BR*k*SB, PR*k*SB, OPR*k*SB, RR*k*SB, and GR*k*SB, to answer R*k*SB retrieval, which employ offline pre-computation, reuse, and pruning techniques to improve the query performance
- We propose a variant of reverse skyline queries, namely, ranked reverse skyline (RRS) query, and extend our techniques to handle RRS search.
- We conduct extensive experiments with both real and synthetic datasets to verify the efficiency and scalability of our proposed algorithms under a variety of settings.

A preliminary version of this work has been published in [26]. In this paper, we extend that work by (1) including a more comprehensive review of the related work, (2) improving the R*k*SB query processing algorithm via employing the reuse

mechanism and the new pruning heuristic, (3) proposing the RRS query and its processing algorithms, and (4) enhancing experimental evaluation to incorporate the new classes of queries.

The rest of the paper is organized as follows. Section 2 reviews the related work. Section 3 formalizes the R*k*SB query, presents five R*k*SB query processing algorithms, and analyzes their correctness. The RRS query and its corresponding algorithms are proposed in Section 4. Section 5 presents the performance evaluation of the proposed algorithms and reports our findings. Finally, Section 6 concludes the paper with some directions for future work.

2. Related work

In this section, we first overview the previous work on the skyline query and its variants in Section 2.1, and then survey the reverse skyline query and its variations in Section 2.2.

2.1. Skyline queries

The skyline query is a popular paradigm for extracting interesting objects from multi-dimensional databases. The skyline was first investigated by Kung et al. [19] in the computational geometry, and then was introduced to the database community by Borzsony et al. [4]. The first two algorithms developed are *Block Nested Loop* (BNL) and *Divide-and-Conquer* (D&C) [4]. Later, *Sort-First-Skyline* (SFS) algorithm is proposed as an improved version of BNL [7]. In [12], an optimized version of SFS, namely, *Linear-Elimination-Sort for Skyline* (LESS), is presented, which has attractive worst-case asymptotical performance. Other representatives include *Sort and limit skyline algorithm* (SaLSa) [2] and *object-based space partitioning* (OSP) scheme [42]. These algorithms mentioned above do not assume any index on the dataset. On the other hand, indexing techniques have also been exploited to accelerate skyline queries. Representative works include *Bitmap* and *Index* [34] that can retrieve skyline points instantly, *Nearest Neighbor* (NN) [18] and *Branch-and-Bound Skyline* (BBS) [29] that can return skyline points progressively, and *Z-SKY* [20] that is a generic framework to support skyline and many of its variants including skyband queries, top-ranked skyline queries, *k*-dominant skyline queries, and subspace skyline queries.

Papadias et al. [29] propose several interesting variations of skyline queries, such as dynamic skyline query, *k*-skyband query, ranked skyline query, and so forth. Given a dataset *P* and a parameter *k*, a *k*-skyband query reports the set of the points that are dominated by *at most k* points. Conceptually, *k* represents the thickness of the skyline. If $k = 0$, it corresponds to the traditional skyline. Ranked skyline query also calls top-*K* skyline retrieval. Given a dataset *P*, a parameter *K* and a preference function *f*, a ranked skyline query returns the *K* skyline points that have the minimum score according to the input function *f*.

Moreover, a variety of skyline query variants are studied as well, e.g., skyline queries on data streams [10,24,35], uncertain data [30,43], incomplete data [17,27], time series data [15], and keyword-matched data [1,6]; skyline search in peer-to-peer (P2P) networks [8,38], subspaces [22,31,32,36], and metric spaces [5,11]; skycube computation [16,40,41], continuous skyline retrieval [13,21,28,44], stochastic skyline query [25], to name but a few.

It is worth pointing out that the above skyline query and its variants are different from the R*k*SB and RRS queries studied in this paper. Therefore, the algorithms proposed for skyline retrieval and its variants can not applied to answer R*k*SB and RRS queries.

2.2. Reverse skyline queries

The concept of reverse skyline is originally introduced in [9]. In order to compute the reverse skyline of an arbitrary query point, *Branch and Bound Reverse Skyline algorithm* (BBRS) and *Reverse Skyline using Skyline Approximations algorithm* (RSSA) are proposed. Specifically, BBRS is an improved customization of the original BBS algorithm. It can be divided into two steps: first it uses a heap to retrieve the global skyline of a specified query point *q*, which is the superset of the actual reverse skyline set. Second, BBRS performs a window query for every global skyline point to examine whether or not it is a

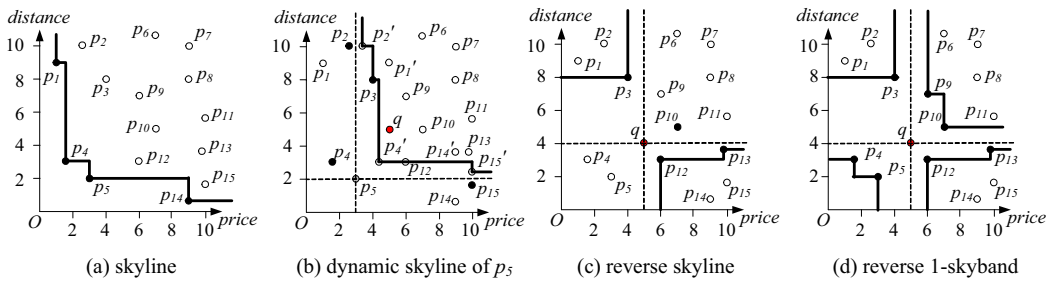
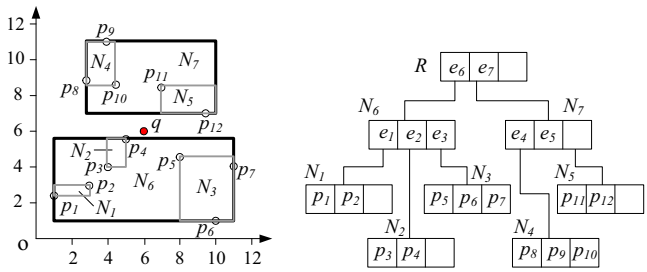


Fig. 1. Example of skyline, dynamic skyline, reverse skyline, and reverse 1-skyband.



(a) Dataset (b) The R-tree

Action	Heap contents	S
access root	$< e_6, e_7 >$	$\varnothing$
expand e_6	$< e_2, e_7, e_3, e_1 >$	$\varnothing$
expand e_2	$< \mathbf{p_4}, e_7, e_3, p_3, e_1 >$	$\{p_4\}$
expand e_7	$< e_5, e_3, p_3, e_4, e_1 >$	$\{p_4\}$
expand e_5	$< \mathbf{p_{11}}, e_3, p_3, e_4, p_{12}, e_1 >$	$\{p_4, p_{11}\}$
expand e_3	$< \mathbf{p_5}, \cancel{p_3}, e_4, p_{12}, e_1, p_7, p_6 >$	$\{p_4, p_{11}, p_5\}$
expand e_4	$< p_{10}, \mathbf{p_{12}}, p_8, e_1, p_9, p_7, p_6 >$	$\{p_4, p_{11}, p_5, p_{12}\}$

(c) Procedure of a reverse skyline query

Fig. 2. Example of a reverse skyline query.

real answer object. If the window query returns no answer, the corresponding point is a reverse skyline point. Consider, for example, the dataset depicted in Fig. 2 (a), which is indexed by the R-tree shown in Fig. 2(b). The heap contents during the reverse skyline computation are illustrated in Fig. 2(c), where the set $S = \{p_4, p_{11}, p_5, p_{12}\}$ represents the final reverse skyline set. As for RSSA, it is based on the well known *filter-refinement* paradigm. Before the algorithm starts, it computes the dynamic skyline for each database object, which is kept on disk with a fixed-size. By using this approximation in the filter step, it can be able to identify the partial real reverse skyline points and the unqualified reverse skyline points. The remaining candidate points are then examined in the refinement step using a window query.

In addition to the traditional reverse skyline query, the reverse skyline computation in different environments is also studied in the literature. The monochromatic and bichromatic reverse skyline search on *uncertain* data is investigated in [23], where each object is modeled as a probability distribution function. *Probabilistic Reverse Furthest Skyline* (PRFS) is proposed in [23] as a complimentary query to probabilistic reverse skyline. The bichromatic reverse skyline retrieval for the traditional dataset is explored in [39], in which each object is a *precise* point. In that work, several non-trivial heuristics are identified, which can optimize the access order to achieve stronger pruning power, and a new algorithm called *bichromatic reverse skyline algorithm* (BRS) is developed accordingly. More recently, techniques for reverse skyline computation over wireless sensor networks [37], data streams [45], and arbitrary non-metric similarity measures [33] have also been explored in the literature. In wireless sensor networks (WSNs), the power of the sensor node is very precious. Therefore, the reverse skyline algorithm in such environments should be not only effective but also energy efficient. An energy-efficient approach based on skyband is proposed in [37] to answer the reverse skyline query in WSNs. As data in data streams update rapidly, it is very difficult to maintain the index in a dynamic dataset. Toward this, an algorithm termed *Divide and Conquer Reverse Skyline algorithm* (DCRS) is developed in [45] to support reverse skyline retrieval on data streams. The DCRS uses the DC-Tree as the index, and employs effective pruning methods to shrink the search space. The reverse skyline with arbitrary non-metric similarity measures is studied in [33]. Since the attributes do not have a total ordering among their values, many indexes like R-tree are inapplicable. New algorithms, viz., *Block Reverse Skyline* (BRS), *Sort Reverse Skyline* (SRS), and *Tree Reverse Skyline* (TRS), are proposed to answer the reverse skyline query with arbitrary non-metric similarity measures. Nowadays, users may be interested in answering why-not questions in reverse skyline queries [14], which aims to find out why a particular point is not in a reverse skyline, and what actions we should take to put the point into a reverse skyline. To this end, Islam et al. [14] show how to modify the why-not point and the query point to include the why not point in the reverse skyline of the query point, and propose techniques that incur minimum changes to both the why-not point and the query point.

It is worth mentioning that the reverse skyline query and its variants are different from the R*k*SB query in terms of problem definitions and the details of query processing, and thus, the existing algorithms can not be directly applied to compute the R*k*SB efficiently.

3. Reverse *k*-skyband query processing

As mentioned in Section 1, the result set of the reverse skyline query may be very small, far from users' requests. Hence, we introduce the reverse *k*-skyband (R*k*SB) query. In the sequel, we present the formal definition of R*k*SB retrieval, propose five efficient algorithms to answer R*k*SB queries, and analyze the correctness of these algorithms.

3.1. Problem formulation

Let P be a d-dimensional data set. For any point $p \in P$, we use $p[i]$ to denote the i-th dimensional value. A point $p \in P$ is said to dominate another point $p' \in P$, denoted as $p \prec p'$, iff (i) $\forall i \in [1, d]$, $p[i] \leqslant p'[i]$; and (ii) $\exists\, j \in [1, d]$, $p[j] < p'[j]$. For instance, in Fig. 3(a), point p_5 dominates point p_6 as $p_5[1] < p_6[1]$ and $p_5[2] < p_6[2]$.

Definition 1 (*Dynamic k-Skyband*). Given a d-dimensional data set P, a query point q, and a parameter k, if a point $p \in P$ belongs to the *dynamic k-skyband* of q, there are *at most* k points in P, denoted by O, such that for each point $o \in O$, it satisfies: (1) $\forall i \in [1, d]$, $|q[i] - o[i]| \leqslant |q[i] - p[i]|$; and (2) $\exists\, j \in [1, d]$, $|q[j] - o[j]| < |q[j] - p[j]|$.

A dynamic k-skyband query retrieves all the points that are dynamically dominated by at most k points. Fig. 3(b) illustrates the dynamic 1-skyband of a point p_5. As shown in Fig. 3, although a point q is not in the dynamic skyline of p_5, it is in p_5's dynamic 1-skyband since the number of the points that dynamically dominate q is 1. Note that, dynamic k-skyband is a generalization of dynamic skyline [29]. Based on the dynamic k-skyband, we formalize the reverse k-skyband query below.

Definition 2 (*Reverse k-Skyband Query*). Given a d-dimensional data set P, a query point q, and a parameter k, a *reverse k-skyband* (RkSB) *query* returns all the points in P whose dynamic k-skyband contains q. Formally, a point $p \in P$ is in the reverse k-skyband of q, iff there are *at most* k points in P, denoted as O, such that $\forall o \in O$, it holds: (1) $\forall i \in [1, d]$, $|p[i] - o[i]| \leqslant |p[i] - q[i]|$; and (2) $\exists\, j \in [1, d]$, $|p[j] - o[j]| < |p[j] - q[j]|$.

Take Fig. 3(b) as an example again. Since a point q is included in the dynamic 1-skyband of a point p_5, the point p_5 belongs to the reverse 1-skyband of q according to Definition 2. Similarly, we can obtain the complete reverse 1-skyband of q, which contains $\{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8\}$, as depicted in Fig. 3(c). It is worth noting that, conceptually, k represents the *thickness* of the reverse skyline. Hence, the case $k = 0$ corresponds to a conventional reverse skyline query. As mentioned in Section 1, the user might change the location of the query point to increase the number of reverse skyline points. However, finding a good query point location such that the number of identified reverse skyline points matches the application need is not easy, because the size of the result set is affected by many factors. Thus, the RkSB query can solve this problem by relaxing the condition and returning more answer points, especially for those cases where the change of the query point is impossible. As an example, a real-estate agent would like to promote a new project to the customers, and he/she can employ the reverse skyline query to locate the potential customers as the promotion targets. Nonetheless, the reverse skyline set may be too small. On the other hand, the properties (e.g., the floor height, the size, the build time, etc.) of the project are fixed. In this case, the agent can utilize the RkSB query to relax the conditions and thus find more promotion targets.

3.2. RkSB query processing algorithms

A naive solution to tackle RkSB retrieval is to, for each point in a specified data set P, perform a dynamic k-skyband query, and then return those points $p \in P$ with $q \in DSB(p)$, in which $DSB(p)$ represents the set of p's dynamic k-skyband points. Nevertheless, this method is *very inefficient* as it needs to traverse the data set P *multiple times* (i.e., $|P|$ times), resulting in high I/O and CPU costs, especially for larger P. To this end, we propose five efficient algorithms for processing the RkSB query, namely, *Branch-bound-based RkSB algorithm* (BRkSB), *Pre-computation-based RkSB algorithm* (PRkSB), *Optimized PRkSB*(OPRkSB), *Reused-based RkSB algorithm* (RRkSB), and *Global-skyband-based RkSB algorithm* (GRkSB). We assume that the data set P is indexed by an R-tree. Furthermore, in order to facilitate the understanding of different RkSB query processing algorithms, a running example is employed, as shown in Fig. 4, where a 2D data point set $P = \{p_1, p_2, \ldots, p_{15}\}$ is depicted in Fig. 4(a), and they are organized in the R-tree with node capacity set to 3, as illustrated in Fig. 4(b).

3.2.1. Branch-bound-based RkSB algorithm

Before presenting the BRkSB algorithm, an improved customization of the original BBS algorithm, we first define the global k-skyband, and then propose two lemmas to identify candidate RkSB points.

Definition 3 (*Global k-Skyband*). Given a d-dimensional data set P, a query point q, and a parameter k, if a point $p \in P$ is in the global k-skyband of q, there exist at most k points in P, denote as O, such that $\forall o \in O$, it satisfies: (1) $\forall i \in [1, d]$, $(p[i] - q[i])(o[i] - q[i]) > 0$; (2) $\forall i \in [1, d]$, $|q[i] - o[i]| \leqslant |q[i] - p[i]|$; and (3) $\exists\, j \in [1, d]$, $|q[j] - o[j]| < |q[j] - p[j]|$.

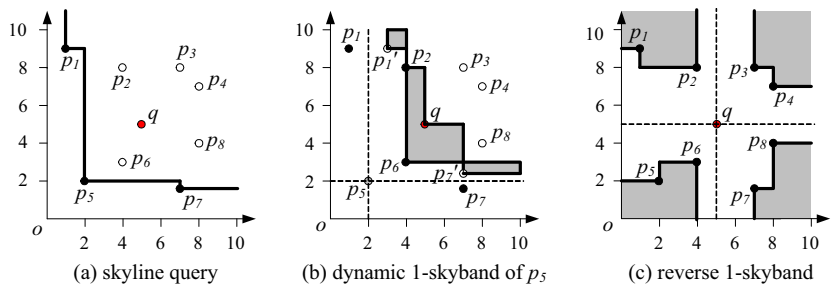


Fig. 3. Illustration of skyline, dynamic 1-skyband, and reverse 1-skyband.

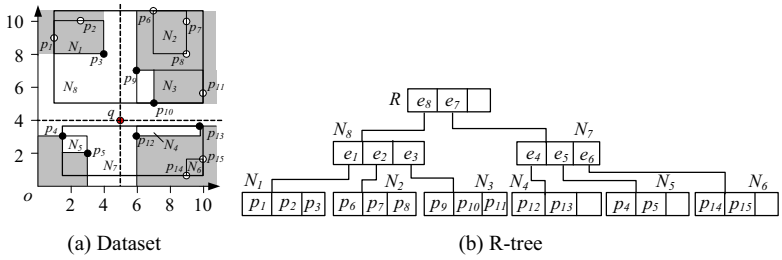


Fig. 4. A running example.

A global k-skyband query retrieves all the points that are globally dominated by at most k points. Note that global k-skyband is an extension of global skyline [9]. As shown in Fig. 5, point p_{11} is only globally dominated by p_{10}, and thus, it is a global 1-skyband point of q. On the other hand, point p_8 is globally dominated by points p_9 and p_{10}, and hence, it is not a global 1-skyband point of q. In Fig. 5, all the global 1-skyband points of q include $\{p_1, p_2, p_3, p_4, p_5, p_9, p_{10}, p_{11}, p_{12}, p_{13}, p_{14}\}$.

Lemma 1. *Let q be a query point, $GSB_k(q)$ be the global k-skyband point set of q, and $RSB_k(q)$ be the set of reverse k-skyband points w.r.t. q. If a point $p \notin GSB_k(q)$, $p \notin RSB_k(q)$.*

Proof. If a point $p \notin GSB_k(q)$, we can find at least $(k+1)$ points (stored in a set S) that globally dominate p, i.e., for every point $s \in S$, it holds (1) $\forall i \in [1, d]$, $|s[i] - q[i]| \leqslant |p[i] - q[i]|$; and (2) $\exists\, j \in [1, d]$, $|s[j] - q[j]| < |p[j] - q[j]|$, which contradicts with Definition 2 (i.e., $p \notin RSB_k(q)$). Consequently, the proof completes. □

Lemma 2. *Given a d-dimensional data set P, a query point q, and a parameter k, suppose a rectangle Rect is centered at a point $p \in P$, and its extent is defined by the coordinate-wise distances to q. If there are more than k points within Rect, p is not a real reverse k-skyband point of q.*

Proof. If there are more than k points inside the rectangle *Rect*, we can find $(k+1)$ points (preserved in a set S) such that, each point $s \in S$ satisfies: (1) $\forall i \in [1, d]$, $|p[i] - s[i]| \leqslant |p[i] - q[i]|$; and (2) $\exists\, j \in [1, d]$, $|p[j] - s[j]| < |p[j] - q[j]|$. According to Definition 1, the query point q does not belong to the dynamic k-skyband of the point p. Therefore, p is not a reverse k-skyband point of q. □

Consider, for instance, Fig. 6(a), in which the shaded area is the rectangle of a point p. As depicted in Fig. 6(a), point q is dynamically dominated by points p_1 and p_2, which are located inside p's rectangle. Thus, the point q is not in the dynamic 1-skyband of the point p, and p does not belong to the reverse 1-skyband of q.

We can utilize Lemma 1 to efficiently retrieve a superset of the actual result, i.e., no false missing, and use Lemma 2 to prune away unqualified candidate reverse k-skyband points. Our first method, namely, BRkSB, employs lemmas 1 and 2 to process the RkSB query. The pseudo-code of BRkSB is presented in Algorithm 1. The basic idea is as follows. First, BRkSB computes the set $GSB_k(q)$ of global k-skyband points that is guaranteed to include all the actual reverse k-skyband points according to Lemma 1. Then, the algorithm executes a window query for each candidate global k-skyband point $p \in GSB_k(q)$. If the

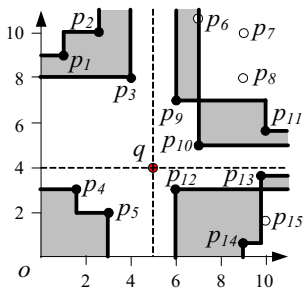


Fig. 5. Global 1-skyband.

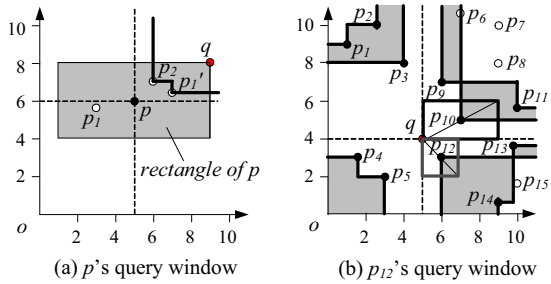


Fig. 6. Example of the query window.

window query returns no more than k points, $p \in RSB_k(q)$. To simplify our discussion, we use the term p's *query window* to refer to the rectangle centered at p with its extent defined by the coordinate-wise distances to q, denoted as $W(p,q)$. It is worth mentioning that, the window query is implemented as a boolean/empty range query. Specifically, we use a *counter* to count the number of the points located into the $W(p,q)$, and the window query terminates once it finds $(k+1)$ points, which can be handled efficiently.

Algorithm 1. Branch-bound-based Reverse k-Skyband Algorithm (BRkSB)

Input: an R-tree R on a set of data points, a query point q, a parameter k
Output: the result set *RSB* that contains all the points belonging to the RkSB of q
/∗ *GSB*: the set of global k-skyband points; *RSB*: the set of reverse k-skyband points; H_g, H_w: min-heaps, sorted in ascending order of their distances (i.e., L_1-norm) to q. ∗/
1: initialize sets $GSB = RSB = \varnothing$ and min-heaps $H_g = H_w = \varnothing$
2: insert all entries of the root R into H_g
3: **while** $H_g \neq \varnothing$ **do**
4: de-heap the top entry e of H_g
5: **if** e is globally dominated by $(k+1)$ points in *GSB* **then** discard e // by Lemma 1
6: **else** // e is globally dominated by less than $(k+1)$ points in *GSB*
7: **if** e is an intermediate node **then**
8: **for** each child entry $e_i \in e$ **do**
9: **if** e_i is globally dominated by at most k point in *GSB* **then** insert e_i into H_g
10: **else** // e is a data point
11: add e to *GSB* // for pruning later
12: perform the window query (using H_w) based on e and q
13: **if** the window query finds less than $(k+1)$ points **then**
14: add e to *RSB* // e is a RkSB point of q by Lemma 2
15: **else** discard e // the window query contains more than k points
16: return *RSB*

Next, we illustrate BRkSB using the running example. Initially, BRkSB visits the root node of the R-tree R, and inserts all its entries e_7, e_8 into a min-heap H_g (line 2). Then, the entry e_7 with the minimum distance to q is expanded. As e_7 is not globally dominated by any point (line 5) and it is an intermediate node, BRkSB removes the entry e_7 from the heap and adds its children e_4, e_5, e_6 to H_g (lines 7–9). Similarly, the next two expanded entries are e_8 and e_4 respectively, in which the first global 1-skyband point p_{12} is found. Since p_{12} is a data point, BRkSB adds p_{12} to the set *GSB* for pruning later, and then runs a window query on it (lines 11–12). As illustrated in Fig. 6(b), there is no point in p_{12}'s query window, and thus, p_{12} is inserted into the result set *RSB*. The algorithm proceeds in the same manner until the heap becomes empty. The final answer points are p_3, p_4, p_5, p_9, p_{10}, p_{12}, and p_{13}. Table 1 depicts the procedure of BRkSB during the processing of the query.

3.2.2. Pre-computation-based RkSB algorithm

Given the fact that the global k-skyband is a superset of the reverse k-skyband as well as it may contain many unqualified points, we present an enhanced algorithm, called *Pre-computation-based Reverse k-Skyband algorithm* (PRkSB). To be more specific, PRkSB employs the traditional (i.e., 0-th) dynamic skyline and the k-th dynamic skyline, which are *offline* pre-computed and stored on disk, to identify the reverse k-skyband points. Below, we define the k-th dynamic skyline, and then present Lemma 3 and Heuristic 1 to support our proposed PRkSB algorithm.

Table 1
Procedure of BRkSB.

Action	Heap contents	*RSB*
Access root	$\langle e_7, e_8 \rangle$	$\emptyset$
Expand e_7	$\langle e_8, e_5, e_4, \mathrm{v}e_6 \rangle$	$\emptyset$
Expand e_8	$\langle e_4, e_3, e_5, e_1, e_2, e_6 \rangle$	$\emptyset$
Expand e_4	$\langle \boldsymbol{p}_{12}, e_3, e_5, p_{13}, e_1, e_2, e_6 \rangle$	$\{p_{12}\}$
Expand e_3	$\langle \boldsymbol{p}_{10}, e_5, p_{13}, p_9, e_1, e_2, p_{11}, e_6 \rangle$	$\{p_{12}, p_{10}\}$
Expand e_5	$\langle \boldsymbol{p}_5, \boldsymbol{p}_4, \boldsymbol{p}_{13}, \boldsymbol{p}_9, e_1, e_2, p_{11}, e_6 \rangle$	$\{p_{12}, p_{10}, p_5, p_4, p_{13}, p_9\}$
Expand e_1	$\langle \boldsymbol{p}_3, \cancel{e_2}, \cancel{p_{11}}, e_6, p_2, p_1 \rangle$	$\{p_{12}, p_{10}, p_5, p_4, p_{13}, p_9, p_3\}$
Expand e_6	$\langle \cancel{p_{14}}, \cancel{p_{15}}, \cancel{p_2}, \cancel{p_1} \rangle$	$\{p_{12}, p_{10}, p_5, p_4, p_{13}, p_9, p_3\}$

Definition 4 (*k-th Dynamic Skyline*). Given a d-dimensional data set P, a query point q, and a parameter k, if a point $p \in P$ is in the k-th dynamic skyline of q, there are *exactly* k points in P, denoted by O, such that for every point $o \in O$, it holds: (1) $\forall i \in [1, d]$, $|q[i] - o[i]| \leqslant |q[i] - p[i]|$; and (2) $\exists\, j \in [1, d]$, $|q[j] - o[j]| < |q[j] - p[j]|$.

The k-th dynamic skyline query retrieves the set of the points in P that are dynamically dominated by k points. Fig. 7(a) illustrates the example of the k-th dynamic skyline, where the 0-th dynamic skyline contains points p_1, p_2, p_5, and p_9; the 1st dynamic skyline includes points p_3 and p_6; and the 2nd dynamic skyline consists of points p_4 and p_{10}. Note that, all of them form the dynamic 2-skyband of p.

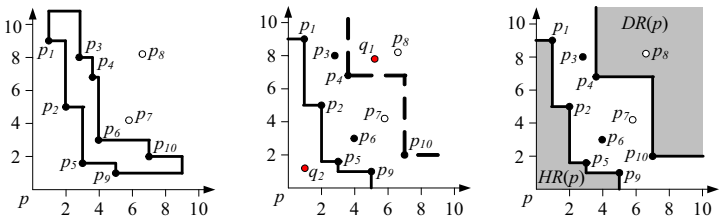
Lemma 3. *For a given point p and a query point q, let DSL(p) be the set of 0-th dynamic skyline points of p and kDSL(p) be the set of k-th dynamic skyline points of p. If a point $s \in DSL(p)$ is dynamically dominated by q, p belongs to the reverse k-skyband of q. If a point $s' \in kDSL(p)$ dynamically dominates q, p is not in the reverse k-skyband of q.*

Proof. If a point $s \in DSL(p)$ is dynamically dominated by q, it means that q is not dynamically dominated by any other point. Hence, q is in the dynamic k-skyband of p, and p belongs to the reverse k-skyband of q. If $s' \in kDSL(p)$, there are exactly k points dynamically dominating s'. As q is dynamically dominated by s', it is also dynamically dominated by the k points which dynamically dominate s'. Consequently, q is not in the dynamic k-skyband of p, and p does not belong to the reverse k-skyband of q. □

Algorithm 2. Pre-computation-based Reverse k-Skyband Algorithm (PRkSB)

Input: an R-tree R on a set of data points, a query point q, a parameter k, the dynamic skyline of the dataset, the k-th dynamic skyline of the dataset
Output: the result set *RSB* that contains all the points belonging to the RkSB of q
/∗ *DR*(p): the discard region of p containing the points that are dynamically dominated by at least (k + 1) points; *HR*(p): the hit region of p including the points that are not dynamically dominated by any other point. ∗/

```
initialize sets GSB = RSB = ∅ and min-heaps H_g = H_w = ∅
insert all entries of the root R into H_g
while H_g ≠ ∅ do
  de-heap the top entry e of H_g
  if e is globally dominated by (k + 1) points in GSB then discard e   // by Lemma 1
  else   // e is globally dominated by less than (k + 1) points in GSB
    if e is an intermediate node then
      for each child e_i ∈ e do
        if e_i is globally dominated by at most k points in GSB then insert e_i into H_g
    else   // e is a data point
      add e to GSB   // for pruning later
      if q is in the HR (e) then add e to RSB   // Heuristic 1
      else if q is in the DR_{k+1}(e) then discard e
      else
        perform the window query (using H_w) based on e and q
        if the window query finds less than (k + 1) points then add e to RSB   // by Lemma 2
        else discard e
return RSB
```



(a) The k-th dynamic skyline (b) Example for Lemma 3 (c) $DR(p)$ and $HR(p)$

Fig. 7. Illustration of the k-th dynamic skyline, Lemma 3, and $DR(p)$ and $HR(p)$.

Consider, for example, Fig. 7(b), where points q_1 and q_2 are query points. Since point $p_4 \in 2\ DSL\ (p)$ dynamically dominates q_1, p is not in the reverse 2-skyband of q_1. On the other hand, q_2 is not dynamically dominated by any other point, and thus, p belongs to q_2's reverse 2-skyband. Lemma 3 can help prune away unqualified points. To facilitate the point pruning, two regions w.r.t. a specified point p are defined, namely, the *Discard Region of* p (i.e., $DR_{k+1}(p)$) and the *Hit Region of* p (i.e., $HR(p)$). The former $DR_{k+1}(p)$ is the region that contains the points dynamically dominated by *at least* $(k+1)$ points w.r.t. p; and the latter $HR\ (p)$ is the region containing all the points not dynamically dominated by any other point w.r.t. p. As shown in Fig. 7(c), the 0-th dynamic skyline contains points p_1, p_2, p_5, and p_9, and all these four points that are not dynamically dominated form $HR\ (p)$. On the other hand, suppose $k = 2$, the 2nd dynamic skyline points p_4 and p_{10} form $DR_3(p)$, in which all the points enclosed are dynamically dominated by at least three points w.r.t. p. For instance, point p_8 is located inside $DR_3(p)$ and it is dynamically dominated by seven points p_2, p_3, p_4, p_5, p_6, p_7, and p_9. Based on these two regions, a given point p can be evaluated, as stated in Heuristic 1.

Heuristic 1. Given a query point q, a parameter k, and a point p,$DR_{k+1}(p)$ and $HR(p)$ can be derived. If q falls into $DR_{k+1}(p)$, p is not in the reverse k-skyband of q (and thus is discarded). If q locates within $HR(p)$, p belongs to the reverse k-skyband of q. If q is not inside either $DR_{k+1}(p)$ or $HR(p)$, p needs to be further evaluated.

Our second approach, i.e., PR*k*SB, utilizes the above Heuristic 1 to tackle the R*k*SB query. The pseudo-code of PR*k*SB is depicted in Algorithm 2. The main idea is as follows. Before the algorithm starts, for every data point $p \in P$, its corresponding 0-th dynamic skyline and k-th dynamic skyline are pre-computed and stored on disk. When a query is issued at a point q, PR*k*SB computes its global k-skyband (i.e., $GSB_k(q)$) that is a superset of the final query result, and then, for each point $p \in GSB_k(q)$, we derive $DR_{k+1}(p)$ and $HR(p)$. If q is within $DR_{k+1}(p)$, p can be pruned safely. If q is inside $HR(p)$, p is an actual answer point. Otherwise, a window query on the current candidate point p is performed based on Lemma 2, as does in BR*k*SB.

Algorithm 3. Optimized PR*k*SB Algorithm (OPR*k*SB).

Input: an R-tree R on a set of data points, a query point q, a parameter k, the dynamic skyline of the dataset, the k-th dynamic skyline of the dataset
Output: the result set *RSB* that contains all the points belonging to the R*k*SB of q
/∗ *PGSB*: the set of global k-skyband points after pruned by Heuristic 1. ∗/
initialize sets $GSB = PGSB = RSB = \varnothing$ and min-heaps $H_g = H_w = \varnothing$
insert all entries of the root R into H_g
while $H_g \neq \varnothing$ **do**
  de-heap the top entry e of H_g
  if e is globally dominated by $(k+1)$ points in *GSB* **then** discard e // by Lemma 1
  else // e is globally dominated by less than $(k+1)$ points in *GSB*
    if e is an intermediate node **then**
      for each child $e_i \in e$ **do**
        if e_i is globally dominated by at most k point in *GSB* **then** insert e_i into H_g
    else // e is a data point
      add e to *GSB* // for pruning later
      if q is within $HR(e)$ **then** add e to *RSB* // Heuristic 1
      else if q is inside $DR_{k+1}(e)$ **then** discard e
      else add e to *PGSB* // for the next pruning
for each point $p \in PGSB$ **do** // Heuristic 2
  if the window based on p and q contains more than k global k-skyband points **then** discard p
  else
    perform the window query (using H_w) based on p and q
    if the window query finds less than $(k+1)$ points **then** add p to *RSB*
    else discard p
return *RSB*

3.2.3. Optimized PRkSB algorithm

As discussed above, if a specified point p cannot be returned as an answer point or discarded as a non-answer point based on Heuristic 1, PR*k*SB employs window queries to decide whether p is a real answer point. As shown in Fig. 6(b), the query point q partitions the search space into four quadrants. Assume that a point p is currently located in the quadrant *qua* and it is observed that all the points located inside quadrants except for *qua* will not be in the query window. Meanwhile, the global k-skyband points located into *qua* are closer to q compared with others, and hence have a higher chance to be within the query window. As the global k-skyband points are only determined by the query point q and k, they are fixed during the evaluation of different point p. We can utilize the global k-skyband points to improve the performance of window queries, as stated in Lemma 4.

Lemma 4. *Given a query point q, a parameter k, and a data point p, if there are more than k global k-skyband points inside p's query window, p is not in the reverse k-skyband of q.*

Proof. The proof is similar as that of Lemma 2, and thus is omitted. □

Lemma 4 can save all the window queries with their corresponding query windows containing more than k global k-skyband points. For instance, assume that a reverse 1-skyband query is issued at q as depicted in Fig. 8. We can locate all the global 1-skyband points, and they are $\{p_1,p_2,p_3,p_4,p_6,p_9,p_{10},p_{11},p_{12}\}$. If p_1 is the point currently evaluated, we need to evaluate how many global 1-skyband points are located within the query window centered at p_1. Based on the positions of p_1 and q, we only need to evaluate the global 1-skyband points p_3, p_4, and p_6 that are located in the same quadrant as p_1. Since p_3 and p_4 are located inside the query window, it is guaranteed that p_1 is not in the reverse 1-skyband of q according to Lemma 4. Hence, the window query of p_1 is saved without using the boolean/empty range query. Actually, the global $(k-i)$-skyband $(0 < i \leqslant k)$ of q, or the global $(k+j)$-skyband $(j > 0)$ of q can also be used for pruning. However, the pruning efficiency of the global $(k-i)$-skyband is not as well as that of the global k-skyband because it contains fewer points. As for the global $(k+j)$-skyband of q, its time cost is larger than that of the global k-skyband. Consequently, we employ the global k-skyband of q to prune away, as stated in Heuristic 2 below.

Heuristic 2. Given a query point q, a parameter k, and a data point p, if there are more than k global k-skyband points in the query window of p, p does not belong to the reverse k-skyband of q and thus can be discarded; otherwise, p needs to be further evaluation.

Based on Heuristic 2, we propose an improved PR*k*SB algorithm, namely, *optimized PRkSB algorithm* (OPR*k*SB), by using Heuristic 2 before issuing a window query for evaluating a point p. Algorithm 3 presents the pseudo-code of OPR*k*SB. In particular, OPR*k*SB computes the global k-skyband of a given query point q, and then utilizes Heuristic 1 to prune away unqualified global k-skyband points that cannot be the actual answer points. Thereafter, for every remaining candidate point, the algorithm does not directly execute a window query on it, but employs the global k-skyband of q (instead of the remaining global k-skyband) as an additional pruning heuristic based on Heuristic 2.

3.2.4. Reused-based RkSB algorithm

All the above three algorithms, viz., BR*k*SB, PR*k*SB, and OPR*k*SB, rely on the window query to refine certain candidate points. Each window query has to traverse the R-tree to determine whether there are $(k+1)$ points in the window of the candidate point. Since there are many candidate points need evaluation, *multiple* window queries are issued, and the R-tree is visited repeatedly. Motivated by this observation, we present the *Reused-based RkSB algorithm* (RR*k*SB) by using the *reuse mechanism* to further boost OPR*k*SB.

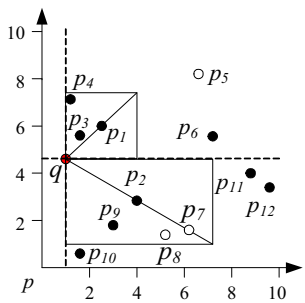


Fig. 8. Illustration of Lemma 4.

Algorithm 4. Reused-based R*k*SB algorithm (RR*k*SB).

Input: an R-tree R on a set of data points, a query point q, a parameter k, the dynamic skyline of the dataset, the k-th dynamic skyline of the dataset
Output: the result set *RSB* that contains all the points belonging to the R*k*SB of q

```
1:  initialize sets GSB = PGSB = RSB = ∅ and min-heaps H_g = H_w = ∅
2:  insert all entries of the root R into H_g
3:  while H_g ≠ ∅ do
4:    de-heap the top entry e of H_g
5:    if e is globally dominated by (k + 1) points in GSB then   // by Lemma 1
6:      insert e_i into H_w   //for reuse later
7:    else   // e is globally dominated by less than (k + 1) points in GSB
8:      if e is an intermediate node then
9:        for each child e_i ∈ e do
10:         if e_i is globally dominated by at most k point in GSB then insert e_i into H_g
11:         else insert e_i into H_w   // for reuse later
12:     else   // e is a data point
13:       add e to GSB   // for pruning later
14:       insert e into H_w   // for reuse
15:       if q is within HR(e) then add e to RSB   // Heuristic 1
16:       else if q is inside DR_{k+1}(e) then discard e
17:       else add e to PGSB   // for the next pruning
18: for each point p ∈ PGSB do   // Lemma 4
19:   if the window based on p and q contains more than k global k-skyband points then discard p
20:   else
21:     perform the window query (using H_w) based on p and q
22:     if the window query finds less than (k + 1) points then add p to RSB   // Lemma 2
23:     else discard p
24: return RSB
```

Recalled that there are two processes need to traverse the R-tree, i.e., the computation of the global k-skyband and window queries. These two processes use the same R-tree. Furthermore, the dominance relationship of the nodes after expanding does not change. Therefore, as long as we keep the integrity of the R-tree (meaning that neither store the visited nodes repeatedly nor miss some visited nodes), the visited R-tree can be reused in these two processes. In the sequel, we discuss how to maintain the integrity of the R-tree. There are two methods to store the visited R-tree nodes. One is to maintain an R-tree with the entire visited nodes, and the other is only to preserve the deepest level's visited nodes. Since maintaining the whole R-tree takes considerable space, we implement the second approach. Based on this discussion, we propose the fourth R*k*SB algorithm, i.e., RR*k*SB, with its pseudo-code shown in Algorithm 4. As RR*k*SB is very similar to OPR*k*SB, we only highlight their differences here. First, in order to keep the integrity of the R-tree, we should not discard any node unless it is expanded and replaced by its child nodes. Consequently, in Algorithm 4, we maintain all these visited nodes via a heap (lines 6, 11, and 14), while in Algorithm 3, these visited nodes are discarded directly. Second, the heap H_w used in Algorithm 3 is empty because the entries of the heap are released after the termination of the window query; whereas in Algorithm 4, H_w (line 21) is not empty because it stores the deepest level's visited nodes for the reuse later.

3.2.5. Global-skyband-based RkSB algorithm

By employing the reuse technique in OPR*k*SB, the node accesses can be reduced significantly. However, the improved I/O cost comes with an extra cost. With the visited nodes maintained by a heap, the window queries issued at evaluated points need to scan every entry of the heap. If there are lots of entries in the heap, the whole scanning is costly. Motivated by this defect of the reuse technique, we develop the *Global-skyband-based RkSB algorithm* (GR*k*SB), which does not use the R-tree nodes in the window query to prune candidate points. Next, we first give the definition of the k-th global skyline.

Definition 5 (*k-th Global Skyline*). Given a d-dimensional data set P, a query point q, and a parameter k, if a point $p \in P$ is in the k-th global skyline of q, there are k points in P, denoted by O, such that for every point $o \in O$, it holds: (1) $\forall i \in [1, d]$, $(p[i] - q[i])(o[i] - q[i]) > 0$; (2) $\forall i \in [1, d]$, $|q[i] - o[i]| \leqslant |q[i] - p[i]|$; and (3) $\exists\, j \in [1, d]$, $|q[j] - o[j]| < |q[j] - p[j]|$.

The k-th global skyline query retrieves the set of the points in P that are globally dominated by exact k points. As shown in Fig. 5, point p_{14} is globally dominated by point p_{12}, and point p_8 is globally dominated by points p_9 and p_{10}. Therefore, p_{14} is in the 1-st global skyline of q, and p_8 is in the 2-nd global skyline of q. Based on the k-th global skyline, Lemma 6 is developed.

Lemma 5. *Given a d-dimensional data set P, a query point q, and a parameter k, assume that there are k global k-skyband points in the query window W(p,q) of p ∈ P, (i) if all the (k + 1)-th global skyline points are out of W(p,q), then W(p,q) only contains k global k-skyband points, and thus, p is in the reverse k-skyband of q; otherwise, (ii) W(p,q) contains at least one (k + 1)-th global skyline point, and hence, p is not in q's reverse k-skyband.*

Proof. First, assume that the statement (i) is not valid, i.e., if $W(p,q)$ contains k global k-skyband points but none of the $(k+1)$-th global skyline points, it still contains at least one i-th global skyline point p_i with $i \in \{k+2, k+3, \ldots\}$. Since p_i is one i-th global skyline point, it must be globally dominated by i points in $W(p,q)$ according to Definition 5. Nevertheless, there are only $(k+1)$ points, i.e., k global k-skyband points and point p, in $W(p,q)$ when $i > (k+1)$. Therefore, our assumption is invalid. Second, assume that the statement (ii) is invalid, i.e., if $W(p,q)$ contains k global k-skyband points and one $(k+1)$-th global skyline point, p is in q's reverse k-skyband, which contradicts with Lemma 2, and hence, our assumption is invalid. The proof completes. □

Lemma 6. *Given a d-dimensional data set P, a query point q, and a parameter k, if there are less than k global k-skyband points in p's query window W(p,q), then W(p,q) can not contain any other point, and thus, p is a real reverse k-skyband point.*

Proof. Assume that the above statement is not valid, i.e., $W(p,q)$ actually contains at least one i-th global skyline point p' with $i > k$. As p' is one i-th global skyline point, it must be dominated by i points (including p) with all those i points located inside $W(p,q)$. However, the number of global k-skyband points (including p) within $W(p,q)$ is at most k if $i > k$. Consequently, our assumption is invalid, and the proof completes. □

Algorithm 5. Global-skyband-based R*k*SB algorithm (GR*k*SB).

Input: an R-tree R on a set of data points, a query point q, a parameter k, the dynamic skyline of the dataset, the k-th dynamic skyline of the dataset
Output: the result set *RSB* that contains all the points belonging to the R*k*SB of q
/∗ *GS*: the set of $(k+1)$-th global skyline points ∗/

```
initialize sets GSB = GS = PGSB = RSB = ∅ and min-heap H_g = ∅
insert all entries of the root R into H_g
while H_g ≠ ∅ do
  de-heap the top entry e of H_g
  if e is globally dominated by (k + 2) points in GSB then discard e
  else  // e is globally dominated by less than (k + 2) points in GSB
    if e is an intermediate node then
      for each child e_i ∈ e do
        if e_i is globally dominated by at most (k + 1) point in GSB then insert e_i into H_g
    else  // e is a data point
      if e is globally dominated by (k + 1) point in GSB then
        add e to GS
      else  // e is the global k-skyband point
        add e to GSB  // it is the global k-skyband point
        if q is within HR(e) then add e to RSB  // Heuristic 1
        else if q is inside DR_{k+1}(e) then discard e
        else add e to PGSB  // for the next pruning
for each point p ∈ PGSB do  // Heuristic 3
  N_1 = the number of k global k-skyband points within p's query window
  N_2 = the number of (k + 1)-th global skyline points within p's query window
  if N_1 > k then discard p
  else if N_1 = k and N_2 = 0 then add p to RSB
  else if N_1 = k and N_2 > 0 then discard p
  else if N_1 < k then add p to RSB
return RSB
```

To facilitate the understanding of the newly proposed Lemmas, some examples are illustrated in Fig. 9. First, as shown in Fig. 9(a), the global 3-skyband points are p_1, p_5, p_7, p_8, p_9, p_{10}, p_{12}, and p_{13}; and the 4-th global skyline points are p_4 and p_6. As the query window of p_8 contains three global 3-skyband points (i.e., p_7, p_9, p_{10}) but none of the 4-th global skyline, point p_8 is a real reverse 3-skyband point based on Lemma 5. Second, as depicted in Fig. 9(b), the dataset changes with the global 3-skyband points remained but the 4-th global skyline changed to p_6 only. Now the query window of p_8 contains three global

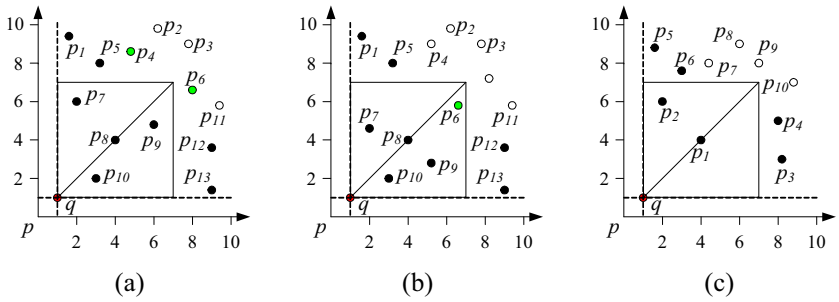


Fig. 9. Illustration of Lemma 5 and Lemma 6.

3-skyband points (i.e., p_7,p_9,p_{10}) and one 4-th global skyline p_6. Hence, according to Lemma 5, point p_8 is not an actual reverse 3-skyband point. Finally, as shown in Fig. 9(c), the global 3-skyband points are p_1, p_2, p_3, p_4, p_5, and p_6, and the query window of p_1 only contains p_1. Based on Lemma 6, p_1 is a real reverse 3-skyband point. In the following, we integrate Lemma 4, 5, and 6 into Heuristic 3, which utilizes the global k-skyband and the $(k + 1)$-th global skyline to refine candidate points.

Heuristic 3. Given a query point q, a data point p, the global k-skyband of q, the $(k + 1)$-th global skyline, and let $W(p,q)$ be p's query window, let N_1 be the number of k global k-skyband points in $W(p,q)$, and N_2 be the number of $(k + 1)$-th global skyline points in $W(p,q)$. Whether p is a reverse k-skyband point of q can be decided based on N_1 and N_2 as follows: (i) if $N_1 > k$, p is not a reverse k-skyband point of q; (ii) if $N_1 = k$ and $N_2 = 0$, p is a real reverse k-skyband point of q; (iii) if $N_1 = k$ and $N_2 > 0$, p is not a reverse k-skyband point of q; and (iv) if $N_1 < k$, p is an actual reverse k-skyband point of q.

Our last method, namely, GRkSB, is proposed based on Heuristic 3 with its pseudo-code listed in Algorithm 5. First, it computes the global k-skyband and the $(k + 1)$-th global skyline of q (lines 3–17), and meanwhile prune away certain unqualified points using Heuristic 1 (lines 15–17). Thereafter, it refines the remaining points using Heuristic 3 (lines 18–24).

3.3. Discussion

In this subsection, we analyze the correctness of our presented algorithms, i.e., BRkSB, PRkSB, OPRkSB, RRkSB, and GRkSB.

Lemma 7. *The proposed five algorithms visit (data point and intermediate) entries of an R-tree in ascending order of their distances to a specified query point q.*

Proof. The proof is straightforward since the algorithm always visits the entries according to their *mindist* (i.e., L_1-norm) preserved by the heap. □

Lemma 8. *Any data point inserted into the result set RSB during the execution of the algorithm is guaranteed to be an actual reverse k-skyband point.*

Proof. This is guaranteed by Lemmas 1 to 6 and Heuristics 1 through 3. □

Lemma 9. *Every data point will be examined, unless one of its ancestor nodes has been pruned away.*

Proof. The proof is obvious because all the entries that are not discarded by the current global k-skyband points (preserved in the set *GSB*) are added to the heap and evaluated. □

Let $|R_P|$ be the cardinality of the R-tree indexing a dataset P, g be the number of global k-skyband objects, g' be the number of $(k + 1)$-th global skyband objects, g_1 be the number of the global k-skyband objects refined by Heuristic 1, g_{12} be the number of the global k-skyband objects refined by Heuristics 1 and 2, and h be the number of maximal heap entries used.

Lemma 10. *The times of traversing the R-tree for BRkSB, PRkSB, OPRkSB, RRkSB, and GRkSB are $(g + 1)$, $(g_1 + 1)$, $(g_{12} + 1)$, 1, and 1, respectively.*

Proof. The global k-skyband for BRkSB, the global k-skyband refined by Heuristic 1 for PRkSB, and the global k-skyband refined by Heuristics 1 and 2 for OPRkSB all need window queries by traversing R-trees. In addition, the computation of global k-skyband should traverse the R-tree. Therefore, the times of traversing the R-tree for BRkSB, PRkSB, and OPRkSB are $(g + 1)$, $(g_1 + 1)$, and $(g_{12} + 1)$, respectively. RRkSB reuses the visited R-tree nodes, and GRkSB only uses the R-tree to compute the global k-skyband and the $(k + 1)$-th global skyline, and thus, these two algorithms traverse the R-tree *only once*. □

Theorem 1. *The time complexities of BRkSB, PRkSB, OPRkSB, RRkSB, and GRkSB are* $O(|R_P| \times (g+1))$, $O(|R_P| \times (g_1+1))$, $O(|R_P| \times (g_{12}+1))$, $O(|R_P| + h \times g_{12})$, *and* $O(|R_P| + (g+g') \times g_{12})$, *respectively.*

Proof. Both the computation of the global k-skyband and the window query need to traverse the R-tree. In the worst case, the two processes have to traverse the whole R-tree. Therefore, the computation of the global k-skyband and the window query should take $O(|R_P|)$ time in the worst case. Based on Lemma 10, the times of traversing the R-tree for BRkSB, PRkSB, and OPRkSB are $(g+1)$, (g_1+1), and $(g_{12}+1)$, respectively. Hence, the overall time complexities of BRkSB, PRkSB, and OPRkSB are $O(|R_P| \times (g+1))$, $O(|R_P| \times (g_1+1))$, and $O(|R_P| \times (g_{12}+1))$, respectively. In addition, RRkSB utilizes the reused nodes that are stored in a heap to refine candidates, and GRkSB employs the global k-skyband and the $(k+1)$-th global skyline to get the final query result. Thus, RRkSB and GRkSB need to take $O(h \times g_{12})$ and $O((g+g') \times g_{12})$ for refinement. Consequently, the overall time complexities of RRkSB and GRkSB are $O(|R_P| + h \times g_{12})$ and $O(|R_P| + (g+g') \times g_{12})$, respectively. □

4. Ranked reverse skyline query processing

The traditional reverse skyline retrieval uses the L_1-norm to compute the distance. In this section, we propose the ranked reverse skyline (RRS) query, which employs the user specified function to compute the distance. Accordingly, we present new algorithms to answer the RRS query and analyze their correctness.

4.1. Problem formulation

First, we formally define the ranked reverse skyline query in Definition 6 below.

Definition 6 (*Ranked Reverse Skyline*). Given a d-dimensional data set P, a query point q, a function f which is monotone on each attribute, and a parameter K, let S be the reverse skyline of q, the ranked reverse skyline (RRS) of q contains at most K points, denoted as S_R, such that for any point $p \in S_R$, $o \in S - S_R$ satisfying (1) $S_R \subseteq S$ and (2) $f(p) \leqslant f(o)$.

The RRS query returns K reverse skyline points that have the minimum scores according to the input function. Consider the example in Fig. 1(c) again, where we suppose $K = 2$ and the preference function is $f(x,y) = x + 5y$. Since the reverse skyline of q in Fig. 1(c) is $S = \{p_3, p_{12}, p_{13}\}$, we can easily get their scores using f, i.e., $f(p_3) = 44$, $f(p_{12}) = 21$, and $f(p_{13}) = 29$. Based on Definition 6, the output 2 ranked reverse skyline points are $\langle p_{12}, 21\rangle$ and $\langle p_{13}, 29\rangle$ in this order, which is different from the result set of the RRS query in Section 1.

Although RRS retrieval is similar as the reverse skyline query, it has its own application base. The reverse skyline assigns the same weight to all the attributes as it employs the L_1-norm distance. In those cases, where some attributes are more important than others, the reverse skyline is not appropriate, because it cannot reflect different weights over various attributes. On the other hand, the RRS query allows the users to customize the function to match their own needs and then uses the user specified function to compute the distance. In addition, RRS retrieval allows the users to specify the (maximum) number of the points returned via a parameter K, while the reverse skyline query has no control of the size of the result set.

Algorithm 6. Reused-based RRS algorithm (RRRS).

Input: an R-tree R on a set of data objects, the query point q, the dynamic skyline of the dataset, a function f, a parameter K.
Output: the result set *RRS* of a ranked reverse skyline
/∗*GS*: the set of global skyline points; *RRS*: the set of ranked reverse skyline points. ∗/

```
initialize sets GS = RRS = ∅ and min-heaps H_g = H_w = ∅
insert all entries of the root into H_g sorted by f
while H_g ≠ ∅ do
  de-heap the top entry e of H_g
  if e is globally dominated by points in GS then
    insert e into heap H_w   // for reuse later
  else   // e is not globally dominated by points in GS
    if e is an intermediate entry then
      for each child e_i of e do
        if e_i is not globally dominated by points in GS then insert e_i into H_g
        else insert e into H_w
    else   // e is a data point
      insert e into H_w
      add e to GS
```

```
        if q is in the DADR(e) then add e to RRS
        else if q is in the DDR(e) then discard e
        else
          execute a window query on e (using H_g and H_w)
          if the query window contains point(s) then discard e
          else   // the query window does not contain any point
            add e to RRS
        if |RRS| = K then break
    return RRS
```

Based on the definition, a naive solution is to find all the reverse skyline points in the dataset, and then return those K points with the smallest values according to the input function. However, the size of the reverse skyline points could be much larger than K, and thus, it is not necessary to retrieve the complete set of reverse skyline points. In view of this, several efficient algorithms are proposed below.

4.2. RRS query processing algorithms

In the sequel, we adapt the algorithms for the traditional reverse skyline query to support RRS queries.

The traditional reverse skyline query processing algorithms, i.e., BBRS and RSSA, can be naturally extended to answer RRS queries. Accordingly, we develop *Branch-bound-based RRS algorithm* (BRRS) and *Pre-computation-based RRS algorithm* (PRRS) for RRS queries, by slightly modifying BBRS and RSSA algorithms, respectively. There are mainly two differences compared with conventional reverse skyline query processing algorithms. First, we change the *mindist* function to the new user-input function f. Due to the monotonicity of f, it is easy to prove that the output points are indeed reverse skyline points. The only change with respect to the original algorithm is the order of entries visited. It also ensures that the entries are visited in ascending order of their scores according to a specified *function f*. Second, the algorithm terminates after K points have been reported. Therefore, it reduces unnecessary traversal of the R-tree. It is worth noting that the maximal score of the returned K points is smaller than the score of any remaining object. As BRRS and PRRS are very similar as BBRS and RSSA, we skip their pseudo-codes for the space saving.

Algorithm 7. Global-skylines-based RRS algorithm (GRRS).

Input: an R-tree R on a set of data objects, the query point q, the dynamic skyline of the dataset, a function f, a parameter K

Output: the result set *RRS* of a ranked reverse skyline

```
/*GS1 is used for storing the 1st global skyline, PGS is used for storing the global skyline after pruned by the dynamic
  skyline. */
  initialize sets GS = GS1 = PGS = RRS = ∅ and a min-heap H_g = ∅
  insert all entries of the root into H_g sorted by f
  while H_g ≠ ∅ do
    de-heap the top entry e of H_g
    if e is globally dominated by 2 points in GS then
      discard e   // e is not in the global skyline
    else   // e is not globally dominated by points in GS
      if e is an intermediate entry then
        for each child e_i of e do
          if e_i is globally dominated by less than 2 points in GS then
            insert e_i into H_g
      else   // e is a data point
        if e is globally dominated by 1 point in GS then add e to GS1
        else add e to GS
        if q is in the DADR(e) then add e to RRS
        else if q is in the DDR(e) then discard e
        else add e to PGS
  for each point p ∈ pgs do   // Heuristic 3
    if the window based on p and q contains GS or GS1 then discard e
    else
      insert e into RRS
      if |RRS| = K then break
  return RRS
```

Since BRRS and PRRS need to traverse the R-tree repeatedly during the processing, we develop the *Reused-based RRS algorithm* (RRRS), which utilizes the reuse mechanism in PRRS. The pseudo-code of RRRS is depicted in Algorithm 6. Compared with PRRS, there are many differences. First, RRRS uses two heaps, i.e., H_g and H_w, to store the visited entries. H_g is to store the entries that are not dominated by the current global skyline and H_w is to store the entries that are dominated by the current global skyline. Consequently, during the processing, RRRS can not discard any entry unless it is expanded (lines 6, 11, and 13). Second, RRRS uses both H_g and H_w to perform the window query, but PRRS only uses H_g.

As mentioned in Section 1, the reverse k-skyband corresponds to the traditional reverse skyline if $k = 0$. Moreover, the RRS query is similar as the reverse skyline query except for the result of RRS. Therefore, some heuristics used for reverse k-skyband computation can also be adapted to the RRS query if we set $k = 0$. For example, we have proposed Heuristic 3 in Section 3.2.5 that uses the global k-skyband to prune unqualified objects. If we suppose $k = 0$, Heuristic 3 can be explained as follows: Given a query point q and a point p, if p's query window does not contain any global skyline point or global 1-skyband of q, p is a reverse skyline point; otherwise, p is not a reverse skyline point. This can be employed during the processing of RRS retrieval. Hence, we develop the *Global-skyline-based RRS algorithm* (GRRS) based on Heuristic 3. It utilizes the global skylines in the window query, and its pseudo-code is depicted in Algorithm 7. Specifically, GRRS uses a heap H_g to find all the global skyline and the global 1-skyline (lines 7–13), during which it also employs the dynamic skyline to discard unqualified points (lines 15–17). Finally, it uses Heuristic 3 to further refine the global skyline and then get the final query result (lines 18–23). GRRS is different from the aforementioned algorithms. To be more specific, when the number of the whole reverse skyline points is larger than K, BRRS, PRRS, and RRRS all do not need to find the whole global skyline, while GRRS still needs to find all the global skyline, because the partial global skyline can not fully refine candidate points.

4.3. Discussion

In this subsection, we prove the correctness of our proposed algorithms for RRS queries, namely, BRRS, PRRS, RRRS, and GRRS.

There are some lemmas presented in [9], which guarantee the correctness of BBRS algorithm. For instance, the algorithm visits the R-tree entries in ascending order of their distances to a specified query point; the number of the candidates examined by the algorithm is minimized; each data point will be evaluated unless one of its ancestor nodes has been pruned, and so on. Since all the four algorithms proposed above, i.e., BRRS, PRRS, RRRS, and GRRS, are adapted from BBRS, the lemmas mentioned in [9] can also be applied to those algorithms. Next, we present several lemmas that only apply to them.

Let $|P|$ be the cardinality of a data set P (i.e. the number of total data objects), K be the input parameter, s be the number of reverse skyline objects, g be the number of whole global skyline objects, g'' be the number of currently found global skyline objects, g_1 be the number of the global skyline objects refined by the dynamic skyline, g''_1 be the number of the currently found global skyline objects refined by the dynamic skyline.

Lemma 11. *If $K > s$, BRRS, PRRS, RRRS, and GRRS traverse the R-tree $(g + 1)$, $(g_1 + 1)$,1, and 1 time(s), respectively; if $K \leqslant s$, BRRS, PRRS, RRRS, and GRRS traverse the R-tree $(g'' + 1), (g''_1 + 1), 1$, and 1 time(s), respectively.*

Proof. The number that BRRS and PRRS traverse the R-tree depends on the number of the computed global skyline objects. Hence, BRRS and PRRS traverse the R-tree $(g + 1)$ and $(g_1 + 1)$ times, respectively. RRRS reuses the visited nodes, and GRRS only utilizes the R-tree to compute the global skyline and the 1st global skyline. Consequently, they traverse the R-tree a single once. The proof completes. □

5. Experimental evaluation

In this section, we evaluate the efficiency and scalability of our proposed algorithms via the experiments on both real and synthetic datasets. In what follows, we first present the experimental setup in Section 5.1, and then report our experimental results and findings for RkSB and RRS queries in Section 5.2 and Section 5.3, respectively.

5.1. Experimental settings

We use two real datasets, namely, the *CarDB* dataset and the *NBA* dataset. Specifically, *CarDB* is a 45 K-tuple dataset extracted from *Yahoo! Autos*, and we select two attributes (i.e., *price* and *mileage*); *NBA* extracted from http://www.databasebasketball.com includes 15,272 records about 3542 players, and we select four attributes (i.e., *number of games played* (GP), *total points* (PTS), *total rebounds* (REB), and *total assists* (AST)). We also generate many synthetic datasets in order to examine the scalability of the algorithms. These synthetic datasets follow four different distributions, viz., *Independent* (*IN*), *Clustered* (*CL*), *Correlated* (*CO*), and *Anti-correlated* (*AC*). All attribute values of *IN* datasets are generated independently using a uniform distribution; *CL* dataset comprises ten randomly centered clusters, and each of them follows a Gaussian distribution with the same number of points; *CO* dataset represents an environment, in which points that are good in one dimension are also good in other dimensions; *AC* dataset represents an environment, where points that are good in one dimension are bad in one or

all of the other dimensions. Fig. 10 illustrates four distributions of the synthetic datasets with 100,000 points in a 2-dimensional space. In the experiments, each dataset is indexed by an R-tree [3], with a page size of 4096 bytes.

We investigate the performance of the proposed algorithms under a variety of parameters, including the number t of dynamic skyline points, the thickness k of the reverse k-skyband, the number K of returned reverse skyline points, the dimensionality *dim*, the cardinality N, and the buffer size. Note that, in every experiment, only one factor varies, whereas the others are fixed to their default values. The range of the parameters and their default values are summarized in Table 2. We employ the number of node accesses (# node accesses), the CPU time, and the number of global k-skyband points pruned (*NGP*) as the main performance metrics. Each reported value in the following diagrams is the average of 100 queries, and the query object also follows the corresponding dataset distribution.

All algorithms were implemented by C++, and all the experiments were conducted on a PC with Intel Core 4 Duo 2.8 GHz processor and 4 GB RAM, running Microsoft Windows XP Professional Edition.

5.2. Results on RkSB queries

The first set of experiments focuses on the R*k*SB query. It contains two parts, with the first one aiming at verifying the effectiveness of our presented pruning heuristics and the other part studying the performance of our proposed algorithms (i.e., BR*k*SB, PR*k*SB, OPR*k*SB, RR*k*SB, and GR*k*SB) in answering R*k*SB queries.

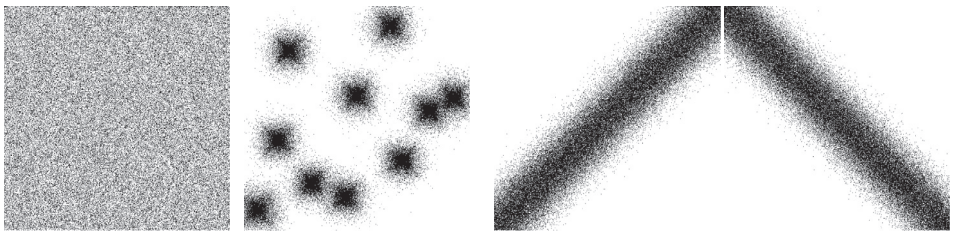
First, we evaluate the pruning efficiency of all proposed heuristics. Fig. 11 shows the results under different t values, with k fixed at 3 (the median value of Fig. 12) for real and synthetic datasets. Obviously, each heuristic prunes a large number of points, which validates its effectiveness. Take Heuristic 2 under the *CL* distribution as an example. It saves the detailed examination of 407 out of 544 points when $t = 50$. Compared with Heuristic 1, Heuristic 2 has a more powerful pruning capability. This is because, as mentioned in Section 3.2, the global k-skyband points are closer to a given query point, compared against other points, and thus, it has a high probability of falling inside the query window for pruning.

Heuristic 3 is even more powerful than Heuristic 2, since it can discard all the global skyband points. Observe that the *NGP* of Heuristic 1 is *zero* when $t = 0$, because there is no *DR* and *HR* that can be used to prune away unqualified candidate points. Then, we vary other parameters and report the pruning efficiency of heuristics w.r.t. k, *dim*, N, and *buffer size* using both real and synthetic datasets in Figs. 12–15 respectively. The diagrams confirm the observations and their corresponding explanations in Fig. 11.

The second set of experiments studies the performance of our proposed algorithms. Fig. 16 depicts the impact of t on PRkSB, OPR*k*SB, RR*k*SB, and GR*k*SB. Note that, since BR*k*SB does not employ dynamic skylines to prune points, it is excluded from this experiment. Clearly, OPR*k*SB outperforms PR*k*SB in all cases because it integrates Heuristics 1 and 2 to discard unqualified candidate points. Furthermore, as t grows, the I/O and CPU time of PR*k*SB drop. The reason is that the bigger t, the more candidates will be pruned. Therefore, the I/O and CPU time of PR*k*SB decrease. As expected, RR*k*SB and GR*k*SB are both better than OPR*k*SB and PR*k*SB in term of I/O. This is because RR*k*SB and GR*k*SB traverse the R-tree only once while OPR*k*SB and PR*k*SB traverse the R-tree multiple times. In addition, the CPU time of RR*k*SB is the worst among the five algorithms, since RR*k*SB needs to maintain the reuse heap during the processing, incurring more CPU cost.

Then, we verify the effect of k on the algorithms and report the performance in Fig. 17 for both real and synthetic datasets. GR*k*SB clearly outperforms RR*k*SB, OPR*k*SB, PR*k*SB, and BR*k*SB in all cases, and the advantage of GR*k*SB becomes more significant as k increases. This is because, when k becomes larger, more candidate points need to be checked, which results in more expensive query cost. The I/O cost of BR*k*SB is the worst since it does not utilize any pruning heuristic, and every candidate point triggers one window query. Again the CPU time of RR*k*SB is the worst, while its I/O overhead is very similar to GR*k*SB.

Next, we investigate the influence of *dim* on the algorithms. Towards this, we use the synthetic datasets with $k = 3$, $N = 100$ K, and *buffer size* = 0, and *dim* varied from 2 to 5, and report the results in Fig. 18. As expected, the performance of all algorithms degrades with the growth of *dim*. This degradation is due to the poor efficiency of R-trees in high dimensions. Another reason is that with the growth of dimensionality, the point has a lower possibility to be dominated. Therefore, the final result intends to include more points with increasing I/O and CPU cost. However, GR*k*SB still performs the best among all the algorithms.

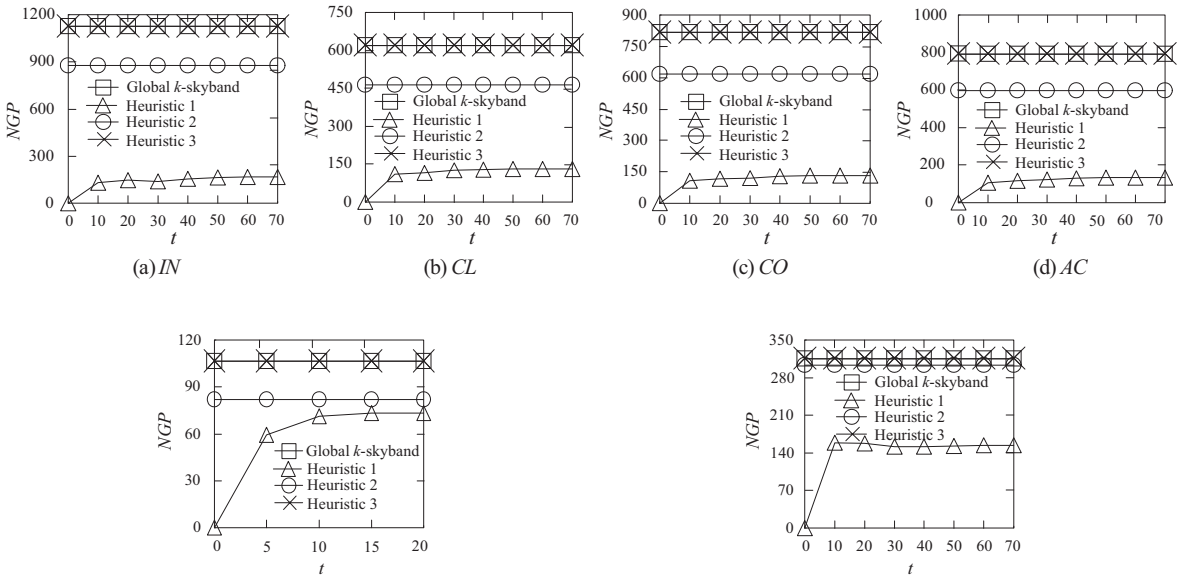


(a) *IN* (b) *CL* (c) *CO* (d) *AC*

Fig. 10. Dataset distribution.

Table 2
Parameters used in our experiments.

Parameter	Range	Default
Number t of dynamic skyline points	0, 10, 30, 50, 70	50
Thickness k of the reverse k-skyband	0, 1, 2, 3, 4, 5, 6	3
Number K of returned reverse skyline points	10, 20, 30, 40, 50	30
Dimensionality dim	2, 3, 4, 5	3
Cardinality N	40 K, 80 K, 120 K, 160 K, 200 K	100 K
Buffer size (KB)	0, 16, 32, 64, 128, 256, 512	0



(a) *IN* (b) *CL* (c) *CO* (d) *AC*

(e) *CarDB* ($k = 3$, $dim = 2$, $N = 45$K, buffer size = 0) (f) *NBA* ($k = 3$, $dim = 4$, $N = 15$K, buffer size = 0)

Fig. 11. Pruning efficiency vs. t(k = 3, dim = 3, N = 100 K, buffer size = 0).

We inspect the effect of N on the algorithms, by fixing t = 50, k = 3, dim = 3, and buffer size = 0, and employing synthetic datasets with cardinality N varied between 40 K and 200 K. Fig. 19 shows the results. Again, GRkSB outperforms the other four algorithms in all cases. Moreover, the costs of algorithms grow as CN increases. This is because the size of the final query result set ascends with N.

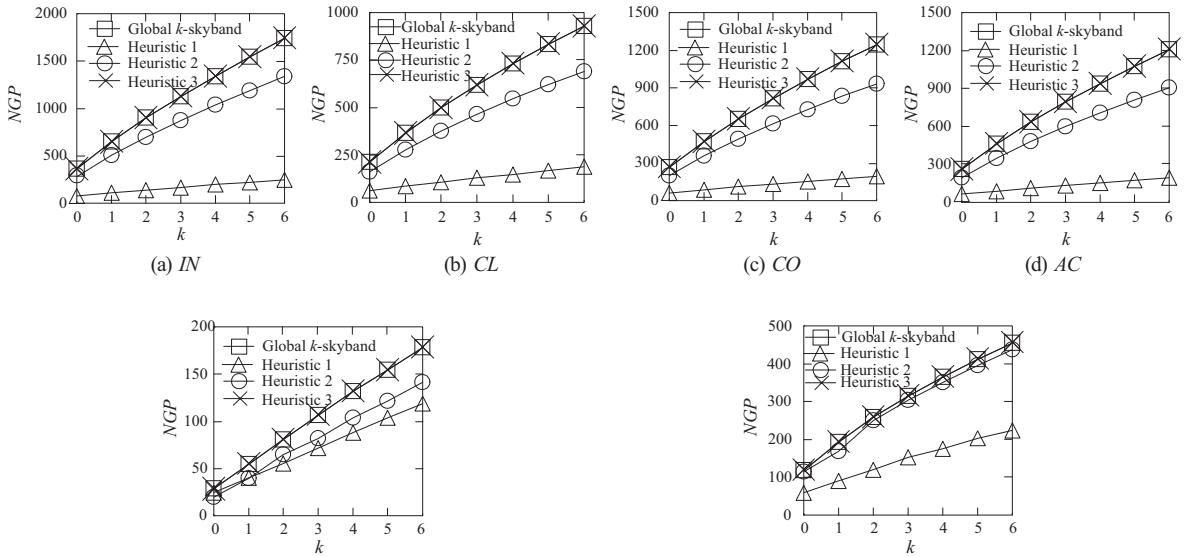
Finally, we explore the effect of buffer size on the algorithms, by fixing t = 50, k = 3, dim = 3, and N = 100 K. Fig. 20 shows the results for both real and synthetic datasets. Obviously, as the buffer size grows, the I/O cost drops gradually, whereas the CPU time remains almost the same. This is due to the mechanism of the buffer. Again, GRkSB outperforms the other four algorithms in all cases.

In summary, from the above experimental results on both real and synthetic datasets, we can conclude that GRkSB and RRkSB perform the best in terms of I/O cost, followed by OPRkSB, and PRkSB, and BRkSB is the worst. As for CPU time, GRkSB is still the best while the RRkSB is the worst.

5.3. Results on RRS queries

In the third sets of experiments, we verify the performance of the algorithms for RRS queries. Five parameters are also evaluated, i.e., the number K of returned ranked reverse skyline points, the number t of dynamic skyline, the dimensionality dim, the cardinality N, and the buffer size.

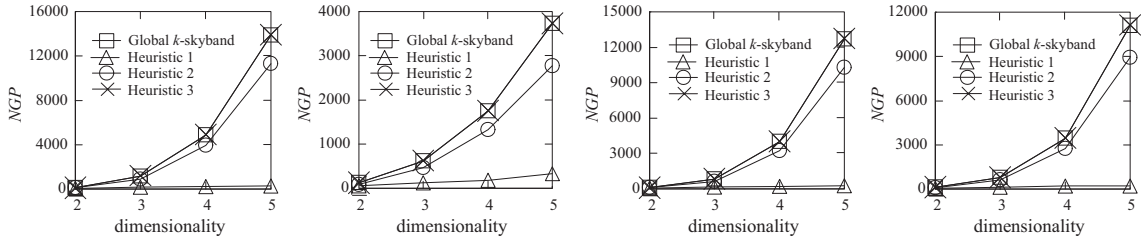
First, we study the effect of K on the algorithms and report the results in Fig. 21. The costs of the algorithms for RRS queries increase with the growth of K except GRRS that remains the same. This is because, when K grows, more candidate points need to be checked, which incurs more window queries with increased cost. As for GRRS, its cost mainly comes from the computation of global skyline and global 1-skyline. For a given dataset, the global skyline and the global 1-skyline are fixed. Consequently, the wall clock time remains the same. Observe that, in Fig. 21, when K is small, the I/O overhead of GRRS is worse than that of other algorithms. With the growth of K, the I/O cost of GRRS is better than that of BRRS and PRRS, but RRRS still is the best. The reason is that when getting a global skyline point, BRRS, PRRS, and RRRS examine whether it is in the final



(a) *IN* (b) *CL* (c) *CO* (d) *AC*

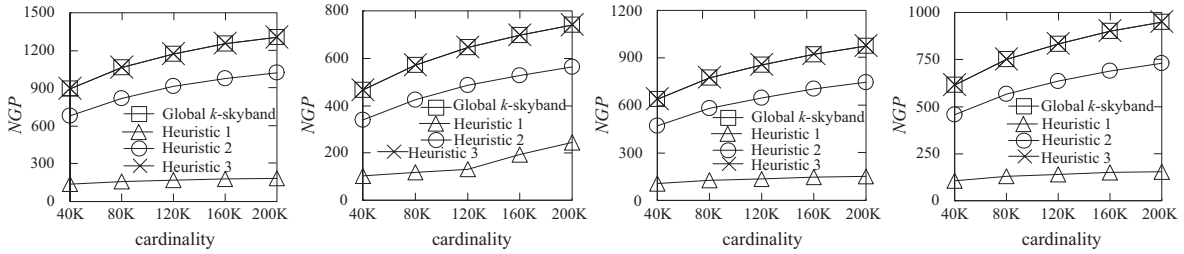
(e) *CarDB* (t = 10, dim = 2, N = 45K, buffer size = 0) (f) *NBA* (t = 50, dim = 4, N = 15K, buffer size = 0)

Fig. 12. Pruning efficiency vs. k(t = 50, dim = 3, N = 100 K, buffer size = 0).



(a) *IN* (b) *CL* (c) *CO* (d) *AC*

Fig. 13. Pruning efficiency vs. dim(t = 50, k = 3, N = 100 K, buffer size = 0).



(a) *IN* (b) *CL* (c) *CO* (d) *AC*

Fig. 14. Pruning efficiency vs. N(t = 50, k = 3, dim = 3, buffer size = 0).

result, and terminate the process once K ranked reverse skyline points are retrieved. In other words, BRRS, PRRS, and RRRS do not need to find the whole global skyline. However, GRRS is different, and it has to retrieve the complete set of the global skyline and the global 1-skyline. When K is small, although BRRS and PRRS traverse the R-tree repeatedly, their I/O overhead is still smaller than that of GRRS. Nevertheless, as K grows, more candidate points should be examined, and the traversal cost of the R-tree increases, which makes BRRS and PRRS perform worse than GRRS. The point at which GRRS exceeds BRRS and PRRS is different under various datasets, as shown in Fig. 21. This is caused by different distributions of the datasets. Although the I/O cost of RRRS performs best, its CPU time is the worst. This is because the RRRS should take more time in maintaining the reuse heap.

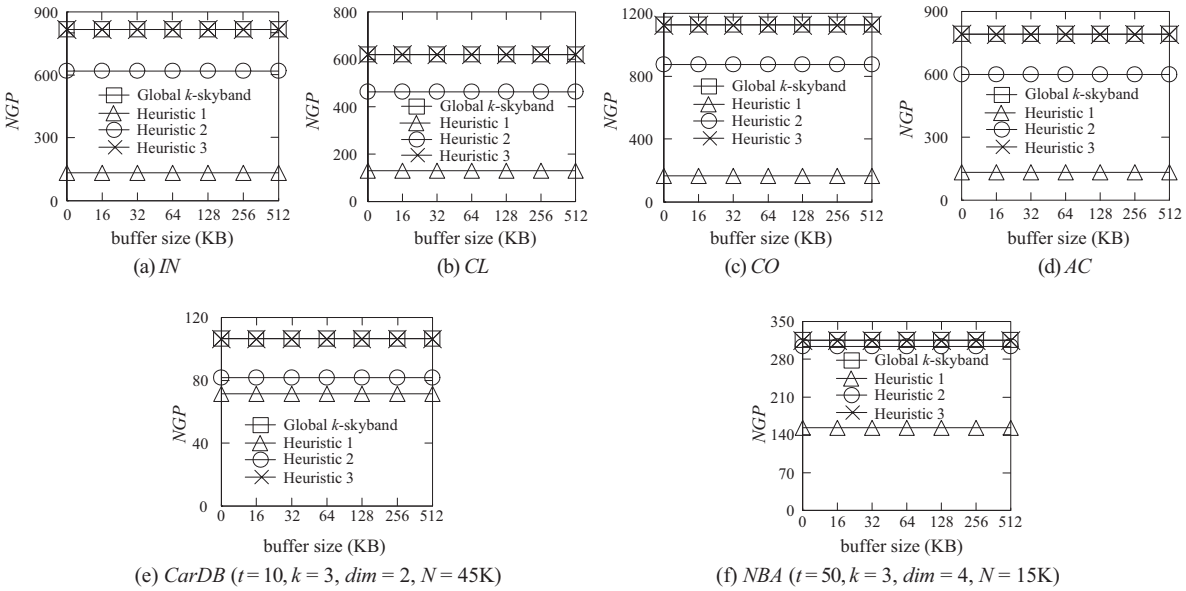


Fig. 15. Pruning efficiency vs. buffer size (t = 50, k = 3, dim = 3, N = 100 K).

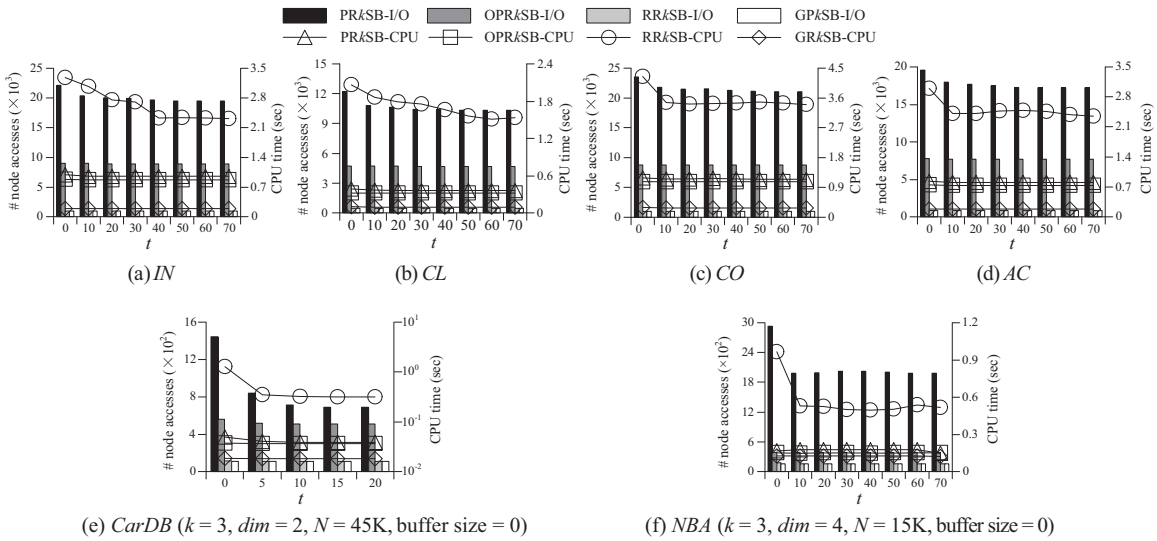


Fig. 16. RkSB cost vs. t(k = 3, dim = 3, N = 100 K, buffer size = 0).

Second, we explore the impact of t on PRRS, RRRS, and GRRS, and report the results in Fig. 22. Note that, since BRRS does not utilize dynamic skylines to prune points, it is excluded from this experiment. As t grows, the cost of PRRS drops, while those of RRRS and GRRS remain almost the same. This is because the number of dynamic skylines increases with the growth of t, which helps to prune away more points. PRRS only uses the dynamic skyline to prune candidates, while RRRS and GRRS employ other heuristics or the reuse technique apart from Heuristic 1. Therefore, the decrease tendency of RRRS and GRRS is not so evident, compared with that of PRRS. Obviously, the I/O cost of RRRS outperforms both GRRS and PRRS in all cases. In addition, the I/O overhead of that of GRRS is always better than PRRS for *CL*, *CO*, and *AC* datasets, but not *IN* dataset. This phenomenon is in accord with the result shown in Fig. 21(a) when K = 30. As for the CPU time, RRRS performs the worst.

Next, we investigate the influence of dim on the algorithms. Towards this, we change dim from 2 to 5 and fix K = 30 and N = 100 K, with results depicted in Fig. 23. The tendency of *IN* and *AC* datasets drops when dim grows, while that under *CL* and *CO* datasets is different. As explained before, when dim increases, it is more likely that an object is not dominated, and hence the size of final query result enlarges. Since the cardinality is fixed, there are more reverse skyline for the unit number of points. As r is also fixed, the number of candidate points that are found and refined decreases with the growth of dim. On

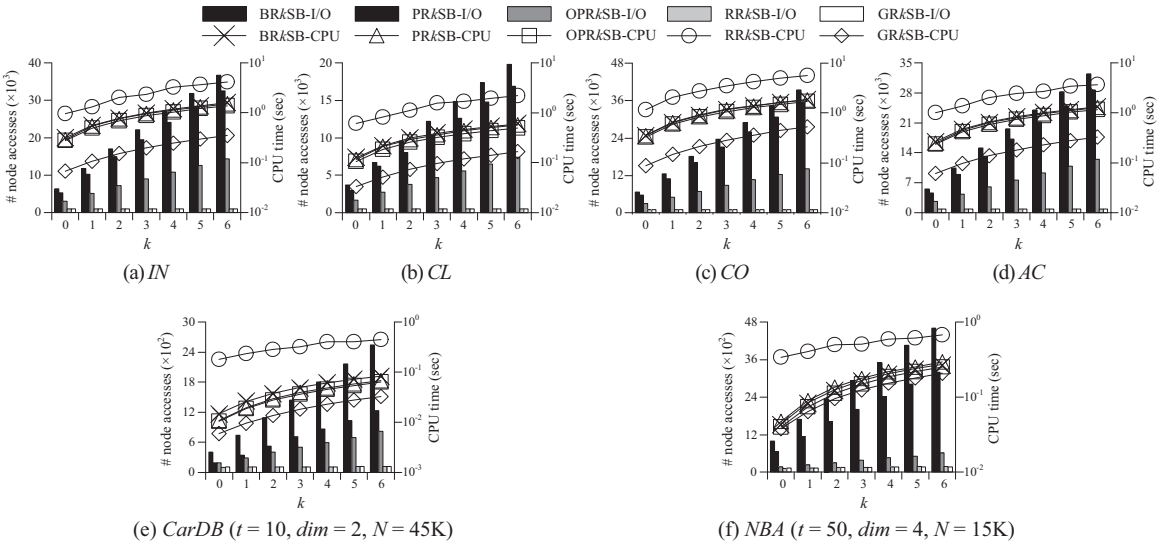


Fig. 17. R*k*SB cost vs. *k*(*t* = 50, *dim* = 3, *N* = 100 K, buffer size = 0).

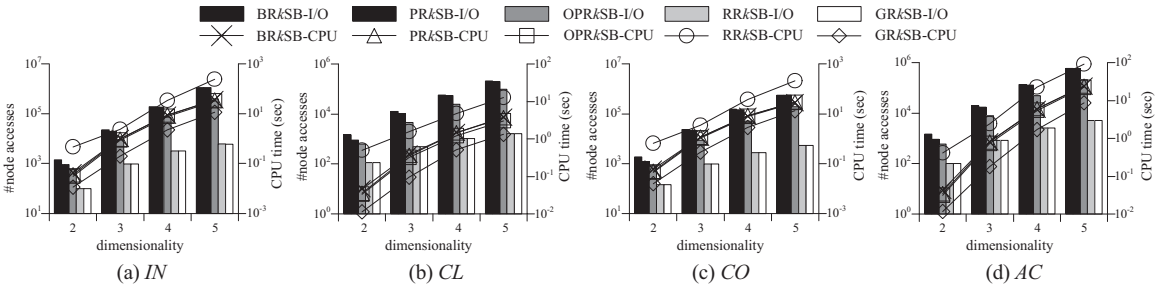


Fig. 18. R*k*SB cost vs. *dim*(*t* = 50, *k* = 3, *N* = 100 K, buffer size = 0).

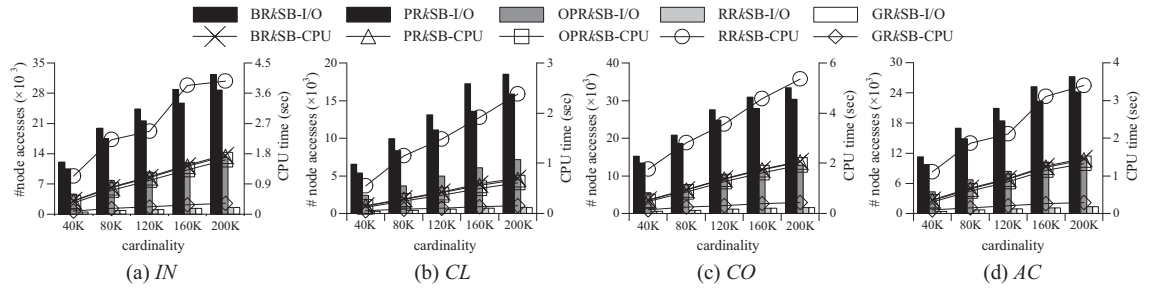


Fig. 19. R*k*SB cost vs. *N*(*t* = 50, *k* = 3, *dim* = 3, buffer size = 0).

the other hand, the performance of R-trees is poor in high dimensions, which also affects the efficiency of algorithms. For different data distributions, the influence caused by these two factors is different. For *IN* and *AC* datasets, the first factor plays a more important role, while for *CL* and *CO* datasets, the second factor is more important. BRRS degrades as *dim* grows in all cases. The reason is that it does not use any pruning technique, and the times it traverses the R-tree is far more than other algorithms. Similar as previous experiments, the I/O cost of RRRS is still the best for all cases.

Then, we inspect the effect of *N* on the algorithms, by fixing *t* = 50, *K* = 30, *dim* = 3, buffer size = 0, and employing synthetic datasets with cardinality *N* varied between 40 K and 200 K. Fig. 24 plots the performance of the algorithms as a function of *N*. Again, RRRS always outperforms other algorithms in terms of I/O overhead. With the growth of *N*, the performance of GRRS degrades sharply, and it performs even worse than BRRS and PRRS. This is because when *r* is fixed, BRRS, PRRS, and GRRS all retrieve the results incrementally. However, GRRS finalizes its result set until it finds all the global skyline and the

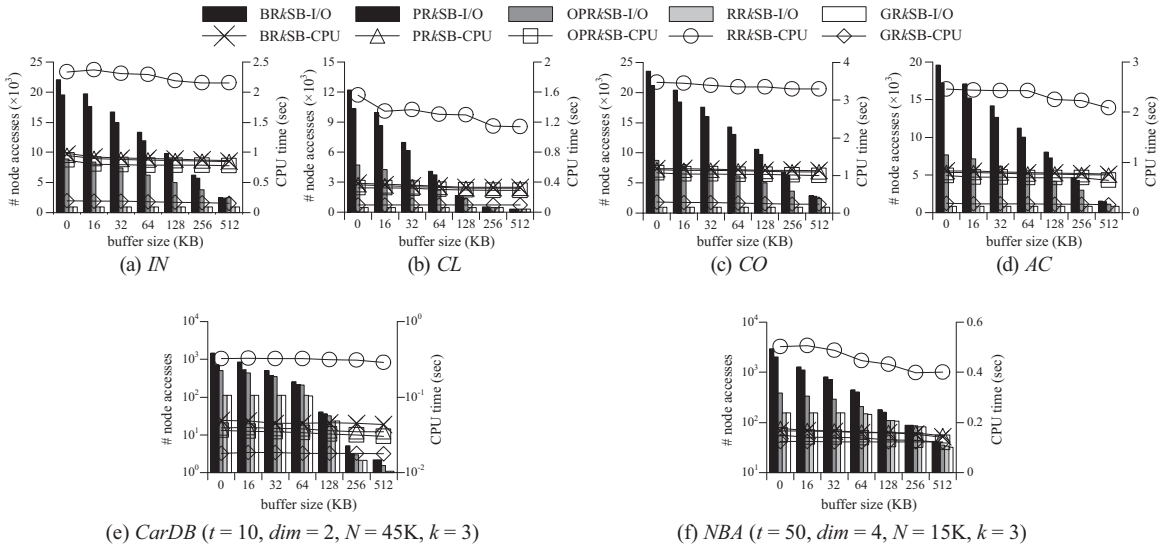


Fig. 20. R*k*SB cost vs. buffer size (t = 50, dim = 3, N = 100 K, k = 3).

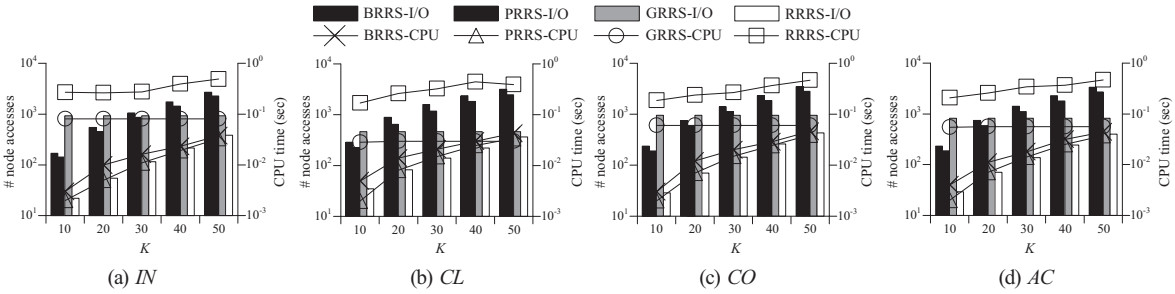


Fig. 21. RRS cost vs. K(t = 50, dim = 3, N = 100 K, buffer size = 0).

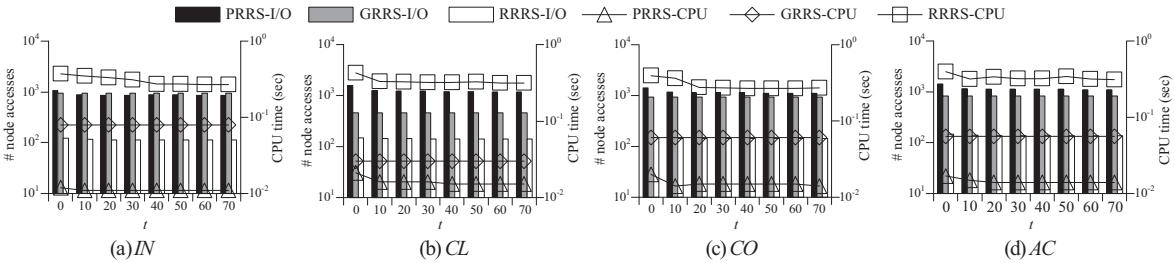


Fig. 22. RRS cost vs. t(K = 30, dim = 3, N = 100 K, buffer size = 0).

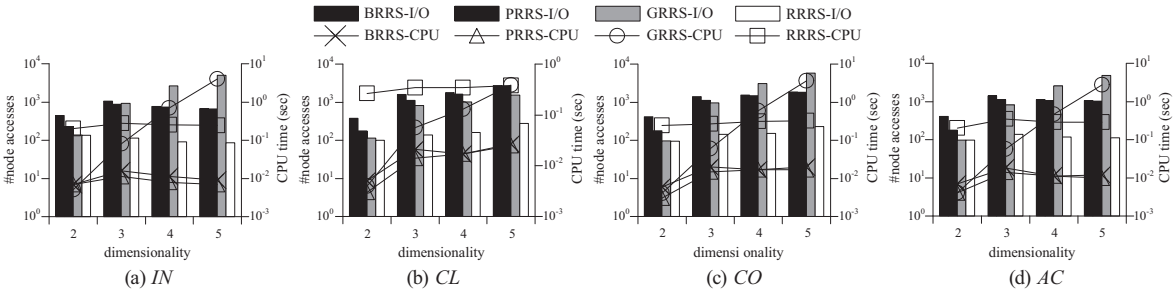


Fig. 23. RRS cost vs. dim(t = 50, K = 30, N = 100 K, buffer size = 0).

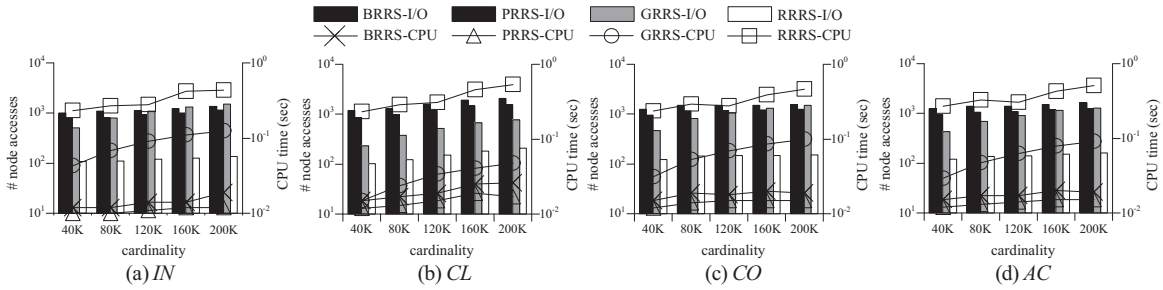


Fig. 24. RRS cost vs. N(t = 50, K = 30, dim = 3, buffer size = 0).

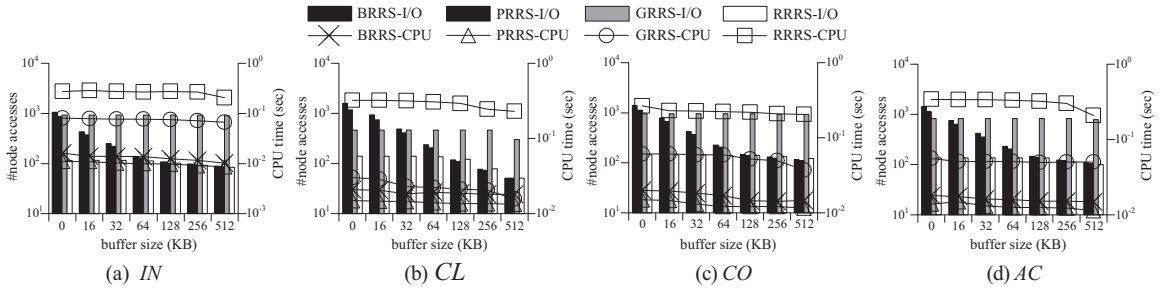


Fig. 25. RRS cost vs. buffer size (t = 50, K = 30, dim = 3, N = 100 K).

global 1-skyline, which incurs larger I/O cost. As N increases, the cost of the computation for the global skyline and the global 1-skyline also grows, resulting in the poor performance of GRRS.

Finally, we explore the effect of buffer size on the algorithms, by fixing t = 50, K = 30, dim = 3, and N = 100 K. Fig. 25 shows the performance of the algorithms as a function of buffer size. As expected, the I/O cost drops gradually with the growth of buffer size. The phenomenon is caused by the mechanism of the buffer.

To sum up, from the above experimental results on both real and synthetic datasets, we can conclude that the I/O overhead of RRRS performs the best. The performance of GRRS is varying based on the values of parameters.

6. Conclusions

Although the reverse skyline query has been received much attentions in the database community, its applications under some specified circumstances are limited. To this end, in this paper, we introduce RkSB and RRS queries, and propose a suite of supporting algorithms based on R-trees. In particular, we investigate the RkSB query, whose dynamic k-skyband contains a given query point. It corresponds to the traditional reverse skyline retrieval when k = 0. Five algorithms, i.e., BRkSB, PRkSB, OPRkSB, RRkSB, and GRkSB are developed to answer the RkSB query efficiently. As a second step, we study the RRS query, which returns the K reverse skyline points whose scores are the minimum. Since RRS retrieval is similar as the traditional reverse skyline query, we extend BBRS and RSSA to support the RRS query, namely, BRRS and PRRS. Moreover, to boost PRRS, two improved algorithms RRRS and GRRS are also proposed. In the future, we intend to investigate the reverse skyline query and its variants in metric spaces and over incomplete data, respectively.

Acknowledgments

Yunjun Gao was supported in part by NSFC Grants 61379033 and 61003049, the Fundamental Research Funds for the Central Universities under Grant 2013QNA5020, and the Key Project of Zhejiang University Excellent Young Teacher Fund (Zijin Plan).

References

[1] K. Banafaa, R. Wen, X. Li, Keyword-matched data skyline in Peer-to-Peer systems, in: Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA), 2013, pp. 71–85.
[2] I. Bartolini, P. Ciaccia, M. Patella, Efficient sort-based skyline evaluation, ACM Trans. Datab. Syst. 33 (4) (2008) 1–45.
[3] N. Beckmann, H.P. Kriegel, R. Schneider, B. Seeger, The R*-tree: an efficient and robust access method for points and rectangles, in: Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data (SIGMOD), 1990, pp. 322–331.
[4] S. Borzsony, D. Kossmann, K. Stocker, The skyline operator, in: Proceedings of the 17th International Conference on Data Engineering (ICDE), 2001, pp. 421–430.

[5] L. Chen, X. Lian, Efficient processing of metric skyline queries, IEEE Trans. Knowl. Data Eng. (TKDE) 21 (3) (2008) 351–365.
[6] H. Choi, H. Jung, K. Lee, Y. Chung, Skyline queries on keyword-matched data, Inform. Sci. 232 (2013) 449–463.
[7] J. Chomicki, P. Godfrey, J. Gryz, D. Liang, Skyline with presorting, in: Proceedings of the 19th International Conference on Data Engineering (ICDE), 2003, pp. 717–719.
[8] B. Cui, L. Chen, L. Xu, H. Lu, G. Song, Q. Xu, Efficient skyline computation in structured peer-to-peer systems, IEEE Trans. Knowl. Data Eng. (TKDE) 21 (7) (2009) 1059–1072.
[9] E. Dellis, B. Seeger, Efficient computation of reverse skyline queries, in: Proceedings of the 33rd International Conference on Very Large Data Base (VLDB), 2007, pp. 291–302.
[10] X. Ding, X. Lian, L. Chen, H. Jin, Continuous monitoring of skylines over uncertain data streams, Inform. Sci. 184 (1) (2012) 196–214.
[11] D. Fuhry, R. Jin, D. Zhang, Efficient skyline computation in metric space, in: Proceedings of the 12th International Conference on Extending Database Technology (EDBT), 2009, pp. 1042–1051.
[12] P. Godfrey, R. Shipley, J. Gryz, Maximal vector computation in large data sets, in: Proceedings of the 31st International Conference on Very Large Data Base (VLDB), 2005, pp. 229–240.
[13] Z. Huang, H. Lu, B.C. Ooi, A.K.H. Tung, Continuous skyline queries for moving objects, IEEE Trans. Knowl. Data Eng. (TKDE) 18 (12) (2006) 1645–1658.
[14] M. Islam, R. Zhou, C. Liu, On answering why-not questions in reverse skyline queries, in: Proceedings of the 29th International Conference on Data Engineering (ICDE), 2013, pp. 973–984.
[15] B. Jiang, J. Pei, Online interval skyline queries on time series, in: Proceedings of the 25th International Conference on Data Engineering (ICDE), 2009, pp. 1036–1047.
[16] G.T. Kailasam, J. Lee, J. Rhee, J. Kang, Efficient skycube computation using point and domain-based filtering, Inform. Sci. 180 (7) (2010) 1090–1103.
[17] M.E. Khalefa, M.F. Mokbel, J.J. Levandoski, Skyline query processing for incomplete data, in: Proceedings of the 24th International Conference on Data Engineering (ICDE), 2008, pp. 556–565.
[18] D. Kossmann, F. Ramsak, S. Rost, Shooting stars in the sky: An online algorithm for skyline queries, in: Proceedings of the 28th International Conference on Very Large Data Base (VLDB), 2002, pp. 275–286.
[19] H.T. Kung, F. Luccio, F.P. Preparata, On finding the maxima of a set of vectors, J. ACM 22 (4) (1975) 469–476.
[20] K.C.K. Lee, B. Zheng, H. Li, W.-C. Lee, Approaching the skyline in z order, in: Proceedings of the 33rd International Conference on Very Large Data Base (VLDB), 2007, pp. 279–290.
[21] M.-W. Lee, S.-W. Hwang, Continuous skylining on volatile moving data, in: Proceedings of the 25th International Conference on Data Engineering (ICDE), 2009, pp. 1568–1575.
[22] J. Lee, J. Kim, S. Hwang, Supporting efficient distributed skyline computation using skyline views Original, Inform. Sci. 194 (1) (2012) 24–37.
[23] X. Lian, L. Chen, Reverse skyline search in uncertain databases, ACM Trans. Datab. Syst. 35 (1) (2010) (article 3).
[24] X. Lin, Y. Yuan, W. Wang, H. Lu, Stabbing the sky: Efficient skyline computation over sliding windows, in: Proceedings of the 21st International Conference on Data Engineering (ICDE), 2005, pp. 502–513.
[25] X. Lin, Y. Zhang, W. Zhang, M.A. Cheema, Stochastic skyline operator, in: Proceedings of the 27th International Conference on Data Engineering (ICDE), 2011, pp. 721–732.
[26] Q. Liu, Y. Gao, G. Chen, Q. Li, On efficient reverse *k*-skyband query processing, in: Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA), 2012, pp. 544-559.
[27] C. Lofi, K. Maarry, W. Balke, Skyline queries in crowd-enabled databases, in: Proceedings of the 16th International Conference on Extending Database Technology (EDBT), 2013, pp. 465–476.
[28] M. Morse, J.M. Patel, W.I. Grosky, Efficient continuous skyline computation, Inform. Sci. 177 (17) (2007) 3411–3437.
[29] D. Papadias, Y. Tao, G. Fu, B. Seeger, Progressive skyline computation in database systems, ACM Trans. Datab. Syst. 30 (1) (2005) 41–82.
[30] J. Pei, B. Jiang, X. Lin, Y. Yuan, Probabilistic skylines on uncertain data, in: Proceedings of the 33rd International Conference on Very Large Data Base (VLDB), 2007, pp. 15–26.
[31] J. Pei, W. Jin, M. Ester, Y. Tao, Catching the best views of skyline: a semantic approach based on decisive subspaces, in: Proceedings of 31st International Conference on Very Large Data Base (VLDB), 2005, pp. 253–264.
[32] J. Pei, Y. Yuan, X. Lin, W. Jin, M. Ester, Q. Liu, Towards multidimensional subspace skyline analysis, ACM Trans. Datab. Syst. 31 (4) (2006) 1335–1381.
[33] M.D. Prasad, P. Deepak, Efficient reverse skyline retrieval with arbitrary non-metric similarity measures, in: Proceedings of the 14th International Conference on Extending Database Technology (EDBT), 2011, pp. 319–330.
[34] K.-L. Tan, P.-K. Eng, B.C. Ooi, Efficient progressive skyline computation, in: Proceedings of the 27th International Conference on Very Large Data Base (VLDB), 2001, pp. 301–310.
[35] Y. Tao, D. Papadias, Maintaining sliding window skylines data streams, IEEE Trans. Knowl. Data Eng. (TKDE) 18 (3) (2006) 377–391.
[36] Y. Tao, X. Xiao, J. Pei, SUBSKY: Efficient computation of skylines in subspaces, in: Proceedings of the 22nd International Conference on Data Engineering (ICDE), 2006, pp. 65–65.
[37] G. Wang, J. Xin, L. Chen, Y. Liu, Energy-efficient reverse skyline queries processing over wireless sensor networks, IEEE Trans. Knowl. Data Eng. (TKDE) 24 (7) (2012) 1259–1391.
[38] S. Wang, B.C. Ooi, A.K.H. Tung, L. Xu, Efficient skyline query processing on peer-to-peer networks, in: Proceedings of the 23rd International Conference on Data Engineering (ICDE), 2007, pp. 1126–1275.
[39] X. Wu, Y. Tao, R.C.-W. Wong, L. Ding, J.X. Yu, Finding the influence set through skylines, in: Proceedings of the 12th International Conference on Extending Database Technology (EDBT), 2009, pp. 1030–1041.
[40] T. Xia, D. Zhang, Refreshing the Sky: The compressed skycube with efficient support for frequent updates, in: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), 2006, pp. 491–502.
[41] Y. Yuan, X. Lin, Q. Liu, W. Wang, J.X. Yu, Q. Zhang, Efficient computation of the skyline cube, in: Proceedings of the 31st International Conference on Very Large Data Base (VLDB), 2005, pp. 241–252.
[42] S. Zhang, N. Mamoulis, D.W. Cheung, Scalable skyline computation using object-based space partitioning, in: Proceedings of the 35th SIGMOD International Conference on Management of Data (SIGMOD), 2009, pp. 483–494.
[43] W. Zhang, X. Lin, Y. Zhang, W. Wang, J.X. Yu, Probabilistic skyline operator over sliding windows, in: Proceedings of the 25th International Conference on Data Engineering (ICDE), 2009, pp. 1060–1071.
[44] Z. Zhang, R. Cheng, D. Papadias, A.K.H. Tung, Minimizing the communication cost for continuous skyline maintenance, in: Proceedings of the 35th SIGMOD international conference on Management of data (SIGMOD), 2009, pp. 495–508.
[45] L. Zhu, C. Li, H. Chen, Efficient computation of reverse skyline on data stream, in: Proceedings of the International Joint Conference on Computational Sciences and Optimization, 2009, pp. 735–739.