6-2014

On Efficient Reverse Skyline Query Processing

Yunjun GAO
Zhejiang University

Qing LIU
Zhejiang University

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

Gang CHEN
Zhejiang University

Expert Systems with Applications 41 (2014) 3237–3249

Contents lists available at ScienceDirect

Expert Systems with Applications

journal homepage: www.elsevier.com/locate/eswa

On efficient reverse skyline query processing

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

[a] College of Computer Science, Zhejiang University, Hangzhou 310027, China
[b] School of Information Systems, Singapore Management University, Singapore 178902, Singapore

ARTICLE INFO

Keywords:
Skyline
Reverse skyline
Constrained reverse skyline
Query processing
Algorithm

ABSTRACT

Given a D-dimensional data set P and a query point q, a *reverse skyline query* (RSQ) returns all the data objects in P whose dynamic skyline contains q. It is important for many real life applications such as business planning and environmental monitoring. Currently, the state-of-the-art algorithm for answering the RSQ is the *reverse skyline using skyline approximations* (RSSA) algorithm, which is based on the precomputed approximations of the skylines. Although RSSA has some desirable features, e.g., applicability to arbitrary data distributions and dimensions, it needs for *multiple accesses* of the same nodes, incurring *redundant* I/O and CPU costs. In this paper, we propose several efficient algorithms for *exact* RSQ processing over *multidimensional* datasets. Our methods utilize a conventional data-partitioning index (e.g., R-tree) on the dataset P, and employ *precomputation*, *reuse*, and *pruning* techniques to boost the query performance. In addition, we extend our techniques to tackle a natural variant of the RSQ, i.e., *constrained reverse skyline query* (CRSQ), which retrieves the reverse skyline inside a specified *constrained region*. Extensive experimental evaluation using both real and synthetic datasets demonstrates that our proposed algorithms *outperform* RSSA by *several orders of magnitude* under all experimental settings.

1. Introduction

The skyline query is a research hot topic in database community. For a specified dataset P, a skyline query returns the objects/points in P that are not dominated by others. In particular, for a given D-dimensional dataset P, if a point $p_1 \in P$ *dominates* another point $p_2 \in P$, it must hold that (1) $i \in [1,D]$, $p_1[i] \leqslant p_2[i]$, and (2) $\exists j \in [1,D]$, $p_1[j] < p_2[j]$. Here, $p[i]$ is the ith dimensional value of the point p, and we assume the *smaller* the *better*. For example, we have a set $P = \{a,b,\ldots,h,i\}$ of hotels, as depicted in Fig. 1(a), with x-axis corresponding to the room price of every hotel, and y-axis representing its distance to the beach. Note that, the price of hotel a is *cheaper* than that of hotel b, and it is *closer* to the beach compared with b. Hence, a dominates b. All the hotels a, g, h, and i *not dominated* by any other hotel constitute the skyline of P. The skyline query is very useful in many multi-criteria decision making and user preference queries. For instance, in the above hotel example, it can return, for the tourists, a small set of interesting hotels from a big pool, to save the tourists' time, when they need to find a cheap hotel yet close to the beach to stay.

The traditional skyline query is *static* as it takes into account the static attribute values of data points in a dataset P. In other words, the skyline of P is *fixed*. However, if we specify a query point q and consider points' dominance relationships w.r.t. q, the skyline of P (w.r.t. q) is *not fixed*, and thus, it is referred to as the *dynamic skyline query*. Specifically, for a given D-dimensional dataset P and a query point q, if a point $p_1 \in P$ dominates another point $p_2 \in P$ w.r.t. q, it must hold that (1) $i \in [1,D]$, $|p_1[i] - q[i]| \leqslant |p_2[i] - q[i]|$, and (2) $\exists j \in [1,D]$, $|p_1[j] - q[j]| < |p_2[j] - q[j]|$. An example of the dynamic skyline query is illustrated in Fig. 1(b), where every green point denotes the transformed point w.r.t. q (i.e., a point p is transformed to the point p' w.r.t. q via $p'[i] = |p[i] - q[i]|$), and the dynamic skyline of the specified dataset w.r.t. q consists of hotels e, g, and h. Compared with the traditional skyline query, the dynamic skyline query offers users more *flexibility* in specifying the search criteria. Back to the aforementioned hotel example again, if a certain tourist prefers a hotel with room price around \$100 and its distance to beach around 300 m, he/she can specify his/her ideal hotel as the query point, and then, the dynamic skyline query is ran to return the hotels that match his/her preference best.

In general, if a specified query point q belongs to the dynamic skyline of a given dataset P w.r.t. a certain point $p \in P$, p is said to be in the *reverse skyline* of P w.r.t. q. A *reverse skyline* query (RSQ) finds all the points in P that have q as a member of their dynamic skylines. Take Fig. 1 as an example. Suppose a dynamic skyline query is issued at point g, and hence, its result set contains hotels e, h, and q, as shown in Fig. 1(c). If an RSQ is issued at point q, hotels e, f, g, h, and i are returned as the result, as depicted in Fig. 1(d). Intuitively, the reverse skyline indicates the influence of a query

* Corresponding author. Address: College of Computer Science, Zhejiang University, 38 Zheda Road, Hangzhou 310027, China. Tel.: +86 571 8765 1613; fax: +86 571 8795 1250.
E-mail addresses: gaoyj@zju.edu.cn (Y. Gao), liuqing1988@zju.edu.cn (Q. Liu), bhzheng@smu.edu.sg (B. Zheng), cg@zju.edu.cn (G. Chen).

0957-4174/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.eswa.2013.11.012

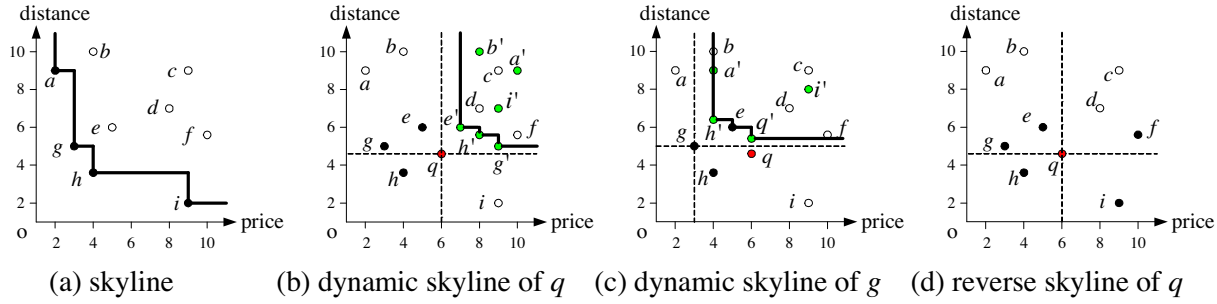


Fig. 1. Example of skyline, dynamic skyline, and reverse skyline.

point on a dataset, i.e., how many objects in the dataset may be interested in the query point. Consequently, RSQ can be applied to the applications from the companies' perspective, and it is different from the skyline and dynamic skyline queries because they are performed from the perspective of customers. As an example, a certain real estate agent has a set of potential customers with different preferences. In order to locate the target customers who might be interested in the real estate listings he/she has, the agent can use real estate listings as the query points, and perform the reverse skyline query over the dataset storing the customers' preferences. The result is just the customers in potential. Another example is the business planning. Assume that the decision-maker of a computer manufacturer has designed a new type of computers, and he/she would like to know how many customers may be interested in this new/forthcoming computer. If only few people have an interest on it, the decision-maker of the computer manufacturer might change quickly some parameters of the current one or redesign a new one. In this case, he/she can take the new computer as a query point q and the potential customer preferences (e.g., CPU, main-memory, etc.) as a dataset P, and then perform the reverse skyline query. The cardinality of the query result will help him/her decide whether the new computer needs to be put into production.

Currently, the *branch and bound reverse skyline* (BBRS) algorithm and the *reverse skyline using skyline approximations* (RSSA) algorithm are the *state-of-the-art* algorithms for answering reverse skyline queries. In particular, BBRS is an improved customization of the original BBS algorithm (Papadias, Tao, Fu, & Seeger, 2005); RRSA is based on accurate precomputed approximations of the skylines; and as demonstrated in Dellis and Seeger (2007), RRSA *exceeds* BBRS significantly in all the cases. Although RSSA has some desirable features, e.g., applicability to arbitrary data distributions and dimensions, it has to traverse the index R-tree (Beckmann, Kriegel, Schneider, & Seeger, 1990) on a dataset *repeatedly*, incurring *unnecessary* I/O and CPU costs. Hence, their performance has much room for improvement. In addition, as demonstrated in Papadopoulos and Manolopoulos (1997), the efficiency of an R-tree is poor in high dimensions. If there exists *repeated traversal* of the R-tree, the performance of the algorithms using R-trees deteriorate dramatically, which also inspires us to boost the efficiency of current reverse skyline algorithms. Moreover, in some business planning applications, the users require getting the query result within a limited time, for quick decision making.

In addition to the traditional reverse skyline query, many reverse skyline variants are also propose in the literature, such as *bichromatic reverse skyline* (Wu, Tao, Wong, Ding, & Yu, 2009), reverse skyline queries over *uncertain* data (Lian & Chen, 2010) and *data stream* (Zhu, Li, & Chen, 2009), reverse skyline query with *arbitrary non-metric* similarity measures (Prasad & Deepak, 2011), reverse skyline queries in *wireless sensor networks* (Wang, Xin, Chen, & Liu, 2012), *reverse k-skyband* (Liu, Gao, Chen, Li, & Jiang, 2012). As to be discussed in Section 2, all these variants are inherently different from the traditional RSQ. Hence, the techniques developed for these variants are not directly applicable to tackle the RSQ. Motivated by this, in this paper, we develop several algorithms for *exact* RSQ processing over *multidimensional* datasets. Our methods utilize a conventional data-partitioning index (e.g., R-tree) on the dataset, and employ *precomputation*, *reuse*, and *pruning* techniques to boost the query performance.

In some real applications, users might enforce some *constraints* (e.g., *spatial region*, *distance*, etc.) on reverse skyline queries. Take the real estate agent as an example. In order to keep the profits and absorb as many customers as possible, the price of the house should be neither too low nor too high. Also, the agent may want to know the desired customers within a certain price range. Hence, we propose a natural variant of RSQ, namely, *constrained reverse skyline query* (CRSQ), which retrieves the reverse skyline inside a specified *constrained region* (instead of the *whole* data space). In addition, we extend our approaches to tackle it since the reverse skyline algorithms without taking the region constraints into consideration cannot efficiently handle CRSQ. In brief, the key contributions of this paper can be summarized as follows:

- We utilize the *reuse mechanism* to achieve a *single* traversal of the R-tree, which significantly reduces the I/O overhead, and the CPU time accordingly.
- We develop an effective global skyline based *pruning heuristic* to further improve the computational performance of RSQ, and carry out a comprehensive theoretical analysis.
- We introduce CRSQ, an intuitive RSQ variant, and extend our techniques to handle it efficiently.
- We conduct extensive experiments with both real and synthetic datasets to show that our proposed algorithms *outperform* RSSA by *several orders of magnitude* under all experimental settings.

The rest of this paper is organized as follows. Section 2 reviews the related work. Section 3 proposes efficient algorithms to process RSQ, and analyzes their correctness. Section 4 presents efficient algorithms for answering CRSQ, and theoretical analysis accordingly. Considerable experimental results and our findings are reported in Section 5. Finally, Section 6 concludes the paper with some directions for future work.

2. Related work

In this section, we survey the previous work on skyline query and its variations, focusing mostly on traditional skyline and reverse skyline queries.

2.1. Skyline queries

The skyline query is a popular paradigm for extracting interesting objects from multidimensional databases. Since it was first introduced into the database community in Borzsony, Kossmann,

and Stocker (2001), many algorithms for computing the skyline have been proposed in the literature. They can be classified into two categories: (1) *non-index* based approaches, and (2) *index* based methods. The first one involves solutions that do not assume any index on the underlying dataset, but they retrieve the skyline by scanning the dataset. The existing algorithms belonging to this category include *Block Nested Loop* (BNL) (Borzsony et al., 2001), *Divide-and-Conquer* (D&C) (Borzsony et al., 2001), *Sort-Filter-Skyline* (SFS) (Chomicki, Godfrey, Gryz, & Liang, 2003), *Linear Elimination Sort for Skyline* (LESS) (Godfrey, Shipley, & Gryz, 2005), *Sort and Limit Skyline algorithm* (SaLSa) (Bartolini, Ciaccia, & Patella, 2008), and *Object-based Space Partitioning* (OSP) (Zhang, Mamoulis, & Cheung, 2009). In particular, the most efficient skyline computation algorithm for non-indexed data is OSP, which recursively divides the D-dimensional space into 2^D separate partitions w.r.t. a reference skyline object, and facilitates progressive skyline retrieval on high dimensional datasets. Methods of the other category exploit an appropriate index structure, e.g., an R-tree (Beckmann et al., 1990), to accelerate skyline computation. The existing algorithms in this second category contain *Bitmap* (Tan, Eng, & Ooi, 2001), *Index* (Tan et al., 2001), *Nearest Neighbor* (NN) (Kossmann, Ramsak, & Rost, 2002), *Branch and Bound* (BBS) (Papadias et al., 2005), and *ZSearch* (Lee, Zheng, Li, & Lee, 2007). Note that, BBS is the state-of-the-art indexed approach for skyline retrieval using the R-tree, and its I/O cost is proven to be optimal in Papadias et al. (2005).

In addition to conventional skyline operator, numerous skyline query variants have also been studied in the literature. Examples include *constrained* skyline query (Chen, Cui, & Lu, 2011; Dellis, Vlachou, Vladimirskiy, Seeger, & Theodoridis, 2006; Papadias et al., 2005), *dynamic* skyline query (Papadias et al., 2005; Sacharidis, Bouros, & Sellis, 2008), *k-Skyband* query (Papadias et al., 2005), *SkyCluster* query (Huang, Xiang, Zhang, & Liu, 2011), *subspace* skyline query (Pei et al., 2006; Tao, Xiao, & Pei, 2011), *metric* skyline query (Chen & Lian, 2009; Fuhry, Jin, & Zhang, 2009), *probabilistic* skyline query (Pei, Jiang, Lin, & Yuan, 2007; Zhang, Lin, Zhang, Wang, & Yu, 2009), *representative* skyline query (Lin, Yuan, Zhang, & Zhang, 2007; Tao, Ding, Lin, & Pei, 2009), *stochastic* skyline query (Lin, Zhang, Zhang, & Cheema, 2011), *parallel* skyline query (Gao, Chen, Chen, & Chen, 2006; Kohler, Yang, & Zhou, 2011; Vlachou, Doulkeridis, & Kotidis, 2008; Wu et al., 2006), and skyline retrieval in *distributed* environments (such as P2P systems and web information systems) (Hose & Vlachou, 2012), to name but a few.

2.2. Reverse skyline queries

The reverse skyline query (RSQ), a novel variation of skyline operator, introduced in Dellis and Seeger (2007), has received considerable attention from the database research community in the past few years, due to its importance in a wide spectrum of applications such as business planning (Dellis & Seeger, 2007; Prasad & Deepak, 2011), environmental monitoring (Lian & Chen, 2010; Wang et al., 2012), preference-based marketing (Dellis & Seeger, 2007), profile-based investment (Wu et al., 2009), and habitat monitoring (Wang et al., 2012). The existing RSQ processing algorithms include BBRS and RSSA. In order to reduce the search space, the two algorithms introduce the concept of global skyline (Dellis & Seeger, 2007). Given a D-dimensional data set P and a query point q, the global skyline of q contains the points that are not globally dominated by any other point w.r.t. q. Specifically, if a point $p_1 \in P$ globally dominates $p_2 \in P$ w.r.t. q, it must hold that (1) $\forall i \in [1, D]$, $(p_1[i] - q[i])(p_2[i] - q[i]) > 0$; (2) $\forall i \in [1, D], |p_1[i] - q[i]| \leqslant |p_2[i] - q[i]|$; mand (3) $\exists j \in [1, D], |p_1[j] - q[j]| < |p_2[j] - q[j]|$. Fig. 2 illustrates an example of global skyline of q, which includes points c, e, f, h, j and k.

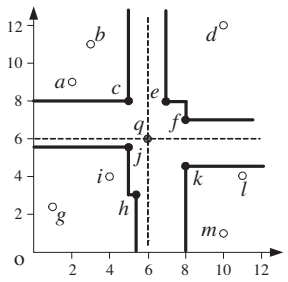


Fig. 2. Example of global skyline.

Specifically, BBRS is an improved customization of the original BBS algorithm, and it works as follows. First, BBRS uses a *min-heap* (sorted by the entry distances to a given query point q) to retrieve the *global skyline* of q, which is the *superset* of the actual reverse skyline set (proved in Dellis and Seeger (2007)). Subsequently, BBRS performs, for every global skyline point, a window query for refinement. For the global skyline point p, its query window is the area that is centered at p and has the coordinate-wise distance to a given query point q as its extent. Actually, the window query is a *Boolean window query*, which only needs to determine whether there is any point located inside the window. The window query works as follows. Initially, it initializes the heap based on the root. It then proceeds to de-heap the top entry in the heap for evaluation until the heap is empty. If the entry is within the window, there must be point in the window, and thus, the window query terminates; if the entry intersects the window, it has to be expanded, and its child entries are inserted into the heap; and otherwise, the entry must be out of the window, and hence, it is discarded. Take the dataset depicted in Fig. 3 as an example for BBRS. The heap contents during the reverse skyline computation are shown in Table 1, where the set $S = \{j, c, k\}$ represents the final reverse skyline.

On the other hand, RSSA is an enhanced version of BBRS, and it utilizes the *dynamic skyline* to prune away *unqualified* points. Note that, the dynamic skyline of each point is *precomputed* and kept on disk. Then, similar as BBRS, RSSA retrieves the *global skyline*. Thereafter, every global skyline point p is evaluated based on the concepts of *Dynamic Dominance Region* (DDR: the region that contains the points dominated by at least one skyline point) and *Dynamic Anti-Dominance Region* (DADR: the region that contains the points dominating a certain skyline point), which are determined by the precomputed dynamic skyline of p. If the query point q is located within the DADR of some global skyline point p, p is definitely a result point. If q is located inside the DDR of a certain global skyline point p, p definitely is not a result point. If a global skyline point cannot be refined by either DDR or DADR, it has to be further verified by running the window query, just like BBRS

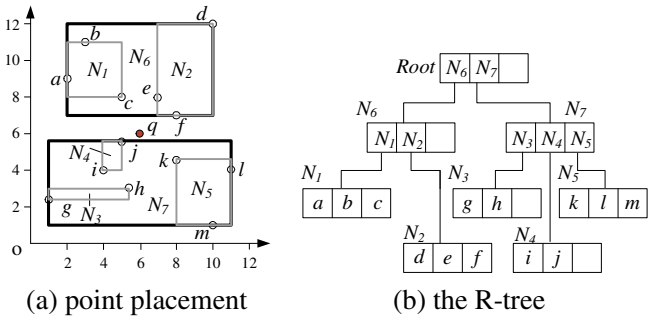
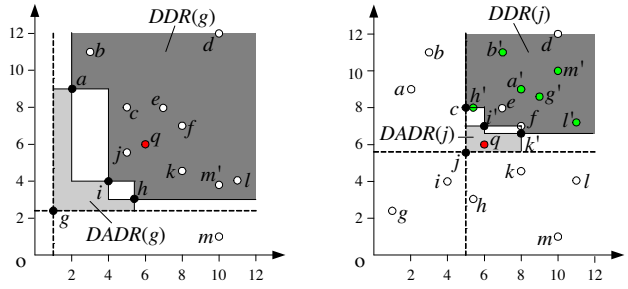


Fig. 3. Example of BBRS.

Table 1
Heap contents of BBRS.

Action	Heap contents	S
Access *Root*	$\langle N_7, N_6\rangle$	$\varnothing$
Expand N_7	$\langle N_6, N_4, N_5, N_3\rangle$	$\varnothing$
Expand N_6	$\langle N_4, N_2, N_1, N_5, N_3\rangle$	$\varnothing$
Expand N_4	$\langle \mathbf{j}, N_2, N_1, N_5, i, N_3\rangle$	{j}
Expand N_2	$\langle$ ~~e, f~~, $N_1, N_5, i, N_3, d\rangle$	{j}
Expand N_1	$\langle \mathbf{c}, N_5, \mathbf{i}, N_3, a, b, d\rangle$	{j, c}
Expand N_5	$\langle \mathbf{k}, i, N_3, a, l, b, m, d\rangle$	{j, c, k}
Expand N_3	$\langle$ ~~k~~, ~~a~~, ~~l~~, ~~b~~, ~~g~~, ~~m~~, ~~d~~$\rangle$	{j, c, k}



(a) *DDR*(g) and *DADR*(g) (b) *DDR*(j) and *DADR*(j)

Fig. 4. Example of DDR and DADR.

does. For illustration, Fig. 4 depicts the DDR and DADR of data points g and j, respectively. In Fig. 4(a), since a query point q falls completely into *DDR*(g), g is not a reverse skyline point; whereas q lies in *DADR*(j), as shown in Fig. 4(b), and thus, j is certainly a reverse skyline point. As demonstrated in Dellis and Seeger (2007), RSSA outperforms BBRS under all instances. However, since every window query traverses the R-tree (on a specified dataset) from the root, it needs to traverse the R-tree *multiple times*, especially for a large number of global skyline points *not refined* by either DDR or DADR, which results in *unnecessary* I/O and CPU costs. In this paper, we use *precomputation*, *reuse*, and *pruning* techniques to achieve a *single* traversal of the R-tree, and boost the computational performance of the reverse skyline accordingly.

In addition, reverse skyline queries under different environments are explored as well. Wu et al. (2009) investigate *bichromatic reverse skyline* (BRS) retrieval for traditional dataset, in which each object is a *precise* point. They propose the BRS algorithm that is designed for precise points, and seamlessly integrates several *non-trivial heuristics* in order to reduce the I/O cost. The BRS retrieval is different from RSQ since it takes two datasets into consideration while RSQ only considers one dataset. Hence, BRS algorithm cannot be applied to RSQ directly. Lian and Chen (2010) study *monochromatic* and *bichromatic* reverse skyline queries over *uncertain* data, where every object is modeled as a probability distribution function. They develop *spatial* and *probabilistic pruning* methods to reduce the search space of the reverse skyline query, and employ *precomputation* technique to further improve the query performance. Zhu et al. (2009) present a *Divide and Conquer Reverse Skyline* (DCRS) algorithm to compute the reverse skyline on *data stream*, which provides *continuous* and *high-speed* data elements. More recently, Prasad and Deepak (2011) consider the reverse skyline query with *arbitrary non-metric* similarity measures, i.e., the attributes do not have a total ordering among their values. Prasad and Deepak explore block-based processing of objects and pre-processing to speed-up computational and IO costs. Wang et al. (2012) discuss how to process reverse skyline queries energy-efficiently in *wireless sensor networks*, and propose a sky-band-based approach to tackle this problem. Liu et al. (2012) study the problem of *reverse k-skyband* (RkSB) query processing, and develop several efficient algorithms for computing the RkSB of an *arbitrary* query point by using *precomputation* and *pruning* techniques. Moreover, Islam, Zhou, and Liu (2013) answer the why-not questions in reverse skyline queries. Specifically, they show how to modify the why-not point (data point) and query point to include the why-not point in the reverse skyline of the query point. Towards it, Islam et al. (2013) propose the safe region of a query point where it can be moved while keeping its existing reverse skyline. All these reverse skyline variations are proposed under certain circumstances and are different from RSQ. Therefore these algorithms cannot be applied to tackle the RSQ directly. On the other hand, these algorithms of reverse skyline variations are different from the one we propose in this paper.

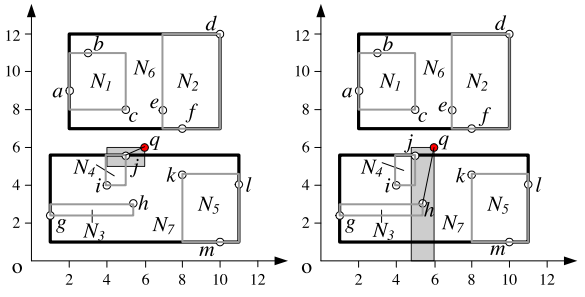
3. Reverse skyline computation

In this section, we formally define the reverse skyline query (RSQ), and then present our proposed two RSQ processing algorithms and their corresponding analysis.

3.1. Problem formulation

Let P be a D-dimensional dataset. For any point $p \in P$, we use $p[i]$ to denote the ith dimensional value of p. A point $p_1 \in P$ is said to *dominate* another point $p_2 \in P$, denoted as $p_1 \prec p_2$, if (i) for every $i \in \{1,2,\ldots,D\}$, $p_1[i] \leqslant p_2[i]$; and (ii) for *at least one* $j \in \{1,2,\ldots,D\}$, $p_1[j] < p_2[j]$. For example, in Fig. 1(a), point g dominates point e, i.e., $g \prec e$.

Definition 3.1 (*Reverse skyline query*). Given P and a query point q, a reverse skyline query (RSQ) finds all the points in P which take q as one of their dynamic skyline points, that is, if a point $p \in P$ is a



(a) Query window of j (b) Query window of h

Action	Heap contents
access Root	{ N_7, N_6 }
expand N_7	{ N_4, N_6, N_5, N_3 }
expand N_4	{ ~~j, N_6, i, N_5, N_3~~ }

(c) Window query for j

Action	Heap contents
access Root	{ N_7, N_6 }
expand N_7	{ N_4, N_3, N_6, N_5 }
expand N_4	{ j, N_3, i, N_6, N_5 }

(d) Window query for h

Fig. 5. Example of window queries.

reverse skyline point of q, there does not exist any other point o $(\neq p) \in P$, such that (1) $\forall i \in [1, D], |o[i] - p[i]| \leqslant |q[i] - p[i]|$; and (2) $\exists j \in [1, D], |o[j] - p[j]| < |q[j] - p[j]|$.

Take Fig. 1 as an example again. Since the dynamic skyline of g contains the query point q, g is a reverse skyline point of q. As mentioned earlier, so far RSSA is the most efficient RSQ processing algorithm, whereas it has to traverse the index R-tree on the dataset *repeatedly*, incurring *unnecessary* I/O and CPU costs. To address this, in what follows, we propose several enhanced algorithms by using *precomputation*, *reuse*, and *pruning* techniques.

3.2. Reverse skyline query processing

In this subsection, we describe two efficient algorithms, namely, *full-reuse-based reverse skyline* (FRRS) algorithm, and *global-skyline-based reverse skyline* (GSRS) algorithm, for computing the reverse skyline. In particular, FRRS is based on the *reuse* mechanism to avoid *multiple* accesses of the same node, which significantly reduces the I/O and CPU costs. GSRS utilizes a novel *global-skyline-based pruning heuristic* to eliminate evaluating candidate global skyline points via window queries, and thus, further improve the query performance.

3.2.1. The FRRS algorithm

Recall that if a global skyline point p cannot be refined by DDR/DADR, existing RSSA algorithm applies a *window* query to verify it. As an example, the query window of j is shown in Fig. 5(a) with its corresponding window query processing steps and heap contents listed in Fig. 5(c), and the query window of h is depicted in Fig. 5(b) with its corresponding window query processing steps and heap contents listed in Fig. 5(d). In this case, we can observe that some entries (e.g., *Root*, N_4, and N_7) are visited *multiple times* when performing *window queries* for different points. However, if we store the visited R-tree nodes of previous window query and directly use it in the next time, it can avoid *redundant* node accesses for all the window queries. By using the reuse technique for all the window queries, it can achieve to traverse the R-tree *only once*, which reduces the I/O cost significantly. Nonetheless, there is still some room for further improvement. Recall that the first step of RSSA is to compute the global skyline, which also needs to traverse the R-tree. If we apply the reuse mechanism in both the global skyline computation and the window query processing, we can cut down the R-tree traversal to *only once*.

To employ the reuse technique, we should preserve the visited R-tree nodes. To this end, we can either maintain all the visited nodes in the heap during the query processing or only store the deepest level nodes of R-tree. Since maintaining all the visited nodes takes considerable space, we adopt the second way in this paper. When we utilize the reuse technique during the processing of global skyline computation and window query, the visited R-tree nodes can be classified into two categories: the first category includes the node entries *not* globally dominated by the retrieved global skyline points; and the other category contains the node entries globally dominated by the retrieved global skyline points. Therefore, in this paper, we use two heaps (H_g for the first category node entries, and H_w for the second category node entries) to maintain the visited R-tree nodes. The combination of H_g and H_w is just the deepest level's nodes visited currently. One issue needs to be addressed is that any node entry should not be discarded during the processing unless it is expanded. Hence, it guarantees that we do *not miss* any node entry in the whole query processing. In the following, we illustrate the details of how the reuse technique is used.

We integrate the reuse technique into RSSA algorithm to develop our first RSQ query processing algorithm, i.e., FRRS, whose pseudo-code is presented in Algorithm 1. Firstly, FRRS utilizes H_g to find the global skyline of query point, during which the node entries globally dominated by the retrieved global skyline points are inserted into the heap H_w for the reuse later (lines 6 and 11 of Algorithm 1). Then, FRRS uses dynamic skyline to refine the global skyline and insert the refined points into the heap H_w as well (lines 14 and 15 of Algorithm 1). Finally, the remained global skyline is further refined by window query using both heaps H_g and H_w (lines 17–19 of Algorithm 1). Algorithm 2 shows the pseudo-code of a window query. In the traditional window query, if a node is located outside the window, it can be pruned away. Nevertheless, when employing the reuse mechanism, this node has to be kept because it may be useful for other window queries (lines 10 of Algorithm 2). We employ an auxiliary heap H_a to store this node temporarily. Once the current window query stops, the entries in the auxiliary heap H_a should be shifted to the work heap for the reuse later (lines 11 of Algorithm 2).

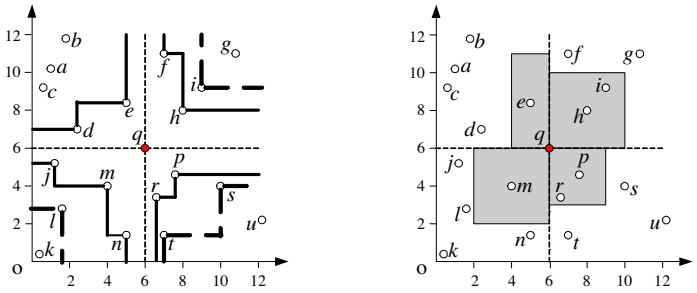
Algorithm 1 full-reuse-based reverse skyline algorithm (FRRS)

Input: an R-tree R on a data set, a query point q, the dynamic skyline of the dataset
Output: the result set S_r of an RSQ

```
1:  initialize sets S_g = S_r = ∅ and min-heaps H_g = H_w = ∅
2:  insert all entries of the root 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 any point in S_g then
6:      insert e into H_w // for the reuse later
7:    else
8:      if e is an intermediate entry then
9:        for each child e_i ∈ e do
10:         if e_i is globally dominated by any point in S_g
      then
11:           insert e_i into H_w // for the reuse later
12:       else insert e_i into H_g
13:     else // e is a data point
14:       if q is in DADR(e) then add e to S_r and add e to H_w
15:       else if q is in DDR(e) then insert e into H_w
16:       else
17:         if Window-Query(H_g, e, q) AND Window-
      Query (H_w, e, q) then
18:           add e to S_r // e is a result point
19:         insert e into H_w // for the reuse later
20:     add e to S_g // for the next round
21: return S_r
```

Algorithm 2 Window-query(H, p, q)

```
1:  initialize tag = TRUE and an auxiliary min-heap H_a = ∅
2:  while H ≠ ∅ do
3:    de-heap the top entry e of H
4:    if e is in the window based on p and q then
5:      tag = FALSE and insert e into H_a // e can be pruned
6:      break // function terminates
7:    else if e crosses the window based on p and q then
8:      for each child e_i ∈ e do
9:        insert e_i into H
10:   else insert e into H_a// e is out of the window
11:   shift from elements in H_a to H if H_a ≠ ∅ // for the reuse
      later
12:   return tag
```



(a) Global skyline and global 1-skyline (b) Lemma 3.1

Fig. 6. Example of Definition 3.2, and Lemma 3.1.

3.2.2. The GSRS algorithm

Although the reuse technique can avoid *multiple accesses* of the same nodes, the window query processing is not very efficient. For example, even though *none* of the heap entries is actually inside the query window of the real reverse skyline point, an exhaustive evaluation is still triggered, which needs to scan all the entries in the heaps H_g and H_w. In addition, the FRRS algorithms have a very important shortcoming. It needs to maintain in memory any data object or node that has been retrieved from disk, and cannot discard any node that qualifies for the global skyline or window query. Even though the R-tree is traversed at most once, this may lead to maintain the entire data set in memory, which is not feasible for large data sets. To address these deficiencies, we develop a *novel heuristic* to prune away all the *unqualified* points using the global skyline. Our second algorithm, namely, GSRS, employs this heuristic, and it also traverses the R-tree *only once*. Before we present GSRS, we first introduce an important concept, i.e., the *global* 1-*skyline*, which is derived based on the global skyline.

Definition 3.2 (*Global 1-skyline*). Given a D-dimensional data set P and a query point q, if a point $p \in P$ is in the global 1-skyline of q, there exists *only one* point o $(\neq p) \in P$ such that (1) $\forall i \in [1, D], (p[i] - q[i])(o[i] - q[i]) > 0$; (2)$\forall i \in [1, D], |o[i] - q[i]| \leqslant |p[i] - q[i]|$; and (3)$\exists j \in [1, D], |o[j] - q[j]| < |p[j] - q[j]|$.

In other words, global 1-skyline query finds all the points that are globally dominated by *exact one* point. It is different from the global skyline in which global skyline does not globally dominated by other points. Fig. 6(a) shows an example of global skyline and global 1-skyline. Points d, e, f, h, j, m, n, r and p constitute the global skyline since they are not dominated by any other point. Point i is *only* globally dominated by point h, whereas the point g is globally dominated by both point h and point i. Consequently, based on Definition 3.2, point i, but not point g, is a global 1-skyline point. Similarly, we can find the whole global 1-skyline points, i.e., i, l, s and t. Note that, the global 1-skyline is an extension of the global skyline, and has some relationships with it: (i) the global 1-skyline point must be globally dominated by *one* global skyline point, i.e., the non global skyline point cannot globally dominate it; and (ii) a global skyline point may globally dominates many global 1-skyline points or none. Based on the characteristics of global skyline and global 1-skyline, we offer the following lemma to support GSRS.

Lemma 3.1. *Given a D-dimensional data set P and a query point q, for a query window corresponding to any point p, if it does not contain any global skyline point or global 1-skyline point, the query window must be empty.* □

Proof. Assume, on the contrary, that there is at least one point p' in the query window. According to Lemma 3.1, p' is *neither* a global skyline point *nor* a global 1-skyline point, and thus, it is globally dominated by at least one global skyline point p_i or global 1-skyline point p_j $(\neq p_i)$. Based on the formation of the query window and the dominance relationship defined, p_i or p_j must be within the query window, which contradicts the condition of Lemma 3.1. Consequently, our assumption is *invalid*, and the proof completes. □

As shown in Fig. 6(b), the query windows for points m and e do not include any global skyline point or global 1-skyline point and their windows are all empty. On the other hand, the window of h contains global 1-skyline point i, and that of p contains global skyline point r. According to the result of window query, we can easily determine that points e and m, not points h and p, are the reverse skyline points. Therefore, based on Lemmas 3.1, we propose the Heuristic 3.1 below.

Heuristic 3.1. Given a D-dimensional data set P and a query point q, if p's query window does not contain any global skyline point or global 1-skyline of q, p is a real reverse skyline point; otherwise, p is not a reverse skyline point. □

Heuristic 3.1 suggests that, for *no* window query, we only need to check whether any global skyline point or global 1-skyline point is located inside the query window, rather than to perform

Algorithm 3 global-skyline-based reverse skyline algorithm (GSRS)

Input: an R-tree R on a data set, a query point q, the dynamic skyline of the dataset
Output: the result set S_r of an RSQ
/* S_{g1}: the set of global 1-skyline; S_c: the set of candidate global skyline points that need to apply Heuristic 3.1 to further verify. */
1: initialize sets $S_g = S_{g1} = S_r = S_c = \varnothing$ and a min-heap $H_g = \varnothing$
2: insert all entries of the root 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 at least 2 points in $S_g \cup S_{g1}$ **then**
6: discard e // e is not a global skyline and global 1-skyline point
7: **else**
8: **if** e is an intermediate entry **then**
9: **for** each child $e_i \in e$ **do**
10: **if** e_i is globally dominated by at most 1 point in $S_g \cup S_{g1}$ **then**
11: insert e_i into H_g
12: **else** // e is a data point

```
13:          if e is globally dominated by 1 point in S_g ∪ S_g1 then
14:             insert e into S_g1 // for the next round
15:          else
16:             insert e into S_g // for the next round
17:             if q is in DADR(e) then add e to S_r // e is a result point
18:             else if q is in DDR(e) then discard e
19:             else insert e into S_c
20:    for each candidate c_i ∈ S_c do // Huristic 3.1
21:    if the window based on c_i and q contains any point in S_g or S_g1 then
22:       discard c_i
23:    else add c_i to S_r // c_i is a result point
24:    return S_r
```

the window query via traversing the R-tree. GSRS applies Heuristic 3.1 to improve the query performance of RSSA. When getting a point/node, we can compute the times T it is globally dominated by the current found global skyline and global 1-skyline. If $T = 0$, it is/contains global skyline point. If $T = 1$, it is/contains global 1-skyline point. Otherwise, it is/contains neither global skyline point nor global 1-skyline point. Therefore, the global 1-skyline can be computed based on the global skyline and itself, and hence, GSRS traverses the R-tree only once. The pseudo-code of GSRS is presented in Algorithm 3, and its basic idea is as follows. First, it computes the global skyline and the global 1-skyline (lines 4–16 of Algorithm 3). Second, the global skyline points are refined by the dynamic skylines that are pre-computed (lines 17–18 of Algorithm 3). For all the remaining points that need further refinement, GSRS utilizes the entire global skyline and the global 1-skyline to determine the final result, using Heuristic 3.1 (lines 20–23 of Algorithm 3).

3.3. Analysis

In the sequel, for our proposed RSQ algorithms, namely, FRRS and GSRS, we analyse their properties and prove their correctness.

Lemma 3.2. *The FRRS and GSRS visit (data point and intermediate) entries of an R-tree in ascending order of their distances to the specified query point q.*

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

Lemma 3.3. *Every data point will be examined, unless one of its ancestor nodes has been pruned away.*

Proof. The proof is obvious because all entries that are not discarded by existing global skyline points (preserved in the set S_g) are added to the heap and examined. □

Lemma 3.4. *FRRS and GSRS can return exactly the real reverse skyline, i.e., every algorithm has no false negative, no false positive, and the reported result set S_r contains no duplicate points.*

Proof. First, no result point is *missed* (i.e., *no false negative*) as only *unqualified* entries, which are globally dominated by the existing global skyline point(s) or/and global 1-skyline point(s), are pruned away. Second, all the entries that can contain or be the actual reverse skyline point(s) are verified to ensure *no false positive*. Third, *no duplicate points* are guaranteed since each qualified point is evaluated *a single once*. Therefore, the proof completes. □

Lemma 3.5. *FRRS and GSRS visit the R-tree only once.*

Proof. First, FRRS reuses the nodes visited in the computation of global skyline when executing the window queries, and hence, it incurs *one* R-tree traversal. Second, GSRS traverses the R-tree *a single once* to compute the global skyline and the global 1-skyline, and then employs Heuristic 3.1 to validate the final result. The proof completes. □

Lemma 3.6. *The number of node accesses involved in FRRS and GSRS is minimized (no redundancy and repeated traverse).*

Proof. The proof is obvious since FRRS and GSRS traverse the R-tree *only once* according to Lemma 3.5, and only the entries that are not pruned by existing global skyline points and global 1-skyline points are inserted into the heap and examined. □

4. Constrained reverse skyline computation

In this section, we extend our techniques presented above to handle a natural variant of reverse skyline queries, namely, *constrained reverse skyline query* (CRSQ), which aims to compute the reverse skyline in a specified region. In what follows, we formalize the CRSQ, and then propose several algorithms for answering CRSQ and offer their theoretical analysis.

In some real applications, users might enforce some *constraints* (e.g., *spatial region*, *distance*, etc.) on reverse skyline queries, and thus, we introduce the CRSQ, which is formally defined in Definition 4.1 below.

Definition 4.1 (*Constrained reverse skyline*). Given a D-dimensional data set P, a query point q, and a constrained region CR, a constrained reverse skyline query (CRSQ) returns the reverse skyline inside CR, that is, if a point $p \in P$ is a constrained reverse skyline point of q, there does not exist any other point o ($\neq p$) $\in P$, such that (1) o is within CR; (2) $i \in [1, D]$, $|o[i] - p[i]| \leqslant |q[i] - p[i]|$; and (3) $\exists j \in [1, D]$, $|o[j] - p[j]| < |q[j] - p[j]|$.

As depicted in Fig. 7, points d, e, h, j, m, and r constitute the constrained reverse skyline. Notice that, the traditional reverse skyline does not contain points h and r. Moreover, although point f is a reverse skyline point, it is not a constrained reverse skyline point, since f is not located inside the constrained region. CRSQ has a *large application base*. For instance, it can help the real estate agent to identify the customers whose price preferences are in a given range.

A naive solution to tackle CRSQ is to index the points in a given constrained region in a *single* R-tree, and then, we perform the RSQ based on the constructed R-tree to get the final result. In practice,

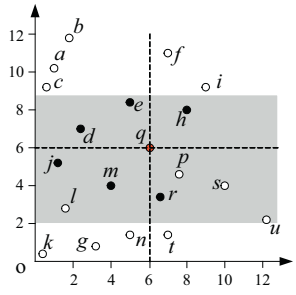


Fig. 7. Example of constrained reverse skyline.

however, the constrained region is not *fixed*, and thus, the R-tree that stores all the points located inside the constrained region has to be *re-constructed repeatedly*, which is *very inefficient*. To address this, we propose three algorithms, namely, *BBS-based constrained reverse skyline* (BCRS) algorithm, *reuse-based constrained reverse skyline* (RCRS) algorithm, and *global-skyline-based constrained reverse skyline* (GCRS) algorithm, for answering the CRSQ. In particular, BCRS, RCRS, and GCRS are extended from BBRS, FRRS, and GSRS, respectively, and they will be presented below.

The proposed algorithms for RSQ, i.e., BBRS, FRRS, and GSRS, can be easily used for dealing with CRSQ by integrating constrained conditions (i.e., *CR*) during the execution of the query. Note that, we do not extend RSSA to tackle CRSQ. This is because the RSSA algorithm utilizes the precomputed dynamic skyline to prune away unqualified candidates. If we extend RSSA to handle CRSQ, we have to the precomputed dynamic skyline within the constrained region *CR*, which is not *fixed*. Consequently, it is not feasible to extend RSSA to answer CRSQ. Similarly, when extending FRRS and GSRS to tackle CRSQ, they cannot employ the dynamic skyline to prune the candidates. In addition, since the final result of CRSQ must satisfy a given region constraint, all the entries not intersecting the constrained region are discarded (i.e., not inserted into the heap). We do not describe them in detail because our proposed algorithms for answering CRSQ are similar as the reverse skyline query processing algorithms.

According to Lemma 3.5, we can easily know that BCRS traverses the R-tree *multiple times*, while both RCRS and GCRS visit the R-tree *only once*. Based on Lemma 3.4, BCRS, RCRS, and GCRS can return *exactly* actual constrained reverse skyline, i.e., every algorithm has *no false negative*, *no false positive*, and the reported result set contains *no duplicate points*.

5. Experimental evaluation

In this section, we experimentally evaluate the effectiveness and efficiency of our proposed algorithms for reverse skyline and constrained reverse skyline queries, using both real and synthetic data sets. Section 5.1 describes the experimental settings. Section 5.2 verifies the performance of the presented RSQ processing algorithms, namely, FRRS and GSRS, by comparing them against RSSA, which, according to the evaluation in Dellis and Seeger (2007), is the *most efficient* existing algorithm for computing the reverse skyline. Section 5.3 reports the experimental results on CRSQ and our findings.

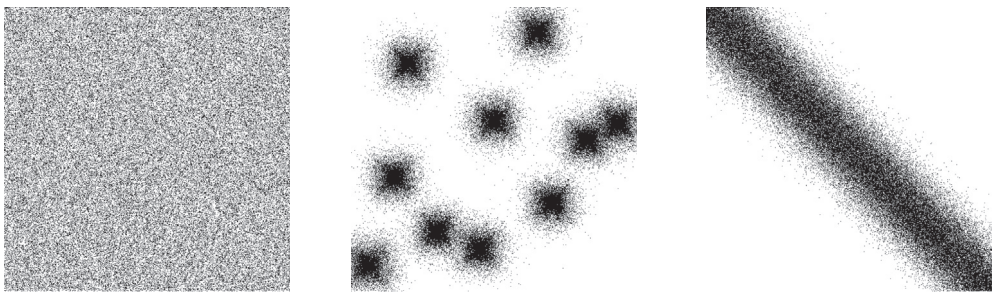
5.1. Experimental setup

We employ two *real* datasets, namely, *CarDB* and *NBA*. Specifically, *CarDB* is a 6D dataset, containing 45,311 tuples, which is extracted from *Yahoo! Autos*. In our experiments, we only select *two* numerical attributes (i.e., *Price* and *Mileage*) of every car. *NBA* includes 15,272 records about 3542 players on 17 attributes, which is available at the website www.databasebasketball.com each record provides the statistics of a player in a season. Four attributes, including *number of games played* (GP), *total points* (PTS), *total rebounds* (REB), and *total assists* (AST), are considered in the experiments. We also create three *synthetic* datasets, i.e., *Independent* (*Ind*), *Clustered* (*Clu*), and *Anti-correlated* (*Aco*), with the dimensionality *dim* in the range [2,5] and the cardinality *N* in the range [40K, 200K]. Specifically, for the *Ind* dataset, all attribute values are generated independently, using a uniform distribution; the *Clu* dataset comprises *ten* randomly centered clusters, each of them follows a Gaussian distribution with the equal number of points; and the *Aco* dataset denotes an environment, where points *good* in one dimension are *bad* in one or all of the other dimension(s). Fig. 8 illustrates the three datasets of different distributions with 100,000 points in 2*D* space. Note that, for all datasets, every dimension of the data space is *normalized* to the range [0,10,000]. Each dataset is indexed by an R-tree (Beckmann et al., 1990), with a page size of 4096 bytes. All the algorithms were implemented in C++, and all the experiments were conducted on a PC with an Intel Core 4 Duo 2.8 GHz PC with 4 GB RAM, running Microsoft Windows XP Professional Edition.

The experiments investigate the performance of the proposed algorithms under a variety of parameters, containing the number *t* of dynamic skyline points, the constrained region *CR*, dimensionality *dim*, and cardinality *N*. It is worth noting that, in each experiment, only one factor varies, whereas the others are fixed to their default values. The settings of the parameters and their default values are listed in Table 2. The *wall clock time* (i.e., the sum of I/O cost and CPU time, where the I/O cost is computed by charging 10 ms for each page access, as with (Lian & Chen, 2010), the *number of node/page accesses* (*NA*), the maximum number of entries in the reuse heap (*MH*), the cardinality of global skyline and global 1-skyline (*CG*), and the cardinality of constrained global skyline and constrained global 1-skyline (*CCG*) are used as the major performance metrics. Each reported value in the following diagrams is the average of 100 queries, whose locations follow the corresponding dataset distribution.

5.2. Results on RSQ

The first set of the experiments verifies the performance of our proposed two new algorithms for reverse skyline queries. First, we study the effect of *t* on FRRS and GSRS, compared with RSSA, using both real and synthetic datasets. Notice that, the setting of *t* for *CarDB* is *different* from other settings. This is because, the number of dynamic skyline points in *CarDB* is *small*, and hence, the corre-



(a) *Ind* (b) *Clu* (c) *Aco*

Fig. 8. Synthetic dataset distribution.

Table 2
Parameter ranges and default values.

Parameter	Range	Default
t (The number t of dynamic skyline points)	0, 10, 30, 50, 70	50
CR (% of the space)	2, 4, 8, 16, 32, 64	32
dim (Dimensionality)	2, 3, 4, 5	3
N (cardinality)	40K, 80K, 120K, 160K, 200K	100 K

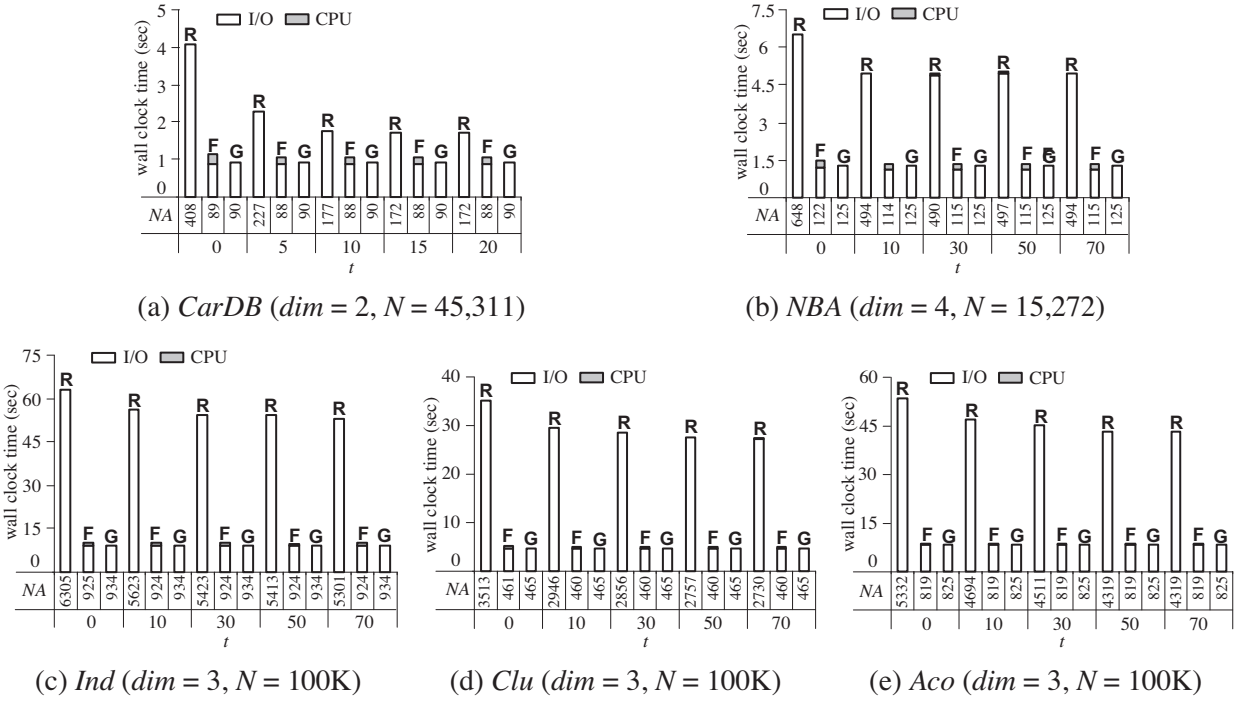
sponding t is set to 0, 5, 10, 15, and 20. The wall clock time (in seconds) of the three algorithms as a function of t, for real and synthetic datasets, is shown in Fig. 9. Here, the wall clock time is broken into *two components*, corresponding to the I/O and CPU costs, respectively. Furthermore, on the top of each bar, we list the *abbreviations* of the algorithms (R for RSSA, F for FRRS, and G for GSRS), and *NA* for every algorithm at the bottom of the bar. It is observed that, as t grows, the wall clock time and *NA* both decrease. The reason is that, the number of dynamic skyline points increases with the growth of t, which helps to prune away *more unqualified* points, and boost the performance accordingly. In addition, it is clear that all the algorithms are I/O bounded. Compared with RSSA and GSRS, the CPU time under FRRS actually occupies a *larger* portion of the wall clock time because, FRRS spend more time on the heap management for the reuse. On the other hand, we can observe that, the *NA* under FRRS and GSRS almost remains *unchanged*. This is because the *NA* of RSSA is mainly caused by the window queries, which visit the R-tree *multiple times*. As both FRRS and GSRS only need to traverse the R-tree *only once*, their *NA* is the *best*. Note that, the *NA* of FRRS is *slightly less* than that of GSRS. The *NA* of FRRS and GSRS consists of *two* parts. The first part is the computation of global skyline, which is the same. For the second part, the *NA* of FRRS comes from the window query, and GSRS comes from the computation of global 1-skyline, which is *more costly* than the former. Therefore, the *NA* of FRRS is slightly less than that of GSRS. However, since the FRRS has to take *more* CPU time in maintaining the reuse heap, the total time of FRRS is *more* than that of GSRS. Overall, both FRRS and GSRS *outperform* RSSA *significantly*, except their advantages under the *CarDB* dataset is not so *obvious*, due to the distribution of *CarDB*.

Table 3 shows the maximum number of entries in the reuse heap (*MH*: the sum of H_g and H_w) of FRRS and the cardinality of global skyline and global 1-skyline (*CG*) of GSRS with respect to t. The *MH* decreases with the growth of t because the increasing dynamic skyline points helps to prune away more unqualified points and thus the nodes visited decrease. However, the *CG* does not change with the variation of t. This is because the global skyline and global 1-skyline are determined only by the given dataset and query point. Hence, t does not influence it. We can find that *MH*, which causes the major run-time memory consumption, is larger than *CG*. It can also illustrate the phenomenon in Fig. 9, confirming that FRRS take *more* CPU time than GSRS.

Next, we explore the impact of dimensionality *dim* on the performance of the algorithms, using synthetic data sets. Towards this, we vary *dim* from 2 to 5, and fix N = 100 K and t = 50. The efficiency of different algorithms under various *dim* is depicted in Fig. 10. Note that, the wall clock time is illustrated in *logarithmic* form. As expected, the performance of all the algorithms degrades with the growth of *dim*. This is because, in a low-dimensional space, each point has a high probability of being dominated by others. Nevertheless, as *dim* increases, more and more points are not dominated by any other point, incurring *more* I/O and CPU costs.

Table 3
The *MH* of FRRS and *CG* of GSRS vs. t.

t	*MH*			*CG*		
	Ind	*Clu*	*Aco*	*Ind*	*Clu*	*Aco*
0	25,764	13,618	27,630	646	366	459
10	17,097	7004	16,822	646	366	459
30	16,405	6608	15,947	646	366	459
50	16,229	6501	15,313	646	366	459
70	16,160	6435	15,149	646	366	459



(a) *CarDB* (dim = 2, N = 45,311)

(b) *NBA* (dim = 4, N = 15,272)

(c) *Ind* (dim = 3, N = 100K)

(d) *Clu* (dim = 3, N = 100K)

(e) *Aco* (dim = 3, N = 100K)

Fig. 9. RSQ cost vs. the number t of dynamic skyline points.

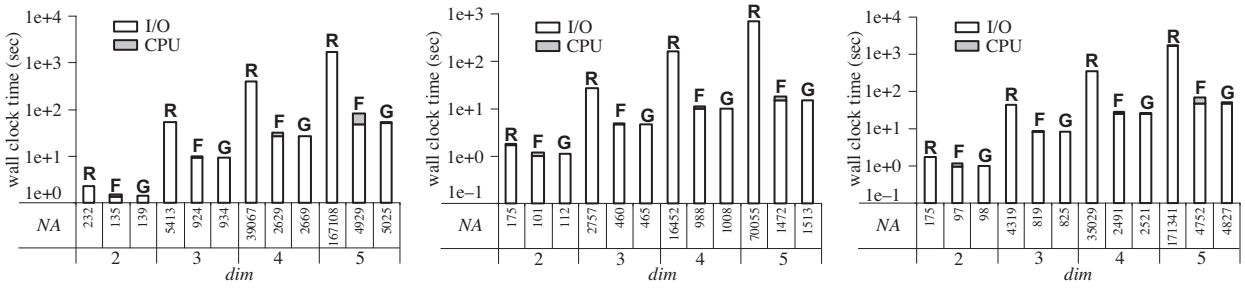


Fig. 10. RSQ cost vs. dimensionality.

Table 4
The *MH* of FRRS and *CG* of GSRS vs. dimensionality.

dim	*MH*			*CG*		
	Ind	*Clu*	*Aco*	*Ind*	*Clu*	*Aco*
2	1348	979	1107	68	61	66
3	16,229	6501	15,313	646	366	459
4	43,835	14,395	51,877	3008	1116	2134
5	77,898	20,398	88,818	9307	2582	7383

Table 5
The *MH* of FRRS and *CG* of GSRS vs. cardinality.

N (K)	*MH*			*CG*		
	Ind	*Clu*	*Aco*	*Ind*	*Clu*	*Aco*
40	8689	3199	7352	515	280	364
80	13,705	5265	13184	607	336	436
120	18,904	7232	16839	669	378	483
160	22,381	9415	21327	709	406	516
200	25,281	10346	24252	739	430	548

Moreover, the *poor* performance of the R-tree in high dimensions (Papadopoulos & Manolopoulos, 1997) leads to this degradation. However, FRRS and GSRS still exceed RSSA.

Table 4 lists the *MH* of FRRS and *CG* of GSRS with respect to *dim*. With the growth of *dim*, both *MH* and *CG* increase. This is because in high dimensionality, the point is more likely to be dominated, and therefore, the global skyline and global 1-skyline become larger. On the other hand, reverse skyline also will increase in high dimensionality, which result in larger *MH*. Again, the size of *CG* is smaller than that of *MH*.

Finally, we evaluate the effect of cardinality *N* on the efficiency of the algorithms, with $t = 50$, $dim = 3$, and *N* varying between 40K and 200K. The experimental results are plotted in Fig. 11. Evidently, the cost of algorithms grows as *N* ascends. The reason is that, the size of the R-tree increases with *N*, which forces the algorithms to access more entries when computing the reverse skyline. Again, GSRS performs the best and RSSA is the worst.

Table 5 presents the *MH* of FRRS and *CG* of GSRS with respect to *N*. As expected, the size of *MH* and *CG* grow when *N* increases. The reason behind is that, the bigger dataset has a bigger global skyline, global 1-skyline, and reverse skyline, which lead to the bigger *MH* and *CG*. Also, *MH* is larger than *CG*. As shown in Table 5, the maximal heap contents, which are stored in main memory, increase when *N* grows. According to the tendency, if *N* ascends continually, the main memory may not keep all the heap contents, which will limit the performance of FRRS.

In summary, GSRS performs the *best* among all the algorithms, and it is usually 2–3 *times faster* than RSSA in most cases. FRRS is *worse* than GSRS. Anyway, all our proposed algorithms outperform RSSA significantly in all cases. As for FRRS, it is more suitable for the small cardinality and low dimensionality.

5.3. Results on CRSQ

In the second set of the experiments, we examine the effectiveness and efficiency of our proposed BCRS, RCRS, and GCRS algorithms for supporting constrained reverse skyline queries. When the given constrained region *CR* changes, the dynamic skyline w.r.t. *CR* varies as well. Consequently, our developed algorithms do not rely on the dynamic skyline for pruning, and thus the parameter *t* is not considered in this set of the experiments. For all the diagrams shown below, the letters on the top of the bars are the abbreviations of the algorithms, i.e., BC for BCRS, RC for RCRS, and GC for GCRS.

First, we inspect the influence of *CR* on the performance of the algorithms, using real and synthetic datasets. The wall clock time and *NA* of the algorithms with respect to *CR* under different datasets are depicted in Fig. 12. Notice that, *CR* for synthetic datasets varies from 2% to 64% (of the data space), whereas that under real datasets changes from 60% to 100% because, the distribution of the real datasets differs from that of the synthetic datasets. It is

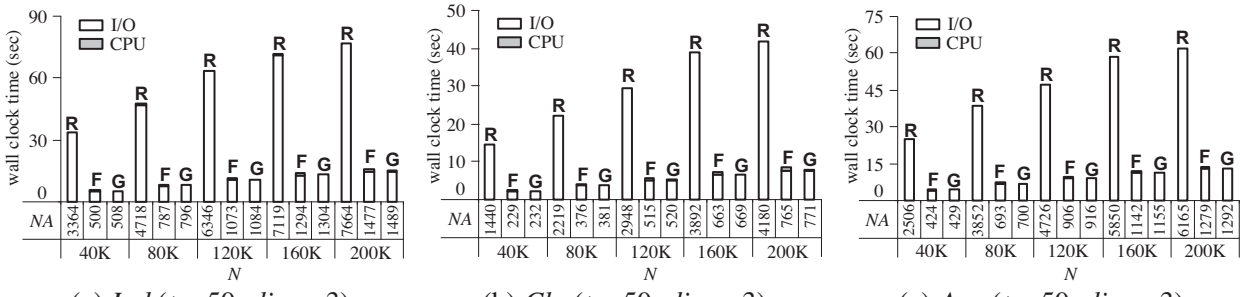


Fig. 11. RSQ cost vs. cardinality.

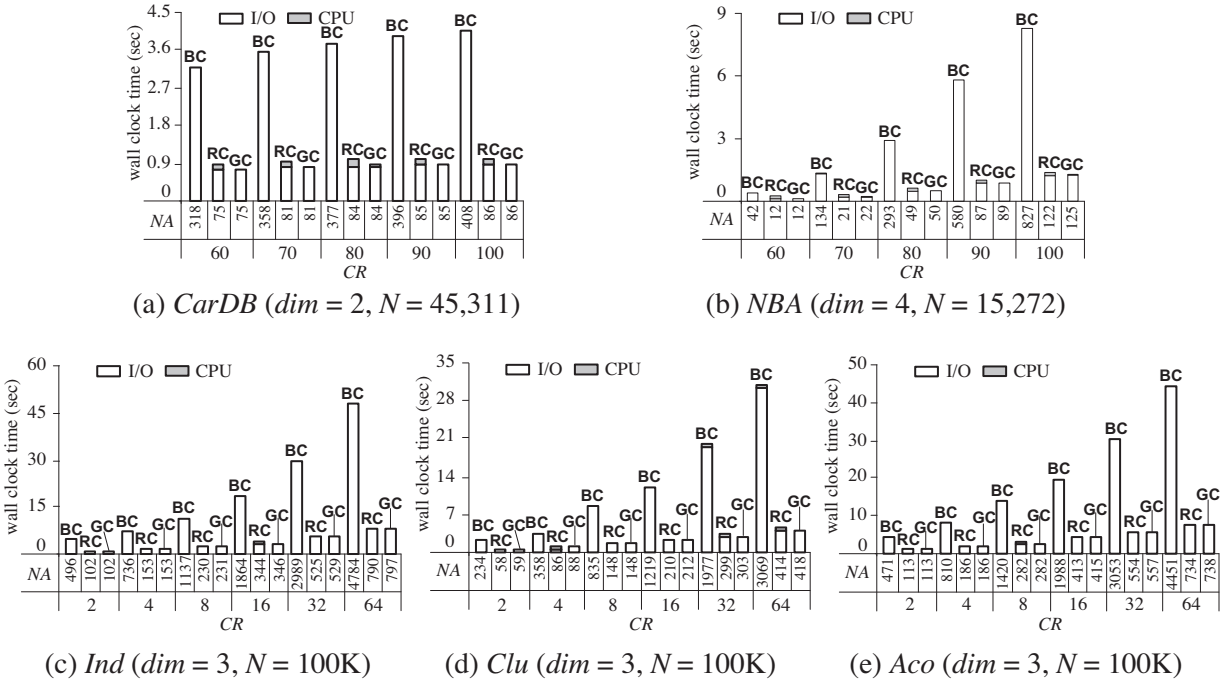


Fig. 12. CRSQ cost vs. constrained region.

Table 6
The *MH* of RCRS and *CCG* of GCRS vs. *CR*.

CR	*MH*			*CCG*		
	Ind	*Clu*	*Aco*	*Ind*	*Clu*	*Aco*
2	1212	843	971	95	47	70
4	2000	1274	2115	124	75	104
8	3318	2732	2955	169	132	159
16	5630	4302	5077	241	192	196
32	9613	6377	9614	348	250	293
64	16,656	9873	17280	495	318	379

Table 7
The *MH* of RCRS and *CCG* of GCRS vs. dimensionality.

dim	*MH*			*CCG*		
	Ind	*Clu*	*Aco*	*Ind*	*Clu*	*Aco*
2	2228	1994	2051	95	45	53
3	9613	6377	9614	348	250	293
4	21,028	9794	12,633	1438	673	1223
5	23,272	15,448	11,695	4014	1611	3874

obvious that, all the algorithms are I/O bounded, and both the wall clock time and *NA* grow with *CR*. The reason is that, as the constrained region expands, it contains more points, and hence, more points need to be checked, which results in higher I/O and CPU overheads. Moreover, RCRS and GCRS are significantly faster than BCRS. This is because, both RCRS and GCRS traverse the R-tree *only once*, while BCRS traverses it *repeatedly*.

Table 6 lists the maximum number of entries in the heap (*MH*) of RCRS and the cardinality of constrained global skyline and constrained global 1-skyline (*CCG*) of GCRS with respect to *CR*. With the expanding of *CR*, *MH* and *CCG* also increase. This is because when the constrained region becomes larger, more point will fall in it. Therefore, more points may become the constrained global skyline, constrained global 1-skyline, and constrained reverse skyline, and the nodes visited ascend as well.

Then, we evaluate the effect of dimensionality *dim* on the efficiency of the algorithms. Fig. 13 shows the performance of the algorithms in terms of the wall clock time (by the logarithmic form) and *NA*. As expected, the larger the *dim*, the higher the cost. This tendency can be explained by the following two factors. First, the points in high dimensions are more likely to be not dominated by others. Second, the R-tree is *inefficient* in high dimensions. Even

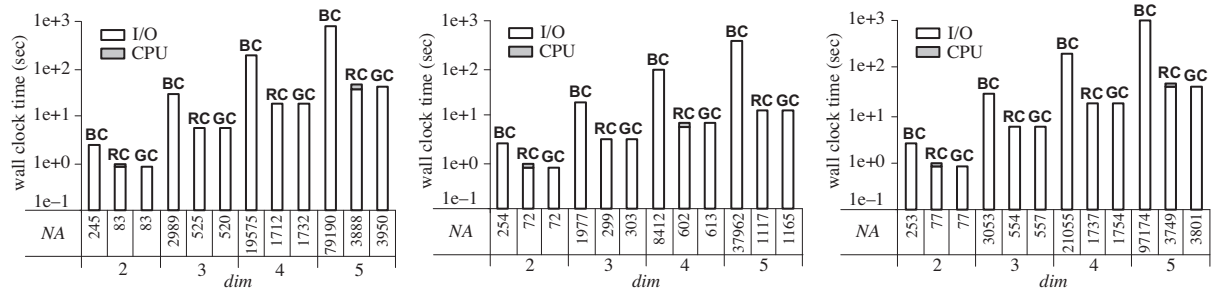
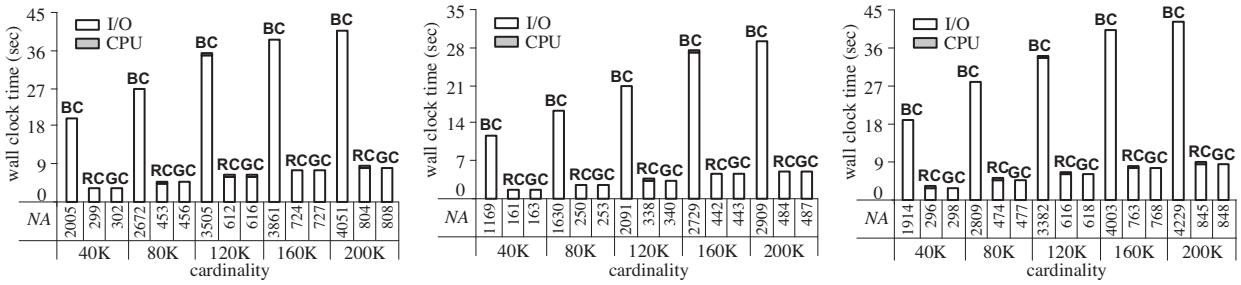


Fig. 13. CRSQ cost vs. dimensionality.



(a) *Ind* (*CR*=32%, *dim*=3) (b) *Clu* (*CR*=32%, *dim*=3) (c) *Aco* (*CR*=32%, *dim*=3)

Fig. 14. CRSQ cost vs. cardinality.

Table 8
The *MH* of RCRS and *CCG* of GCRS vs. cardinality.

N (K)	*MH*			*CCG*		
	Ind	*Clu*	*Aco*	*Ind*	*Clu*	*Aco*
40	5332	3299	4498	276	195	235
80	8296	5304	8183	325	230	269
120	11,311	7232	10,507	365	255	296
160	13,471	9476	12,802	388	275	319
200	15,363	10,422	14,912	401	288	338

though these, RCRS and GCRS are still over BCRS. Table 7 displays *MH* of RCRS and *CCG* of GCRS with respect to *dim*. Again, both *MH* and *CCG* increase. The explanations are the same as those in Table 4, and thus omitted.

Finally, we investigate the effect of cardinality *N* on the performance of the algorithms, and report the experimental results in Fig. 14. Observe that, the cost of algorithms grows as *N* increases. The reason is that, more points falling into the specified constrained region as *N* ascends, which results in more points to be examined and hence longer wall clock time and higher *NA*. Table 8 shows *MH* of RCRS and *CCG* of GCRS with respect to *N*. The phenomena and their explanations are the same as those in Table 5, and hence omitted.

To sum up, GCRS performs the best, follows by RCRS, and BCRS is the worst. Moreover, both GCRS and PCRS are several orders of magnitude faster than BCRS.

6. Conclusions

In this paper, we propose several efficient algorithms to answer the reverse skyline query (RSQ), by using the *precomputation*, *reuse*, and *pruning* techniques. In addition, we study a *new* form of RSQ, namely, *constrained* RSQ (CRSQ), and extend our methods to tackle it efficiently. Extensive experiments with both real and synthetic datasets demonstrate that our proposed RSQ algorithms achieves *several orders of magnitude* performance gain over the *state-of-the-art* RSSA algorithm under all experimental settings, and our presented CRSQ algorithms can efficiently compute the constrained reverse skyline. In the future, we intend to further extend our approaches to handle other variants of RSQ, such as *ranked* reverse skyline and *group-by* reverse skyline queries. Also, we plan to explore RSQ and its variations in *metric spaces*. In addition, we are interested in further improving the performance of reverse skyline using cash.

Acknowledgments

Yunjun Gao was supported in part by NSFC Grants 61379033 and 61003049, the Fundamental Research Funds for the Central Universities under Grants 2012QNA5018 and 2013QNA5020, and the Key Project of Zhejiang University Excellent Young Teacher Fund (Zijin Plan).

References

Bartolini, I., Ciaccia, P., & Patella, M. (2008). Efficient sort-based skyline evaluation. *ACM Transactions on Database Systems, 33*(4), 1–45.
Beckmann, N., Kriegel, H. P., Schneider, R., & Seeger, B. (1990). The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD international conference on management of data* (pp. 322–331).
Borzsony, S., Kossmann, D., & Stocker, K. (2001). The skyline operator. In *Proceedings of the international conference on data engineering* (pp. 421–430).
Chen, L., Cui, B., & Lu, H. (2011). Constrained skyline query processing against distributed data sites. *IEEE Transactions on Knowledge and Data Engineering, 23*(2), 204–217.
Chen, L., & Lian, X. (2009). Efficient processing of metric skyline queries. *IEEE Transactions on Knowledge and Data Engineering, 21*(3), 351–365.
Chomicki, J., Godfrey, P., Gryz, J., & Liang, D. (2003). Skyline with presorting. In *Proceedings of the international conference on data engineering* (pp. 717–719).
Dellis, E., & Seeger, B. (2007). Efficient computation of reverse skyline queries. In *Proceedings of the international conference on very large data base* (pp. 291–302).
Dellis, E., Vlachou, A., Vladimirskiy, I., Seeger, B., & Theodoridis, Y. (2006). Constrained subspace skyline computation. In *Proceedings of the ACM international conference on information and knowledge management* (pp. 415–424).
Fuhry, D., Jin, R., & Zhang, D. (2009). Efficient skyline computation in metric space. In *Proceedings of the international conference on extending database technology* (pp. 1042–1051).
Gao, Y., Chen, G., Chen, L., & Chen, C. (2006). Parallelizing progressive computation for skyline queries in multi-disk environment. In *Proceedings of the international conference on database and expert systems applications* (pp. 697–706).
Godfrey, P., Shipley, R., & Gryz, J. (2005). Maximal vector computation in large data sets. In *Proceedings of the international conference on very large data base* (pp. 229–240).
Hose, K., & Vlachou, A. (2012). A survey of skyline processing in highly distributed environments. *VLDB Journal, 21*(3), 359–384.
Huang, Z., Xiang, Y., Zhang, B., & Liu, X. (2011). A clustering based approach for skyline diversity. *Expert Systems with Applications, 38*, 7984–7993.
Islam, M., Zhou, R., & Liu, C. (2013). On answering why-not questions in reverse skyline queries. In *Proceedings of the international conference on data engineering* (pp. 973–984).
Kohler, H., Yang, J., & Zhou, X. (2011). Efficient parallel skyline processing using hyperplane projections. In *Proceedings of the ACM SIGMOD international conference on management of data* (pp. 85–96).
Kossmann, D., Ramsak, F., & Rost, S. (2002). Shooting stars in the sky: An online algorithm for skyline queries. In *Proceedings of the international conference on very large data base* (pp. 275–286).
Lee, K. C. K., Zheng, B., Li, H., & Lee, W.-C. (2007). Approaching the skyline in *z* order. In *Proceedings of the international conference on very large data base* (pp. 279–290).
Lian, X., & Chen, L. (2010). Reverse skyline search in uncertain databases. *ACM Transactions on Database Systems, 35*(1), 3.
Lin, X., Yuan, Y., Zhang, Q., & Zhang, Y. (2007). Selecting stars: The k most representative skyline operator. In *Proceedings of the international conference on data engineering* (pp. 86–95).
Lin, X., Zhang, Y., Zhang, W., & Cheema, M. A. (2011). Stochastic skyline operator. In *Proceedings of the international conference on data engineering* (pp. 721–732).
Liu, Q., Gao, Y., Chen, G., Li, Q., & Jiang, T. (2012). On efficient reverse k-skyband query processing. In *Proceedings of the international conference on database systems for advanced applications* (pp. 544–559).
Papadias, D., Tao, Y., Fu, G., & Seeger, B. (2005). Progressive skyline computation in database systems. *ACM Transactions on Database Systems, 30*(1), 41–82.

Papadopoulos, A., & Manolopoulos, Y. (1997). Performance of nearest neighbor queries in R-trees. In *Proceedings of the international conference on database theory* (pp. 394–408).

Pei, J., Jiang, B., Lin, X., & Yuan, Y. (2007). Probabilistic skylines on uncertain data. In *Proceedings of the international conference on very large data base* (pp. 15–26).

Pei, J., Yuan, Y., Lin, X., Jin, W., Ester, M., & Liu, Q. (2006). Towards multidimensional subspace skyline analysis. *ACM Transactions on Database Systems, 31*(4), 1335–1381.

Prasad, M. D., & Deepak, P. (2011). Efficient reverse skyline retrieval with arbitrary non-metric similarity measures. In *Proceedings of the international conference on extending database technology* (pp. 319–330).

Sacharidis, D., Bouros, P., & Sellis, T. (2008). Caching dynamic skyline queries. In *Proceedings of the international conference on scientific and statistical database management* (pp. 455–472).

Tan, K.-L., Eng, P.-K., & Ooi, B. C. (2001). Efficient progressive skyline computation. In *Proceedings of the international conference on very large data base* (pp. 301–310).

Tao, Y., Ding, L., Lin, X., & Pei, J. (2009). Distance-based representative skyline. In *Proceedings of the international conference on data engineering* (pp. 892–903).

Tao, Y., Xiao, X., & Pei, J. (2011). Efficient skyline and top-*k* retrieval in subspaces. *IEEE Transactions on Knowledge and Data Engineering, 23*(2), 204–217.

Vlachou, A., Doulkeridis, C., & Kotidis, Y. (2008). Angle-based space partitioning for efficient parallel skyline computation. In *Proceedings of the ACM SIGMOD international conference on management of data* (pp. 227–238).

Wang, G., Xin, J., Chen, L., & Liu, Y. (2012). Energy-efficient reverse skyline queries processing over wireless sensor networks. *IEEE Transactions on Knowledge and Data Engineering, 24*(7), 1259–1275.

Wu, P., Zhang, C., Feng, Y., Zhao, B. Y., Agrawal, D., & Abbadi, A. E. (2006). Parallelizing progressive skyline queries for scalable distribution. In *Proceedings of the international conference on extending database technology* (pp. 112–130).

Wu, X., Tao, Y., Wong, R. C.-W., Ding, L., & Yu, J. X. (2009). Finding the influence set through skylines. In *Proceedings of the international conference on extending database technology* (pp. 1030–1041).

Zhang, W., Lin, X., Zhang, Y., Wang, W., & Yu, J. X. (2009). Probabilistic skyline operator over sliding windows. In *Proceedings of the international conference on data engineering* (pp. 1060–1071).

Zhang, S., Mamoulis, N., & Cheung, D. W. (2009). Scalable skyline computation using object-based space partitioning. In *Proceedings of the ACM SIGMOD international conference on management of data* (pp. 483–494).

Zhu, L., Li, C., & Chen, H. (2009). Efficient computation of reverse skyline on data stream. In *International joint conference on computational sciences and optimization* (pp. 735–739).