An Optimal and Progressive Algorithm for Skyline Queries

Dimitris Papadias[§] Yufei Tao[†] Greg Fu[§] Bernhard Seeger[*]

[§]Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{dimitris,greg}@cs.ust.hk

[†]Department of Computer Science
Carnegie Mellon University
Pittsburgh, USA
taoyf@cs.cmu.edu

[*]Dept. of Mathematics and Computer Science
Philipps-University Marburg
Marburg, Germany
seeger@mathematik.uni-marburg.de

ABSTRACT

The skyline of a set of *d*-dimensional points contains the points that are not dominated by any other point on all dimensions. Skyline computation has recently received considerable attention in the database community, especially for progressive (or online) algorithms that can quickly return the first skyline points without having to read the entire data file. Currently, the most efficient algorithm is NN (nearest neighbors), which applies the divide-and-conquer framework on datasets indexed by R-trees. Although NN has some desirable features (such as high speed for returning the initial skyline points, applicability to arbitrary data distributions and dimensions), it also presents several inherent disadvantages (need for duplicate elimination if *d*>2, multiple accesses of the same node, large space overhead). In this paper we develop BBS (branch-and-bound skyline), a progressive algorithm also based on nearest neighbor search, which is IO optimal, i.e., it performs a single access only to those R-tree nodes that may contain skyline points. Furthermore, it does not retrieve duplicates and its space overhead is significantly smaller than that of NN. Finally, BBS is simple to implement and can be efficiently applied to a variety of alternative skyline queries. An analytical and experimental comparison shows that BBS outperforms NN (usually by orders of magnitude) under all problem instances.

1. INTRODUCTION

The skyline operator is important for several applications involving multi-criteria decision making. Given a set of objects $p_1, p_2,.., p_N$, the operator returns all objects p_i such that p_i is not *dominated* by another object p_j. Using the common example in the literature, assume in Figure 1.1 that we have a set of hotels and for each hotel we store its distance from the beach (*x* axis) and its price (*y* axis). The most *interesting* hotels are the ones (*a*, *i*, *k*) for which there is no point that is better on both dimensions. Borzsonyi et al. [BKS01] propose an SQL syntax for the skyline operator, according to which the above query would be expressed as: [*Select* *, *From* Hotels, *Skyline* of Price *min*, Distance *min*], where *min* indicates that the price and the distance attributes should be minimized. The syntax can also capture different conditions (such as *max*), joins, group-by and so on. For simplicity, we assume that skylines are computed with respect to *min* conditions on all dimensions; however, all methods discussed can be applied with any combination of conditions.

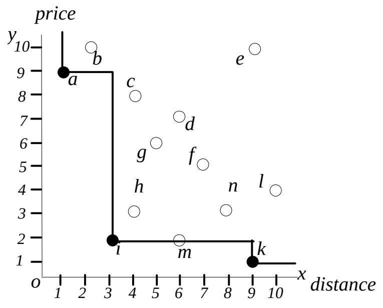


Figure 1.1: Example dataset and skyline

Using the *min* condition, a point p_i *dominates*[1] another point p_j if and only if the coordinate of p_i on any axis is not larger than the corresponding coordinate of p_j. Informally, this implies that p_i is preferable to p_j according to any *preference* (*scoring*) function which is monotone on all attributes. For instance, hotel *a* in Figure 1.1 is better than hotels *b and e* since it is closer to the beach and cheaper (independently of the relative importance of the distance and price attributes). Furthermore, for every point *p* in the skyline there exists a monotone function *f* such that *p* minimizes *f* [BKS01].

Skylines are related to several other well-known problems, including *convex hulls*, *top-K* queries and *nearest neighbor* search. In particular, the convex hull contains the subset of skyline points that may be optimal only for linear preference functions (as opposed to any monotone function). Böhm and Kriegel [BK01] propose an algorithm for convex hulls, which applies branch and bound search on datasets indexed by R-trees. In addition, several main-memory algorithms have been proposed for the case that the whole dataset can fit in memory [PS85].

Top-*K* (or ranked) queries retrieve the best *K* objects that minimize a specific preference function. As an example, given the preference function $f(x,y)=x+y$, the top-3 query, for the dataset in Figure 1.1, retrieves <*i*,5>, <*h*,7>, <*m*,8> (in this order), where the number with each point indicates its score. The difference from skyline queries is that the output changes according to the input function and the retrieved points are not guaranteed to be part of the skyline (*h* and *m* are dominated by *i*). Recent database techniques for top-*K* queries include *Prefer* [HKP01] and *Onion* [CBC+00] that are based on pre-materialization and convex hulls, respectively. Several methods have been proposed for combining the results of multiple top-*K* queries [F98, NCS+01].

Nearest neighbor queries specify a query point *q* and output the objects closest to *q*, in increasing order of their distance. Existing database algorithms assume that the objects are indexed by an R-tree (or some other data-partition method) and apply

[1] According to this definition two, or more, points with the same coordinates can be part of the skyline.

branch-and-bound search. In particular, the depth-first algorithm of [RKV95] starts from the root of the R-tree and recursively visits the entry closest to the query point. Entries, which are farther than the nearest neighbor already found, are pruned. The best-first algorithm of [HS99] inserts the entries of the visited nodes in a heap, and follows the one closest to the query point. The relation between skyline queries and nearest neighbor search has been exploited by previous skyline algorithms and will be discussed in Section 2.

Skylines, and other directly related problems such as multi-objective optimization [S86], maximum vectors [KPL75, SM88, M91] and the contour problem [M74], have been extensively studied and numerous algorithms have been proposed for main-memory processing. To the best our knowledge, however, the first work that addresses skylines in the context of databases is [BKS01], which develops algorithms based on block nested loops, divide-and-conquer and index scanning. Tan et al. [TEO01] propose *progressive* algorithms that can output skyline points without having to scan the entire data input. Finally, Kossmann et al. [KRR02] present an improved algorithm, called NN due to its reliance on nearest neighbor search, which applies the divide-and-conquer framework on datasets indexed by R-trees. The experimental evaluation of [KRR02] shows that NN outperforms previous algorithms in terms of overall performance and general applicability independently of the dataset characteristics, while it supports on-line processing efficiently. Despite its advantages, NN has also some serious shortcomings such as need for duplicate elimination, multiple node visits and large space requirements.

Motivated by this fact, we propose a progressive algorithm called BBS (branch and bound skyline), which, like NN, is based on nearest neighbor search on multi-dimensional access methods, but (unlike NN) it is optimal in terms of node accesses. BBS incorporates the advantages of NN, without sharing its shortcomings. We experimentally and analytically show that BBS outperforms NN (usually by orders of magnitudes) both in terms of CPU and IO costs for all problem instances, while incurring less space overhead. In addition to its efficiency, the proposed algorithm is simple and easily extendible to several variations of skyline queries.

The rest of the paper is organized as follows: Section 2 reviews previous secondary-memory algorithms for skyline computation, focusing more on NN since it is the most recent and efficient algorithm. Section 3, analyzes the shortcomings of NN and introduces BBS, providing a cost model for its expected performance and a proof of its optimality. Section 4 proposes alternative skyline queries and discusses their processing using BBS. Section 5 experimentally evaluates BBS, comparing it against NN under a variety of settings. Finally, Section 6 concludes the paper with some directions for future work.

2. RELATED WORK

This section surveys existing secondary-memory algorithms for computing skylines, namely: (1) *block nested loop*, (2) *divide-and-conquer*, (3) *bitmap*, (4) *index* and (5) *nearest neighbor*. Specifically, (1-2) are proposed in [BKS01], (3-4) in [TEO01] and (5) in [KRR02]. We do not consider the *sorted list scan*, and the *B-tree* algorithms of [BKS01] due to their limited applicability (only for two dimensions) and poor performance, respectively.

2.1 Block Nested Loop (BNL)

Intuitively, a straightforward approach to compute the skyline is to compare each point p with every other point; if p is not dominated, then it is a part of the skyline. BNL builds on this concept by scanning the data file and keeping a list of candidate skyline points in main memory. The first data point is inserted into the list. For each subsequent point p, there are three cases:
(i) If p is dominated by any point in the list, it is discarded as it is not part of the skyline.
(ii) If p dominates any point in the list, it is inserted into the list, and all points in the list dominated by p are dropped.
(iii) If p is neither dominated, nor dominates, any point in the list, it is inserted into the list as it may be part of the Skyline.

The list is self-organizing because every point found dominating other points is moved to the top. This reduces the number of comparisons as points that dominate multiple other points are likely to be checked first. A problem of BNL is that the list may become larger than the main memory. When this happens, all points falling in third case (cases (i) and (ii) do not increase the list size), are added to a temporary file. This fact necessitates multiple passes of BNL. In particular, after the algorithm finishes scanning the data file, only points that were inserted in the list before the creation of the temporary file are guaranteed to be in the skyline and are output. The remaining points must be compared against the ones in the temporary file. Thus, BNL has to be executed again, this time using the temporary (instead of the data) file as input.

The advantage of BNL is its wide applicability, since it can be used for any dimensionality without indexing or sorting the data file. Its main problems are the reliance on main memory (a small memory may lead to numerous iterations) and its inadequacy for on-line processing (it has to read the entire data file before it returns the first skyline point).

2.2 Divide-and-Conquer (D&C)

The D&C approach divides the dataset into several partitions so that each partition fits in memory. Then, the partial skyline of the points in every partition is computed using a main-memory algorithm (e.g., [SM88, M91]), and the final skyline is obtained by merging the partial ones. Figure 2.1 shows an example using the dataset of Figure 1.1. The data space is divided into 4 partitions s_1, s_2, s_3, s_4, with partial skylines $\{a,c,g\}$, $\{d\}$, $\{i\}$, $\{m,k\}$, respectively. In order to obtain the final skyline, we need to remove those points that are dominated by some point in other partitions. Obviously all points in the skyline of s_3 must appear in the final skyline, while those in s_2 are discarded immediately because they are dominated by any point in s_3 (in fact s_2 needs to be considered only if s_3 is empty). Each skyline point in s_1 is compared only with points in s_3, because no point in s_2 or s_4 can dominate those in s_1. In this example, points c,g are removed because they are dominated by i. Similarly, the skyline of s_4 is also compared with points in s_3, which results in the removal of m. Finally, the algorithm terminates with the remaining points $\{a,i,k\}$. D&C is efficient only for small datasets (e.g., if the entire dataset fits in memory then the algorithm requires only one application of a main-memory skyline algorithm). For large datasets, the partitioning process requires reading and writing the entire dataset at least once, thus incurring significant IO cost. Further, this approach is not suitable for on-line processing because it cannot report any skyline until the partitioning phase completes.

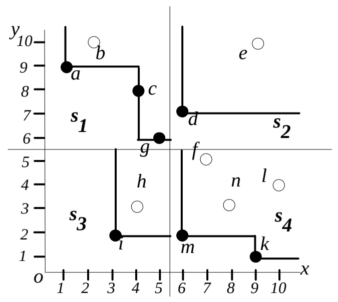


Figure 2.1: Divide and conquer

2.3 *Bitmap*

This technique encodes in bitmaps all the information required to decide whether a point is in the Skyline. A data point $p = (p_1, p_2, ..., p_d)$, where d is the number of dimensions, is mapped to a m-bit vector, where m is the total number of distinct values over all dimensions. Let k_i be the total number of distinct values on the ith dimension (i.e., $m = \sum_{i=1\sim d} k_i$). In Figure 1.1, for example, there are $k_1 = k_2 = 10$ distinct values on the x-, y-dimensions and $m = 20$. Assume that p_i is the j_i-th smallest number on the ith axis; then, it is represented by k_i bits, where the $(k_i - j_i + 1)$ most significant bits are 1, and the remaining ones 0. Table 2.1 shows the bitmaps for points in Figure 1.1. Since point a has the smallest value (1) on the x-axis, all bits of a_1 are 1. Similarly, since a_2 (=9) is the 9-th smallest on the y-axis, the first 10–9+1=2 bits of its representation are 1, while the remaining ones are 0.

id	coordinate	bitmap representation
a	(1,9)	(1111111**1**11, 11**0**0000000)
b	(2,10)	(1111111**1**10, 10**0**0000000)
c	**(4,8)**	(1111111**1**000, 11**1**0000000)
d	(6,7)	(111110**0**000, 11**1**1000000)
e	(9,10)	(110000**0**000, 10**0**0000000)
f	(7,5)	(111100**0**000, 11**1**1110000)
g	(5,6)	(111111**0**000, 11**1**1100000)
h	(4,3)	(111111**1**000, 11**1**1111100)
i	(3,2)	(111111**1**100, 11**1**1111110)
k	(9,1)	(110000**0**000, 11**1**1111111)
l	(10,4)	(100000**0**000, 11**1**1111000)
m	(6,2)	(111110**0**000, 11**1**1111110)
n	(8,3)	(111000**0**000, 11**1**1111100)

Table 2.1: The *bitmap* approach

Consider now that we want to decide whether a point, e.g., c with bitmap representation (1111111000, 1110000000), belongs to the skyline. The most significant bits whose value is 1, are the 4th and the 8th, on dimensions x and y, respectively. The algorithm creates two bit-strings, c_X = 1110000110000 and c_Y = 0011011111111, by juxtaposing the corresponding bits (i.e., 4th and 8th) of every point. In Table 2.1, these bit-strings (shown in bold) contain 13 bits (one from each object, starting from a and ending with n). The 1's in the result of $c_X \& c_Y$=0010000110000, indicate the points that dominate c, i.e., c, h and i. Obviously, if there is more than a single 1, the considered point is not in the skyline[2]. The same operations are repeated for every point in the dataset, to obtain the entire skyline.

[2] The result of "&" will contain several 1's if multiple skyline points coincide. This case can be handled with an additional "*or*" operation [TEO01].

The efficiency of *bitmap* relies on the speed of bit-wise operations. The approach can quickly return the first few skyline points according to their insertion order (e.g., alphabetical order in Table 2.1), but cannot adapt to different user preferences, which is an important property of a good skyline algorithm [KRR02]. Furthermore, the computation of the entire skyline is expensive because, for each point inspected, it must retrieve the bitmaps of all points in order to obtain the juxtapositions. Also the space consumption may be prohibitive, if the number of distinct values is large. Finally, the technique is not suitable for dynamic datasets where insertions may alter the rankings of attribute values.

2.4 Index

The "index" approach organizes a set of d-dimensional points into d lists such that a point $p = (p_1, p_2, …, p_d)$ is assigned to the ith list ($1 \le i \le d$), if and only if, its coordinate p_i on the ith axis is the minimum among all dimensions, or formally: $p_i \le p_j$ for all $j \ne i$. Table 2.2 shows the lists for the dataset of Figure 1.1. Points in each list are sorted in ascending order of their minimum coordinate (*minC*, for short) and indexed by a B-tree. A *batch* in the ith list consists of points that have the same ith coordinate (i.e., *minC*). In Table 2.2, every point of list 1 constitutes an individual batch because all x-coordinates are different. Points in list 2 are divided into 5 batches {k}, {i,m}, {h,n}, {l} and {f}.

list 1		**list 2**	
a (1, 9)	*minC*=1	*k* (9, 1)	*minC*=1
b (2, 10)	*minC*=2	*i* (3, 2), *m* (6, 2)	*minC*=2
c (4, 8)	*minC*=4	*h* (4, 3), *n* (8, 3)	*minC*=3
g (5, 6)	*minC*=5	*l* (10, 4)	*minC*=4
d (6, 7)	*minC*=6	*f* (7, 5)	*minC*=5
e (9, 10)	*minC*=9		

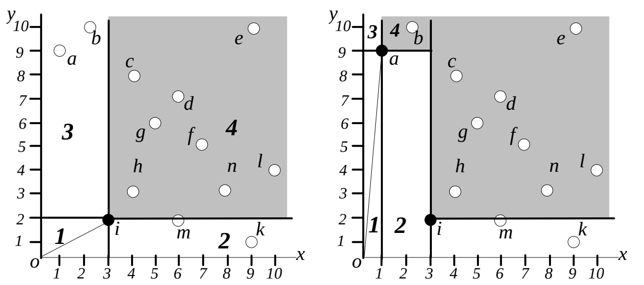
Table 2.2: The index approach

Initially, the algorithm loads the first batch of each list, and handles the one with the minimum *minC*. In Table 2.2, the first batches {a}, {k} have identical *minC*=1, in which case the algorithm handles the batch from list 1. Processing a batch involves (i) computing the skyline inside the batch, and (ii) among the computed points, it adds the ones not dominated by any of the already-found skyline points into the skyline list. Continuing the example, since batch {a} contains a single point and no skyline point is found so far, a is added to the skyline list. The next batch {b} in list 1 has *minC*=2; thus, the algorithm handles batch {k} from list 2. Since k is not dominated by a, it is inserted in the skyline. Similarly, the next batch handled is {b} from list 1, where b is dominated by point a (already in the skyline). The algorithm proceeds with batch {i,m}, computes the skyline inside the batch that contains a single point i (i.e., i dominates m), and adds i to the skyline. At this step the algorithm does not need to proceed further, because both coordinates of i are smaller than or equal to the *minC* (i.e., 4, 3) of the next batches (i.e., {c}, {h,n}) of lists 1 and 2. This means that all the remaining points (in both lists) are dominated by i and the algorithm terminates with {a,i,k}.

Although this technique can quickly return skyline points at the top of the lists, it has several disadvantages. First, as with the *bitmap* approach, the order that the skyline points are returned is fixed, not supporting user-defined preferences. Second, as indicated in [KRR02], the lists computed for d dimensions cannot be used to retrieve the skyline on any subset of the dimensions. In general, in order to support queries for arbitrary dimensionality subsets, an exponential number of lists must be pre-computed.

2.5 Nearest Neighbor (NN)

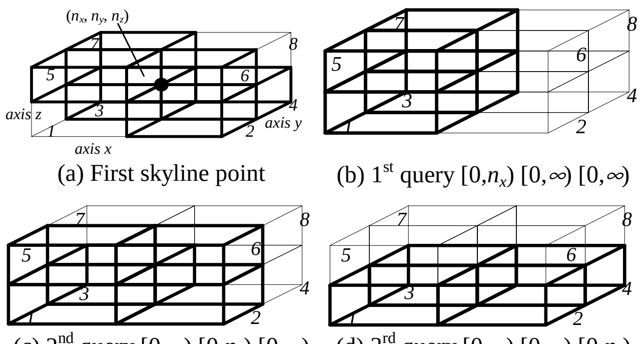
NN uses the results of nearest neighbor search to partition the data universe recursively. As an example, consider the application of the algorithm to the data set of Figure 1.1, which is indexed by an R-tree. NN performs a nearest neighbor query (using an existing algorithm such as [RKV95, HS99]) on the R-tree, to find the point with the minimum distance (*mindist*) from the beginning of the axes (point *o*). Without loss of generality, we assume that distances are computed according to L_1 norm, i.e., the *mindist* of a point *p* from the beginning of the axes equals the sum of the coordinates of *p*. It can be shown that the first nearest neighbor (point *i* with *mindist* 5) is part of the skyline. On the other hand, all the points in the *dominance region* of *i* (shaded area in Figure 2.2a) can be pruned from further consideration. The remaining space is split in two partitions based on the coordinates (i_x,i_y) of point *i*: (i) $[0,i_x)$ $[0,\infty)$ and (ii) $[0,\infty)$ $[0,i_y)$. In Figure 2.2a, the first partition contains subdivisions 1 and 3, while the second one, subdivisions 1 and 2.



(a) discovery of point *i* (a) discovery of point *a*

Figure 2.2: Example of NN

The set of partitions resulting after the discovery of a skyline point are inserted in a *to-do* list. While the *to-do* list is not empty, NN removes one of the partitions from the list and recursively repeats the same process. For instance, point *a* is the nearest neighbor in partition $[0,i_x)$ $[0,\infty)$, which causes the insertion of partitions $[0,a_x)$ $[0,\infty)$ (subdivisions 1 and 3 in Figure 2.2b) and $[0,i_x)$ $[0,a_y)$ (subdivisions 1 and 2 in Figure 2.2b) in the *to-do* list. If a partition is empty, it is not subdivided further. In general, if *d* is the dimensionality of the data-space, each skyline point discovered causes *d* recursive applications of NN. Figure 2.3a shows a 3D example, where point *n* with coordinates (n_x,n_y,n_z) is the first nearest neighbor (i.e., skyline point).



(a) First skyline point (b) 1st query $[0,n_x)$ $[0,\infty)$ $[0,\infty)$

(c) 2nd query $[0,\infty)$ $[0,n_y)$ $[0,\infty)$ (d) 3rd query $[0,\infty)$ $[0,\infty)$ $[0,n_z)$

Figure 2.3: NN partitioning for 3 dimensions

The NN algorithm will be recursively called for the partitions (i) $[0,n_x)$ $[0,\infty)$ $[0,\infty)$ (Figure 2.3b), (ii) $[0,\infty)$ $[0,n_y)$ $[0,\infty)$ (Figure 2.3c) and (iii) $[0,\infty)$ $[0,\infty)$ $[0,n_z)$ (Figure 2.3d). Among the eight space subdivisions shown in Figure 2.3, the 8th one will not be searched by any query since it is dominated by point *n*. Each of the remaining subdivisions however, will be searched by two queries, e.g., a skyline point in subdivision 2, will be discovered by both the 2nd and 3rd query. In general, for *d*>2, the overlapping of the partitions necessitates duplicate elimination. Kossmann et al. [KRR02] propose the following elimination methods:

Laisser-faire: A main memory hash table stores the skyline points found so far. When a point *p* is discovered, it is probed and if it already exists in the hash table, *p* is discarded; otherwise, *p* is inserted into the hash table. The technique is straightforward and incurs minimum CPU overhead, but results in very high IO cost since large parts of the space will be accessed by multiple queries.

Propagate: When a point *p* is found, all the partitions in the *to-do* list that contain *p* are removed and re-partitioned according to *p*. The new partitions are inserted into the *to-do* list. Although *propagate* does not discover the same skyline point twice, it incurs high CPU cost because the *to-do* list is scanned every time a skyline point is discovered.

Merge: The main idea is to merge partitions in the *to-do*, thus reducing the number of queries that have to be performed. Partitions that are contained in other ones can be eliminated in the process. Like *propagate*, *merge* also incurs high CPU cost since it is expensive to find good candidates for merging.

Fine-grained Partitioning: The original NN algorithm generates *d* partitions after a skyline point is found. An alternative approach is to generate 2^d non-overlapping subdivisions. In Figure 2.3 for instance, the discovery of point *n* will lead to 6 new queries (i.e., 2^3-2 since subdivisions 1 and 8 cannot contain any skyline points). Although *fine grain partitioning* avoids duplicates, it generates the more complex problem of false hits, i.e., it is possible that points in one subdivision (e.g., 4) are dominated by points in another (e.g., 2) and should be eliminated.

According to the experimental evaluation of [KRR02], the performance of *laisser-faire* and *merge* is unacceptable, while *fine grain partitioning* was not implemented due to the false hits problem. *Propagate* is significantly more efficient, but the best results were achieved by a *hybrid* method that combines *propagate* and *laisser-faire*. Compared to previous algorithms, NN is significantly faster for up to 4 dimensions. In particular, NN returns the entire skyline faster than *index* and their difference increases (sometimes to orders of magnitudes) with the size of the skyline. On the other hand, *index* has better performance for returning skyline points progressively, as it simply scans through the extended B-tree to return points that are good in one dimension. However, as claimed in [KRR02], these points are not representative of the whole skyline because certain dimensions are favored. For higher than 3 dimensions, the cost of NN increases due to the growth of the overlapping area between partitions and, to a lesser degree, due to the performance deterioration of R-trees. For these cases, *index* is also inapplicable due to its extreme space requirements (if skylines on subsets of the dimensions are allowed). D&C and *bitmap* are not favored by correlated datasets (where the skyline is small) as the overhead of merging and loading the bitmaps, respectively, does not pay-off. BNL performs well for small skylines, but its cost increases fast with the skyline size (e.g., anti-correlated datasets, high dimensionality) due to the large number of iterations that must be performed.

3. Branch and Bound Skyline Algorithm

Despite its performance advantages compared to previous skyline algorithms, NN has some serious shortcomings, which are presented in Section 3.1. Then, Section 3.2 describes BBS and Section 3.3 illustrates its IO optimality.

3.1 Motivation

A recursive call of the NN algorithm terminates when the corresponding nearest neighbor query does not retrieve any point within the corresponding space. Lets call such a query *empty*, to distinguish it from *non-empty* queries that return results, each spawning d new recursive applications of the algorithm (where d is the dimensionality of the data space). Figure 3.1 shows a query processing tree, where empty queries are illustrated as transparent cycles. For the second level of recursion, for instance, the second query does not return any results, in which case the recursion will not proceed further.

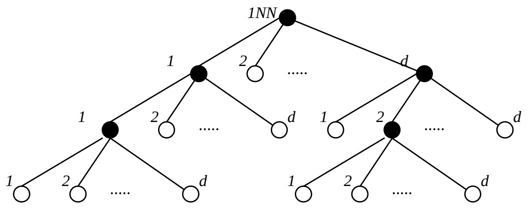


Figure 3.1: Recursion tree

Some of the non-empty queries may be *redundant*, meaning that they return skyline points already found by previous queries. Let s be the number of skyline points in the result, e the number of empty queries, ne the number of non-empty ones, and r the number of redundant queries. Since every non-empty query either retrieves a skyline point, or it is redundant, then $ne=s+r$. Furthermore, the number of empty queries in Figure 3.1 equals the number of leaf nodes in the recursion tree, i.e., $e = ne\cdot(d\text{-}1)+1$. By combining the two equations we get $e=(s+r)\cdot(d\text{-}1)+1$. Each query must traverse a whole path from the root to the leaf level of the R-tree before it terminates; therefore, its IO cost is at least h node accesses, where h is the height of the tree.

Summarizing the above observations, the total number of accesses for NN is: $NA_{NN} \geq (e+s+r)\cdot h = (s+r)\cdot h\cdot d+h > s\cdot h\cdot d$. The value $s\cdot h\cdot d$ is a rather optimistic lower bound since, for $d>2$, the number r of redundant queries may be very high (depending on the duplicate elimination method used), and queries normally incur more than h node accesses. On the other hand, as will be shown shortly, BBS is at least d times faster than the lower bound of NN.

Another problem of NN concerns the *to-do* list size, which can exceed that of the dataset for as low as 3 dimensions, even without considering redundant queries. Consider, for instance, a 3D uniform dataset (cardinality N) and a skyline query with the preference function[3] $f(x,y,z)=x$. The first skyline point n (n_x,n_y,n_z) has the smallest x coordinate among all data points, and adds partitions $P_x=[0,n_x)$ $[0,\infty)$ $[0,\infty)$, $P_y=[0,\infty)$ $[0,n_y)$ $[0,\infty)$, $P_z=[0,\infty)$ $[0,\infty)$ $[0,n_z)$ in the *to-do* list. Note that the NN query in P_x is empty because there is no other point whose x-coordinate is below n_x. On the other hand, the expected volume of P_y (P_z) is ½ (assuming unit axis length on all dimensions), because the nearest neighbor is decided solely on x-coordinates, and hence n_y (n_z) distributes uniformly in [0,1]. Following the same reasoning, a NN in P_y finds the second skyline point that introduces three new partitions such that one partition leads to an empty query, while the volumes of the other two are ¼. P_z is handled similarly, after which the *to-do* list contains 4 partitions with volumes ¼, and 2 empty partitions. In general, after the ith level of recursion, the *to-do* list contains 2^i partitions with volume $1/2^i$, and $2^{i\text{-}1}$ empty partitions. The algorithm terminates when $1/2^i<1/N$ (i.e., $i>\log N$) so that all partitions in the *to-do* list are empty. Assuming that the empty queries are performed at the end, the size of the *to-do* list can be obtained by summing the number e of empty queries at each recursion level i:

$$\sum_{i=1}^{\log N} 2^{i-1} = N\text{-}1$$

The implication of the above equation is that even in 3D, NN may *behave like a main-memory algorithm* (since the *to-do* list, which resides in memory, is at the same order of size as the input dataset). Using the same reasoning, for arbitrary dimensionality $d>2$, $e= \Theta\ ((d–1)^{\log N})$, i.e., the *to-do* list may become orders of magnitude larger than the dataset, which seriously limits the applicability of NN. In fact, as shown in Section 5, the algorithm does not terminate in the majority of experiments involving 4 and 5 dimensions.

3.2 Description

Like NN, BBS is also based on nearest neighbor search. Although both algorithms can be used with any data-partition method, in this paper we use R-trees due to their simplicity and popularity. The same concepts can be applied with other multi-dimensional access methods for high-dimensional spaces, where the performance of R-trees is known to deteriorate. Furthermore, as claimed in [KRR02], most applications involve up to 5 dimensions, for which R-trees are still efficient. For the following discussion, we use the set of 2D data points of Figure 1.1, organized in the R-tree of Figure 3.2 with node capacity=3. An intermediate entry e_i corresponds to the minimum bounding rectangle (MBR) of a node N_i at the lower level, while a leaf entry corresponds to a data point. Distances are computed according to L_1 norm, i.e., the *mindist* of a point equals the sum of its coordinates and the *mindist* of a MBR (i.e., intermediate entry) equals the *mindist* of its lower-left corner point.

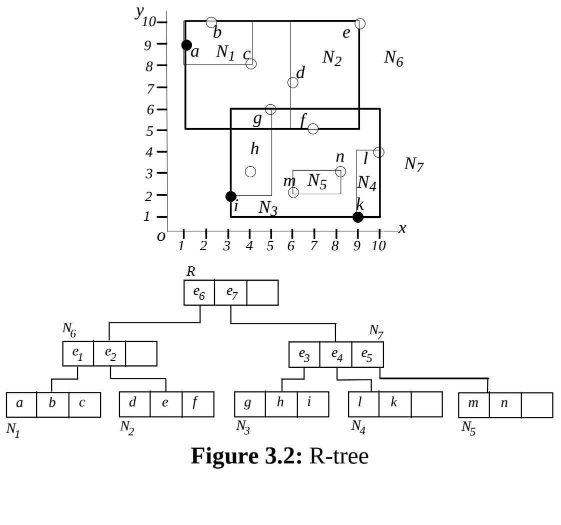


Figure 3.2: R-tree

[3] NN (and BBS) can be applied with any monotone function; the skyline points are the same, but the order that they are discovered may be different.

BBS, similar to previous algorithms for nearest neighbors [RKV95, HS99] and convex hulls [BK01], is based on the branch-and-bound paradigm. Specifically, it starts from the root node of the R-tree and inserts all its entries (e_6, e_7) in a heap sorted according to their *mindist*. Then, the entry with the minimum *mindist* (e_7) is "expanded". This expansion process removes the entry (e_7) from the heap and inserts its children (e_3, e_4, e_5). The next expanded entry is again the one with the minimum *mindist* (e_3), in which the first nearest neighbor (i) is found. This point (i) belongs to the skyline, and is inserted to the list S of skyline points.

Notice that up to this step BBS behaves like the *best-first nearest neighbor* algorithm of [HS99]. The next entry to be expanded is e_6. Although the best-first algorithm would now terminate since the *mindist* (6) of e_6 is greater than the distance (5) of the nearest neighbor (i) already found, BBS will proceed because node N_6 may contain skyline points (e.g., a). Among the children of e_6, however, only the ones that are not dominated by some point in S are inserted into the heap. In this case, e_2 is pruned because it is dominated by point i. The next entry considered (h) is also pruned because it is dominated by point i. The algorithm proceeds in the same manner until the heap becomes empty. Figure 3.3 shows the ids and the *mindist* of the entries inserted in the heap (skyline points are bold and pruned entries are shown with strikethrough fonts).

action	**heap contents**	***S***
access root	<e_7,4><e_6,6>	∅
expand e_7	<e_3,5><e_6,6><e_5,8><e_4,10>	∅
expand e_3	**<*i*,5>**<e_6,6><h,7><e_5,8> <e_4,10><g,11>	{i}
expand e_6	~~<h,7><e_5,8>~~<e_1,9><e_4,10><g,11>	{i}
expand e_1	**<*a*,10>** <e_4,10><g,11><b,12><c,12>	{i,a}
expand e_4	**<*k*,10>**~~<g,11><b,12><c,12><l,14>~~	{i,a,k}

Figure 3.3: Heap Contents

The pseudo-code for BBS is shown in Figure 3.4. Notice that an entry is checked for dominance twice: before it is inserted in the heap and before it is expanded. The second check is necessary because an entry (e.g., e_5) in the heap may become dominated by some skyline point discovered after its insertion (therefore it does not need to be visited).

```
Algorithm BBS (R-tree R)
1. S=∅ // list of skyline points
2. insert all entries of the root R in the heap
3. while heap not empty
4.  remove top entry e
5.  if e is dominated by some point in S discard e
6.    else // e is not dominated
7.          if e is an intermediate entry
8.              for each child e_i of e
9.                  if e_i is not dominated by some point in S
10.                     insert e_i into heap
11.         else // e is a data point
12.             insert e_i into S
13. end while
End BNN
```

Figure 3.4: BBS algorithm

Next we present a proof of correctness for BBS.

■ Lemma 1*:* BBS visits (leaf and intermediate) entries of an R-tree in ascending order of their distance to the origin of the axis.

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

■ Lemma 2: Any data point added to S during the execution of the algorithm is guaranteed to be a final skyline point.

Proof: Assume, on the contrary, that point p_j was added into S, but it is not a final skyline point. Then, p_j must be dominated by a (final) skyline point, say, p_i, whose coordinate on any axis is not larger than the corresponding coordinate of p_j, and at least one coordinate is smaller (since p_i and p_j are different points). This in turn means that *mindist*(p_i)< *mindist*(p_j). By Lemma 1, p_i must be visited before p_j. In other words, at the time p_j is processed, p_i must have already appeared in the skyline list, and hence p_j should be pruned, which contradicts the fact that p_j was added in the list.

■ Lemma 3: 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 skyline point are inserted into the heap and examined.

Lemmas 2 and 3 guarantee that if BBS is allowed to execute until its termination, it will correctly return all skyline points, without reporting any false hits. An important issue regards the dominance checking, which can be expensive if the skyline contains numerous points. In order to speed up this process we insert the skyline points found in a main-memory R-tree. Continuing the example of Figure 3.2, for instance, only points i, a, k will be inserted (in this order) to the main-memory R-tree. Checking for dominance can now be performed in a way similar to traditional window queries. An entry (i.e., node MBR or data point) is dominated by a skyline point p, if its lower left point falls inside the *dominance region* of p, i.e., the rectangle defined by p and the edge of the universe.

Figure 3.5 shows the dominance regions for points i, a, k and two entries; e is dominated by i and k, while e' is not dominated by any point (therefore is should be expanded). Notice that, in general, most dominance regions will cover a large part of the data space, in which case there will be significant overlap between the intermediate nodes of the main-memory R-tree. Unlike traditional window queries that must retrieve all results, this is not a problem here because we only need to retrieve a single dominance region in order to determine that the entry is dominated (by at least one skyline point).

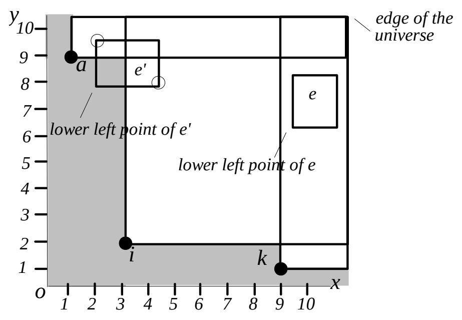


Figure 3.5: Entries of the main-memory R-tree

As a conclusion of this section we informally evaluate BBS with respect to the criteria of [HAC+99, KRR02]:

(i) *Progressiveness*: the first results should be output to the user almost instantly and the algorithm should produce more and more results the longer the execution time.

(ii) *Absence of false misses*: given enough time, the algorithm should generate the entire skyline.
(iii) *Absence of false hits*: the algorithm should not insert into *S* points that will be later replaced.
(iv) *Fairness*: the algorithm should not favor points that are particularly good in one dimension.
(v) *Incorporation of preferences*: the algorithm should allow the users to determine the order according to which skyline points are returned.
(vi) *Universality*: the algorithm should be applicable to any dataset distribution and dimensionality, using some standard index structure.

BBS satisfies property (i) as it returns skyline points instantly in ascending order of their distance to the beginning of the axes, without having to visit a large part of the R-tree. Lemma 3 ensures property (ii), since every data point is examined unless some of its ancestors is dominated (in which case the point is dominated too). Lemma 2 guarantees property (iii). Property (iv) is also fulfilled because BBS outputs points according to their *mindist*, which takes into account all dimensions. Regarding user preferences (v), as we discuss in Section 4.1, the user can specify the order of skyline points to be returned by appropriate preference functions. Furthermore, BBS also satisfies property (vi) since it does not require any specialized indexing structure, but (like NN) it can be applied with R-trees or any other data-partition method. Furthermore, the same index can be used for any subset of the *d*-dimensions that may be relevant to different users.

3.3 Analysis

In this section we first prove that BBS is IO optimal, meaning that (i) it visits only the nodes that may contain skyline points and (ii) it does not access the same node twice. Then, we provide a qualitative comparison with NN in terms of node accesses and space overhead (i.e., the heap versus the *to-do* list sizes).

Central to the analysis of BBS is the concept of *skyline search region (SSR)*, i.e., the part of the data space that may contain skyline points. Consider for instance the running example (with skyline points *i*, *a*, *k*). The *SSR* is the area (shaded in Figure 3.5) defined by the skyline and the two axes.

■ Lemma 4: Any skyline algorithm based on R-trees must access all the nodes whose MBRs intersect the *SSR*.

For instance, although entry *e'* in Figure 3.5 does not contain any skyline points, this cannot be determined unless the node of *e'* is visited.

■ Lemma 5: If an entry *e* does not intersect the *SSR*, then there is a skyline point *p* whose distance from the origin of the axes is smaller than the *mindist* of *e*.

Proof: Since *e* does not intersect the *SSR*, it must be dominated by at least a skyline point *p*, meaning that *p* dominates the lower-left corner point of *e*. This implies that the distance of *p* to the origin of the axes is smaller than the *mindist* of *e*.

■Theorem: The number of node accesses performed by BBS is optimal.

Proof: First we prove that BBS only accesses nodes that may contain skyline points. Assume, to the contrary, that the algorithm also visits an entry (let it be *e* in Figure 3.5) that does not intersect the *SSR*. Clearly, *e* should not be accessed because it cannot contain skyline points. Consider a skyline point that dominates *e* (e.g., *k*). Then, by Lemma 5, the distance of *k* to the origin is smaller than the *mindist* of *e*. According to Lemma 1, BBS visits the entries of the R-tree in ascending order of their *mindist* to the origin. 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 only 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*. ■

To quantify the actual cost of BBS, next we derive the number of node accesses for computing the entire skyline. Let $P_i(\xi, \psi)$ be the probability that the MBR of a level-*i* node intersects the rectangle with corner points (0, 0) and (ξ, ψ); then, the *node density* $D_i(\xi, \psi)$ at level-*i* is the derivative of $P_i(\xi, \psi)$, or formally $D_i(\xi,\psi) = \partial^2 P_i(\xi,\psi)/\partial\xi\partial\psi$ [TSS00]. The number NA_i of node accesses at the *i*th level (leaf nodes are at level 0) equals:

$$NA_i = \frac{N}{f^{i+1}} P_{intr-i}\text{, and } P_{intr-i} = \iint_{(x,y)\in SSR} D_i(x,y)\,dxdy$$

where *N* is the cardinality of the dataset, *f* the node fan-out (N/f^{i+1} is the total number of nodes at level *i*), and P_{intr-i} is the probability that a level-*i* node intersects the SSR. As analyzed in [TSS00], the value of $D_i(\xi, \psi)$ depends on the data density at location (ξ, ψ), i.e., the number of nodes covering point (ξ, ψ) increases with the data $\mathcal{D}(\xi, \psi)$ density around (ξ, ψ). The crucial observation is that, $\mathcal{D}(\xi, \psi)=0$ for every $(\xi, \psi) \in SSR$, because there cannot be any point in SSR (otherwise such a point would appear on the skyline). It follows that $D_i(\xi, \psi)$ is also low (but may not be zero, see [TSS00] for deriving $D_i(\xi, \psi)$ from $\mathcal{D}(\xi, \psi)$), resulting in a small NA_i. The total number NA_{BBS} of node accesses performed by BBS is the sum of accesses NA_i at each level. Similar conclusions also hold for higher dimensionality.

Assuming that each leaf node visited contains some skyline point, NA_{BBS} is below $s{\cdot}h$. This bound corresponds to a rather pessimistic case, where BBS has to access a complete path for each skyline point. Many skyline points, however, may be found in the same leaf nodes, or in the same branch of a non-leaf node (e.g., the root of the tree!), so that these nodes only need to be accessed once. Therefore, BBS is at least *d* ($=s{\cdot}h{\cdot}d$ / $s{\cdot}h$) times faster than NN. In practice, for $d>2$, the speed-up is much larger than *d* (several orders of magnitude) as $NA_{NN} = s{\cdot}h{\cdot}d$ does not take into account the number *r* of redundant queries.

Finally, we compare the memory overhead of the heap in BBS and the *to-do* list in NN. The number of entries n_{heap} in the heap is at most $(f-1){\cdot} NA_{BBS}$. This is a pessimistic upper bound, because it assumes that a node expansion removes from the heap the expanded entry and inserts all its *f* children (in practice most children will be dominated by some discovered skyline point and pruned). Since for independent dimensions the expected number of skyline points is $s=\Theta((\ln N)^{d-1}/(d-1)!)$ [B89], $n_{heap} \leq (f-1){\cdot} NA_{BBS} \approx (f-1) \cdot h \cdot s \approx (f-1){\cdot} h \cdot(\ln N)^{d-1}/(d-1)!$. For $d\geq3$ and typical values of *N* and *f* (e.g., N=100k and $f\approx100$), the heap size is much smaller that the corresponding *to-do* list size, which as discussed in Section 3.1 can be in the order of $(d-1)^{\log N}$. Furthermore, a heap entry stores d+2 numbers (i.e., entry id, *mindist*, and the coordinates of the lower-left corner), as opposed to 2*d* numbers for *to-do* list entries (i.e., *d*-dimensional ranges).

In summary, the main-memory requirement of BBS is at the same order as the size of the skyline, since both the heap and the main-memory R-tree sizes are at this order. This is a reasonable assumption because (i) skylines are normally small and (ii) previous algorithms, such as *index*, are based on the same principle. Nevertheless, specialized heap management techniques (e.g., [HS99]) can be applied for very limited memory.

4. VARIATIONS OF SKYLINE QUERIES

Next we propose novel variations of skyline queries and illustrate how BBS can be applied for their processing. In particular, Section 4.1 discusses ranked skylines, Section 4.2 constrained skyline queries, Section 4.3 dynamic skylines, and Section 4.4 enumerating and *K*-dominating queries.

4.1 Ranked skyline queries

Given a set of points in the *d*-dimensional space $[0, 1]^d$, a ranked (top-*K*) skyline query (i) specifies a parameter *K*, and a preference function *f* which is monotone on each attribute, (ii) and returns the *K* skyline points *p* that have the minimum score according to the input function. Consider the running example, where *K*=2 and the preference function is $f(x,y)=x+3y^2$. The output skyline points should be <*k*,12>, <*i*, 15> in this order (the number with each point indicates its score).

BBS can easily handle such queries by modifying the *mindist* definition to reflect the preference function (i.e., the *mindist* of a point with coordinates *x* and *y* equals $x+3y^2$). The *mindist* of an intermediate entry equals the score of its lower left point. Furthermore, the algorithm terminates after exactly *K* points have been inserted into *S*. Due to the monotonicity of *f*, it is easy to prove that the points returned are skyline points. The only change with respect to the original algorithm is the order of entries visited, which does not affect the correctness or optimality of BBS because in any case an entry will be considered after all entries that dominate it.

None of the previous skyline algorithms (see Section 2) supports ranked skyline efficiently. Specifically, BNL, D&C, *bitmap*, and the *index* methods require first retrieving the entire skyline, sorting the skyline points by their scores, and then outputting the best *K* points. On the other hand, although NN can also be used with any monotone function, its application to ranked skyline may incur almost the same cost as that of a complete skyline. This is because, due its divide-and-conquer nature, it is difficult to establish the termination criterion. If, for instance, *K*=2, NN must perform *d* queries after the first nearest neighbor (skyline point) is found, compare their results, and return the one with the minimum score. The situation is more complicated when *K* is large because the output of numerous queries must be compared.

4.2 Constrained skyline queries

Given a set of constraints, a constrained skyline query returns the most interesting points in the data space defined by the constraints. Typically, each constraint is expressed as a range along a dimension and the conjunction of all constraints forms a hyper-rectangle (referred to as the *constraint region*) in the *d*-dimensional attribute space. Consider the hotel example, where a user is interested only in hotels whose price (*y*- axis) is in the range 4-7. The skyline in this case contains points *g*, *f* and *l* (Figure 4.1), as they are the most interesting hotels in the specified range.

BBS can easily process such queries. The only difference with respect to the original algorithm is that entries not intersecting the constraint region are pruned (i.e., not inserted in the heap). Figure 4.2 shows the contents of the heap during the processing of the query in Figure 4.1. The NN algorithm can also support constrained skylines with a similar modification. In particular, the first nearest neighbor (e.g., *g*) is retrieved in the constraint region using constrained nearest neighbor search [FSAA01]. Then, each space subdivision is the intersection of the original subdivision (area to be searched by NN for the un-constrained query) and the constraint region.

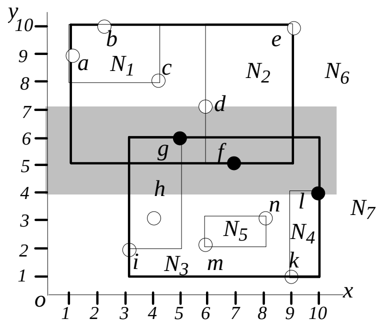


Figure 4.1: Constrained query example

action	heap contents	*S*
access root	<e_7,4><e_6,6>	∅
expand e_7	<e_3,5><e_6,6><e_4,10>	∅
expand e_3	<e_6,6> <e_4,10><*g*,11>	∅
expand e_6	<e_4,10><*g*,11><e_2,11>	∅
expand e_4	**<*g*,11>**<e_2,11><*l*,14>	{*g*}
expand e_2	**<*f*,12>**~~<*d*,13>~~**<*l*,14>**	{*g*,*f*,*l*}

Figure 4.2: Heap contents for constrained query

The *index* method can be modified for constrained skylines, by processing the batches starting from the beginning of the constraint ranges (instead of the top of the lists). Bitmap can avoid loading the juxtapositions (see Section 2.3) for points that do not satisfy the query constraints. D&C may discard, during the partitioning step, points that do not belong to the constraint region. For BNL, the only difference with respect to regular skylines is that only points in the constrained region are inserted in the self-organizing list.

4.3 Dynamic skyline queries

Assume a database containing points in *d*-dimensional space with axes $d_1, d_2, \ldots, d_d$. A dynamic skyline query specifies *m dimension functions* $f_1, f_2, \ldots, f_m$ such that each function f_i ($1 \le i \le m$) takes as parameters the coordinates of the data points along a *subset* of the *d* axes. The goal is to return the skyline in the new data space with dimensions defined by $f_1, f_2, \ldots, f_m$. Consider a database that stores the following information for each hotel: (i) its *x*-, (ii) *y*-coordinates, and (iii) its price (i.e., the database contains 3 dimensions). Then, a user specifies his/her current location (u_x,u_y), and requests the most interesting hotels, where preference must take into consideration the hotels' proximity to the user (in terms of Euclidean distance) and the price. Each point *p* with co-ordinates (p_x,p_y,p_z) in the original 3D space is transformed to a point *p'* in the 2D space with coordinates $(f_1(p_x,p_y), f_2(p_z))$, where the dimension functions f_1 and f_2 are defined as:

$$f_1\left(p_x, p_y\right)=\sqrt{\left(p_x-u_x\right)^2+\left(p_y-u_y\right)^2}, \text{ and } f_2(p_z)= p_z.$$

The terms *original* and *dynamic space* refer to the original *d*-dimensional data space and the space with computed dimensions (from $f_1, f_2, \ldots, f_m$), respectively. Correspondingly, we refer to the coordinates of a point in the original space as *original coordinates*, while to those of the point in the dynamic space as *dynamic coordinates*.

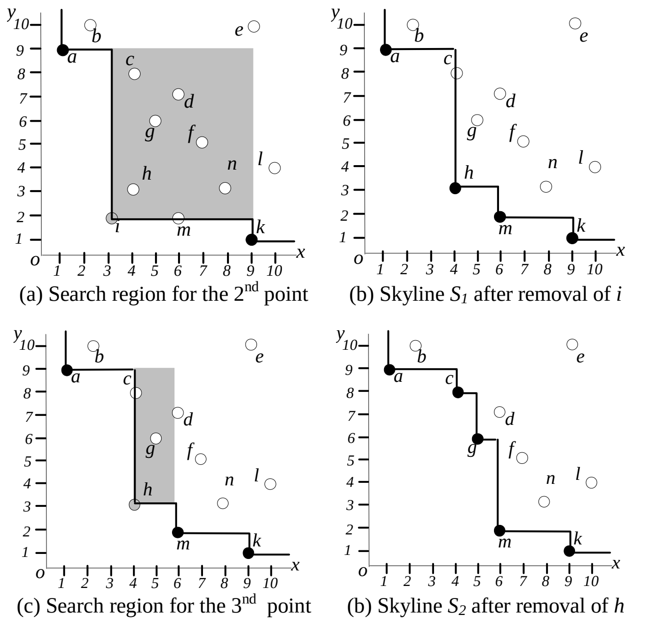
BBS is applicable to dynamic skylines by expanding entries in the heap according to their *mindist* in the dynamic space (which is computed on-the-fly when the entry is considered for the first time). In particular, the *mindist* of a leaf entry (data point) e with original coordinates (e_x,e_y,e_z), equals $\sqrt{(e_x-u_x)^2+(e_y-u_y)^2}+e_z$, and the *mindist* of an intermediate entry e whose MBR has ranges [e_{x0},e_{x1}] [e_{y0},e_{y1}] [e_{z0},e_{z1}] is computed as *mindist*([e_{x0},e_{x1}] [e_{y0},e_{y1}], (u_x,u_y))+e_{z0}, where the first term equals the mindist between point (u_x,u_y) to the 2D rectangle [e_{x0},e_{x1}] [e_{y0},e_{y1}]. Furthermore, notice that the concept of dynamic skylines can be employed in conjunction with ranked and constraint queries (i.e., find the top-5 hotels within 1km, given that the price is twice as important as the distance). BBS can process such queries by appropriate modification of the *mindist* definition (the z coordinate is multiplied by 2) and by constraining the search region ($f_1(x,y)$≤1km).

Regarding the applicability of the previous methods, BNL still applies because it evaluates every point, whose dynamic coordinates can be computed on-the-fly. D&C and NN can also be modified for dynamic queries with the transformations described above, suffering, however, similar problems of the original algorithms. *Bitmap* and *index* are not applicable because these methods rely on pre-computation, which provides little help when the dimensions are defined dynamically.

4.4 Enumerating and *K*-dominating queries

Enumerating queries return, for each skyline point p, the number of points dominated by p. This information may be relevant for some applications as it provides some measure of "goodness" for the skyline points. In the running example, for instance, hotel i, may be more interesting than the other skyline points since it dominates 9 hotels as opposed to 2 for hotels a and k. Lets call *num*(p) the number of points dominated by point p. A straightforward approach to process such queries involves two steps: (i) first compute the skyline and (ii) for each skyline point p apply a query window in the data R-tree and count the number of points *num*(p) falling inside the dominance region of p. Notice that since all (except for the skyline) points are dominated, all the nodes of the R-tree will be accessed by some query. Furthermore, due to the large size of the dominance regions, numerous R-tree nodes will be accessed by several window queries. In order to avoid multiple node visits, we apply the inverse procedure, i.e., we scan the data file and for each point we perform a query in the main-memory R-tree to find the dominance regions that contain it. The corresponding counters *num*(p) of the skyline points are then increased accordingly.

An interesting variation of the problem is the K-dominating query, which retrieves the K points that dominate the largest number of other points. Strictly speaking, this is not a skyline query, since the result does not necessarily contain skyline points. If K=3, for instance, the output should include hotels i, h and m, since *num*(i)=9, *num*(h)=7 and *num*(m)=5. In order to obtain the result, we first perform an enumerating query that returns the skyline points and the number of points that they dominate. This information for the first K=3 points is inserted into a *list* sorted according to *num*(p), i.e., *list* = <i,9>, <a,2>, <k,2>. Clearly, the first element of the *list* (point i) is the first result of the 3-dominating query. Any other point potentially in the result, should be in the dominance region of i, but not in the dominance region of a, or k (i.e., in the shaded area of Figure 4.3a); otherwise, it would dominate fewer points than a, or k. In order to retrieve the candidate points we perform a *local* skyline query S' in this region (i.e., a constrained skyline query), after removing i from S and outputting it to the user. S' contains points h and m. The new skyline S_1 = (S-{i}) ∪ S' is shown in Figure 4.3b.



(a) Search region for the 2nd point (b) Skyline S_1 after removal of i

(c) Search region for the 3rd point (b) Skyline S_2 after removal of h

Figure 4.3: Example of *3*-dominating query

Since h and m do not dominate each other, they may each dominate at most 7 points (i.e., *num*(i)-2), meaning that they are candidates for the 3-dominating query. In order to find the actual number of points dominated, we perform a window query in the data R-tree using the dominance regions of h and m as query windows. After this step, <h,7> and <m,5> replace the previous candidates <a,2>, <k,2> in the *list*. Point h is the second result of the 3-dominating query and is output to the user. Then, the process is repeated for the points that belong to the dominance region of h, but not in the dominance regions of other points in S_1 (i.e., shaded area in Figure 4.3c). The new skyline S_2 = (S_1-{h})∪{c,g} is shown in Figure 4.3d. Points c and g may dominate at most 5 points each (i.e., *num*(h)-2), meaning that they cannot outnumber m. Hence, the query terminates with <i,9> <h,7> <m,5> as the final result. In general, the algorithm can be thought of as skyline "peeling", since it computes local skylines at the points that have the largest dominance. Figure 4.4 shows the pseudo-code for K-dominating queries.

Obviously all existing algorithms can be employed for enumerating queries, since the only difference with respect to regular skylines is the second step (i.e., counting the number of points dominated by each skyline point). Actually, the *bitmap* approach can avoid scanning the actual dataset, since information about *num*(p) for each point p, can be obtained directly by appropriate juxtapositions of the bitmaps. On the other hand, K-dominating queries require an effective mechanism for skyline "peeling", i.e., discovery of skyline points in the dominance region of the last point removed from the skyline. Since this requires the application of a constrained skyline query, the relative performance of algorithms is similar to that for constrained skylines, discussed in Section 4.2.

```
Algorithm K-dominating_BBS (R-tree R, int K)
1. compute skyline S using BBS
2. for each point in S compute the number of dominated points
3. insert the top-K points of S in list sorted on num(p)
3. counter=0
4. while counter < K
5.     p = remove first entry of list
6.     output p
7.     S' = set of local skyline points in the dominance region of p
8.     if (num(p)-|S'|)> num(last element of list)
         // S' may contain candidate points
9.         for each point p' in S'
10.            find num(p') // perform a window query in data R-tree
11.            if num(p') > num(last element of list)
12.                update list // remove last element and insert p'
13.    counter=counter+1;
14. end while
End K-dominating_BBS
```

Figure 4.4: *K*-dominating_BBS algorithm

5. EXPERIMENTAL EVALUATION

In this section we verify the effectiveness and efficiency of BBS by comparing it against NN under a variety of settings. NN applies a combination of *laisser-faire* and *propagate* for duplicate elimination, since as discussed in [KRR02], it gives the best results. Specifically, only the first 20% of the *to-do list* is searched for duplicates using *propagate* and the rest of the duplicates are handled with *laisser-faire*. Following the common methodology in the literature, we employ independent (uniform) and anti-correlated datasets with dimensionality *d* in the range [2,5] and cardinality *N* in the range [100K, 10M]. Datasets are indexed by R*-trees [BKSS90] using a page size of 4Kbytes resulting in node capacities between 204 (*d*=2) and 94 (*d*=5). A Pentium 4 CPU at 2.4GHz with 512Mbytes Ram is used for all experiments.

We evaluate several factors that affect the performance of the algorithms. In particular, Sections 5.1 and 5.2 study the effect of dimensionality and cardinality, respectively. Section 5.3 compares the progressive behavior of the algorithms and, finally, Section 5.4 evaluates the performance of BBS and NN on constrained queries. We do not perform experiments with the other query types as their cost can be predicted by the presented results. In particular, the cost of a top-*K* skyline query is the same as that of a progressive query, in which BBS terminates after the first *K* points are returned. For dynamic skylines the only difference with respect to regular queries is in the computation of *mindist*. Enumerating queries, in addition to a regular skyline query, require a scan of the data file. Finally, *K*-dominating queries combine enumerating and constrained queries.

5.1 The effect of dimensionality

In order to study the effect of dimensionality we use the datasets with cardinality *N*=1M and vary *d* between 2 and 5. Figure 5.1 shows the number of node accesses as a function of dimensionality, for independent (5.1a) and anti-correlated (5.1b) datasets. Figure 5.2 illustrates a similar experiment that compares the algorithms in terms of CPU-time under the same settings. NN could not terminate successfully for *d*>4 in case of independent, and for *d*>3 in case of anti-correlated datasets due the prohibitive size of the *to-do* list (to be discussed shortly). BBS clearly outperforms NN and the difference increases fast with dimensionality. The degradation of NN is caused mainly by the growth of the number of partitions (i.e., queries), as well as the number of duplicates. The degradation of BBS is due to the growth of the skyline and the poor performance of R-trees in high dimensions. Notice that these factors also influence NN, but their effect is small compared to the inherent deficiencies of the algorithm itself. Furthermore, although the existence of an LRU buffer will reduce the node accesses of NN (BBS will not be affected since it visits every node at most once), its disadvantage compared to NN will still be very large due to the CPU overhead.

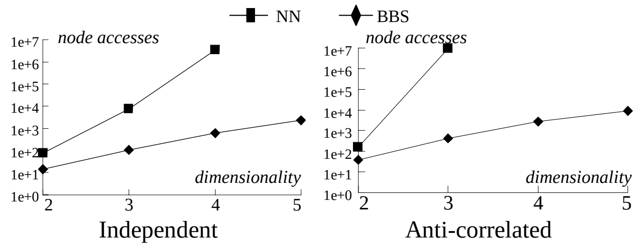


Figure 5.1: Node accesses vs. *d* (*N*=1M)

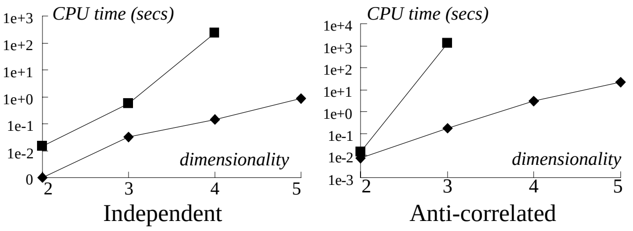


Figure 5.2: CPU-time vs. *d* (*N*=1M)

Figure 5.3 shows the maximum sizes (in Kbytes) of the heap, the *to-do* list and the dataset, as a function of dimensionality. For *d*=2, the *to-do* list is smaller than the heap, and both are negligible compared to the size of the dataset. For *d*=3, however, the *to-do* list surpasses the heap (for independent data) and the dataset (for anti-correlated data). Clearly, the maximum size of the *to-do* list exceeds the main-memory of most existing systems for *d*≥4 (anti-correlated data), which also explains the missing numbers about NN in the diagrams for high dimensions. Notice that [KRR02] report the cost of NN for returning up to the first 500 skyline points using anti-correlated data in 5 dimensions. NN can return a number of skyline points (but not the complete skyline), because the *to-do* list does not reach its maximum size until a sufficient number of skyline points have been found (and a large number of partitions have been added). This will be further discussed in Section 5.3, where we study the size of the *to-do* list as a function of the points returned.

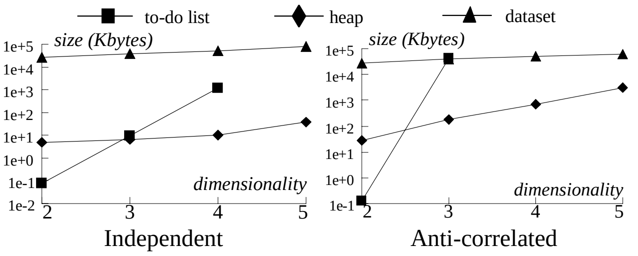


Figure 5.3: Heap and *to-do* list size vs. *d* (*N*=1M)

Figure 5.4 compares the CPU-time (as a function of *d*) of BBS using main-memory R-trees and an alternative implementation that exhaustively scans the list of current skyline points to determine if an entry is dominated. The gain of R-trees increases with the dimensionality and is higher for anti-correlated data, because in both cases the number of skyline points (and dominance checks) increases.

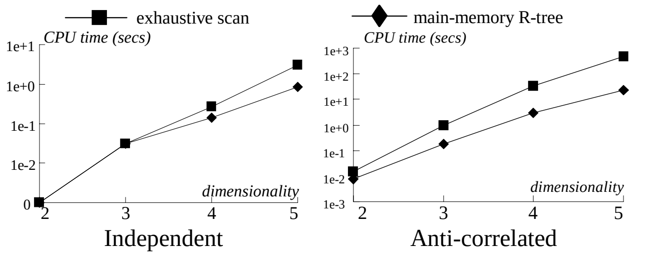


Figure 5.4: Main-memory R-tree gains vs. *d* (*N*=1M)

5.2 The effect of cardinality

Figures 5.5 and 5.6 show the number of node accesses and CPU time, respectively, versus the cardinality for 3D datasets. Even though the effect of cardinality is not as important as that of dimensionality, in all cases BBS is several orders of magnitude faster than NN. For anti-correlated data, NN does not terminate successfully for $N \geq 5M$, again due to the prohibitive size of the *to-do* list. Some irregularities in the diagrams (a small dataset may be more expensive than a larger one) are due to the positions of the skyline points and the order in which they are discovered. If for instance, the first nearest neighbor is very close to the beginning of the axes, both BBS and NN will prune a large part of their respective search spaces (and reduce the total cost).

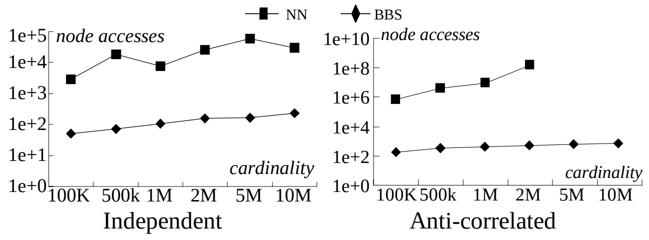


Figure 5.5: Node accesses vs. *N* (*d*=3)

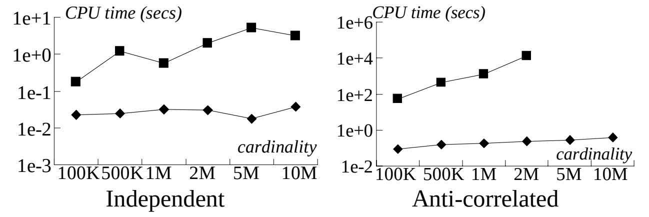


Figure 5.6: CPU-time vs. *N* (*d*=3)

5.3 Progressive behavior

Next we evaluate the speed of the algorithms in returning skyline points incrementally. Figures 5.7 and 5.8 show the node accesses and CPU time of BBS and NN as a function of the points returned for datasets with *N*=1M and *d*=3 (the number of points in the final skyline is 119 and 977, for independent and anti-correlated datasets, respectively). Both algorithms return the first point with the same cost (since they both apply nearest neighbor search to locate it). Then, BBS starts to gradually outperform NN and the difference increases with the number of points returned.

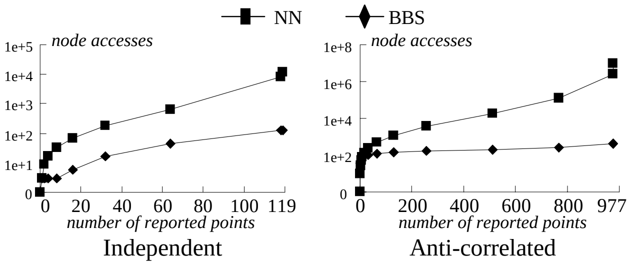


Figure 5.7: Node accesses vs. # points returned (*N*=1M, *d*=3)

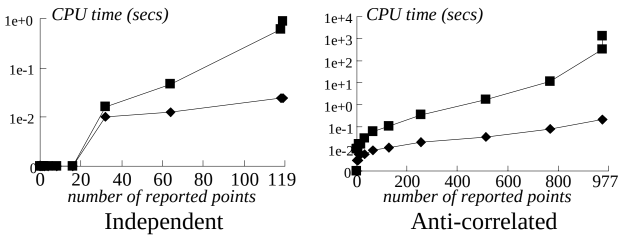


Figure 5.8: CPU-time vs. # points returned (*N*=1M, *d*=3)

Figure 5.9 presents an interesting experiment that compares the sizes of the heap and *to-do* lists as a function of the points returned. The heap reaches its maximum size at the beginning of BBS, whereas the *to-do* list towards the end of NN. This happens because before BBS discovers the first skyline point, it inserts all the entries of the visited nodes in the heap (since no entry can be pruned by existing skyline points). The more skyline points are discovered, the more heap entries are pruned, until the heap eventually becomes empty. On the other hand, the *to-do* list size is dominated by empty queries, which occur towards the late phases NN when the space subdivisions become too small to contain any points. Thus, NN could still be used to return a number of skyline points (but not the complete skyline) even for relatively high dimensionality.

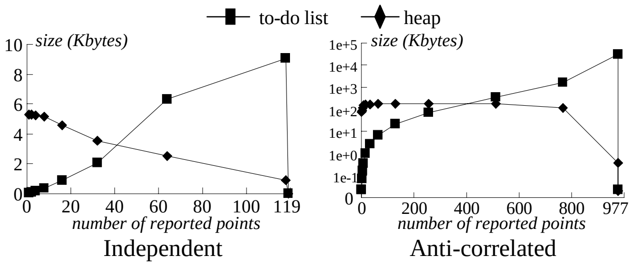


Figure 5.9: Heap, *to-do* list vs. # points returned (*N*=1M, *d*=3)

5.4 Constrained skyline queries

Finally, we present a comparison between BBS and NN on constrained skyline queries. Figure 5.10 shows the node accesses of BBS and NN as a function of the constraint region volume (*N*=1M, *d*=3), which is measured as a percentage of the volume of the data universe. The locations of constraint regions are uniformly generated and the results are computed by taking the average of 50 queries. Again BBS is several orders of magnitude faster than NN (similar results are obtained for CPU-time). The counter-intuitive observation here is that constrained queries are usually more expensive than regular skylines. To verify this consider Figure 5.11a that illustrates the node accesses of BBS on independent data, when the volume of the constraint region ranges between 98% and 100% (i.e., regular skyline). Even a range very close to 100% is much more expensive than a regular query. This can be explained by the skyline search region (*SSR*). As discussed in Section 3.3, for regular queries, the number of nodes that intersect the *SSR* (and must be visited by BBS) is very small. On the other hand, a constrained query has to visit many nodes at the boundary of the constraint region since they may all contain skyline points. Similar observations hold for anti-correlated data and NN (see Figure 5.11b).

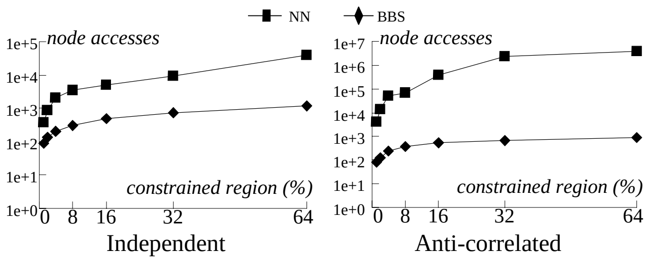


Figure 5.10: Node accesses vs. constraint region (*N*=1M, *d*=3)

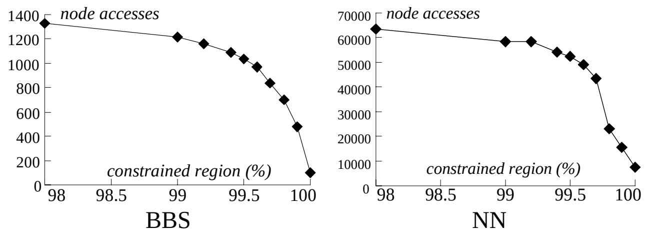


Figure 5.11: Node accesses vs. constraint region 98-100% (Independent, *N*=1M, *d*=3)

6. CONCLUSION

All existing database algorithms for skyline computation have several deficiencies, which severely limit their applicability. BNL and D&C are very sensitive to main memory size and the dataset characteristics. Furthermore, neither algorithm is progressive. *Bitmap* is applicable only for datasets with small attribute domains and cannot efficiently handle updates. *Index* (like *bitmap*) does not support user-defined preferences and cannot be used for skyline queries on a subset of the dimensions. Although NN was presented as a solution to these problems, it introduces new ones, namely poor performance and prohibitive space requirements for more than three dimensions.

We believe that BBS overcomes all these deficiencies since (i) it is efficient for both progressive and complete skyline computation, independently of the data characteristics (dimensionality, distribution), (ii) it can easily handle user preferences and process numerous alternative skyline queries (e.g., ranked, constrained skylines), (iii) it does not require any pre-computation (besides building the R-tree), (iv) it can be used for any subset of the dimensions, and (v) it has limited main-memory requirements.

Although in this implementation of BBS we used R-trees in order to perform a direct comparison with NN, the same concepts are applicable to any data-partition access method. In the future, we plan to investigate alternatives for high dimensional spaces, where R-trees are inefficient. Another interesting topic is the fast retrieval of approximate skyline points, i.e., points that do not necessarily belong to the skyline but are very "close". Finally, we want to explore new variations of skyline queries, in addition to the ones proposed in Section 4.

ACKNOWLEDGEMENTS

This work was supported by grants HKUST 6081/01E, HKUST 6197/02E from Hong Kong RGC and Se 553/3-1 from DFG. We would like to thank Mordecai Golin for his helpful comments.

REFERENCES

[B89] Buchta, C. On the Average Number of Maxima in a Set of Vectors. Information Proc. Letters, 33, 1989.

[BKS01] Borzsonyi, S, Kossmann, D., Stocker, K. The Skyline Operator. ICDE, 2001.

[BKSS90] Beckmann, N., Kriegel, H., Schneider, R., Seeger, B. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. SIGMOD, 1990.

[BK01] Böhm, C., Kriegel, H. Determining the Convex Hull in Large Multidimensional Databases. DAWAK, 2001.

[CBC+00] Chang, Y., Bergman, L., Castelli, V., Li, C., Lo, M., Smith, J. The Onion Technique: Indexing for Linear Optimization Queries. SIGMOD, 2000.

[F98] Fagin, R. Fuzzy Queries In Multimedia Database Systems. PODS, 1998.

[FSAA01] Ferhatosmanoglu, H., Stanoi, I., Agrawal, D., Abbadi, A. Constrained Nearest Neighbor Queries. SSTD, 2001.

[HAC+99] Hellerstein, J. Anvur, R., Chou, A., Hidber, C., Olston, C., Raman, V., Roth, T., Haas, P. Interactive Data Analysis: the Control Project. IEEE Computer, 32(8), 1999.

[HKP01] Hristidis, V., Koudas, N., Papakonstantinou, Y. PREFER: A System for the Efficient Execution of Multi-parametric Ranked Queries. SIGMOD, 2001.

[HS99] Hjaltason, G., Samet, H. Distance Browsing in Spatial Databases. ACM TODS, 24(2):265-318, 1999.

[KPL75] Kung, H., Luccio, F., Preparata, F. On Finding the Maxima of a Set of Vectors. Journal of the ACM, 22(4), 1975.

[KRR02] Kossmann, D., Ramsak, F., Rost, S. Shooting Stars in the Sky: an Online Algorithm for Skyline Queries. VLDB, 2002.

[M74] McLain D. Drawing Contours from Arbitrary Data Points. Computer Journal, 17(4), 1974.

[M91] Matousek, J. Computing Dominances in E^n. Information Processing Letters, 38(5), 1991.

[NCS+01] Natsev, A., Chang, Y., Smith, J., Li., C., Vitter. J. Supporting Incremental Join Queries on Ranked Inputs. VLDB, 2001.

[PS85] Preparata, F., Shamos, M. *Computational Geometry - An Introduction*. Springer 1985.

[RKV95] Roussopoulos, N., Kelly, S., Vincent, F. Nearest Neighbor Queries. SIGMOD, 1995.

[S86] Steuer, R. *Multiple Criteria Optimization*. Wiley, New York, 1986.

[SM88] Stojmenovic, I., Miyakawa, M. An Optimal Parallel Algorithm for Solving the Maximal Elements Problem in the Plane. Parallel Computing, 7(2), 1988.

[TEO01] Tan, K., Eng, P. Ooi, B. Efficient Progressive Skyline Computation. VLDB, 2001.

[TSS00] Theodoridis, Y., Stefanakis, E., Sellis, T. Efficient Cost Models for Spatial Queries Using R-trees. TKDE, 12(1):19-32, 2000.