# Progressive Skyline Computation in Database Systems

Dimitris Papadias
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
dimitris@cs.ust.hk

Yufei Tao
Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
taoyf@cs.cityu.edu.hk

Greg Fu
JP Morgan Chase
277 Park Avenue, New York
NY 10172-0002, USA
gregory.c.fu@jpmchase.com

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

**Abstract**

The skyline of a *d*-dimensional dataset 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 methods that can quickly return the initial results without reading the entire database. All the existing algorithms, however, have some serious shortcomings which limit their applicability in practice. In this paper we develop BBS (branch-and-bound skyline), an algorithm based on nearest neighbor search, which is I/O optimal, i.e., it performs a single access only to those nodes that may contain skyline points. BBS is simple to implement and supports all types of progressive processing (e.g., user preferences, arbitrary dimensionality etc). Furthermore, we propose several interesting variations of skyline computation, and show how BBS can be applied for their efficient processing.

To appear in **ACM-TODS**, special issue on best of SIGMOD/PODS 2003
Long version of [PTFS03]

**Keywords:** Skyline Query, Branch and Bound Algorithms, Multi-dimensional Access Methods

## 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 *a*, *i* and *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.

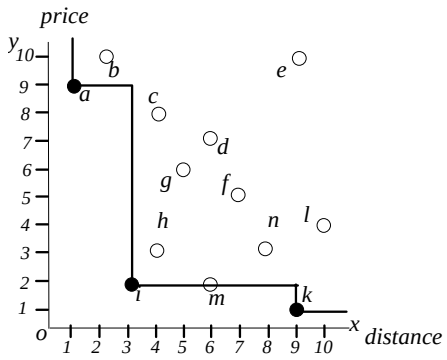



**Figure 1.1:** Example dataset and skyline

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. 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 fits in memory [PS85].

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

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*). 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 [FLN01, 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-partitioning method) and apply 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 [H94, 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, M91] and the contour problem [M74], have been extensively studied and numerous algorithms have been proposed for main-memory processing. To the best of our knowledge, however, the first work addressing skylines in the context of databases is [BKS01], which develops algorithms based on block nested loops, divide-and-conquer and index scanning. An improved version of block nested loops is presented in [CGGL03]. Tan et al. [TEO01] propose *progressive* (or *on-line*) algorithms that can output skyline points without having to scan the entire data input. Kossmann et al. [KRR02] present an 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. We experimentally and analytically show that BBS outperforms NN (usually by orders of magnitudes) both in terms of CPU and I/O 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 practical variations of skyline queries.

The rest of the paper is organized as follows: Section 2 reviews previous secondary-memory algorithms for skyline computation, discussing their advantages and limitations. Section 3 introduces BBS, providing a cost model for its expected performance and a proof of its optimality. Section 4 proposes alternative skyline queries and illustrates their processing using BBS. Section 5 introduces the concept of approximate skylines and Section 6 experimentally evaluates BBS, comparing it against NN under a variety of settings. Finally, Section 7 concludes the paper with directions for future work.

## 2. RELATED WORK

This section surveys existing secondary-memory algorithms for computing skylines, namely: (1) *divide-and-conquer,* (2) *block nested loop,* (3) *sort first skyline* (4) *bitmap*, (5) *index* and (6) *nearest neighbor*. Specifically, (1-2) are proposed in [BKS01], (3) in [CGGL03], (4-5) in [TEO01] and (6) 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 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., [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 I/O cost. Further, this approach is not suitable for on-line processing because it cannot report any skyline until the partitioning phase completes.

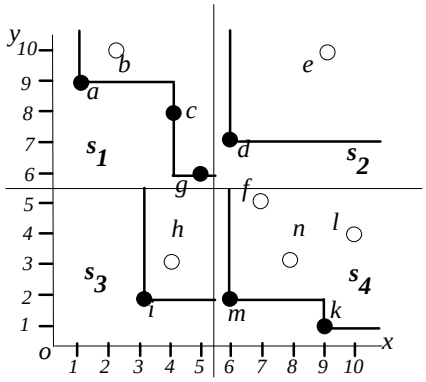



**Figure 2.1**: Divide and conquer

### 2.2 Block Nested Loop (BNL) and Sort First Skyline (SFS)

A straightforward approach to compute the skyline is to compare each point *p* with every other point, and report *p* as part of the skyline if it is not dominated. BNL builds on this concept by scanning the data file and keeping a list of candidate skyline points in main memory. At the beginning the list contains the first data point, while 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, and all points in the list dominated by *p* are dropped, and (iii) if *p* is neither dominated by, nor dominates, any point in the list it is simply inserted without dropping any point.

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 the 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 progressive processing (it has to read the entire data file before it returns the first skyline point). The SFS variation of BNL alleviates these problems by first sorting the entire dataset according to a (monotone) preference function. Candidate points are inserted into the list in ascending order of their scores, because points with lower scores are likely to dominate a large number of points, thus rendering the pruning more effective. SFS exhibits progressive behavior because the pre-sorting ensures that a point *p* dominating another *p'* must be visited before *p'*; hence we can immediately output the points inserted to the list as skyline points. Nevertheless, SFS has to

scan the entire data file to return a complete skyline, because even a skyline point may have very large score and thus appear at the end of the sorted list (e.g., in Figure 1.1, point *a* has the 3rd largest score for the preference function 0·*distance*+1·*price*). Another problem of SFS (and BNL) is that the order that the skyline points are reported is fixed (and decided by the sort order), while as discussed in Section 2.6, a progressive skyline algorithm should be able to report points according to user-specified scoring functions.

### 2.3 *Bitmap*

This technique encodes in bitmaps all the information needed to decide whether a point is in the skyline. Towards this, 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 $i$-th 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 $i$-th axis; then, it is represented by $k_i$ bits, where the leftmost $(k_i - j_i + 1)$ 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) | (1111111111, 11**0**0000000) |
| *b* | (2,10) | (1111111110, 10**0**0000000) |
| *c* | (4,8) | (1111111000, 11**1**0000000) |
| *d* | (6,7) | (1111100000, 11**1**1000000) |
| *e* | (9,10) | (1100000000, 10**0**0000000) |
| *f* | (7,5) | (1111000000, 11**1**1110000) |
| *g* | (5,6) | (1111110000, 11**1**1100000) |
| *h* | (4,3) | (1111111000, 11**1**1111100) |
| *i* | (3,2) | (1111111100, 11**1**1111110) |
| *k* | (9,1) | (1100000000, 11**1**1111111) |
| *l* | (10,4) | (1000000000, 11**1**1111000) |
| *m* | (6,2) | (1111100000, 11**1**1111110) |
| *n* | (8,3) | (1110000000, 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 rightmost bits equal to 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.

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, as with BNL and SFS, it cannot adapt to different user preferences. 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 $p = (p_1, p_2, \ldots, p_d)$ is assigned to the *i*-th list ($1 \le i \le d$), if and only if its coordinate $p_i$ on the *i*-th 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 *i*-th list consists of points that have the same *i*-th 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

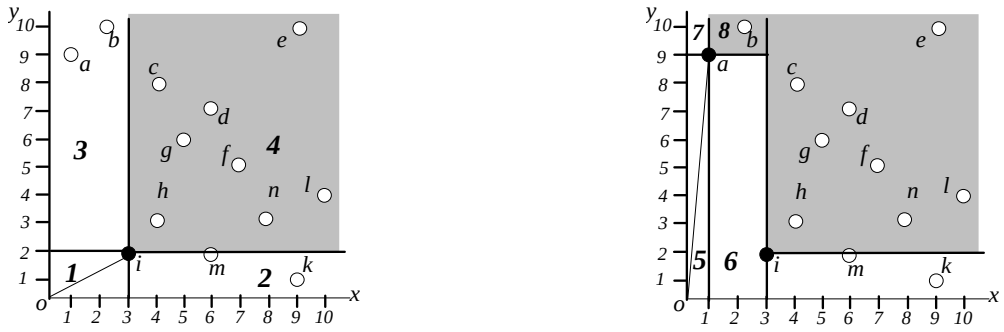
[2] The result of "&" will contain several 1's if multiple skyline points coincide. This case can be handled with an additional "*or*" operation [TEO01].

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, the order that the skyline points are returned is fixed, not supporting user-defined preferences. Furthermore, as indicated in [KRR02], the lists computed for *d* dimensions cannot be used to retrieve the skyline on any subset of the dimensions because the list that an element belongs to may change according the subset of selected dimensions. In general, for supporting queries on arbitrary dimensions, 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 dataset of Figure 1.1, which is indexed by an R-tree [G84, SRF87, BKSS90]. 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[3], 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,∞) and (ii) [0,∞) [0,$i_y$). In Figure 2.2a, the first partition contains subdivisions 1 and 3, while the second one, subdivisions 1 and 2.



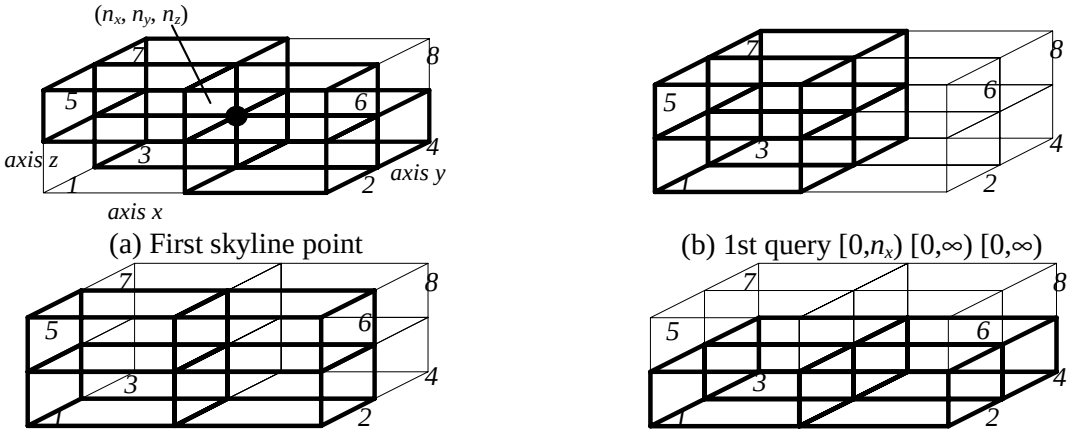

(a) Discovery of point *i* (b) Discovery of point *a*

**Figure 2.2:** Example of NN

[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.

The 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 5 and 7 in Figure 2.2b) and $[0,i_x)$ $[0,a_y)$ (subdivisions 5 and 6 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, a new skyline point causes *d* recursive applications of NN. In particular, each co-ordinate of the discovered point splits the corresponding axis, introducing a new search region to towards the origin of the axis.

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). 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.




(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

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 I/O 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 *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* was unacceptable, while *fine grain partitioning* was not implemented due to the false hits problem. *Propagate* was significantly more efficient, but the best results were achieved by a *hybrid* method combining *propagate* and *laisser-faire*.

### 2.6 Discussion about the existing algorithms

We summarize this section with a comparison of the existing methods, based on the experiments of [TEO01, KRR02, CGGL03]. Tan et al. [TEO01] examine BNL, D&C, *bitmap*, *index*, and suggest that *index* is the fastest algorithm for producing the entire skyline under all settings. D&C and *bitmap* are not favored by correlated datasets (where the skyline is small) as the overhead of partition-merging and bitmap-loading, respectively, does not pay-off. BNL performs well for small skylines, but its cost increases fast with the skyline size (e.g., for anti-correlated datasets, high dimensionality, etc.) due to the large number of iterations that must be performed. [TEO01] also shows that *index* has the best performance in returning skyline points progressively, followed by *bitmap*. The experiments of [CGGL03] demonstrate that SFS is in most cases faster than BNL without, however, comparing it with other algorithms. According to the evaluation of [KRR02], NN returns the entire skyline faster than *index* (hence also faster than BNL, D&C and *bitmap*) for up to 4 dimensions, and their difference increases (sometimes to orders of magnitudes) with the skyline size. Although *index* can produce the first few skyline points in shorter time, these points are not representative of the whole skyline (as they are good on only one axis while having large coordinates on the others).

Kossmann et al. [KRR02] also suggest a set of criteria (adopted from [HAC+99]) for evaluating the behavior and applicability of progressive skyline algorithms:

(i) *Progressiveness*: the first results should be reported to the user almost instantly and the output size should gradually increase.

(ii) *Absence of false misses*: given enough time, the algorithm should generate the entire skyline.

(iii) *Absence of false hits*: the algorithm should not discover temporary skyline 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 users should be able to determine the order according to which skyline points are reported.

(vi) *Universality*: the algorithm should be applicable to any dataset distribution and dimensionality, using some standard index structure.

All methods satisfy criterion (ii), as they deal with exact (as opposed to approximate) skyline computation. Criteria (i) and (iii) are violated by D&C and BNL since they require at least a scan of the data file before reporting skyline points and they both insert points (in partial skylines or the self-organizing list) that are later removed. Furthermore, SFS and *bitmap* need to read the entire file before termination, while *index* and NN can terminate as soon as all skyline points are discovered. Criteria (iv) and (vi) are violated by *index* because it outputs the points according to their minimum coordinate in some dimension and cannot handle skylines in some subset of the original dimensionality. All algorithms, except NN, defy criterion (v); NN can incorporate preferences by simply changing the distance definition according to the input scoring function.

Finally, note that progressive behavior requires some form of pre-processing, i.e., index creation (*index*, NN), sorting (SFS) or bitmap creation (*bitmap*). This pre-processing is a one-time effort since it can be used by all subsequent queries provided that the corresponding structure is updateable in the presence of record insertions and deletions. The maintenance of the sorted list in SFS can be performed by building a B+-tree on top of the list. The insertion of a record in *index* simply adds the record in the list that corresponds to its minimum co-ordinate; similarly, deletion removes the record from the list. NN can also be updated incrementally as it is based on a fully dynamic structure (i.e., the R-tree). On the other hand, *bitmap* is aimed at static datasets because a record insertion/deletion may alter the bitmap representation of numerous (in the worst case, of all) records.

## 3. BRANCH AND BOUND SKYLINE ALGORITHM

Despite its general applicability and performance advantages compared to existing skyline algorithms, NN has some serious shortcomings, which are described in Section 3.1. Then, Section 3.2 proposes the

branch and bound skyline (BBS) algorithm and proves its correctness. Section 3.3 analyzes the performance of BBS and illustrates its I/O optimality. Finally, Section 3.4 discusses the incremental maintenance of skylines in the presence of database updates.

### 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. 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 is redundant, we have $ne=s+r$. Furthermore, the number of empty queries in Figure 3.1 equals the number of leaf nodes in the recursion tree, i.e., $e=ne\cdot(d-1)+1$. By combining the two equations we get $e=(s+r)\cdot(d-1)+1$. Each query must traverse a whole path from the root to the leaf level of the R-tree before it terminates; therefore, its I/O 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.

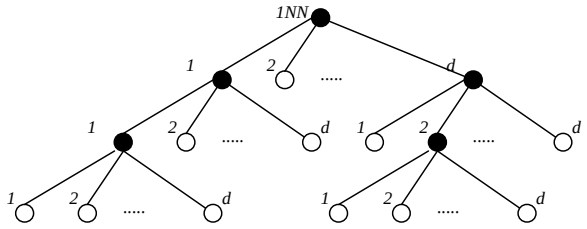



**Figure 3.1:** Recursion tree

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. Assume, for instance, a 3D uniform dataset (cardinality $N$) and a skyline query with the preference function $f(x,y,z)=x$. 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 $i$-th level of recursion, the *to-do* list contains $2^i$ partitions with volume $1/2^i$, and $2^{i-1}$ 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 6, the algorithm does not terminate in the majority of experiments involving 4 and 5 dimensions.

### 3.2 Description of BBS

Like NN, BBS is also based on nearest neighbor search. Although both algorithms can be used with any data-partitioning 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.

BBS, similar to the previous algorithms for nearest neighbors [RKV95, HS99] and convex hulls [BK01], adopts 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 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.

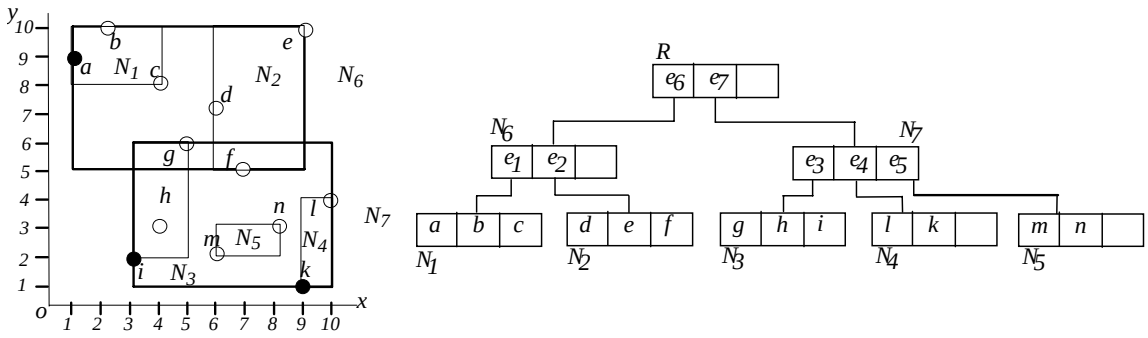



Figure 3.2: R-tree

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 nearest neighbor 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 as it also 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).

Next we prove the 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.

```
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 insert e_i into heap
10.                   else // e is a data point
11.                       insert e_i into S
12. end while
End BBS
```

**Figure 3.4:** BBS algorithm

■ 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).

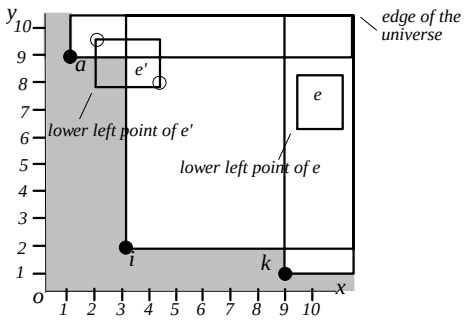



**Figure 3.5:** Entries of the main-memory R-tree

To conclude this section, we informally evaluate BBS with respect to the criteria of [HAC+99, KRR02], presented in Section 2.6. BBS satisfies property (i) as it returns skyline points instantly in ascending order of their distance to the origin, 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-partitioning 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 of BBS

In this section we first prove that BBS is I/O 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 theoretical comparison with NN in terms of the number of node accesses and memory consumption (i.e., the heap versus the *to-do* list sizes). Central to the analysis of BBS is the concept of the *skyline search region* (*SSR*), i.e., the part of the data space that is not dominated by any skyline point. Consider for instance the running example (with skyline points *i*, *a*, *k*). The *SSR* is the shaded area in Figure 3.5 defined by the skyline and the two axes. We start with the following observation.

■ 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 child 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 one skyline point $p$, meaning that $p$ dominates the lower-left corner of $e$. This implies that the distance of $p$ to the origin 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 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*. ■

Assuming that each leaf node visited contains exactly a skyline point, the number $NA_{BBS}$ of node accesses performed by BBS is at most $s \cdot h$ (where $s$ is the number of skyline points, and $h$ the height of the R-tree). 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 (our experiments show that in most cases the number of node accesses at each level of the tree is much smaller than $s$). Therefore, BBS is at least $d$ ($=s \cdot h \cdot d \,/\, s \cdot h$) times faster than NN (as explained in Section 3.1, the cost $NA_{NN}$ of NN is at least $s \cdot h \cdot d$). 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.

Regarding the memory overhead, the number of entries $n_{heap}$ in the heap of BBS 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 $d \geq 3$ and typical values of $N$ and $f$ (e.g., $N=10^5$ and $f \approx 100$), the heap size is much smaller than 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, the size of the heap can be further reduced at the expense of some CPU overhead. Consider that in Figure 3.6 intermediate node $e$ is visited first and its children (e.g., $e_1$) are inserted into the heap. When $e'$ is visited afterwards ($e$ and $e'$ have the same *mindist*), $e'_1$ can be immediately pruned, because there must exist at least a (not yet discovered) point in the bottom edge of $e_1$ that dominates $e'_1$. A similar situation happens if node $e'$ is accessed first. In this case $e'_1$ is inserted into the heap, but it is removed (before its expansion) when $e_1$ is added. BBS can easily incorporate this mechanism by checking the contents of the heap before the insertion of an entry $e$: (i) all entries dominated by $e$ are removed (ii) if $e$ is dominated by some entry, it is not inserted. We chose not to implement this optimization because it induces some CPU overhead without affecting the number of node accesses, which is optimal (in the above example $e'_1$ would be pruned during its expansion since by that time $e_1$ will have been visited).

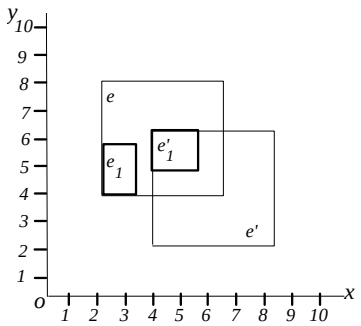
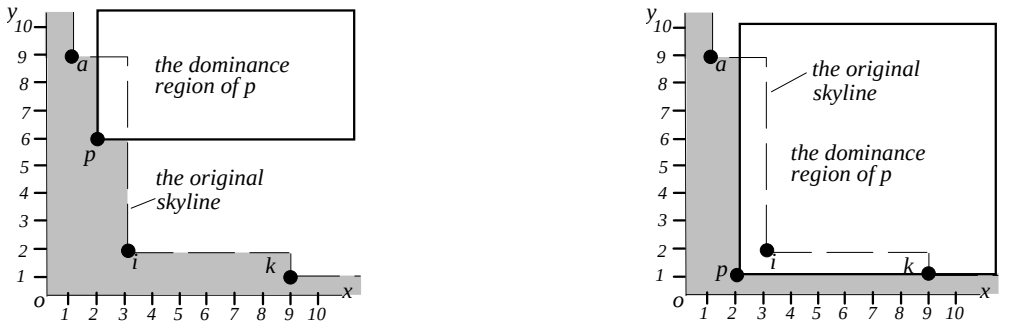



**Figure 3.6:** Reducing the size of the heap

### 3.4 Incremental maintenance of the skyline

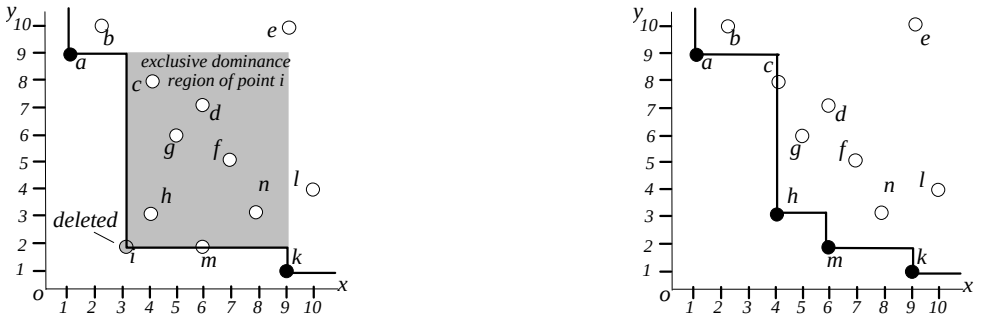
The skyline may change due to subsequent updates (i.e., insertions and deletions) to the database, and hence should be incrementally maintained to avoid re-computation. Given a new point $p$ (e.g., a hotel added to the database) our incremental maintenance algorithm first performs a dominance check on the main-memory R-tree. If $p$ is dominated (by an existing skyline point), it is simply discarded (i.e., it does not affect the skyline); otherwise, BBS performs a window query (on the main-memory R-tree) using the dominance region of $p$, to retrieve the skyline points that will become *obsolete* (i.e., those dominated by $p$). This query may not retrieve anything (e.g., Figure 3.7a), in which case the number of skyline points increases by one. Figure 3.7b shows another case, where the dominance region of $p$ covers two points $i$, $k$, which are removed (from the main-memory R-tree). The final skyline consists of only points $a$, $p$.




(a) Skyline cardinality increases (b) Skyline cardinality decreases

**Figure 3.7**: Incremental skyline maintenance for insertion

Handling deletions is more complex. First, if the point removed is not in the skyline (which can be easily checked by the main-memory R-tree using the point's coordinates), no further processing is necessary. Otherwise, part of the skyline must be re-constructed. To illustrate this, assume that point *i* in Figure 3.8a is deleted. For incremental maintenance, we need to compute the skyline with respect only to the points in the *constrained* (shaded) area, which is the region *exclusively* dominated by *i* (i.e., not including areas dominated by other skyline points). This is because, points (e.g., *e*, *l*) outside the shaded area cannot appear in the new skyline, as they are dominated by at least one other point (i.e., *a* or *k*). As shown in Figure 3.8b, the skyline within the exclusive dominance region of *i* contains two points *h* and *m*, which substitute *i* in the final skyline (of the whole dataset). In Section 4.1, we discuss skyline computation in a constrained region of the data space.




(a) Finding a skyline within a region (b) The skyline after removing *i*

**Figure 3.8**: Incremental skyline maintenance for deletion

Except for the above case of deletion, incremental skyline maintenance involves only main-memory operations. Given that the skyline points constitute only a small fraction of the database, the probability of deleting a skyline point is expected to be very low. In extreme cases (e.g., bulk updates, large number of skyline points) where insertions/deletions frequently affect the skyline, we may adopt the following "lazy" strategy to minimize the number of disk accesses: after deleting a skyline point *p*, we do not compute the constrained skyline immediately, but add *p* to a buffer. For each subsequent insertion, if *p* is dominated by a new point *p'*, we remove it from the buffer because all the points potentially replacing *p* would become obsolete anyway as they are dominated by *p'* (the insertion of *p'* may also render other

skyline points obsolete). When there are no more updates or a user issues a skyline query, we perform a *single* constrained skyline search, setting the constraint region to the *union* of the exclusive dominance regions of the remaining points in the buffer, which is emptied afterwards.

## 4. VARIATIONS OF SKYLINE QUERIES

In this section we propose novel variations of skyline search, and illustrate how BBS can be applied for their processing. In particular, Section 4.1 discusses constrained skylines, Section 4.2 ranked skylines, Section 4.3 group-by skylines, Section 4.4 dynamic skylines, Section 4.5 enumerating and *K*-dominating queries, and Section 4.6 skybands.

### 4.1 Constrained skyline

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 price range. Note that *d* (which also satisfies the constraints) is not included as it is dominated by *g*. The constrained query can be expressed using the syntax of [BKS01] and the *where* clause: *Select* *, *From* Hotels, *Where* Price∈[4,7], *Skyline* of Price *min*, Distance *min*. In addition, constrained queries are useful for incremental maintenance of the skyline in the presence of deletions (as discussed in Section 3.4).

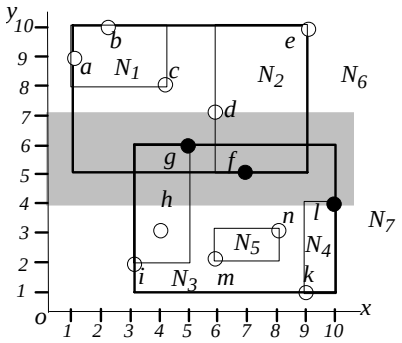



**Figure 4.1:** Constrained query example

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 same concept can also be applied when the constraint region is not a (hyper-) rectangle, but an arbitrary area in the data-space.

| 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 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. The *index* method can benefit from the constraints, by starting with the batches at 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, and D&C may discard, during the partitioning step, points that do not belong to the constraint region. For BNL and SFS, the only difference with respect to regular skyline retrieval is that only points in the constraint region are inserted in the self-organizing list.

### 4.2 Ranked skyline

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). Such *ranked skyline queries* can be expressed using the syntax of [BKS01] combined with the *order by* and *stop after* clauses: *Select* *, *From* Hotels, *Skyline* of Price *min*, Distance *min*, *order by* Price+3·*sqr*(Distance), *stop after* 2.

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 reported. Due to the monotonicity of $f$, it is easy to prove that the output points are indeed 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 other algorithms can answer this query efficiently. Specifically, BNL, D&C, *bitmap*, and *index* (as well as SFS if the scoring function is different from the sorting one) require first retrieving the entire skyline, sorting the skyline points by their scores, and then outputting the best $K$ ones. On the other

hand, although NN can be used with all monotone functions, its application to ranked skyline may incur almost the same cost as that of a complete skyline. This is because, due to 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 where the output of numerous queries must be compared.

### 4.3 Group-by skyline

Assume that for each hotel, in addition to the *price* and *distance*, we also store its *class* (i.e., 1-star, 2-star, …, 5-star). Instead of a single skyline covering all three attributes, a user may wish to find the individual skyline in each class. Conceptually, this is equivalent to grouping the hotels by their classes, and then computing the skyline for each group; i.e., the number of skylines equals the cardinality of the group-by attribute domain. Using the syntax of [BKS01], the query can be expressed as: *Select* *, *From* Hotels, *Skyline* of Price *min*, Distance *min*, Class *diff* (i.e., the group-by attribute is specified by the keyword *diff*).

One straightforward way to support group-by skylines is to create a separate R-tree for the hotels in the same class, and then invoke BBS in each tree. Separating one attribute (i.e., class) from the others, however, would compromise the performance of queries involving all the attributes[4]. In the following, we present a variation of BBS which operates on a single R-tree that indexes all the attributes. For the above example, the algorithm (i) stores the skyline points already found for each class in a separate main-memory 2D R-tree and (ii) maintains a single heap containing all the visited entries. The difference is that the sorting key is computed based only on price and distance (i.e., excluding the group-by attribute). Whenever a data point is retrieved, we perform the dominance check at the corresponding main-memory R-tree (i.e., for its class), and insert it into the tree only if it is not dominated by any existing point.

On the other hand the dominance check for each intermediate entry $e$ (performed before its insertion into the heap, and during its expansion) is more complicated, because $e$ is likely to contain hotels of several classes (we can identify the potential classes included in $e$ by its projection on the corresponding axis). First, its MBR (i.e., a 3D box) is projected onto the price-distance plane and the lower-left corner $c$ is obtained. We need to visit $e$, only if $c$ is not dominated in some main-memory R-tree corresponding to a class covered by $e$. Consider, for instance, that the projection of $e$ on the class dimension is [2,4] (i.e., $e$ may contain only hotels with 2, 3 and 4 stars). If the lower-left point of $e$ (on the price-distance plane) is dominated in all three classes, $e$ cannot contribute any skyline point. When the number of distinct values

[4] A 3D skyline in this case should maximize the value of the *class* (e.g., given two hotels with the same *price* and *distance*, the one with more stars is preferable).

of the group-by attribute is large, the skylines may not fit in memory. In this case, we can perform the algorithm in several passes, each pass covering a number of *continuous* values. The processing cost will be higher as some nodes (e.g., the root) may be visited several times.

It is not clear how to extend NN, D&C, *index*, or *bitmap* for group-by skylines beyond the naïve approach, i.e., invoke the algorithms for every value of the group-by attribute (e.g., each time focusing on points belonging to a specific group), which, however, would lead to high processing cost. BNL and SFS can be applied in this case by maintaining separate temporary skylines for each class value (similar to the main memory R-trees of BBS).

### 4.4 Dynamic skyline

Assume a database containing points in a $d$-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 $f_i$ ($1 \leq i \leq m$) 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, for instance, 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 coordinates $(p_x,p_y,p_z)$ 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:

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

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 $\sqrt{(e_x - u_x)^2 + (e_y - u_y)^2} + e_z$. 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 ($f_1(x,y) \leq 1$km).

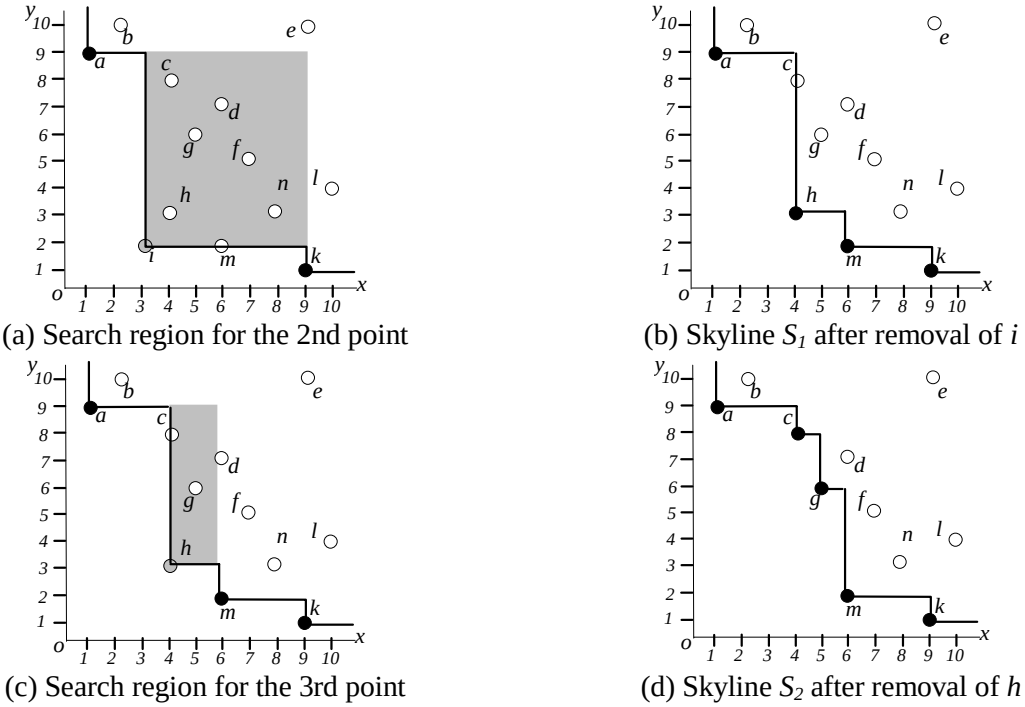
Regarding the applicability of the previous methods, BNL still applies because it evaluates every point, whose dynamic coordinates can be computed on-the-fly. The optimizations, of SFS, however, are now useless since the order of points in the dynamic space may be different from that in the original space. D&C and NN can also be modified for dynamic queries with the transformations described above, suffering, however, from the same problems as 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.

### 4.5 Enumerating and *K*-dominating queries

Enumerating queries return, for each skyline point *p*, the number of points dominated by *p*. This information 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*, with *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>. 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 (exclusive) 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 query), after removing *i* from *S* and reporting it to the user. *S'* contains points *h*

and $m$. The new skyline $S_1 = (S-\{i\}) \cup S'$ 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 (d) 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 $S_2 = (S_1-\{h\}) \cup \{c,g\}$ 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. It is worth pointing out that the exclusive dominance region of a skyline point for $d$>2 is not necessarily a hyper-rectangle (e.g., in 3D space it may correspond to an "L-shaped" polyhedron derived by removing a cube from another cube). In this case the constraint region can be represented as a union of hyper-rectangles (constrained BBS is still applicable). Furthermore, since we only care about the number of points in the dominance regions (as opposed to their ids), the performance of window queries can be improved by using aggregate R-trees [PKZT01] (or any

other multidimensional aggregate index).

```
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

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, because information about *num*(*p*) for each point *p*, can be obtained directly by appropriate juxtapositions of the bitmaps. *K*-dominating queries require an effective mechanism for skyline "peeling", i.e., discovery of skyline points in the exclusive dominance region of the last point removed from the skyline. Since this requires the application of a constrained query, all algorithms are applicable (as discussed in Section 4.1).

### 4.6 Skyband Query

Similar to *K*-nearest neighbor queries (that return the *K* NNs of a point), a *K*-skyband query reports the set of points which are dominated by at most *K* points. Conceptually, *K* represents the thickness of the skyline; the case that *K*=0 corresponds to a conventional skyline. Figure 4.5 illustrates the result of a 2-skyband query containing hotels {*a*, *b*, *c*, g, *h*, *i*, *k*, *m*}, each dominated by at most 2 other hotels.

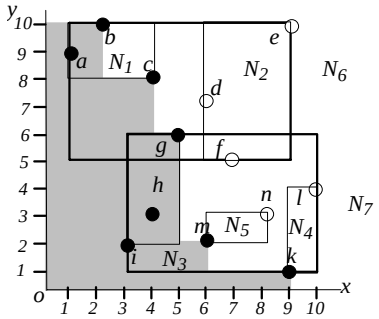



**Figure 4.5:** Example of 2-skyband query

A naïve approach to check if a point $p$ with coordinates ($p_1$, $p_2$, …, $p_d$) is in the skyband would be to perform a window query in the R-tree and count the number of points inside the range [0, $p_1$) [0, $p_2$) … [0, $p_d$). If this number is smaller than or equal to $K$, then $p$ belongs to the skyband. Obviously, the approach is very inefficient, since the number of window queries equals the cardinality of the dataset. On the other hand, BBS provides an efficient way for processing skyband queries. The only difference with respect to conventional skylines is that an entry is pruned only if it is dominated by more than $K$ discovered skyline points. Figure 4.6 shows the contents of the heap during the processing of the query in Figure 4.5. Note that the skyband points are reported in ascending order of their scores, therefore maintaining the progressiveness of the results. BNL and SFS can support $K$-skyband queries with similar modifications (i.e., insert a point in the list if it is dominated by no more than $K$ other points). None of the other algorithms is applicable, at least in an obvious way.

| 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><$e_2$,11><g,11> | {$i$,$h$} |
| expand $e_5$ | <$m$,8><$e_1$,9><$e_4$,10><$n$,11><$e_2$,11><$g$,11> | {$i$,$h$,$m$} |
| expand $e_1$ | <$a$,10><$e_4$,10><$n$,11><$e_2$,11><g,11><$b$,12><$c$,12> | {$i$,$h$,$m$,$a$} |
| expand $e_4$ | <$k$,10>~~<$n$,11><$e_2$,11>~~<$g$,11><$b$,12><$c$,12>~~<$l$,14>~~ | {$i$,$h$,$m$,$a$,$k$,$g$,$b$,$c$} |

**Figure 4.6:** Heap contents of 2-skyband query

### 4.7 Summary

Finally, we close this section with Table 4.1, which summarizes the applicability of the existing algorithms for each skyline variation. A "no" means that the technique is inapplicable, inefficient (e.g., it must perform a post-processing step on the basic algorithm), or its extension is non-trivial. Even if an algorithm (e.g., BNL) is applicable for a query type (group-by skylines), it does not necessarily imply that it is progressive (the criteria of Section 2.6 also apply to the new skyline queries). Clearly, BBS has the widest applicability since it can process all query types effectively.

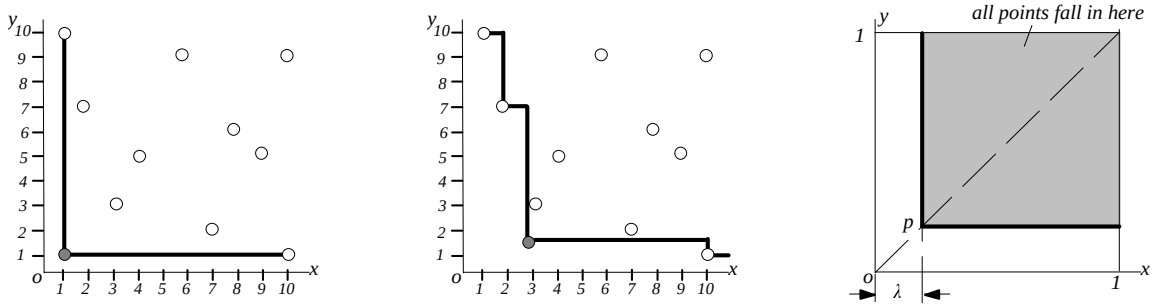
| | D&C | BNL | SFS | Bitmap | Index | NN | BBS |
|---|---|---|---|---|---|---|---|
| **Constrained** | *Yes* | *Yes* | *Yes* | *Yes* | *Yes* | *Yes* | *Yes* |
| **Ranked** | *No* | *No* | *No* | *No* | *No* | *No* | *Yes* |
| **Group-by** | *No* | *Yes* | *Yes* | *No* | *No* | *No* | *Yes* |
| **Dynamic** | *Yes* | *Yes* | *Yes* | *No* | *No* | *Yes* | *Yes* |
| ***K*-dominating** | *Yes* | *Yes* | *Yes* | *Yes* | *Yes* | *Yes* | *Yes* |
| ***K*-skyband** | *No* | *Yes* | *Yes* | *No* | *No* | *No* | *Yes* |

**Table 4.1**: Applicability comparison

## 5. APPROXIMATE SKYLINES

In this section we introduce *approximate skylines*, which can be used to provide immediate feedback to the users (i) without any node accesses (using a histogram on the dataset), or (ii) progressively, after the root visit of BBS. The problem for computing approximate skylines is that, even for uniform data, we

cannot probabilistically estimate the shape of the skyline based only on the dataset cardinality $N$. In fact, it is difficult to predict the actual number of skyline points (as opposed to their order of magnitude [B89]). To illustrate this, Figures 5.1a and 5.1b show two datasets that differ in the position of a single point, but have different skyline cardinalities (1 and 4, respectively). Thus, instead of obtaining the actual shape, we target a *hypothetical point p* such that its $x$- and $y$- coordinates are the minimum among all the expected coordinates in the dataset. We then define the approximate skyline using the two line segments enclosing the dominance region of $p$. As shown in Figure 5.1c, this approximation can be thought of as a "low resolution" skyline.




(a) Actual skyline with 1 point (b) Actual skyline with 4 points (c) Approximate skyline

**Figure 5.1**: Skylines of uniform data

Next we compute the expected coordinates of $p$. First, for uniform distribution, it is reasonable to assume that $p$ falls on the diagonal of the data space (because the data characteristics above and below the diagonal are similar). Assuming, for simplicity, that the data space has unit length on each axis, we denote the coordinates of $p$ as $(\lambda, \lambda)$ with $0 \le \lambda \le 1$. To derive the expected value for $\lambda$, we need the probability $P\{\lambda \le \xi\}$ that $\lambda$ is no larger than a specific value $\xi$. To calculate this, note that $\lambda > \xi$ implies that all the points fall in the dominance region of $(\lambda, \lambda)$ (i.e., a square with length $1-\xi$). For uniform data, a point has probability $(1-\xi)^2$ to fall in this region, and thus $P\{\lambda > \xi\}$ (i.e., the probability that all points are in this region) equals $[(1-\xi)^2]^N$. So, $P\{\lambda \le \xi\} = 1-(1-\xi)^{2N}$, and the expected value of $\lambda$ is given by:

$$E(\lambda) = \int_0^1 \xi \cdot \frac{\mathrm{d}P(\lambda \le \xi)}{\mathrm{d}\xi} \mathrm{d}\xi = 2N \int_0^1 \xi \cdot (1-\xi)^{2N-1} \mathrm{d}\xi \qquad (5\text{-}1)$$

Solving this integral, we have:

$$E(\lambda) = 1/(2N+1) \qquad (5\text{-}2)$$

Following similar derivations for $d$-dimensional spaces, we obtain $E(\lambda) = 1/(d \cdot N+1)$. If the dimensions of the data space have different lengths, then the expected coordinate of the hypothetical skyline point on dimension $i$ equals $AL_i/(d \cdot N+1)$, where $AL_i$ is the length of the axis. Based on the above analysis, we can obtain the approximate skyline for arbitrary data distribution using a multi-dimensional histogram [MD88, APR99], which typically partitions the data space into a set of *buckets* and stores for each bucket the

number (called *density*) of points in it. Figure 5.2a shows the extents of 6 buckets ($b_1$, …, $b_6$) and their densities, for the dataset of Figure 1.1. Treating each bucket as a uniform data space, we compute the hypothetical skyline point based on its density. Then, the approximate skyline of the original dataset is the skyline of all the hypothetical points, as shown in Figure 5.2b. Since the number of hypothetical points is small (at most the number of buckets), the approximate skyline can be computed using existing main-memory algorithms (e.g., [KPL75, M91]). Due to the fact that histograms are widely used for selectivity estimation and query optimization, the extraction of approximate skylines does not incur additional requirements and does not involve I/O cost.

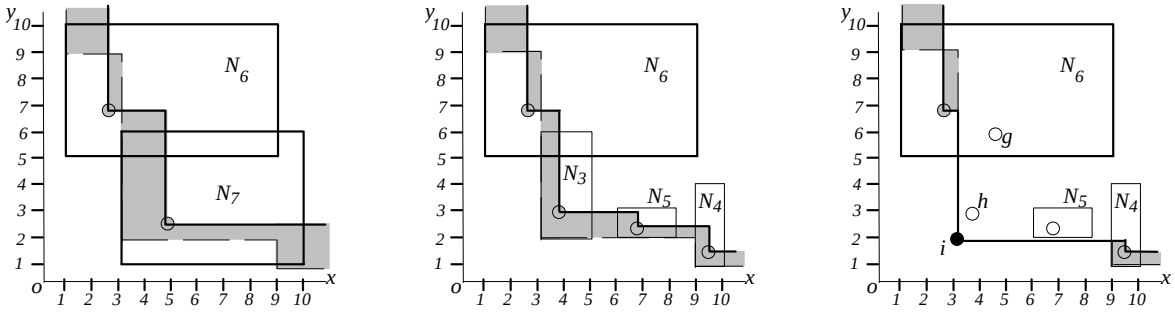

(a) Bucket information (b) Approximate skyline

**Figure 5.2**: Obtaining the approximate skyline for non-uniform data

Approximate skylines using histograms can provide some information about the actual skyline in environments (e.g., data streams, on-line processing systems) where only limited statistics of the data distribution (instead of individual data) can be maintained; thus, obtaining the exact skyline is impossible. When the actual data are available, the concept of approximate skyline, combined with BBS, enables the "drill-down" exploration of the actual one. Consider, for instance, that we want to estimate the skyline (in the absence of histograms) by performing a single node access. In this case, BBS retrieves the data R-tree root and computes by Equation 5-2, for every entry MBR, a hypothetical skyline point (i) assuming that the distribution in each MBR is almost uniform (a reasonable assumption for R-trees [TSS00]), and (ii) using the average node capacity and the tree level to estimate the number of points in the MBR. The skyline of the hypothetical points constitutes a rough estimation of the actual skyline. Figure 5.3a shows the approximate skyline after visiting the root entry as well as the real skyline (dashed line). The *approximation error* corresponds to the difference of the *SSR*s of the two skylines, i.e., the area that is dominated by exactly one skyline (shaded region in Figure 5.3a).

The approximate version of BBS maintains, in addition to the actual skyline *S*, a set *HS* consisting of points in the approximate skyline. *HS* is used just for reporting the current skyline approximation and *not* to guide the search (the order of node visits remains the same as the original algorithm). For each intermediate entry found, if its hypothetical point *p* is not dominated by any point in *HS*, it is added into the approximate skyline and all the points dominated by *p* are removed from *HS*. Leaf entries correspond

to actual data points and are also inserted in *HS* (provided that they are not dominated). When an entry is de-heaped, we remove the corresponding (hypothetical or actual) point from *HS*. If a data point is added to *S*, it is also inserted in *HS*. The approximate skyline is progressively refined as more nodes are visited, e.g., when the second node $N_7$ is de-heaped, the hypothetical point of $N_7$ is replaced with those of its children and the new *HS* is computed as shown in Figure 5.3b. Similarly, the expansion of $N_3$ will lead to the approximate skyline of Figure 5.3c. At the termination of approximate BBS, the estimated skyline coincides with the actual one. To show this, assume, on the contrary, that at the termination of the algorithm there still exists a hypothetical/actual point *p* in *HS*, which does not belong to *S*. It follows that *p* is not dominated by the actual skyline. In this case, the corresponding (intermediate or leaf) entry producing *p* should be processed, contradicting the fact that the algorithm terminates.

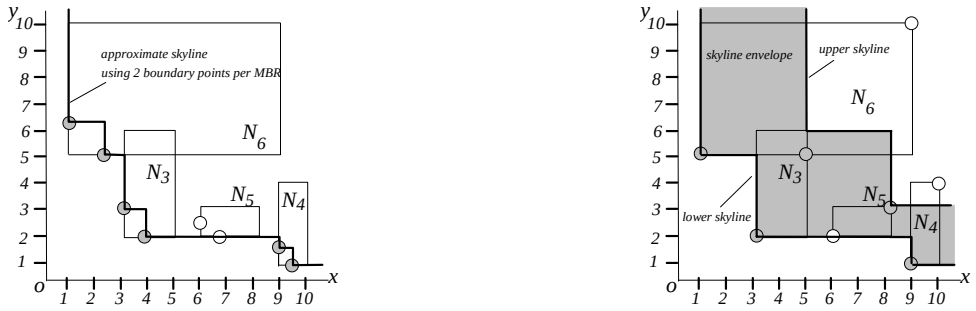



(a) After root retrieval (b) After retrieval of $N_7$ (c) After retrieval of $N_3$

**Figure 5.3**: Approximate skylines as a function of node accesses

Note that for computing the hypothetical point of each MBR we use Equation 5-2, because it (i) is simple and efficient (in terms of computation cost), (ii) provides a uniform treatment of approximate skylines (i.e., the same as in the case of histograms), and (iii) has high accuracy (as shown in Section 6.8). Nevertheless, we may derive an alternative approximation based on the fact that each MBR boundary contains a data point. Assuming a uniform distribution on the MBR projections and that no point is minimum on two different dimensions, this approximation leads to *d* hypothetical points per MBR such that the expected position of each point is $1/((d\text{-}1)\cdot N + 1)$. Figure 5.4a shows the approximate skyline in this case after the first two node visits (root and $N_7$). Alternatively, BBS can output an *envelope* enclosing the actual skyline, where the lower bound refers to the skyline obtained from the lower-left vertices of the MBRs and the upper bound refers to the skyline obtained from the upper-right vertices. Figure 5.4b illustrates the corresponding envelope (shaded region) after the first two node visits. The volume of the envelope is an upper bound for the actual approximation error, which shrinks as more nodes are accessed. The concepts of skyline approximation or envelope permit the immediate visualization of information about the skyline, enhancing the progressive behavior of BBS. In addition, approximate BBS can be easily modified for processing the query variations of Section 4 since the only difference is the maintenance of the hypothetical points in *HS* for the entries encountered by the original algorithm. The

computation of hypothetical points depends on the skyline variation, e.g., for constrained skylines the points are computed by taking into account only the node area inside the constraint region. On the other hand, the application of these concepts to NN is not possible (at least in an obvious way), because of the duplicate elimination problem and the multiple accesses to the same node(s).




(a) Approximation using $d$ (=2) points per MBR (b) Skyline envelope

**Figure 5.4**: Alternative approximations after visiting root and $N_7$

## 6. EXPERIMENTAL EVALUATION

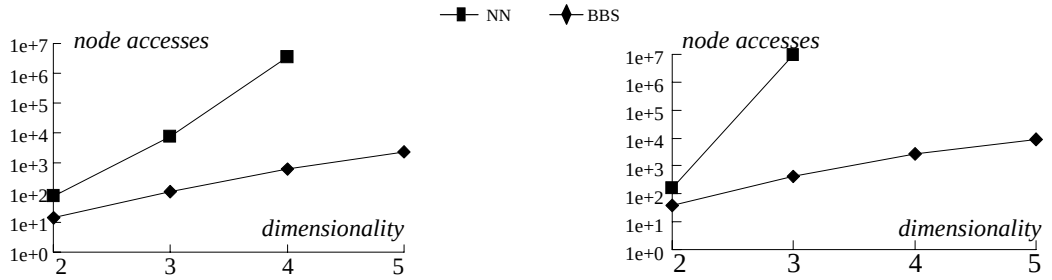
In this section we verify the effectiveness of BBS by comparing it against NN which, according to the evaluation of [KRR02], is the most efficient existing algorithm and exhibits progressive behavior. Our implementation of NN combines *laisser-faire* and *propagate* because, as discussed in Section 2.5, 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[5] datasets (generated in the same way as described in [BKS01]) with dimensionality $d$ in the range [2,5] and cardinality $N$ in the range [100K, 10M]. The length of each axis is 10,000. Datasets are indexed by R*-trees [BKSS90] with a page size of 4Kbytes resulting in node capacities between 204 ($d$=2) and 94 ($d$=5). For all experiments we measure the cost in terms of node accesses since the diagrams for CPU-time are very similar (see [PTFS03]).

Sections 6.1 and 6.2 study the effects of dimensionality and cardinality for conventional skyline queries, whereas Section 6.3 compares the progressive behavior of the algorithms. Sections 6.4, 6.5, 6.6, and 6.7 evaluate constrained, group-by skylines, $K$-dominating and $K$-skyband queries, respectively. Finally, Section 6.8 focuses on approximate skylines. Ranked queries are not included because NN is inapplicable, while the performance of BBS is the same as in the experiments for progressive behavior. Similarly, the cost of dynamic skylines is the same as that of conventional skylines in selected dimension projections and omitted from the evaluation.

[5] For anti-correlated distribution the dimensions are linearly correlated such that, if $p_i$ is smaller than $p_j$ in one axis, then $p_i$ is larger in at least one other dimension (e.g., hotels near the beach are likely to be expensive). An anti-correlated dataset has fractal dimensionality close to 1 (i.e., points lie near the anti-diagonal of the data space).

### 6.1 The effect of dimensionality

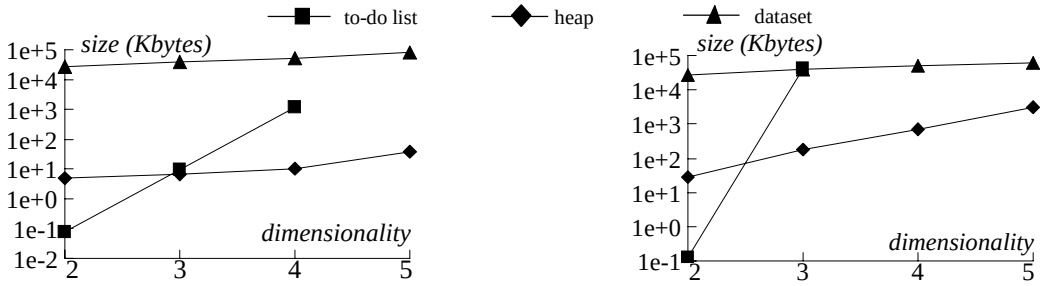
In order to study the effect of dimensionality we use the datasets with cardinality $N$=1M and vary $d$ between 2 and 5. Figure 6.1 shows the number of node accesses as a function of dimensionality, for independent (6.1a) and anti-correlated (6.1b) datasets. NN could not terminate successfully for $d$>4 in case of independent, and for $d$>3 in case of anti-correlated datasets due to 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., each skyline point spawns $d$ partitions), 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. Note that these factors also influence NN, but their effect is small compared to the inherent deficiencies of the algorithm.




(a) Independent (b) Anti-correlated

**Figure 6.1:** Node accesses vs. dimensionality $d$ ($N$=1M)

Figure 6.2 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 explains the missing numbers about NN in the diagrams for high dimensions. Notice that [KRR02] reports 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).

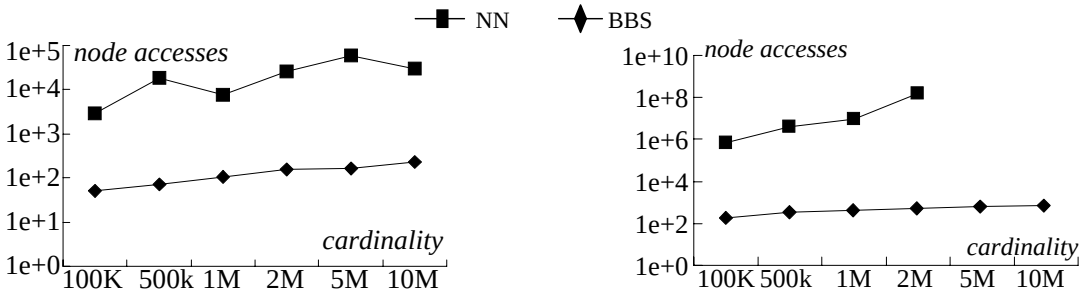



(a) Independent (b) Anti-correlated

**Figure 6.2:** Heap and *to-do* list sizes vs. dimensionality $d$ ($N$=1M)

### 6.2 The effect of cardinality

Figure 6.3 shows the number of node accesses versus the cardinality for 3D datasets. Although 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 origin of the axes, both BBS and NN will prune a large part of their respective search spaces.

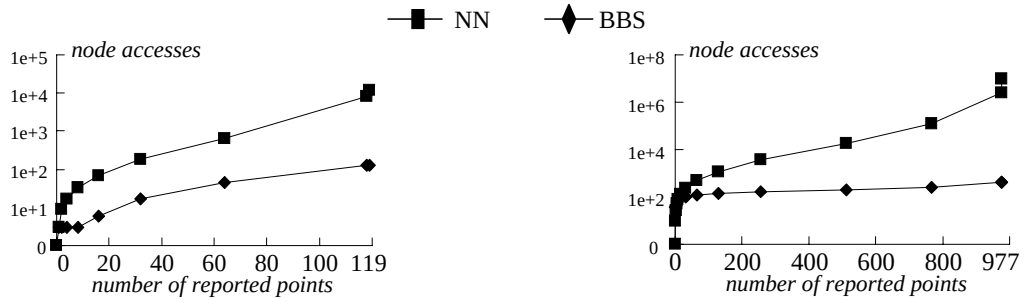



(a) Independent (b) Anti-correlated

**Figure 6.3:** Node accesses vs. cardinality $N$ ($d$=3)

### 6.3 Progressive behavior

Next we compare the speed of the algorithms in returning skyline points incrementally. Figure 6.4 shows the node accesses 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.

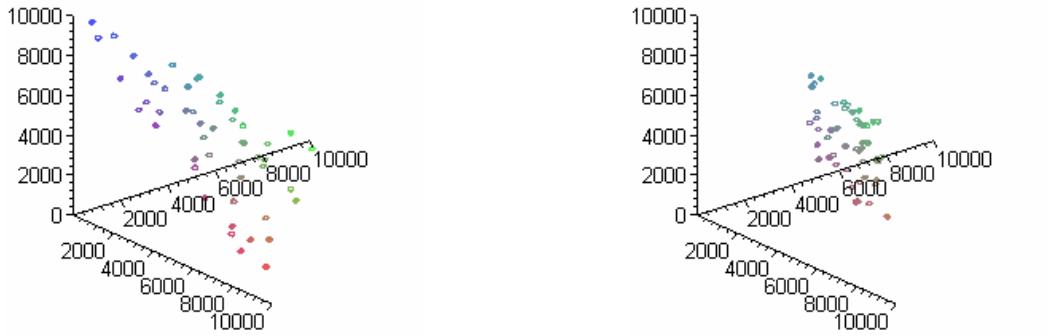



(a) Independent (b) Anti-correlated

**Figure 6.4:** Node accesses vs. number of points reported ($N$=1M, $d$=3)

To evaluate the quality of the results, Figure 6.5 shows the distribution of the first 50 skyline points (out of 977) returned by each algorithm for the anti-correlated dataset with $N$=1M and $d$=3. The initial skyline points of BBS are evenly distributed in the whole skyline, since they are discovered in the order of their

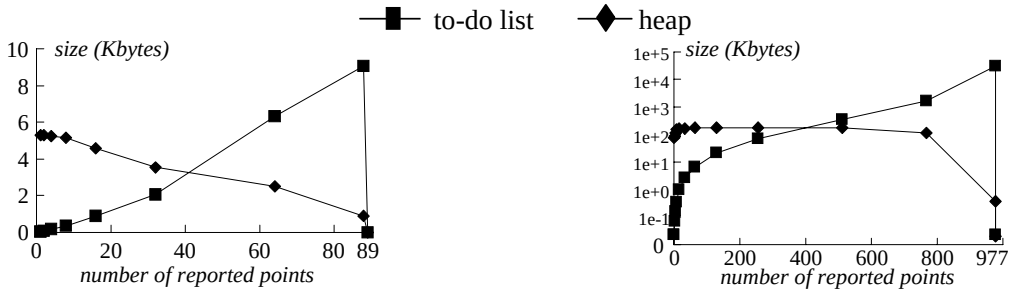
*mindist* (which is independent of the algorithm). On the other hand, NN produces points concentrated in the middle of the data universe because the partitioned regions, created by new skyline points, are inserted to the end of the *to-do* list, and thus nearby points are subsequently discovered.




(a) BBS (b) NN

**Figure 6.5:** Distribution of the first 50 skyline points (Anti-correlated, *N*=1M, *d*=3)

Figure 6.6 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 of 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.

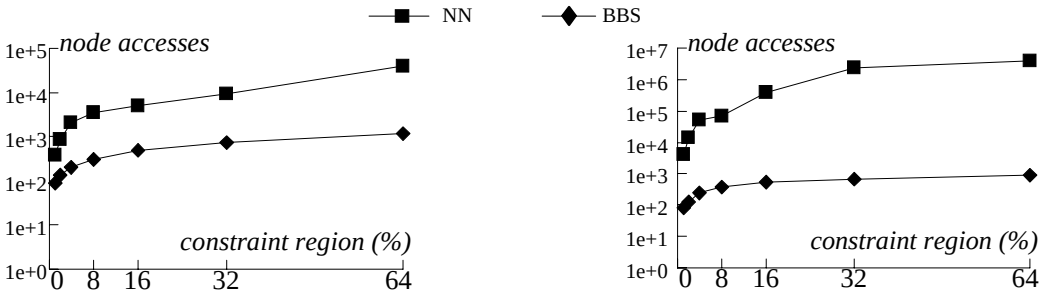



(a) Independent (b) Anti-correlated

**Figure 6.6:** Sizes of the heap and *to-do* list vs. number of points reported (*N*=1M, *d*=3)

### 6.4 Constrained skyline

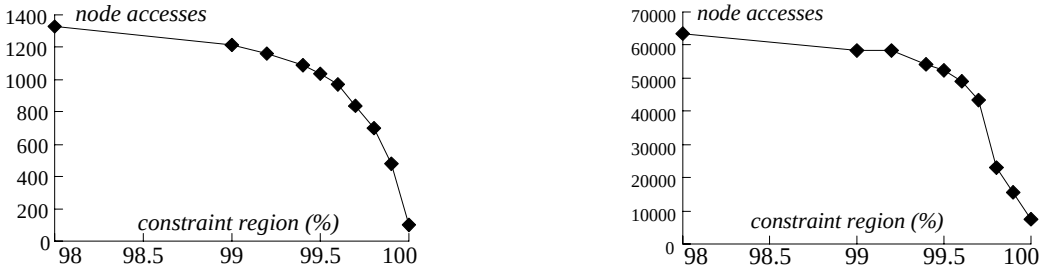
Having confirmed the efficiency of BBS for conventional skyline retrieval, we present a comparison between BBS and NN on constrained skylines. Figures 6.7 shows the node accesses of BBS and NN as a function of the constraint region volume (*N*=1M, *d*=3), which is measured as a percentage of the volume 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.




(a) Independent (b) Anti-correlated

**Figure 6.7:** Node accesses vs. volume of constraint region (*N*=1M, *d*=3)

The counter-intuitive observation here is that constraint regions covering more than 8% of the data space are usually more expensive than regular skylines. Figure 6.8a verifies the observation by illustrating 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 conventional skyline. Similar results hold for NN (see Figure 6.8b) and anti-correlated data.




(a) BBS (b) NN

**Figure 6.8:** Node accesses vs. volume of constraint region 98-100% (Independent, *N*=1M, *d*=3)

To explain this, consider Figure 6.9a that shows a skyline *S* in a constraint region. The nodes that must be visited intersect the constrained skyline search region (shaded area) defined by *S* and the constraint region. In this example, all four nodes $e_1$, $e_2$, $e_3$, $e_4$ may contain skyline points and should be accessed. On the other hand, if *S* were a conventional skyline, as in Figure 6.9b, nodes $e_2$, $e_3$ and $e_4$ could not exist because they should contain at least a point that dominates *S*. In general, the only data points of the conventional *SSR* (shaded area in Figure 6.9b) lie on the skyline implying that for any node MBR, at most one of its vertices can be inside the *SSR*. For constrained skylines there is no such restriction and the number of nodes intersecting the constrained *SSR* can be arbitrarily large.


(a) Constrained skyline (b) Conventional skyline

**Figure 6.9:** Nodes potentially intersecting the *SSR*

It is important to note that the constrained queries issued when a skyline point is removed during incremental maintenance (see Section 3.4) are always cheaper than computing the entire skyline from scratch. Consider, for instance, that the partial skyline of Figure 6.9a is computed for the exclusive dominance area of a deleted skyline point $p$ on the lower-left corner of the constraint region. In this case nodes such as $e_2$, $e_3$, $e_4$ cannot exist, because otherwise they would have to contain skyline points, contradicting the fact that the constraint region corresponds to the exclusive dominance area of $p$.

### 6.5 Group-by skyline

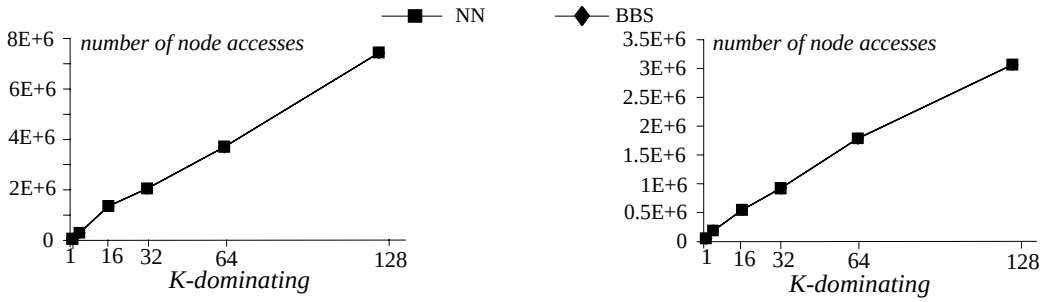
Next we consider group-by skyline retrieval, including only BBS because, as discussed in Section 4, NN is inapplicable in this case. Towards this, we generate datasets (with cardinality 1M) in a 3D space that involves two numerical dimensions and one categorical axis. In particular, the number $c_{num}$ of categories is a parameter ranging from 2 to 64 ($c_{num}$ is also the number of 2D skylines returned by a group-by skyline query). Every data point has equal probability to fall in each category, and for all the points in the same category, their distribution (on the two numerical axes) is either independent or anti-correlated. Figure 6.10 demonstrates the number of node accesses as a function of $c_{num}$. The cost of BBS increases with $c_{num}$ because the total number of skyline points (in all 2D skylines) and the probability that a node may contain qualifying points in some category (and therefore it should be expanded) is proportional to the size of the categorical domain.


(a) Independent (b) Anti-correlated

**Figure 6.10:** BBS node accesses vs. cardinality of categorical axis $c_{num}$ ($N$ =1M, $d$=3)

### 6.6 *K*-dominating skyline

This section measures the performance of NN and BBS on $K$-dominating queries. Recall that each $K$-dominating query involves an enumerating query (i.e., a file scan), which retrieves the number of points dominated by each skyline point. The $K$ skyline points with the largest counts are found and the top-1 is immediately reported. Whenever an object is reported, a constrained skyline is executed to find potential candidates in its exclusive dominance region (see Figure 4.3). For each such candidate, the number of dominated points is retrieved using a window query on the data R-tree. After this process, the object with the largest count is reported (i.e., the second best object), another constrained query is performed and so on. Therefore, the total number of constrained queries is $K$–1 and each such query may trigger multiple window queries. Figure 6.11 demonstrates the cost of BBS and NN as a function of $K$.



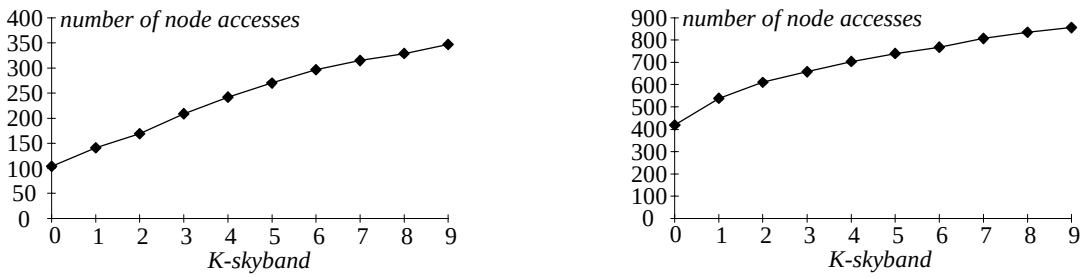

(a) Independent (b) Anti-correlated

**Figure 6.11:** Node accesses vs. number of objects to be reported for *K*-dominating queries (*N* =1M, *d*=2)

The overhead of the enumerating and (multiple) window queries dominates the total cost, and consequently BBS and NN have similar performance. Interestingly, the overhead of the anti-correlated data is lower (than the independent distribution) because each skyline point dominates fewer points (therefore, the number of window queries is smaller). The high cost of *K*-dominating queries (compared to other skyline variations) is due to the complexity of the problem itself (and not the proposed algorithm). In particular, a *K*-dominating query is similar to a semi-join and could be processed accordingly. For instance a nested-loops algorithm would (i) count, for each data point, the number of dominated points by scanning the entire database, (ii) sort all the points in descending order of the counts, and (iii) report the *K* points with the highest counts. Since in our case the database occupies more than 6k nodes, this algorithm would need to access 36E+6 nodes (for any *K*), which is significantly higher than the costs in Figure 6.11 (especially for low *K*).

### 6.7 *K*-Skyband

Next, we evaluate the performance of BBS on *K*-skyband queries (NN is inapplicable). Figure 6.12 shows the node accesses as a function of *K* ranging from 0 (conventional skyline) to 9. As expected, the performance of BBS degrades as *K* increases because a node can be pruned only if it is dominated by more than *K* discovered skyline points, which becomes more difficult for higher *K*. Furthermore, the number of skyband points is significantly larger for anti-correlated data, e.g., for *K*=9, the number is 788 (6778) in the independent (anti-correlated) case, which explains the higher costs in Figure 6.12b.



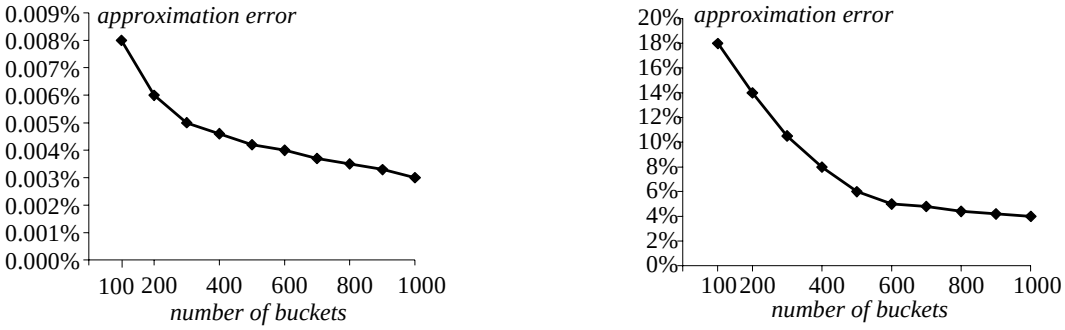

(a) Independent (b) Anti-correlated

**Figure 6.12:** BBS node accesses vs. "thickness" of the skyline for *K*-skyband queries (*N* =1M, *d*=3)

### 6.8 Approximate skylines

This section evaluates the quality of the approximate skyline, using a hypothetical point per bucket or visited node (as shown in the examples of Figures 5.2 and 5.3, respectively). Given an estimated and an actual skyline, the approximation error corresponds to their *SSR* difference (see Section 5). In order to measure this error, we use a numerical approach: (i) we first generate a large number $\alpha$ of points ($\alpha$=$10^4$) uniformly distributed in the data space, and (ii) count the number $\beta$ of points that are dominated by exactly *one* skyline. The error equals $\beta/\alpha$, which approximates the volume of the *SSR* difference divided by the volume of the entire data space. We do not use a relative error (e.g., volume of the *SSR* difference divided by the volume of the actual *SSR*) because such a definition is sensitive to the position of the actual skyline (i.e., a skyline near the origin of the axes would lead to higher error even if the *SSR* difference remains constant).

In the first experiment, we build a *minskew* [APR99] histogram on the 3D datasets by varying the number of buckets from 100 to 1000, resulting in main memory consumption in the range of 3K bytes (100) to 30K bytes (1000 buckets). Figure 6.13 illustrates the error as a function of the bucket number. For independent distribution, the error is very small (less than 0.01%) even with the smallest number of buckets because the rough "shape" of the skyline for a uniform dataset can be accurately predicted using Equation 5-2. On the other hand, anti-correlated data are skewed and require a large number of buckets for achieving high accuracy.

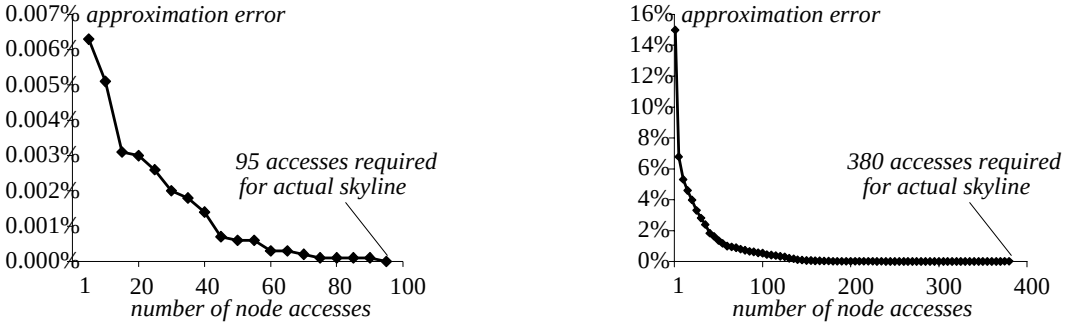



(a) Independent (b) Anti-correlated

**Figure 6.13:** Approximation error vs. number of *minskew* buckets ($N$ =1M, $d$=3)

Figure 6.14 evaluates the quality of the approximation as a function of node accesses (without using a histogram). As discussed in Section 5, the first rough estimate of the skyline is produced when BBS visits the root entry and then the approximation is refined as more nodes are accessed. For independent data, extremely accurate approximation (with error 0.01%) can be obtained immediately after retrieving the root, a phenomenon similar to that in Figure 6.13a. For anti-correlated data, the error is initially large (around 15% after the root visit), but decreases considerably with only a few additional node accesses. Particularly, the error is less than 3% after visiting 30 nodes, and close to 0 with around 100 accesses (i.e., the estimated skyline is almost identical to the actual one with about 25% of the node accesses required

for the discovery of the actual skyline).




(a) Independent (b) Anti-correlated

**Figure 6.14:** Approximation error vs. number of node accesses ($N$ =1M, $d$=3)

## 7. CONCLUSION

The importance of skyline computation in database systems increases with the number of emerging applications requiring efficient processing of preference queries and the amount of available data. Consider, for instance, a bank information system monitoring the attribute values of stock records and answering queries from multiple users. Assuming that the user scoring functions are monotonic functions, the top-1 result of all queries is always a part of the skyline. Similarly, the top-$K$ result is always a part of the $K$-skyband. Thus, the system could maintain only the skyline (or $K$-skyband) and avoid searching a potentially very large number of records. However, all existing database algorithms for skyline computation have several deficiencies, which severely limit their applicability. BNL, SFS and D&C are not progressive. *Bitmap* is applicable only for datasets with small attribute domains and cannot efficiently handle updates. *Index* cannot be used for skyline queries on a subset of the dimensions. SFS, like all above algorithms, does not support user-defined preferences. 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. This paper proposes BBS, a novel algorithm that overcomes all these shortcomings 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, approximate 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-partitioning access method. In the future, we plan to investigate alternatives (e.g., X-trees [BKK96], A-trees [SYUK00]) for high dimensional spaces, where R-trees are inefficient. Another possible solution for high dimensionality would include: (i) converting the data points to subspaces with lower dimensionalities, (ii) computing the skyline in each subspace, and

(iii) merging the partial skylines. Finally, a topic worth studying concerns skyline retrieval in other application domains. For instance, Balke et al. [BGZ04] study skyline computation for web information systems considering that the records are partitioned in several lists, each residing at a distributed server. The tuples in every list are sorted in ascending order of a scoring function, which is monotonic on all attributes. Their processing method uses the main concept of the *threshold* algorithm [FLN01] to compute the entire skyline by reading the minimum number of records in each list. Another interesting direction concerns skylines in temporal databases [ST99] that retain historical information. In this case, a query could ask for the most interesting objects at a past timestamp or interval.

## REFERENCES


[APR99] Acharya, S., Poosala, V., Ramaswamy, S. Selectivity Estimation in Spatial Databases. ACM Conference on the Management of Data (SIGMOD), 13-24, Philadelphia, PA, June 1-3, 1999.

[B89] Buchta, C. On the Average Number of Maxima in a Set of Vectors. Information Processing Letters, 33(2): 63-65, 1989.

[BGZ04] Balke, W., Gunzer, U., Zheng, J. Efficient Distributed Skylining for Web Information Systems. International Conference on Extending Database Technology (EDBT), 256-273, Heraklio, Greece, March 14-18, 2004.

[BK01] Böhm, C., Kriegel, H. Determining the Convex Hull in Large Multidimensional Databases. International Conference on Data Warehousing and Knowledge Discovery (DaWaK), 294-306, Munich, Germany, September 5-7, 2001.

[BKK96] Berchtold, S., Keim, D., Kriegel, H. The X-tree: An Index Structure for High-Dimensional Data. Very Large Data Bases Conference (VLDB), 28-39, Mumbai, India, September 3-6, 1996.

[BKS01] Borzsonyi, S, Kossmann, D., Stocker, K. The Skyline Operator. IEEE International Conference on Data Engineering (ICDE), 421-430, Heidelberg, Germany, April 2-6, 2001.

[BKSS90] Beckmann, N., Kriegel, H., Schneider, R., Seeger, B. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. ACM Conference on the Management of Data (SIGMOD), 322-331, Atlantic City, NJ, May 23-25, 1990.

[CBC+00] Chang, Y., Bergman, L., Castelli, V., Li, C., Lo, M., Smith, J. The Onion Technique: Indexing for Linear Optimization Queries. ACM Conference on the Management of Data (SIGMOD), 391-402, Dallas, TX, May 16-18, 2000.

[CGGL03] Chomicki, J., Godfrey, P., Gryz, J., Liang, D. Skyline with Pre-sorting. IEEE International Conference on Data Engineering (ICDE), 717-719, Bangalore, India, March 5-8, 2003, 2003.

[FLN01] Fagin, R., Lotem, A., Naor, M. Optimal Aggregation Algorithms for Middleware. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), 102 - 113, Santa Barbara, CA, May 21-23, 2001.

[FSAA01] Ferhatosmanoglu, H., Stanoi, I., Agrawal, D., Abbadi, A. Constrained Nearest Neighbor Queries. International Symposium on Spatial and Temporal Databases (SSTD), 257-278, Redondo Beach, CA, July 12-15, 2001.

[G84] Guttman, A. R-Trees: A Dynamic Index Structure for Spatial Searching. ACM Conference on the Management of Data (SIGMOD), 47-57, Boston, MA, June 18-21, 1984.

[H94] Henrich, A. A Distance Scan Algorithm for Spatial Access Structures. ACM Workshop on Geographic Information Systems (ACM GIS), 136-143, Gaithersburg, MD, December, 1994.

[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), 51-59, 1999.

[HKP01] Hristidis, V., Koudas, N., Papakonstantinou, Y. PREFER: A System for the Efficient Execution of Multi-parametric Ranked Queries. ACM Conference on the Management of Data (SIGMOD), 259 - 270, May 21-24, 2001.

[HS99] Hjaltason, G., Samet, H. Distance Browsing in Spatial Databases. ACM Transactions on Database Systems, 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), 469-476, 1975.

[KRR02] Kossmann, D., Ramsak, F., Rost, S. Shooting Stars in the Sky: an Online Algorithm for Skyline Queries. Very Large Data Bases Conference (VLDB), 275-286, Hong Kong, China, 20-23 August, 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), 277-278, 1991.

[MD88] Muralikrishna, M., DeWitt, D. Equi-Depth Histograms for Estimating Selectivity Factors for Multi-Dimensional Queries. ACM Conference on the Management of Data (SIGMOD), 28-36, Chicago, IL, June 1-3, 1988.

[NCS+01] Natsev, A., Chang, Y., Smith, J., Li., C., Vitter. J. Supporting Incremental Join Queries on Ranked Inputs. Very Large Data Bases Conference (VLDB), 281-290, Rome, Italy, September 11-14, 2001.

[PKZT01] Papadias, D., Kalnis, P., Zhang, J., Tao, Y. Efficient OLAP Operations in Spatial Data Warehouses. International Symposium on Spatial and Temporal Databases (SSTD), 443-459, Redondo Beach, CA, July 12-15, 2001.

[PTFS03] Papadias, D., Tao, Y., Fu, G., Seeger, B. An Optimal and Progressive Algorithm for Skyline Queries. ACM Conference on the Management of Data (SIGMOD), 443-454, San Diego, CA, June 9-12, 2003.

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

[RKV95] Roussopoulos, N., Kelly, S., Vincent, F. Nearest Neighbor Queries. ACM Conference on the Management of Data (SIGMOD), 71-79, San Jose, CA, May 22-25, 1995.

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

[SRF87] Sellis, T., Roussopoulos, N., Faloutsos, C. The R+-tree: A Dynamic Index for Multi-Dimensional Objects. Very Large Data Bases Conference (VLDB), 507-518, Brighton, England, September 1-4, 1987.

[ST99] Salzberg, B., Tsotras, V. A Comparison of Access Methods for Temporal Data. ACM Computing Surveys, 31(2): 158-221, 1999.

[SYUK00] Sakurai, Y., Yoshikawa, M., Uemura, S., Kojima, H. The A-tree: An Index Structure for High-Dimensional Spaces Using Relative Approximation. Very Large Data Bases Conference (VLDB), 516-526, Cairo, Egypt, September 10-14, 2000.

[TEO01] Tan, K., Eng, P. Ooi, B. Efficient Progressive Skyline Computation. Very Large Data Bases Conference (VLDB), 301-310, Rome, Italy, September 11-14, 2001.

[TSS00] Theodoridis, Y., Stefanakis, E., Sellis, T. Efficient Cost Models for Spatial Queries Using R-trees. IEEE Transactions on Knowledge and Data Engineering, 12(1):19-32, 2000.