Efficient Computation of Reverse Skyline Queries

Evangelos Dellis
Department of Computer Science
University of Marburg
Marburg, Germany
dellis@mathematik.uni-marburg.de

Bernhard Seeger
Department of Computer Science
University of Marburg
Marburg, Germany
seeger@mathematik.uni-marburg.de

ABSTRACT

In this paper, for the first time, we introduce the concept of *Reverse Skyline Queries*. At first, we consider for a multidimensional data set P the problem of dynamic skyline queries according to a query point q. This kind of dynamic skyline corresponds to the skyline of a transformed data space where point q becomes the origin and all points of P are represented by their distance vector to q. The reverse skyline query returns the objects whose dynamic skyline contains the query object q. In order to compute the reverse skyline of an arbitrary query point, we first propose a Branch and Bound algorithm (called BBRS), which is an improved customization of the original BBS algorithm. Furthermore, we identify a super set of the reverse skyline that is used to bound the search space while computing the reverse skyline. To further reduce the computational cost of determining if a point belongs to the reverse skyline, we propose an enhanced algorithm (called RSSA) that is based on accurate pre-computed approximations of the skylines. These approximations are used to identify whether a point belongs to the reverse skyline or not. Through extensive experiments with both real-world and synthetic datasets, we show that our algorithms can efficiently support reverse skyline queries. Our enhanced approach improves reversed skyline processing by up to an order of magnitude compared to the algorithm without the usage of pre-computed approximations.

1. INTRODUCTION

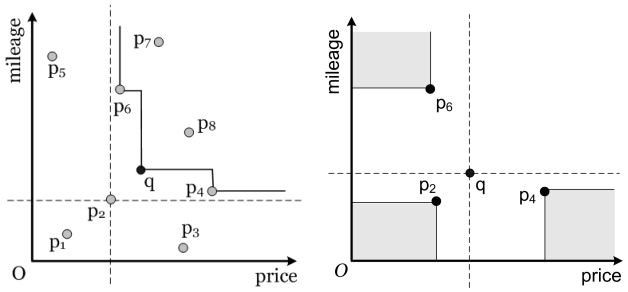
Given a set P of d-dimensional points, the skyline operator returns all points in P that are not dominated by another point. A point p_i *dominates* another point p_j if the coordinate of p_i in each dimension is not greater than that of p_j, and strictly smaller in at least one dimension. The set of skyline points presents a scale-free choice of data points worthy for further considerations in many application contexts. Informally, the skyline of a multidimensional data set contains the "best" tuples according to any preference function that is monotone in each dimension.

Skyline queries [3] are a specific and relevant example of preference queries [6], [12] and have been recognized as a useful and practical way to make database systems more flexible in supporting user requirements [5]. The skyline queries can be either absolute or relative, where 'absolute' means that minimization is based on the static attribute values of data points in P, while 'relative' means that the minimization of the coordinate-wise distances between the data points and a user-given query point is taken into account. Consider for example a used car database with a table CarDB in which the car instances are stored as tuples with attributes Make, Model, Year, Price and Mileage. Obviously, the ideal case is to give the user the opportunity to specify a car of his choice and to request the skyline according to this car. The skyline query becomes a dynamic [16] or relative [9] skyline query, where the domination is defined with respect to the users query point. The skyline is then computed with respect to a new transformed data space where the query point becomes the origin and all points are represented by their coordinate-wise distances to the query point. The dynamic skyline contains all those (transformed) points that are not dominated by any other point with respect to the distances to the query point.

While there are many papers on skyline query processing from the user perspective (selecting the products they like), in this paper, we focus on the companies perspective. To explain, assume that the preferences of users about cars are stored as tuples in a relation with the same attributes as in CarDB. Figure 1 shows the preferences of users as points in a two-dimensional space, considering only attributes *Price* and *Mileage*. A car dealer, who wants to determine the effectiveness of a particular car advertisement in the market, specifies a hypothetical car as a query point, say q. The dealer is interested in which customers find this car interesting, i.e., have this hypothetical car as part of their two-dimensional dynamic skyline. In Figure 1(a), a user interested in cars of type p_2, would have been interested in the hypothetical car q as q is in the dynamic skyline of p_2. The same holds for preferences p_4 and p_6, both are in the reverse skyline of q as the dynamic skylines of p_4 and p_6 contains q. Based on the result of the reversed skyline, the car dealer might offer car q to customers with preferences p_2, p_4 or p_6. If the result set is too small, the dealer might change the price of the hypothetical car to increase the number of reveresed skyline points.

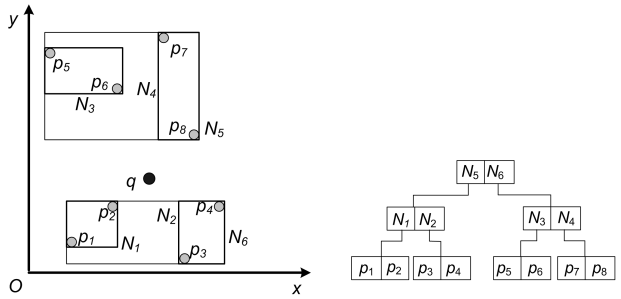
The query retrieving the set of most preferred cars in our motivating example belongs to a broader novel type of *Re-*

VLDB '07, September 23-28, 2007, Vienna, Austria.



(a) Dynamic Skyline (b) Reverse Skyline

Figure 1: Example of Reverse Skyline



(a) Planar Representation (b) R-tree Nodes

Figure 2: Illustration of the BBS Algorithm.

verse Skyline Queries (RSQ). The goal of a reverse skyline query is to identify the influence of a query object on a multidimensional dataset with respect to a vector of distances. This is in contrast to the known problem of reverse nearest neighbors [13] where a single distance function is applied to the multidimensional data. This is however too restrictive for many applications that deal with multidimensional data as in our example above. In general, there is no meaningful and generally applicable distance function that maps a multidimensional point into a single value. Instead, we examine the 'influence' of a data point with respect to dynamically dominated points. Intuitively, given a point q, the points which have q as a member of their dynamic skyline reflect the notion of 'influence'. We call these points the reverse skyline of q.

Besides preference-based marketing, RSQ is highly relevant for many other applications. As another example, let us consider *business location planning* of a new store. Assuming several possible choices for the new store location, the strategy is to select the location that maximize the number of customers. A reverse skyline query posed on a customer database would return those customers who are potentially interested in the new store.

The reverse skyline computation over multidimensional datasets has several technical challenges. We approach the problem of processing RSQ by analytically exploiting the geometric properties of the problem and solution spaces. Based on the concept of the *global skyline* of a point, which is also introduced in this paper, we are able to retrieve a small subset of the database as candidates for RSQ. We prove that all reverse skyline points with respect to q, belong to the global skyline. Outlining the major issues that are addressed in this paper, our main contributions are as follows:

- We address the problem of *Reverse Skyline Queries* (RSQ) and we propose two algorithms for computing RSQ. The Branch and Bound Reverse Skyline (BBRS) algorithm and the enhanced (RSSA) algorithm that employs pre-computed approximations of the skylines.
- We present an optimal algorithm for computing approximations for 2-dimensional skylines and we provide a greedy strategy for higher dimensions.
- Through extensive experiments with both real-world and synthetic datasets, we show that our algorithms can efficiently support reverse skyline queries.

The rest of the paper is organized as follows. Section 2 reviews the previous work related to skyline query processing and reverse NN search. Section 3 presents the problem definition, and Section 4 proposes an efficient Branch and Bound Reverse Skyline algorithm. In Section 5, the RSSA algorithm, which is based on pre-computed skyline approximations, is proposed. Section 6 deals with the problem of computing accurate approximations. Section 7 contains an extensive experimental evaluation that demonstrates the efficiency of the proposed algorithms. Finally, Section 8 concludes the paper with directions for future work.

2. RELATED WORK

Although the proposed algorithms can be used with various indexes, we assume that the dataset is indexed by an R-tree due to the popularity of this structure. Section 2.1 briefly overviews skyline query processing using the R-tree and algorithms for dynamic (or relative) skyline computation. Section 2.2 describes previous work on (monochromatic) reverse nearest neighbor search. Note that the problem of reverse skyline has not been addressed so far.

2.1 Skyline Query Processing

Skyline query processing has received considerable attention in multidimensional databases and has been studied extensively in recent years [17], [24] [22], [7]. Since the introduction of the skyline operator [3], several efficient algorithms have been proposed for the 'general' skyline query [16]. The best known algorithm that can answer *Dynamic Skyline Queries* (DSQ) is the *Branch-and-Bound Skyline* (BBS) algorithm [16]. BBS is a progressive optimal algorithm for the conventional skyline query. In the setting of BBS, a dynamic skyline query specifies a new n-dimensional space based on the original d-dimensional data space. First, each point p in the database is mapped to the point $p' = (f_1(p), ..., f_n(p))$ where each f_i is a function of the coordinates of p. Then, the query returns the general (i.e., static) skyline of the new space (the corresponding points in the original space).

Figure 2 shows the R-tree for the dataset of Figure 1, together with the minimum bounding rectangles (MBR) of the nodes. In order to process the dynamic skyline query according to the query point q (shown in Figure 2), BBS processes the (leaf/intermediate) entries in ascending order of their *mindist* (which is computed on-the-fly in the dynamic space when the entry is considered for the first time) to the query reference point. At the beginning, the root entries are

inserted into a heap H ($= \{N6, N5\}$) using their *mindist* as the sorting key. Then, the algorithm removes the top $N6$ of H, accesses its child node, and enheaps all the entries there (H now becomes $\{N5, N2, N1\}$). Similarly, the next node visited is $N5$, where H ($= \{N4, N2, N1, N3\}$). The next node visited is leaf $N4$, where the data points are also added to H ($= \{p_8, N2, N1, N3, p_7\}$). Since p_8 tops H, it is taken as the first dynamic skyline point, and used for pruning in the subsequent execution. The algorithm proceeds in the same manner until the heap becomes empty.

In analogy to skyline query processing for only one query reference point (absolute or relative), some recent studies considers several query points at the same time. Given a set of data points P and a set of query points Q, a *Spatial Skyline Query* (SSQ) [18] retrieves those points of P which are not dominated by any other point in P considering their derived spatial attributes. The main difference with the regular skyline query is that this spatial domination depends on the location of the query points Q. The spatial skyline query can be defined as a special case of the dynamic skyline query [16]. The authors in [18] proposed two algorithms for the spatial skyline query, the R-tree-based B^2S^2 and the Voronoi-based VS^2. The spatial skyline query is known also as *Multi-Source Skyline Query* (MSSQ) [9]. The authors in [9] extends multi-source skyline queries to road networks where the network distance between two locations needs to be computed on-the-fly. Several algorithms for processing MSSQs are proposed and evaluated in this paper.

Li et. al. in [14] define a novel data cube called *Data Cube for Dominant Relationship Analysis* (DADA), which captures the dominant relationships between products and customers, to help firms delineate market opportunities based on customer preferences and competitive products. The concept of dominance is extended for business analysis from a microeconomic perspective. More specifically, a new form of analysis is proposed, called *Dominant Relationship Analysis* (DRA), which aims to provide insight into the dominant relationships between products and potential buyers. Three new classes of skyline queries called *Dominant Relationship Queries* (DRQs) are consequently proposed for analysis purposes. These types of queries are: (1) *Linear Optimization Queries* (LOQ), (2) *Subspace Analysis Queries* (SAQ), and (3) *Comparative Dominant Queries* (CDQ). Efficient computation for such queries is achieved through a novel data structure.

2.2 Reverse NN Search

Given a point q, a reverse nearest neighbor (RNN) query retrieves all the data points that have q as one of their nearest neighbors. Besides this monochromatic version, there exist also the bichromatic RNN [20] where, given a set Q of queries, the goal is to find the objects $p \in P$ that are closer to some $q \in Q$ than any other point of Q. The algorithms for RNN processing can be classified in two categories depending on whether they require preprocessing, or not: the hypersphere-approaches and the Voronoi-approaches.

The original RNN method [13] pre-computes for each data point p its nearest neighbor NN(p). Then, it represents p as a vicinity circle $(p, dist(p, NN(p)))$ centered at p with radius equal to the Euclidean distance between p and its NN. The MBRs of all circles are indexed by an R-tree, called the RNN-tree. Using the RNN-tree, the reverse nearest neighbors of q can be efficiently retrieved by a point location query, which returns all circles that contain q. Because the RNN-tree is optimized for RNN, but not NN search, the authors in [13] propose to use two trees: (1) a traditional R-tree-like structure for nearest neighbor search (called NN-tree) and (2) the RNN-tree for reverse nearest neighbor search. In order to avoid the maintenance of two separate structures, Yang and Lin [23] combine the two indexes in the RdNN-tree. Similar to the RNN-tree, a leaf node of the RdNN-tree contains vicinity circles of data points. On the other hand, an intermediate node contains the MBR of the underlying points (not their vicinity circles), together with the maximum distance from every point in the sub-tree to its nearest neighbor. These techniques extend a multidimensional index structure to store each object along with its nearest neighbor distance and, thus, actually store hyper spheres rather than points.

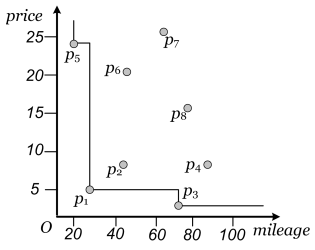
Stanoi et al. [20] eliminate the need for pre-computing all NNs by utilizing geometric properties of RNN retrieval. Tao et. al. [21] propose a method which utilize a conventional data-partitioning index (e.g. R-tree) on the dataset and do not require any pre-computation. Their method is designed for exact processing of RkNN queries with arbitrary values of k on dynamic multidimensional datasets. Their framework follows a filter and refinement strategy. Specifically, the filter step retrieves a set of candidate results that is guaranteed to include all the actual reverse nearest neighbors; the subsequent refinement step eliminates the false hits. The two steps are integrated in a seamless way that eliminates multiple accesses to the same index node (i.e., each node is visited at most once). Basically, this method store the objects in conventional multidimensional index structures without any extension and compute a Voronoi cell during query processing.

Achtert et. al. [1] propose the first approach for efficient RkNN search in arbitrary *metric* spaces where the value of k is specified at query time. Their approach uses the advantages of existing metric index structures as well as conservative and progressive distance approximations to filter out true drops and true hits. In particular, they approximate the k-nearest neighbor distance for each data object by upper and lower bounds using two functions (of two parameters each). Their solution is based on the hypersphere approach for the RkNN problem with an arbitrary k not exceeding a given threshold parameter k_{max} for general metric objects.

3. PROBLEM DEFINITION

Let $D = (D^1, \ldots, D^d)$ be a d-dimensional data space and $P \subseteq D$ be a data set. A point $p \in P$ can be represented as $p = (p^1, p^2, \ldots, p^d)$ with $p^i \in D^i$, $i \in \{1, \ldots, d\}$. A point $p \in P$ is said to dominate another point $q \in P$, denoted as $p \prec q$, if (1) for every $i \in \{1, \ldots, d\} : p^i \leq q^i$; and (2) for at least one $j \in \{1, \ldots, d\} : p^j < q^j$. The skyline of P is a set of points $SL \subseteq P$ which are not dominated by any other point. That is, $SL = \{p \in P | \nexists q \in P : q \prec p\}$. The points in SL are called skyline points of P. Figure 3 illustrates a database of eight objects $P = \{p_1, p_2, ..., p_8\}$ each representing a car with two attributes *Price* and *Mileage*. Figure 3(a) shows the corresponding points in the 2-dimensional space, where x and y axes correspond to the attributes *Mileage* and *Price*, respectively. In Figure 3(a), the point p_1 dominates the point p_2. Overall, the skyline is the set $SL = \{p_5, p_1, p_3\}$.

In the remaining section, we first define the dynamic sky-



(a) Points in a 2-d Space

ID	Price	Mileage
p_1	5,000	30K
p_2	7,500	42K
p_3	2,500	70K
p_4	7,500	90K
p_5	24,000	20K
p_6	20,000	50K
p_7	26,000	70K
p_8	16,000	80K

(b) Database Table

Figure 3: A Database Example.

line query using the above notation. Then we introduce our reverse skyline query based on the definition of the dynamic skyline query.

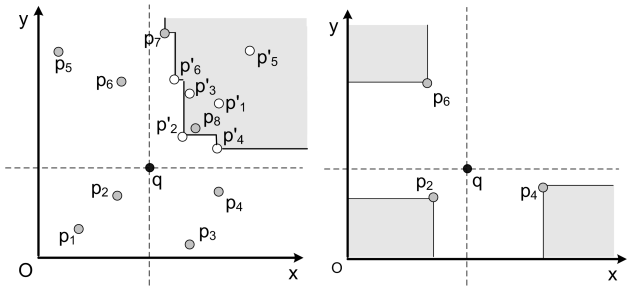
3.1 Dynamic Skyline Query

The general dynamic skyline specifies a new d'-dimensional space based on the original d-dimensional data space. First, each point $p \in P$ is mapped to a point $p' = (f^1(p),\ldots, f^{d'}(p))$ where each f_i is a one-dimensional function. Then, the dynamic skyline of P with respect to functions $f_1, \ldots, f_{d'}$ returns the ordinary skyline of the transformed d'-dimensional space derived from the data set P'. For sake of simplicity, we assume in the following that $d' = d$ and that for a given query point q $f^i(p) = |q^i - p^i|$, i.e., f simply refers to the absolute distance to the query point q in the i-th dimension. Note that the following results still hold for a more general class of distance functions. As an example, let us mention that the left and right part of the absolute distance function can receive different weights. This is important in various applications. For example, a car with 1 liter higher fuel consumption than specified in the user preference might be less preferable than a car with 1 liter less.

Definition 1. (*Dynamic Skyline Query*)
Given a query point q and a data set P, a Dynamic Skyline Query (DSQ) according to q retrieves all data points in P that are not dynamically dominated. A point $p_1 \in P$ dynamically dominates $p_2 \in P$ with regard to the query point q if (1) for all $i \in \{1,\ldots,d\}$: $|q^i - p_1^i| \leq |q^i - p_2^i|$, and (2) at least one $j \in \{1,\ldots,d\}$: $|q^j - p_1^j| < |q^j - p_2^j|$.

In the above definition, it is equivalent to compute the traditional skyline, having transformed all points in the new data space where point q is the origin and the absolute distances to q are used as mapping functions. Consider, for instance, Figure 4(a). A user specifies a preference q for a car to request the most interesting cars with respect to the absolute distances to q where attributes Price and Mileage is taken into account. Each point $p = (p^1, p^2)$ in the original 2-dimensional space is transformed to a point $p' = (|p^1 - q^1|, |p^2 - q^2|)$ in the 2-dimensional distance space. This figure illustrates that each database point is mapped to the new distance space where q becomes the origin. The dynamic skyline consists of points p_7, p_6, p_2 and p_4. Note that point p_8 is not part of the dynamic skyline because it is dynamically dominated by point p_2.

The terms *original space* and *transformed space* refer to the original d-dimensional data space, while the transformed space refers to the data obtained from the d mappings f^1, f^2,..., f^d.



(a) Dynamic Skyline of q (b) Reverse Skyline of q

Figure 4: Dynamic and Reverse Skyline.

3.2 Reverse Skyline Query

Based on the definition of dynamic skyline we now formally define the reverse skyline of a point.

Definition 2. (*Reverse Skyline Query*)
Let P be a d-dimensional data set. A Reverse Skyline Query (RSQ) according to the query point q retrieves all points $p_1 \in P$ where q is in the dynamic skyline of p_1. Formally, a point $p_1 \in P$ is a reverse skyline point of $q \in P$ iff $\nexists p_2 \in P$ such that (a) for all $i \in \{1,\ldots,d\}$: $|p_2^i - p_1^i| \leq |q^i - p_1^i|$ and (b) for at least one $j \in \{1,\ldots,d\}$: $|p_2^j - p_1^j| < |q^i - p_1^i|$.

Let us illustrate the above definition by considering our running example. Figure 4(b) depicts the RSQ of point q. As illustrated, point p_2 is a reverse skyline point of q since, according to above definition, point q is part of the dynamic skyline of point p_2.

The naive brute-force search algorithm for finding the reverse skyline of P given a query point q requires an examination of all points in P. For each point, a dynamic skyline query is performed (e.g. using BBS) to find the points which have q as part of their dynamic skyline. These points constitute the reverse skyline set. A first optimization of this brute-force approach is to stop processing the dynamic skyline of a point when q is already identified as a skyline point. In this case, there is no need to compute the entire skyline. Unfortunately, this simple optimization only leads to marginal improvements.

4. BRANCH AND BOUND PROCESSING OF REVERSE SKYLINE QUERIES

In this section we propose the *Branch and Bound Reverse Skyline* (BBRS) algorithm for reverse skyline computation, which is an improved customization of the original BBS algorithm [16]. Similar to [16], we assume that the data points are indexed by a data partitioning access method (e.g., R*-tree [2]). We start by providing some important lemmas and we continue with a description of our proposed BBRS algorithm. Finally, we provide an analysis of BBRS.

4.1 Preliminaries

In this section we exploit the geometric properties of reverse skyline points with respect to an arbitrary query point. We assume that the data set P and the query point q are given. We first define the global skyline in order to reduce the search space in finding reverse skyline points. For this, we prove one lemma (Lemma 1) that helps to immediately

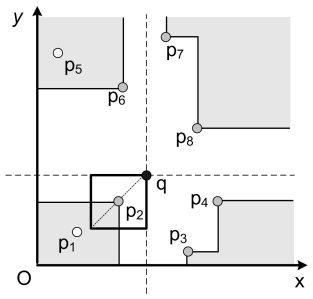


Figure 5: Global Skyline Example

identify candidate reverse skyline points. In addition, we provide another lemma (Lemma 2) to eliminate some of these candidate reverse skyline points not contributing to the result set.

Definition 3. (*Global Skyline*)
A point $p_1 \in P$ globally dominates $p_2 \in P$ with regard to the query point q if (1) for all $i \in \{1, \ldots, d\}$: $(p_i^1 - q_i)(p_i^2 - q_i) > 0$, (2) for all $i \in \{1, \ldots, d\}$: $|p_i^1 - q_i| \leq |p_i^2 - q_i|$ and (3) for at least one $j \in \{1, \ldots, d\}$: $|p_j^1 - q_j| < |p_j^2 - q_j|$. The global skyLine of a point q, $GSL(q)$, contains those points which are not globally dominated by another point according to q.

Figure 5 shows an example of the global skyline of point q and the corresponding dominance regions. Note that the global skyline is different from the dynamic skyline as there is no space transformation. We now present an important lemma which proves that the global skyline set is sufficient to answer the reverse skyline of q correctly.

LEMMA 1. *Let q be the query point and GSL(q) be the set of global skyline points and RSL(q) the set of reverse skyline points. Then, RSL(q)* $\subseteq$ *GSL(q).*

PROOF. Let $x \notin GSL(q)$. Then, there is a point $y \in GSL(q)$ that dominates x. Since for all $i \in \{1, \ldots, d\}$: $|q_i - x_i| \geq |q_i - y_i|$ and there exists a $j \in \{1, \ldots, d\}$: $|q_j - x_j| > |q_j - y_j|$, it follows from Definition 2 that $x \notin RSL(q)$. □

Lemma 1 enables our RSQ algorithms to efficiently retrieve a subset of the data points in P which are potential reverse skyline points of q by simply examining the global dynamic skyline of q. In Section 4.2, we utilize Lemma 1 to find the candidate reverse skyline set. The following lemma help us to eliminate some of these candidate reverse skyline points not contributing to the result set.

LEMMA 2. *Given the global skyline GSL(q) of point q. Assume point* $s \in GSL(q)$ *is a global skyline point. If* $\exists p \in P$ *such that for all* $i \in \{1, \ldots, d\}$ $|p_i - q_i| < |s_i - q_i|$ *then point s is not a reverse skyline point of q.*

PROOF. Assume that, the window query returns p as an answer. This means, p dominates q relative to its distance to s. Therefore, s is not in RSL(q). □

To illustrate, consider the rectangle centered at point p_2 in Figure 5. The extent of the rectangle is defined by the coordinate-wise distances to q. If there is no point inside this rectangle, then p_2 is in RSL(q). Otherwise there is a point in P which dominates q relative to its distance to p_2. Therefore, p_2 is not in RSL(q). With the result of Lemma 2, we reduce the time complexity of our algorithms by disregarding the dominance tests against the global skyline set.

4.2 Description of BBRS

The *Branch and Bound Reverse Skyline* (BBRS) algorithm computes the reverse skyline of a point q by expanding the entries of the heap H according to their distance from q. The algorithm works as follows: First, the algorithm computes (using Lemma 1) the set $GSL(q)$ of candidate reverse skyline points which is an upper bound of the actual reverse skyline set (i.e. no false hits are produced). These candidates are subsequently refined to the actual result. For this we run a window query for each skyline point s of the global skyline and if the query returns no answer we are sure (based on Lemma 2) that point s is a reverse skyline of q.

For the following discussion, we use the set of 2-dimensional data points organized in the R-tree of Figure 2. BBRS starts from the root node of the R-tree and inserts all its entries $(N5, N6)$ in a heap sorted by their distance from q. Then, the entry with the minimum distance $(N5)$ is expanded. This expansion removes the entry $(N5)$ from the heap and inserts its children $(N1, N2)$. The next expanded entry is again the one with the minimum distance from q $(N1)$, in which the first global skyline point (p_2) is found. This point (p_2) belongs to the reverse skyline, as the window centered at p_2 (Figure 5) is empty and is used for pruning in the subsequent execution. Point p_2 is inserted to the list RSL of reverse skyline points. Note, that point p_1 is globally dominated by p_2 and as soon as p_1 is on the top of the heap it is immediately discarded without the window test. The next entry to be expanded is $N6$. BBRS proceeds with the node $N6$ and inserts its children $(N4, N3)$. The heap now becomes $(N4, N2, p_1, N3)$. The algorithm proceeds in the same manner until the heap becomes empty thus all reverse skyline points are inserted in the result set RSL. The pseudo-code for BBRS is shown in Algorithm 1.

Two important issues need to be addressed. Firstly, the window query can be implemented as an empty range query. An *empty range query*, known also as boolean range query [19], is a special case of range query which will return either true or false depending on whether there is any point inside the given range or not. Obviously, empty range queries can be handled more efficiently than equivalent standard range queries and this advantage is exploited in the proposed schemes. For each candidate point, we define a rectangular range with the candidate point as the center and the coordinate-wise distance to the query point as its extent. If this boolean range query returns false then the point belongs to the reverse skyline, otherwise point is not a reverse skyline. The main strength of an empty range query over traditional range queries is that even if multiple MBRs intersect a search region we do not need to access all of them. This is because, for example, if at least one edge of an MBR is already inside the search region then it follows that there is at least a point qualifying the query. Secondly, the dominance test can be expensive if the skyline contains numerous points. In order to speed up this task we insert the skyline points found in a main-memory R-tree. Notice that an entry is tested for dominance twice: before it is inserted in the heap and before it is expanded. The second test is necessary because an entry in the heap may become dominated by

Algorithm 1 BBRS (R-tree R, Query point q)

```
1: RSL ← {} //set of reverse skyline points;
2: insert all entries of the root R in the heap H sorted by
   distance from q;
3: while (heap H is not empty) do
4:   remove top entry e;
5:   if (e is globally dominated by some point in S) then
6:     discard e;
7:   end if
8:   if (e is an intermediate entry) then
9:     for (each child e_i of e) do
10:      if (e_i is not globally dominated by some point in
         S) then
11:        insert e_i into heap H;
12:      end if
13:    end for
14:  else
15:    insert the pruning area of e_i into S;
16:    execute the window query based on e and q;
17:    if window query is empty then
18:      add e to the result set;
19:    end if
20:    discard e;
21:  end if
22:  output RSL
23: end while
```

some skyline point discovered after its insertion (therefore it does not need to be visited).

4.3 Analysis of BBRS

In this subsection we provide an analysis of BBRS. At first we present some important lemmas which guarantee the correctness of our algorithm. After that, we prove the efficiency of the branch and bound algorithm in terms of node accesses.

LEMMA 3. *BBRS visits (leaf and intermediate) entries of an R-tree in ascending order of their distance to the query point q.*

PROOF. The proof is straightforward since the algorithm always visits entries according to their *mindist* order preserved by the heap. □

LEMMA 4. *Any data point added to RSL during the execution of the algorithm is guaranteed to be a final reverse skyline point.*

PROOF. This is guaranteed by Lemma 2. □

LEMMA 5. *Every data point will be examined, unless one of its ancestor nodes has been pruned.*

PROOF. The proof is obvious since all entries that are not pruned by an existing global skyline point are inserted into the heap and examined. □

Lemmas 3 and 4 guarantee that, if BBRS is allowed to execute until its termination, it correctly returns all reverse skyline points, without reporting any false hits. Now, we prove that BBRS retrieves as candidates only the nodes that may contain reverse skyline points, and does not access the same node twice. Central to the analysis of BBRS is the concept of the global skyline search region (GSSR) of point q, that is, the part of the data space that is not globally dominated by any skyline point.

LEMMA 6. *If an entry e does not intersect the GSSR, then there is a skyline point p whose distance from the query point q is smaller than the mindist of e.*

PROOF. Since e does not intersect the GSSR, it must be dominated by at least one skyline point p, meaning that p dominates e. This implies that the distance of p to the query point q is smaller than the *mindist* of e. □

LEMMA 7. *The number of candidates examined by BBRS is minimum.*

PROOF. Assume, to the contrary, that the algorithm also visits an entry e that does not intersect the GSSR. Clearly, this entry should not be accessed because it cannot contain reverse skyline points. Consider a reverse skyline point that dominates e (e.g., k). Then the distance of k to the q is smaller than the *mindist* of e. According to Lemma 3, BBRS visits the entries of the R-tree in ascending order of their *mindist* to the q. Hence, k must be processed before e, meaning that e will be pruned by k, which contradicts the fact that e is visited. In order to complete the proof, we need to show that an entry is not visited multiple times. This is straightforward because entries are inserted into the heap (and expanded) at most once, according to their *mindist* from q. □

5. REVERSE SKYLINE COMPUTATION USING SKYLINE APPROXIMATIONS

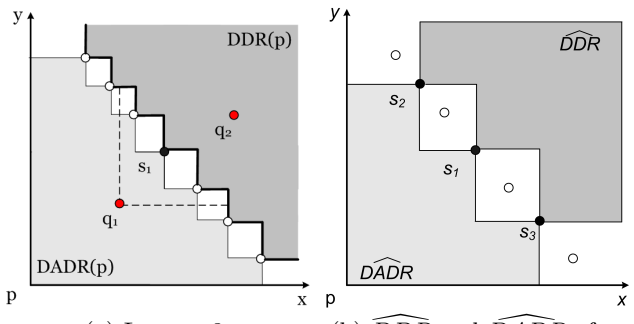
In this section we propose an enhanced algorithm, called *Reverse Skyline using Skyline Approximations* (RSSA) algorithm for supporting reverse skyline queries which is based on the well known filter-refinement paradigm. The main idea is to compute the dynamic skyline for each database object and to keep a fixed-sized approximation of this skyline on disk. By using this approximation in the filter step, we are able to identify points being in the reverse skyline as well as to filter out points not being in the reverse skyline. The remaining candidate points are then further examined in the refinement step. Similar to the method presented in the previous section, a window query is issued for each candidate, but the size of the window can be substantially reduced due to the approximation again. Consequently, this also leads to substantial cost savings.

5.1 Basic Observations

In this subsection we present an important lemma which allows our RSSA algorithm to compute the reverse skyline for every point by eliminating unnecessary dominance tests (i.e. window queries) for (1) definite reverse skyline points and (2) points not belonging to the result.

LEMMA 8. *For a given point p, let DSL(p) be the set of dynamic skyline points of p. Let q be a query point. If there is an s from DSL(p) that dynamically dominates the point q then and only then point p is not a reverse skyline of q.*

A straightforward corollary of the above lemma is the following: if the query point q dominates a point s from DSL(p), then we conclude that point p is a reverse skyline of q. The intuition behind the above lemma is the fact



(a) Lemma 8 (b) $\widehat{DDR}$ and $\widehat{DADR}$ of p

Figure 6: Illustration of $\widehat{DDR}$(p) and $\widehat{DADR}$(p).

that whenever we have to test a point p whether it is a reverse skyline point of q or not, it is sufficient to examine in which of the two regions, defined by the skyline of p, the point q belongs to. The *Dynamic Dominance Region* of p - DDR(p) - contains the points dominated by at least one skyline point. Whereas, the *Dynamic Anti-Dominance Region* - DADR(p) - contains the points dominating some skyline point. If the query point q falls inside DADR(p), then, based on the above observation, we conclude that point p is a reverse skyline point of q. In contrary, if the query point q is in DDR(p), we can discard point p because it is not in the reverse skyline of q. Figure 6(a) illustrates these two cases where the skyline of p contains seven points. In this figure point q_1 is inside the DADR of p and for that reason point p is a reverse skyline of q_1. This is because point q_1 dominates some point of the dynamic skyline of p (in particular point s_1). To explain, after the insertion of the point q_1, the modified skyline (shown with dashed lines in Figure 6(a)) would also contain the point q_1. Quite in contrary, in the same figure point q_2 is dominated by some point of the dynamic skyline of p (i.e. falls inside DDR(p) and is dominated by s_1) and for that reason point p is not a reverse skyline of q_2.

Based on this observation we can design a first algorithm for reverse skyline computation of an arbitrary query point q. In a pre-processing step, we first compute the dynamic skyline for every point and store these skylines on disk. When a query q is issued, we compute its global skyline and check for each point p in the global skyline whether q is in DADR(p) or in DDR(p). The problem of this algorithm is however its huge storage overhead. For independent dimensions the expected number of skyline points is $\theta((log\ n)^{d-1}/(d-1)!)$ [4]; therefore, the total storage cost of keeping all dynamic skylines is then super-linear in n, the number of objects.

In order to overcome this problem, we propose to keep fixed-size progressive approximations of DADR and DDR for each database point rather than keeping the exact regions. The basic idea of our approximation scheme is to select a sample of k points ($k \leq k_{max}$ where k_{max} denotes the actual dynamic skyline points) from the dynamic skyline of point p. Hence, the storage overhead of our approach is linear in the number of objects (as k is considered to be a constant). Then, the union of the dominance regions of these samples (constituting $\widehat{DDR}$(p)) is a progressive approximation of DDR(p), while the union of the anti-dominance regions (forming the $\widehat{DADR}$(p)) is a progressive approximation of DADR(p). Consider Figure 6(b), where seven skyline points (black and white) are shown. The three black ones are selected as samples for the approximation, i.e., points s_1, s_2 and s_3. Consequently, the corresponding regions $\widehat{DADR}$(p) and $\widehat{DDR}$(p) are the union of the dominance region and the anti-dominance region of these three black points, respectively. The part of the dynamic space of a point p, after removing $\widehat{DADR}$(p) and $\widehat{DDR}$(p), constitutes the *approximated skyline*. Figure 7 shows an example of the approximated skyline of p. Methods to compute such approximated skylines, given a value k of skyline points ($k \leq k_{max}$), are discussed in Section 6.

5.2 Description of the Algorithm

The enhanced RSSA algorithm for reverse skyline query processing on our R-tree based framework is explained next. An R-tree is built on the dataset and a skyline approximation of every database point is stored on disk. The initialization step remains the same; a global skyline query GSL(q) is performed returning the data objects belonging to the global skyline of the query and in the next step these objects are efficiently analyzed to answer the query. Note that the query point q is transformed to the new space (according to the candidate) in order to examine in which region of the approximate skyline it belongs to. The remainder of this subsection explains these two steps in more detail with necessary optimizations that can be used.

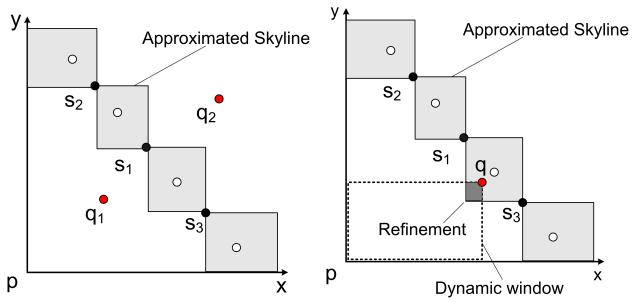
5.2.1 Filter Step

While calculating the global skyline of a query point we need to test those candidate reverse skyline points. However, some of them can be ruled out from further elaboration by examining their skyline approximations. The principle of the elimination is based on Lemma 8 and means that a point from the approximated skyline can dynamically dominate the query point (i.e. query point q is in $\widehat{DDR}$(p)) and therefore this candidate can never be the reverse skyline of the query point in the whole data set. In addition, definite reverse skyline points can immediately be reported by taking advantage of $\widehat{DADR}$(p). In summary, within the filter step we distinguish between the following three cases:

1. A point p can be dropped if $q \in \widehat{DDR}$(p).
2. Otherwise, if $q \in \widehat{DADR}$(p) point p can be added to the result set and is a reverse skyline point.
3. In case that $q \notin \widehat{DADR}$(p) $\cup$ $\widehat{DDR}$(p), point p needs to be refined.

This means, as soon as we find a candidate reverse skyline point s, we read the approximate skyline of s from disk and we check if point q falls inside the dominance region of some approximate skyline point. If this is the case, we can safely drop s. We refer to this step as filter step. Our experiments show that this step can efficiently filter out a significant number of candidates. After the filter step, we check if the query point falls inside the anti-dominance region of some approximated skyline point, which means that point s is a reverse skyline point of q.

5.2.2 Refinement Step



(a) Filter Step (b) Refinement Step

Figure 7: Approximated Skyline of p.

When none of the above cases match, we start the refinement step by issuing an empty range check query. The query region is now the dynamic query region, defined by the query point q and the origin of the dynamic space of p, minus $\widehat{DADR}$(p). Note that the size of the window is much smaller than the original window size and consequently, the query can be performed much faster.

Actually, because of the back-transformation to the original space where point p is the center of the original window, the small window is mapped to 2^d sub-partitions defined by point p. However, in order to further optimize the refinement step, we issue a 2-step empty range query. The first query is the original window as defined above, while the second corresponds to the 2^d projected smaller windows. If the first range query is empty, we don't need to access any nodes. In the latter case, we only test with the smaller windows. Figure 8 illustrates the refinement step in the original space.

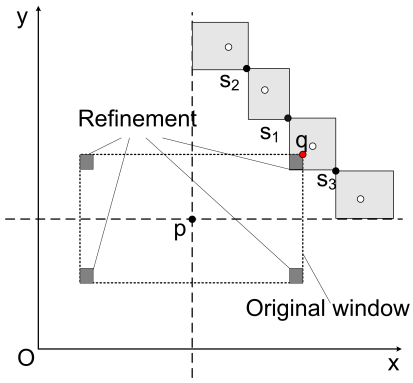


Figure 8: Refinement in the Original Space

We describe the RSSA algorithm using the example shown in Figure 7, where the points (q_1, q_2, q) correspond to our query points. A global skyline query GSL(q) is performed returning the data objects belonging to the global skyline of the query and in the next step these objects are efficiently analyzed to answer the query. Assume now, point p is returned by the global skyline query as a candidate reverse skyline point. Considering Figure 7(a), because query point q_1 is in $\widehat{DADR}$(p), the filter step is sufficient to identify p being in the reverse skyline. On the other hand, query point

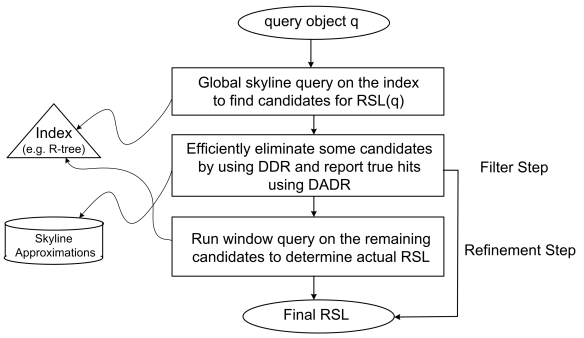


Figure 9: The Multi-Step Algorithm

q_2 is in $\widehat{DDR}$(p) and therefore our filter step ensures that point p can be discarded and is not a reverse skyline point. Now consider 7(b) where point q is neither in $\widehat{DDR}$(p) nor in $\widehat{DADR}$(p) and therefore, a refinement step is necessary by issuing a 2-step empty range query. Note that the lower left corner of the dynamic window is the origin of the data space. As we now can take benefit from $\widehat{DADR}$(p), the window only corresponds to the region with the grid pattern. Though the refinement step is required in this case, the much smaller window leads to substantial performance savings. The sketch depicted in Figure 9 summarizes our general approach for answering reverse skyline queries.

5.3 Updates

In the following, we present different strategies for updating our pre-processed approximations when points are inserted into and removed from the database. It should be noted here that our approach is primarily designed for query-intensive environments with a modest rate of updates and that the dynamic maintenance of our approximation scheme is rather expensive as the computation of skylines is unavoidable.

Let us first discuss the case of inserting a new point x into the database. In addition to inserting x into the R-tree, the approximations of the affected points have to be updated. Therefore, the global skyline GSL(x) is computed and the approximations $\widehat{DADR}$(y) and $\widehat{DDR}$(y) of every point y of GSL(x) have to be checked for updates. There are the following two cases: First, x is in $\widehat{DDR}$(y). Then, x has no impact on the skyline and its approximation. Second, x is not in $\widehat{DDR}$(y), i.e., x may effect the skyline. Then, the dynamic skyline of y and its corresponding approximation are computed. As the latter case might be quite expensive, we can trade in cost for approximation quality in the following way. When x is not in $\widehat{DADR}$(y), we simply do not recompute the approximation. Moreover, when x is in $\widehat{DADR}$(y), i.e., x dominates at least one of the sample points, we remove all the dominated sample points from the approximation and insert x into the sample. Again, there is no need for recomputing the approximation.

Now let us consider the deletion of a point x from the database. First, x is deleted from the R-tree and then, the global skyline GSL(x) is computed and the approximations

of every points y of GSL(x) have to be checked for update. There are now two cases. Firstly, x is in $\widehat{DDR}$(y), i.e., x is not a point of the dynamic skyline. Consequently, there is also no impact on the approximation. Secondly, x is not in $\widehat{DDR}$(y). Then, y could be a member of the dynamic skyline and therefore, the dynamic skyline is computed. If y is indeed a skyline point, the approximation of the skyline is recalculated. A much less expensive alternative for the second case would be to avoid recalculation at all. Instead, we only test whether x is in the approximation. If so, the point is deleted from the approximation.

6. APPROXIMATION OF SKYLINES

The basic idea of our approximation scheme is to precompute the dynamic skyline for each object of the database and to select a fixed number (k) of skyline points. Hence, the storage overhead of our approach is linear in the number of objects (as k is considered to be a constant). This section deals with the problem of computing a good progressive approximation $\widehat{DADR}$(p) and $\widehat{DDR}$(p). The goal is to maximize the probability that the refinement step is not required. This translates into maximizing the volume of $\widehat{DADR}$ and $\widehat{DDR}$ (the volume is defined as the number of points in the regions).

Before presenting non-trivial methods for calculating such approximations, let us first mention two naive approaches. The first naive approach is simply to compute a random sample of size k from the skyline points. The second naive approach is to sort the points first according to a specific dimension. Then, every $(m/k)^{th}$ point is drawn from the sorted sequence where m denotes the number of elements. Both methods require that the entire skyline is available. In order to reduce the overhead of the pre-processing step, yet another approach would be to stop computing skylines after having received the first k. For BBS, this leads to poor approximations and therefore we did not consider it in the following discussions.

6.1 Optimal Approximation for two-dimensional Skylines

An optimal algorithm for the approximation can be provided for two-dimensional data. The basic idea of the optimal algorithm is similar to the ones for approximating histograms [11] and time series [10]. The problem can be solved by a dynamic programming algorithm returning the optimal approximation. The reader is referred to [11] where the idea is nicely presented for approximating histograms.

Suppose that $S = \{(x_1, y_1), ...,(x_m, y_m)\}$ is a collection of m skyline points in a two-dimensional space. We sort the skyline points first in the ascending order of y-coordinate values; consequently, they are also sorted in the descending order of x-coordinate values. Furthermore, we treat the extreme points $(x_0, y_0) = (0, 1)$ and $(x_{m+1}, y_{m+1}) = (1, 0)$ separately.

For simplicity, we consider only the Dynamic Dominance Region (DDR) in the following discussion. For $0 \leq i \leq j \leq m$, the error metric we consider in this paper is:

$$S([i,j]) = (\sum_{k=i}^{j} x_k * (y_{k+1} - y_k)) - x_i * (y_{j+1} - y_i)).$$

This function considers the case in which we choose first the i-th skyline point and thereafter the j+1-th point. It is important that the error metric used in our algorithm is monotone as expressed in the following lemma.

LEMMA 9. *For any skyline S and any values of i, j, k with $0 \leq i \leq k < j \leq m$ the following holds:*

$$S([i,j]) \leq S([i,k]) + S([k,j]).$$

Similar to [11], we calculate the error function SSE*(i, k) using dynamic programming. This corresponds to the best approximation of k points for the first i skyline points, $i \geq k$. The optimal algorithm for approximating skylines is outlined in Algorithm 2.

Algorithm 2 OptimalSelect()

```
1: Input: Skyline S, sample size k_max
2: Output: Sample Sam
3: if (|S| ≤ k) then
4:    return S;
5: end if
6: for (int k = 1; i < k_max; k++) do
7:    for (int i = 1; i < m; i++) do
8:       SSE*(i,k) =
         min_{1<j<i} (SSE*(j,k-1) + SSE([j+1,i]));
9:    end for
10:   Sam = {(x_j, y_j) — index j was selected in
      SSE*(m, k_max)};
11: end for
12: return Sam
```

The optimal algorithm runs in O($m^2 * k_{max}$) iterations and has a space complexity of O($m * k_{max}$) where m is the size of the skyline and k_{max} denotes the sample size.

6.2 A Greedy Algorithm for d-dimensional Skylines

While we can achieve an optimal approximation for 2-dimensional skylines, the problem of an optimal approximation for d-dimensional skylines turns out to be more complex for $d > 2$. As the monotonic property, see Lemma 9, no longer holds, dynamic programming does not guarantee the delivery of an optimal approximation. A naive algorithm is to test all possible subsets of size k which would result in a runtime of $O(m^k)$. The interested reader is referred to [15] where efficient heuristics are discussed for obtaining approximate solutions. In this paper, we limit our discussion to a greedy-based algorithm with $O(m * k_{max})$ iterations. The basic idea is to select the point that has not been examined yet which adds the highest volume increase. The algorithm is outlined in Algorithm 3.

7. EXPERIMENTAL EVALUATION

In this section, we experimentally evaluate the efficiency of the proposed techniques for reverse skyline computation. We used two real datasets, namely CarDB and NBA[1] in our experiments. Specifically, the used car database CarDB, is a 6-dimensional dataset with attributes referring to Make, Model, Year, Price, Mileage and Location. This dataset

[1]These datasets can be downloaded from *autos.yahoo.com* and *www.nba.com*

Algorithm 3 GreedySelect()

```
1: Input: Skyline S, sample size k
2: Output: Sample Sam
3: LSB = {}; USB = {}; Sam = {};
4: if (|S| ≤ k) then
5:    return S;
6: end if
7: for (int i = 0; i < k; i++) do
8:    x = argmax_{p∈S} VOL(ADR(p) ∪ LSB)
      + VOL(DR(p) ∪ USB);
9:    insert(Sam, x);
10:   remove(S, x);
11:   LSB = LSB ∪ ADR(p);
12:   USB = USB ∪ DR(p);
13: end for
```

contains 50,000 tuples extracted from Yahoo! Autos. The two numerical attributes *Price* and *Mileage* of a car are considered in our experiments. NBA contains 17,000 13-dimensional points. Each record provides statistics of a player in a season. We selected four attributes: *number of games played* (GP), *total points* (PTS), *total rebounds* (REB) and *total assists* (AST). Finding the reverse skyline in this players' statistics data set makes excellent sense in practice. Coaches are often interested in finding the best substitutes of a player P. Suitable candidates might be in the reverse skyline of P.

We also used synthetic data sets with two different distributions. Uniformly distributed datasets consist of random points from the unit square, whereas the clustered dataset comprises ten randomly centered clusters, each of them with an equal number of points that follow a Gaussian distribution with variance 0.05 and mean equal to the associated centroid. We generated 2, 3 and 4-dimensional synthetic datasets (uniform and clustered) of varying sizes, ranging from 20,000 to 80,000 points. The values of the attributes are in the range [0; 10,000].

All experiments have been performed on a Windows PC with a 32-bit 3.2 GHz CPU and 2 GB main memory. In each experiment we performed 100 reverse skyline queries to the particular data set and reported the overall result. The queries follow the distribution of the dataset. Each dataset is indexed by an R-tree, where the page size is set to 4KB in all cases. All evaluated methods have been implemented in Java using the XXL libary [8].

7.1 Tuning the Skyline Approximation

The first set of experiments examines the impact of the number k of approximate skyline points on the performance of our RSSA algorithm. For every synthetic dataset (Uniform and Clustered) we created five R-trees by varying k from 10 to $k_{max} = 50$ skyline points. Then, we used the R-trees to process a query workload, and measure the average (per-query) number of page accesses. Figures 10(a) and 10(b) plot the cost as a function of k, for workloads with $d = 3$. Note that the result for $k = 0$ corresponds to the overhead of the basic BBRS that does not approximate the skyline (Section 4). As k becomes larger, the query performance improves continually. This is expected because we are using better approximations of the skylines.

In Figure 10(c) we report the results of our approximation algorithms (optimal and greedy) for selecting the skyline points used in the approximation for the 2-dimensional CAR dataset. We created three R-trees by varying k from 5 to 15 skyline points; in addition we created one R-tree to measure the overhead of the basic BBRS (k=0). Then, we used each tree to process a query workload. We observe that there is only a mild difference between the greedy and the optimal algorithm. In the following, we set k to 10 and 30 for CarDB and NBA, respectively, which provides the best overall performance.

7.2 Examination of Reverse Skyline Queries

We now examine the performance of our algorithms for reverse skyline computation for all datasets. Since there is no competitive approach for reverse skyline search, we only considered both of our algorithms. BBRS does not precompute skyline approximations. Obviously, this approach has less storage overhead than RSSA but needs expensive refinement steps. The optimized RSSA algorithm stores all pre-computed k skyline points on disk and therefore requires at most one page access extra for each examination.

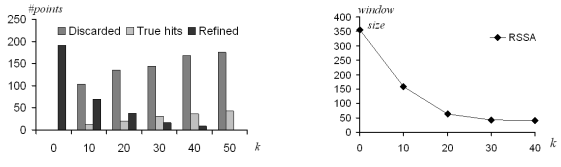
7.2.1 Pruning Capabilities

In Figures 11(a) and 11(b), the results are plotted for the global vs. the reverse skyline size as a function of dimensionality for the synthetic datasets. More precisely, we used datasets with 50,000 points whose d varies from 2 to 4. Based on the fact that the global skyline is a superset of the reverse skyline, we observe that our BBRS algorithm prunes the searched space effectively during the reverse skyline computation. To examine the skyline sizes for the real datasets, we plotted the average number of global and reverse skyline points for the CarDB and NBA datasets in Figure 11(c).

Figure 12(a) shows the pruning capability of RSSA w.r.t. k on the 3-dimensional Clustered data set. Compared with the size of the result set, only a small number of candidates have to be examined, i.e., $\widehat{DDR}$ yields a sound upper bound. Furthermore, the number of true hits we get from our $\widehat{DADR}$ approximation increases with increasing k. For these objects no expensive refinement step is necessary, thus our $\widehat{DADR}$ approximation provides a very accurate lower bound for the skyline. Figure 12(b) shows the average size of the window queries as a function of k for the same dataset.

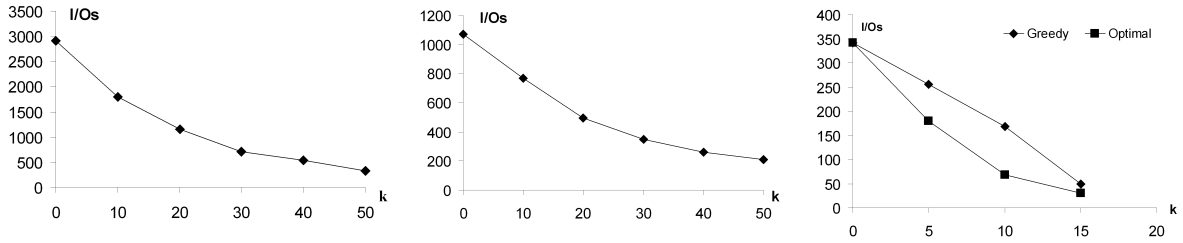
7.2.2 Evaluation of the Algorithms

Next, we examine the cost of BBRS and RSSA for answering each query on various datasets. In Figure 13, we depict the results of a comparison for our real datasets CarDB and NBA. Note that the y-axes are plotted in logarithmic scale.



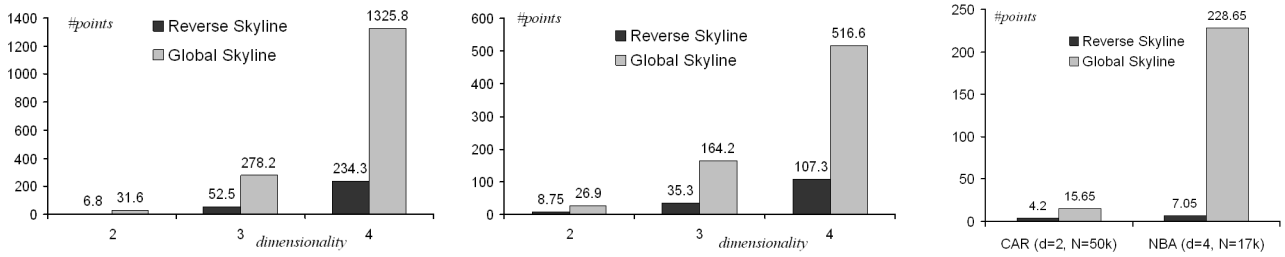
(a) Filter vs. Refinement (b) Window Radius

Figure 12: RSSA performance for different values of k.



(a) Uniform (3-d, 50k) (b) Clustered (3-d, 50k) (c) CAR dataset (2-d, 50k)

Figure 10: Performance for different values of k.



(a) Uniform dataset (N=50,000) (b) Clustered dataset (N=50,000) (c) Real datasets

Figure 11: Reverse vs. Global Skyline Size.

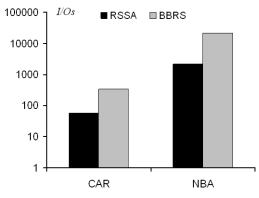
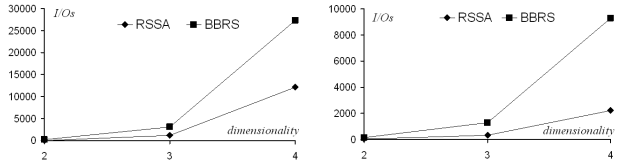


Figure 13: Average cost of our algorithms.



(a) Uniform (N=50k) (b) Clustered (N=50k)

Figure 14: Scalability with dimensionality.

RSSA consistently achieves lower average cost than BBRS (by one order of magnitude in Figure 13). Similar results are observed for other data sets.

7.3 Scalability with the Dimensionality and Database Size

In this section we experientially confirm that our algorithms (BBRS and RSSA) are scalable according to the dimensionality and the size of the database. In this experiment, we only used synthetic datasets. For each dataset, we set (for RSSA) k to the value that optimize the overall performance (through a tuning process similar to Figure 10). Specifically, we set k to 10 for the dataset containing 20,000 points, and k to 30 for the others.

7.3.1 Effect of Dimensionality

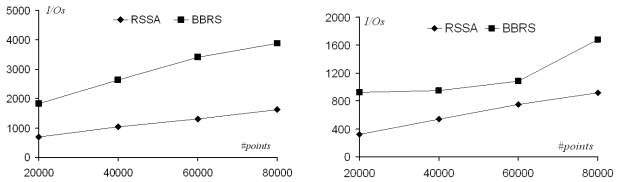
For the purpose of this experiment we used both synthetic datasets. In order to examine the impact of the dimensionality d on our algorithms, we used datasets with 50,000 points with d varying from 2 to 4. In Figure 14(a), the cost of each method (in retrieving reverse skylines) is plotted as a function of d for uniform distributions. Both algorithms scale well, although our optimized RSSA algorithm outperforms BBRS significantly (up to an order of magnitude for d=4). In Figure 14(b), the x-axis represents the dimensionality whereas the y-axis measures the page accesses required to compute the reverse skyline for the clustered dataset. Similar observations hold as for uniformly distributed data.

7.3.2 Effect of Database Size

In this experiment we used the 3-dimensional synthetic datasets and we varied its size between 20,000 and 80,000 points. Figure 15(a) compares the average cost of BBRS and RSSA for uniformly distributed data. Evidently, the optimized method (RSSA) scales better with the database size than BBRS. In particular, RSSA outperforms BBRS for 80,000 points significantly. Figure 15(b) illustrates the corresponding results for Clustered data, confirming similar observations where $k = 10$ was used for the smallest dataset, and $k = 30$ for the others.

8. CONCLUSIONS

In this paper, we introduced the concept of *Reverse Skyline Queries* (RSQ). Given a set of data points P and a query point q, an RSQ returns the data objects that have



(a) Uniform (d=3) (b) Clustered (d=3)

Figure 15: Scalability in the database size.

the query object in the set of their dynamic skyline. It is the complimentary problem to that of finding the dynamic skyline of a query object. Such kind of dynamic skyline corresponds to the skyline of a transformed data space where point q becomes the origin and all points are represented by their distance to q.

In order to compute the reverse skyline of an arbitrary query point, we first proposed a Branch and Bound algorithm (called BBRS), which is an improved customization of the original BBS algorithm. Furthermore, we identified a super set of the reverse skyline that allows us to bound the space searched during the reverse skyline computation. To further reduce the computational cost of determining if a point belongs to the reverse skyline, we proposed an enhanced algorithm (called RSSA), that is based on accurate pre-computed approximations of the skylines. These approximations are used to identify whether a point belongs to the reverse skyline or not. For two-dimensional data, we presented an optimal algorithm, while for higher dimensions a greedy algorithm is proposed. Through extensive experiments with both real-world and synthetic datasets, we showed that our algorithms can efficiently support reverse skyline queries.

In our future work, we will study the problem of supporting reversed k-skyband queries. Moreover, we are interested in efficient algorithms for computing accurate skyline approximations for three- and higher-dimensional data.

9. ACKNOWLEDGMENTS

The authors would like to thank Ilya Vladimirskiy for initiating discussions that led to this paper.

10. REFERENCES

[1] E. Achtert, C. Böhm, P. Kröger, P. Kunath, A. Pryakhin, and M. Renz. Efficient Reverse k-Nearest Neighbor Search in Arbitrary Metric Spaces. In *SIGMOD*, pages 515–526, 2006.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD*, pages 322–331, 1990.

[3] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *ICDE*, pages 421–430, 2001.

[4] C. Buchta. On the average number of maxima in a set of vectors. *Information Processing Letters*, 33(2):63–65, 1989.

[5] S. Chaudhuri, N. N. Dalvi, and R. Kaushik. Robust Cardinality and Cost Estimation for Skyline Operator. In *ICDE*, page 64, 2006.

[6] J. Chomicki. Preference Formulas in Relational Queries. *ACM TODS*, 28(4):427–466, 2003.

[7] E. Dellis, A. Vlachou, I. Vladimirskiy, B. Seeger, and Y. Theodoridis. Constrained Subspace Skyline Computation. In *CIKM*, pages 415–424, 2006.

[8] J. V. den Bercken, B. Blohsfeld, J.-P. Dittrich, J. Krämer, T. Schäfer, M. Schneider, and B. Seeger. Xxl - a Library Approach to Supporting Efficient Implementations of Advanced Database Queries. In *VLDB*, pages 39–48, 2001.

[9] K. Deng, X. Zhou, and H. T. Shen. Multi-source Skyline Query Processing in Road Networks. In *ICDE*, 2007.

[10] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos. Indexing spatiotemporal archives. *VLDB J.*, 15(2):143–164, 2006.

[11] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal Histograms with Quality Guarantees. In *VLDB*, pages 275–286, 1998.

[12] W. Kießling. Foundations of Preferences in Database Systems. In *VLDB*, pages 311–322, 2002.

[13] F. Korn and S. Muthukrishnan. Influence Sets Based on Reverse Nearest Neighbor Queries. In *SIGMOD*, pages 201–212, 2000.

[14] C. Li, B. C. Ooi, A. K. H. Tung, and S. Wang. DADA: a Data Cube for Dominant Relationship Analysis. In *SIGMOD*, pages 659–670, 2006.

[15] S. Muthukrishnan and T. Suel. Approximation algorithms for array partitioning problems. *J. Algorithms*, 54(1):85–104, 2005.

[16] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An Optimal and Progressive Algorithm for Skyline Queries. In *SIGMOD*, pages 467–478, 2003.

[17] J. Pei, W. Jin, M. Ester, and Y. Tao. Catching the Best Views of Skyline: A Semantic Approach Based on Decisive Subspaces. In *VLDB*, pages 253–264, 2005.

[18] M. Sharifzadeh and C. Shahabi. The Spatial Skyline Queries. In *VLDB*, pages 751–762, 2006.

[19] A. Singh, H. Ferhatosmanoglu, and A. S. Tosun. High Dimensional Reverse Nearest Neighbor Queries. In *CIKM*, pages 91–98, 2003.

[20] I. Stanoi, D. Agrawal, and A. E. Abbadi. Reverse Nearest Neighbor Queries for Dynamic Databases. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 44–53, 2000.

[21] Y. Tao, D. Papadias, and X. Lian. Reverse kNN Search in Arbitrary Dimensionality. In *VLDB*, pages 744–755, 2004.

[22] Y. Tao, X. Xiao, and J. Pei. SUBSKY: Efficient Computation of Skylines in Subspaces. In *ICDE*, page 65, 2006.

[23] C. Yang and K.-I. Lin. An Index Structure for Efficient Reverse Nearest Neighbor Queries. In *ICDE*, pages 485–492, 2001.

[24] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang. Efficient Computation of the Skyline Cube. In *VLDB*, pages 241–252, 2005.