Shooting Stars in the Sky: An Online Algorithm for Skyline Queries

Donald Kossmann Frank Ramsak Steffen Rost

Technische Universität München
Orleansstr. 34
81667 Munich
Germany
{kossmann, rost}@in.tum.de, frank.ramsak@forwiss.de

Abstract

Skyline queries ask for a set of *interesting points* from a potentially large set of data points. If we are traveling, for instance, a restaurant might be interesting if there is no other restaurant which is nearer, cheaper, and has better food. Skyline queries retrieve all such interesting restaurants so that the user can choose the most promising one. In this paper, we present a new *online* algorithm that computes the Skyline. Unlike most existing algorithms that compute the Skyline in a batch, this algorithm returns the first results immediately, produces more and more results continuously, and allows the user to give preferences during the running time of the algorithm so that the user can control what kind of results are produced next (e.g., rather cheap or rather near restaurants).

1 Introduction

1.1 Skyline Queries

Recently, there has been a growing interest in so-called Skyline queries [BKS01, TEO01]. The Skyline of a set of points is defined as those points that are not dominated by any other point. A point dominates another point if it is as good or better in all dimensions and better in at least one dimension.

Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002

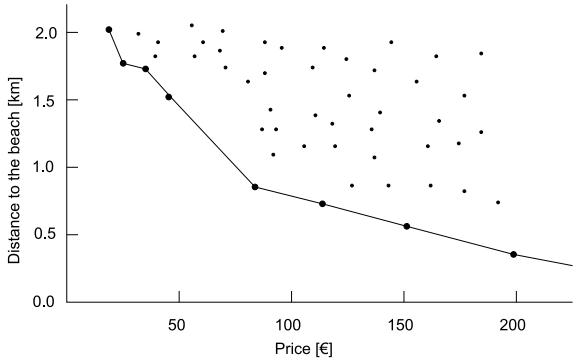


Figure 1: Skyline of hotels in Nassau (Bahamas)

The classic example is shown in Figure 1. The figure shows the Skyline of hotels in Nassau (Bahamas) which are supposed to be cheap and close to the beach. The bold points (which are connected in the graph) represent those hotels which are part of the Skyline. The other hotels are not part of the Skyline because they are dominated in terms of price and distance to the beach by at least one hotel which is part of the Skyline. Such a Skyline could be useful, for instance, for a travel agency; it helps users to get a *big picture* of the interesting options. From the Skyline of hotels, the user can then choose the most promising hotels and make further inquiries.

Skyline queries can also involve more than two dimensions and they could depend on the current position of a user. For instance, (mobile) users could be interested in restaurants that are near, cheap, and have good food (according to some rating system). The distance is based on the current location of the user. Again, the idea is to give the user the *big picture* of interesting options and then let the user make a decision. If the user moves on, the Skyline should be re-computed continuously in order to give the user a choice of interesting restaurants based on the user's new location.

Customer information systems such as travel agencies or mobile city guides are one application area for which Skyline queries are useful. Decision support (i.e., business intelligence) is another area. For instance, a Skyline query can be used in order to determine customers who buy much and complain little. Furthermore, the Skyline operation is very useful for data visualization. With the help of the Skyline, the outline of a geometric object can be determined; in other words, the points of a geometric object that are visible from a certain perspective can be determined using a Skyline query [BKS01]. Another application is distributed query optimization: the set of interesting sites that are potentially useful to carry out a distributed query can be determined using a Skyline query: those *interesting* sites have high computing power and are close to the data needed to execute the query.

There are several related problems: e.g., Top N [CK97], nearest neighbor search [RKV95], convex hull [PS85], the contour problem [McL74], or multi-objective optimization [Ste86, PY01]. Depending on the context, the Skyline is also referred to as the pareto curve [PY01] or a maximum vector [KLP75]. A discussion of these related problems and their relationship to Skyline queries is given in [BKS01].

1.2 Online Skyline Computation

It should have become clear that Skyline queries are often used in *interactive* applications. In such applications, it is important that a Skyline algorithm produces its first results quickly. On the other hand, it is less important that *all* points (possibly thousands) of the Skyline are produced: it is sufficient to give the user a *big picture*. In a mobile environment, this big picture of interesting points (e.g., restaurants) must be recomputed continuously, making it even more important to compute a big picture fast rather than munging on the complete result. These observations motivate the need for an *online* algorithm to compute the Skyline.

Unfortunately, most existing algorithms to compute the Skyline work in a batch-oriented way. These algorithms involve reading the whole data set and return the first results (almost) at the end of their running time. In addition, the running time of these algorithms can be very high and there is little hope to find better batch-oriented algorithms: it can be shown that the best algorithm to compute the full Skyline has a (worst-case) complexity of $\mathcal{O}(n(\log n)^{d-2})$, with n the number of points in the data set and d the number of dimensions [KLP75]; this is a higher complexity than sorting! An online algorithm would probably take much longer than a batch-oriented algorithm to produce the full Skyline, but an online algorithm would produce a subset of the Skyline very quickly. Adopting the criteria set in the Control project [HAC+99], we demand the following properties from an online algorithm:

1. The first results should be returned almost instantaneously. It should be possible to give guarantees that constrain the running time to produce, say, the first 100 results.

2. The algorithm should produce more and more results the longer the algorithm runs. Eventually (if given enough time), the algorithm should produce the full Skyline.

3. The algorithm should only return points which are part of the Skyline. In other words, the algorithm should not return *good* points (e.g., good restaurants) at the beginning and then replace these *good* restaurants with *better* restaurants.

4. The algorithm should be *fair*. In other words, the algorithm should not favor points that are particularly good in one dimension, instead it should continuously compute Skyline points from the whole range.

5. The user should have control over the process. In other words, it should be possible to make preferences *while* the algorithm is running. Using a graphical user interface, the user should be able to click on the screen and the algorithm will return next points of the Skyline which are near the point that the user has clicked on.

6. The algorithm should be universal with respect to the type of Skyline queries and data sets. It should also be based on standard technology (i.e., indexes), and it should be easy to integrate the algorithm into an existing database system. For a given data set (e.g., hotels or restaurants) one index should be enough to consider all dimensions that a user might find *interesting*.

1.3 Related Work

Kung et al. proposed the first Skyline algorithm in [KLP75], referred to as the maximum vector problem. Kung's algorithm is quite complex and based on the divide & conquer principle. In the eighties and nineties, a variety of different algorithms were proposed for specific situations; e.g., algorithms to compute very high dimensional Skylines [Mat91] and parallel Skyline algorithms [SM88]. However, none of these algorithms are applicable in situations in which the data set does not fit into main memory. Furthermore, all these algorithms work in a batch-oriented way.

Börszönyi et al. extended Kung's divide & conquer algorithm so that it works well for large

databases [BKS01]. They also proposed and evaluated several new Skyline algorithms. That work showed that depending on the query, the characteristics of the database, and the availability of main-memory and I/O bandwidth, either a blockwise-nested-loops algorithm or an extension of Kung's divide & conquer algorithm work best. Again, however, that paper only considered batch-oriented Skyline algorithms. In other words, the focus of that work was to find good algorithms to compute the full Skyline rather than finding algorithms that give a big picture quickly.

Tan et al. proposed two *progressive* Skyline algorithms [TEO01]. The first algorithm is based on Bitmaps and the second algorithm is based on an extension of B-trees. Both algorithms meet the first three requirements for an online Skyline algorithm; that is, they return Skyline points very quickly and produce more and more results the longer they run. However, neither of them meet requirements four and five. As a result, it is possible (in fact very likely!) that these algorithms return many cheap hotels at the beginning and that interesting hotels which are near the beach are only returned after a significant amount of time. For the Bitmap algorithm, the order in which Skyline points are returned depends on the clustering of the data; for the B-tree algorithm, the order in which points are returned depends on the value distribution of the data. (We will assess this aspect in our experiments in Section 4.) Furthermore, neither algorithm allows the user to give preferences in which order the results are produced. As a result, both algorithms are not advantageous in an interactive environment. In addition, the applicability of the B-tree algorithm is limited. The algorithm requires that a B-tree-like index structure is constructed for every combination of dimensions that a user might be interested in. For instance, if the data set has five dimensions that are frequently used as criteria in Skyline queries (e.g., price, distance to the beach, rating of rooms, ratings of associated restaurants, and capacity), then 31 indexes need to be constructed. If ten dimensions are potentially interesting, then 1023 indexes are required which is clearly not feasible. There are ways to extend the algorithm so that a single five (or ten)-dimensional index would be sufficient [Tan01]. Applying such extensions, however, involves scanning large portions of the database for each query before returning the first query results so that the extended algorithm violates our first requirement. Furthermore, both the Bitmap and the B-tree algorithm are not applicable in a mobile environment in which properties like *distance* are functions that take the current position of a user as a parameter; both algorithms require that all interesting properties are materialized in the database.

1.4 Overview

The remainder of this paper is organized as follows. Section 2 presents the basic idea of our new online algorithm and shows how it works in order to compute two-dimensional Skylines. Section 3 generalizes the idea and shows how higher-dimensional Skylines can be computed. Section 4 describes the results of our performance experiments. Section 5 contains conclusions and suggestions for future work.

2 An Online Algorithm for Two-dimensional Skylines

In this section, we will describe how our new online algorithm works for two-dimensional Skyline queries, such as the Skyline of hotels shown in Figure 1. In the next section, we will generalize this algorithm to higher-dimensional Skylines. For ease of presentation, we will make three simplifying assumptions. First, we assume that all values are positive real numbers (like distance or price). Second, we assume that there are no duplicates in the data set (e.g., no two hotels have the same price and distance to the beach). Third, we assume that we try to find *minimal* points; in other words, we try to find interesting points which are close to the origin $O = (0, 0)$. However, the techniques can be applied naturally to all ordered domains (including the full range of real numbers), to data sets with duplicates, and to queries that ask for *maxima* in certain dimensions (e.g., quality of food). In order to deal with maxima and/or negative numbers, we simply need to find upper and lower bounds for each dimension in our data set. The techniques cannot be applied to Skyline queries with DIFF annotations which were also proposed in [BKS01]; dealing with DIFF annotations is beyond the scope of this paper.

2.1 Basic Observations: The Relationship Between NN and Skyline

Our algorithm is based on two fundamental observations that show that nearest neighbor search can be applied in order to compute the Skyline.

Observation 1: Let f be an arbitrary monotonic distance function (e.g., Euclidean distance). Let $\mathcal{D}$ be our two-dimensional data set; each point in $\mathcal{D}$ has an x and y field with x and y being positive real numbers. Let $n \in \mathcal{D}$ be a nearest neighbor of $O = (0, 0)$ according to f. Then, n is in the Skyline of $\mathcal{D}$.

This observation can be easily proven by contradiction. Let $n = (x_n, y_n)$. Furthermore, assume that n is not part of the Skyline. As a result, there must be a point $b = (x_b, y_b)$ that *dominates* n. In other words, $x_b < x_n$ and $y_b \leq y_n$ or $x_b \leq x_n$

and $y_b < y_n$. Under these circumstances, however, $f(b, O)$ must be smaller than $f(n, O)$ because f is a monotonic distance function. This is a contradiction to n being the nearest neighbor of O. As a result, n must be part of the Skyline. $\diamond$

Going back to Figure 1, it is quite easy to see that this observation holds. If we use Euclidean distance then the hotel that costs 80 Euros and has a distance to the beach of 0.9 km is the nearest neighbor. This hotel is clearly part of the Skyline. Obviously, this observation also holds for higher-dimensional data sets and higher-dimensional Skyline queries.

The second observation is an extension of the first observation. It states that if the nearest neighbor search is constrained using a region containing O, then we can continue to conclude that all nearest neighbors within that region are in the Skyline of the whole data set.

A region is given by the coordinates of two diagonal points, a bottom-left point and a top-right point. Using our simplifying assumption, the bottom-left point is always O so that we can represent a region by the coordinates of the top-right point.

Observation 2: Let f be an arbitrary monotonic distance function. Let $\mathcal{D}$ be a two-dimensional data set; each point in $\mathcal{D}$ has an x and y field with x and y being positive real numbers. Let $m = (x_m, y_m)$ be a region, and let $\mathcal{D}_m$ be a subset of $\mathcal{D}$ such that $\mathcal{D}_m$ contains all points of $\mathcal{D}$ with $x < x_m$ and $y < y_m$. Let $n \in \mathcal{D}_m$ be a nearest neighbor of the point $O = (0, 0)$ according to f. Then, n is in the Skyline of $\mathcal{D}$. (Naturally, n is also in the Skyline of $\mathcal{D}_m$, using Observation 1.)

The proof of this observation is almost identical with the one of Observation 1. The only additional step is to prove that the (imaginary) point b must also be in the region, which is trivial based on the transitivity of the $<$ relation. $\diamond$

Essentially, the second observation means that if we partition the data set, then it is sufficient to look for Skyline points using nearest neighbor search in each region separately. This observation gives rise to a divide & conquer algorithm using nearest neighbor search.

2.2 The NN Algorithm for 2-dimensional Skylines

We call our online algorithm the NN algorithm because it is based on nearest neighbor search. Before dwelling into the details of this algorithm, we would like to illustrate how it works using an example.

2.2.1 An Example

Consider again our hotel example of Figure 1. The algorithm starts by searching for a nearest neighbor of the point O in the data set using some monotonic distance function. Following Observation 1, this nearest neighbor is guaranteed to be part of the Skyline and can be output to the user immediately. Figure 2 shows the first nearest neighbor (as an asterisk) for the hotel database of Figure 1. Figure 2 also shows that we can divide the data space into three regions using this point:

- Region 1 (depicted with a square in each corner) contains all points of the data set that have a smaller y value than the nearest neighbor. If (n_x, n_y) are the coordinates of the nearest neighbor, then this region is constrained by the following region: (∞, n_y).
- Region 2 (depicted with a quarter of a circle in each corner) contains all points of the data set that have a smaller x value than the nearest neighbor. This region is constrained by the following region: (n_x, ∞).
- Region 3 (depicted with a triangle in each corner) contains all points of the data set that have a greater x value and a greater y value than the nearest neighbor.

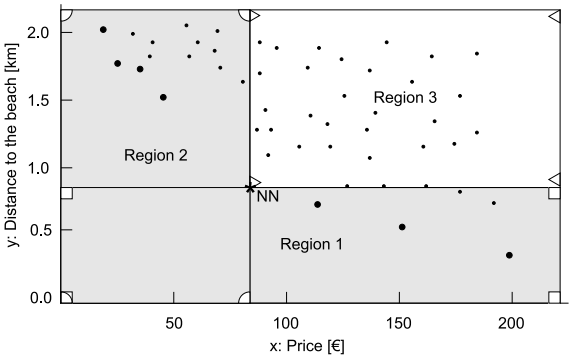


Figure 2: First step of the algorithm

Clearly, all points in Region 3 are dominated by the nearest neighbor so we need not investigate Region 3 any further. To compute more Skyline points, we only need to investigate Regions 1 and 2. In order to find a Skyline point in Regions 1 and 2, we make use of Observation 2 and simply look for nearest neighbors in those regions. Figure 3 shows how this is done for Region 2. Within Region 2, we find the second point of the Skyline by simply looking for a nearest neighbor within that region; that is, we constrain the search by looking only at points of the data set with $x < n_x$. (Note that the first nearest neighbor is not in Region 2 so that the first

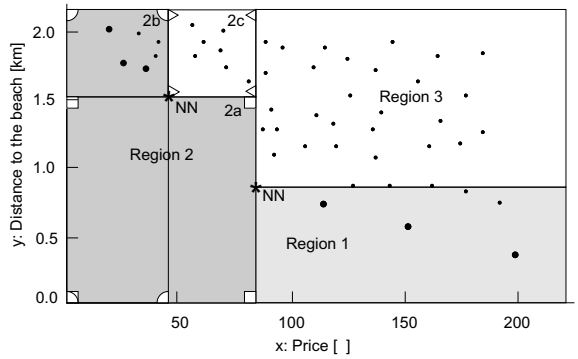


Figure 3: Second step of the algorithm

nearest neighbor will not be considered again.) Using Observation 2, we can conclude that this second nearest neighbor is definitely part of the Skyline and can immediately be output.

As shown in Figure 3, the second nearest neighbor subdivides Region 2 into three new regions: Region 2a, Region 2b, and Region 2c. Following the same principle, we need not investigate Region 2c and we can continue by investigating Regions 2a and 2b by looking for nearest neighbors in those regions. If a region is empty, i.e. no nearest neighbor is found, the region is not subdivided further (for instance, Region 2a in Figure 3 is empty so that that region will not be subdivided). In this way, the algorithm continues until all Skyline points are retrieved, i.e. no regions are left to be processed.

2.2.2 Algorithm Description

We are now ready to present the NN algorithm for 2-dimensional Skylines. The algorithm is shown in Figure 4. The algorithm gets as input a data set $\mathcal{D}$ and a monotonic distance function f. The algorithm maintains a *to-do list* of regions, $\mathcal{T}$. Using our simplifying assumptions (all values are positive real numbers), a region is defined by a two-dimensional vector. At the beginning, the *to-do list* contains only one region: (∞, ∞), i.e., the whole data space. Every time a new nearest neighbor is found for an entry in the *to-do list*, the corresponding region is subdivided into two smaller regions which are added to the *to-do list*. This way, the partitioning goes on and on until the new (smaller) regions are empty. The algorithm terminates when the *to-do list* is empty. If the *to-do list* grows too large, parts of it can be demoted to secondary memory (i.e., disk); however, we expect such cases to be extremely rare because typically a few Skyline points are sufficient in order to get a big picture so that the algorithm is stopped before the *to-do list* floods the main memory.

At the heart of the algorithm is the *boundedNNSearch* function. This function takes as input a point (in our case, O), a data set, $\mathcal{D}$, a region, (m_x, m_y), and a distance function f. It computes

Input: Data set $\mathcal{D}$
Distance function f (e.g., Euclidean distance)

/* Initialization: the whole data space needs to be inspected */
$\mathcal{T} = \{(\infty, \infty)\}$
/* Loop: iterate until all regions have been investigated */
WHILE ($\mathcal{T} \neq \emptyset$) DO
(m_x, m_y) = takeElement($\mathcal{T}$)
IF ($\exists$ boundedNNSearch(O, $\mathcal{D}$, (m_x, m_y), f)) THEN
(n_x, n_y) = boundedNNSearch(O, $\mathcal{D}$, (m_x, m_y), f)
$\mathcal{T} = \mathcal{T} \cup \{(n_x, m_y), (m_x, n_y)\}$
OUTPUT n
END IF
END WHILE

Figure 4: The NN algorithm for 2-d Skyline queries

the nearest neighbor $(n_x, n_y) \in \mathcal{D}$ of O with the additional constraint that this nearest neighbor must be within the region. Fortunately, this function can be computed very efficiently if the data set is indexed using a multi-dimensional index such as an R*-tree [BKSS90] or more modern data structures. For instance, the branch and bound algorithm for nearest neighbor search proposed in [RKV95] could be used for this purpose. Such a multi-dimensional index could also be used to evaluate other predicates of the Skyline query (e.g., *city = Nassau*) and such a multi-dimensional index could also be applied in a mobile environment if the distance of the user to the points in the data set is not materialized in the data set.

2.3 Discussion

Before explaining how the algorithm works for higher dimensional Skyline queries, we evaluate it based on the six requirements stated in the introduction.

1. With the help of a multi-dimensional index structure such as an R*-tree, nearest neighbor search is a cheap operation so that the NN algorithm returns the first results almost instantaneously. In our performance experiments, we always produced the first 10 results in a few seconds, even for very complex Skyline queries that involve ten dimensions. For queries that involve less than six dimensions, the first results are produced in fractions of a second. Due to the curse of dimensionality, nearest neighbor search can become an expensive operation for very high-dimensional data. In practice, however, we do not expect users to specify more than, say, five dimensions as part of their Skyline queries; in particular, in interactive environments. In fact, we believe that Skyline queries that involve two dimensions are the most common case.

2. Ultimately, the algorithm will explore all regions and will find all points of the Skyline.

3. Considering the two observations mentioned at the beginning of this section, all nearest neighbors found by the NN algorithm are part of the Skyline.

4. Fairness: The NN algorithm produces results from the whole range of results very quickly. We will demonstrate this property in Section 4.

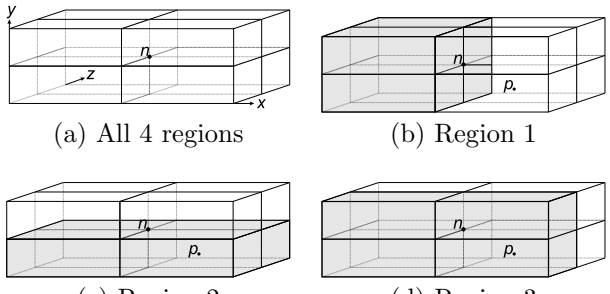
5. Control: There are two ways in which the NN algorithm can react to preferences specified by the user, i.e., change the order in which query results are returned. First, if a user clicks on a particular point in the graphical user interface during the running time of the algorithm, the algorithm can adjust and process those regions of the *to-do list* next that contain the point that the user clicked on. Second, the distance function f is a parameter of the NN algorithm and can be changed any time during the execution of the algorithm. This property can be exploited in the following way. At the beginning, the algorithm starts with some default distance function; e.g., *price + distance*. If the user clicks on the point *(price = 200, distance = 100)* (i.e., the user puts more emphasis on a short distance to the beach), then the distance function is adjusted to *price + 2 · distance*. With every interaction of the user, the distance function can be adjusted accordingly. The Skyline points returned before changing the distance function remain valid as well as the *to-do list*. As a result, the NN algorithm can continue to produce Skyline points using the new distance function without any adaptions.

6. The NN algorithm is universal. As we will see in the following section, it can be extended so that it works for Skyline queries that involve more than two dimensions. It can be applied if the query involves additional predicates (indexed and not indexed) and it can also be applied in the mobile application of the introduction. It is based on multi-dimensional index structures that support nearest neighbor search (e.g., R*-trees); such index structures provide to a large extent scalability, dynamic updates, and independence of the data distribution. If the data set involves, say, five dimensions which are potential criteria for Skyline queries, then a single five-dimensional R*-tree will be sufficient to execute all Skyline queries. As mentioned earlier, for the interactive applications we have in mind, these *ordering* criteria are known in advance and there is a limited number of these criteria in a typical application. Other dimensions of the data set which are part of the Skyline query need not be indexed.

In summary, the NN algorithm fulfills all six requirements. However, there are two situations in which the NN algorithm is not applicable: first, if the Skyline query involves a dimension which is not indexed; and second, if the Skyline query involves large joins, group-bys or other pipeline-breakers that must be carried out before the Skyline operation. In both of these situations, however, there is little hope to find an effective online algorithm. The progressive algorithms proposed in [TEO01] are not applicable in these situations either. In these situations, a batch-oriented algorithm such as those proposed in [BKS01] must be used.

3 The NN Algorithm for d-dimensional Skylines

In this section, we show how the NN algorithm can be applied to Skyline queries that involve more than two dimensions. Figure 5a shows a three-dimensional data space and a nearest neighbor n in that data space, $n = (x_n, y_n, z_n)$. Following the idea presented in the previous section, the data space can be partitioned into four regions: three regions that need to be investigated in further steps of the algorithm and one region that need not to be considered because it contains only points which are dominated by n. The three regions which need to be considered in further steps of the algorithm are defined as follows: Region 1 (x_n, ∞, ∞), Region 2 (∞, y_n, ∞), Region 3 (∞, ∞, z_n). These three regions are depicted individually in Figures 5b to d.



(a) All 4 regions (b) Region 1

(c) Region 2 (d) Region 3

Figure 5: 3-d Skyline computation

Our two observations from Section 2.1 can be generalized to three and more dimensions: the proofs are almost the same as in Section 2.1, just using d instead of two-dimensional points and regions. Thus, n is part of the Skyline and can be output immediately. Furthermore, the three regions can be investigated separately for further Skyline points; i.e., a nearest neighbor in Region 1 is also part of the Skyline and can be output as soon as it is found. As

a result, the algorithm of Figure 4 can be applied to d dimensional Skyline queries ($d > 2$) if it is adjusted to work with d dimensional points and if d regions (rather than 2) are added to the *to-do list* whenever a new nearest neighbor is found. There is only one small subtlety that needs to be addressed. The regions in a d dimensional data space ($d > 2$) overlap in such a way that the same point of the Skyline can be found twice or even more often—such duplicates could not occur in the NN algorithm for 2-dimensional Skyline queries.

In order to see how such duplicates are produced, let us go back to the example of Figure 5 and let us look at point p. Assume that p is also part of the Skyline—p is not dominated by n because it is better in the y and z dimensions. As shown in Figures 5c and 5d, p is contained in Region 2 and Region 3. As a result, p will be produced by the NN algorithm twice: once while processing Region 2 and once while processing Region 3.

Duplicates seem to be a subtlety and there are many alternative ways to carry out duplicate elimination [BD83, Lar97]. However, such duplicates can impact the performance of the NN algorithm severely. In this section, we will present alternative ways to extend the NN algorithm in order to deal with such duplicates. We study the tradeoffs of these alternatives in Section 4.4.

3.1 Laisser-faire

The simplest approach to deal with duplicates is to eliminate them as part of a post-processing step. This duplicate elimination step can best be implemented using a main-memory hash table. Whenever the NN algorithm produces a nearest neighbor, that nearest neighbor is probed with the hash table. If the nearest neighbor is found in the hash table, it is a duplicate and it is not output. Nevertheless, the region must be subdivided and inspected for further Skyline points. If the nearest neighbor is not found in the hash table, the nearest neighbor is not a duplicate: as a consequence, it is recorded in the hash table and output to the user or application. We refer to this approach as *laisser-faire*.

Clearly, this *laisser-faire* approach is simple to implement because it relies on standard ways (i.e., hashing) to eliminate duplicates [Lar97]. We believe that a main-memory hash table is sufficient because the purpose is to produce a *few* Skyline points quickly. Alternatively, of course, an index on secondary storage (e.g., a B-tree) could be used in order to carry out duplicate elimination. The big disadvantage of this algorithm is that it results in a great deal of wasted work. Using this approach, it is possible that the NN algorithm makes little progress because it finds the same Skyline point several times; in the worst case, a point is found $d-1$ times. Another (potential) disadvantage of this approach is that *real* duplicates (i.e., duplicates in the input data set) will be filtered out by this duplicate elimination step; according to the original definition of Skyline queries in [BKS01], such two identical points in the input data set should both be part of the Skyline. In order to remedy this situation, we need to give each point a unique identifier and take this identifier into account during duplicate elimination.

3.2 Propagate

The alternative to *eliminating* duplicates *after* they occur is to *prevent* duplicates *before* they occur. We can do this by using a technique called *propagation*.

Propagation works as follows: Whenever the NN algorithm finds a Skyline point, it scans the whole *to-do list* in order to find those regions in the *to-do list* that contain that point. We can remove those regions from the *to-do list* and partition them using that Skyline point. This is correct as only points that are dominated by a Skyline point are discarded by that operation. If a (main-memory) multi-dimensional index structure is available, that index structure can be used instead of a scan in order to find all regions in the *to-do list* that contain that point.

The big advantage of this technique is that it completely avoids wasted work to find duplicates. On the negative side, however, there is overhead to search in the *to-do list* for regions every time a new nearest neighbor has been found.

3.3 Merge

We now turn to a technique that can be used to improve the *laisser-faire* and *propagate* approaches. The idea is to *merge* (or even eliminate) regions of the *to-do list* under certain circumstances.

The basic idea of merging is quite simple. Assume that two regions $a = (a_1, a_2, \ldots, a_d)$ and $b = (b_1, b_2, \ldots, b_d)$ are in the *to-do list*. Essentially, this means that we still need to look in both of these areas for more Skyline points. Now, we can merge these two regions into a single region: $a \oplus b = (\max(a_1, b_1), \max(a_2, b_2), \ldots, \max(a_d, b_d))$. Region $a \oplus b$ supersedes both regions; thus, we are sure not to miss any points if we investigate $a \oplus b$ only. A particular situation arises, if a supersedes b; i.e., if $a_1 \geq b_1$, $a_2 \geq b_2$, ..., and $a_d \geq b_d$. In this particular situation, $a = a \oplus b$ and thus b can be simply discarded from the *to-do list*. In this situation, b can be discarded from the *to-do list* even if a has already been processed and is not part of the *to-do list*.

Merging reduces the size of the *to-do list*. On the negative side, merging increases the size of the

regions. Furthermore, finding good candidates to merge can become expensive. In addition, eliminating regions which are superseded by other regions involves remembering regions after they have been processed. Therefore, merging must be employed with care. We propose the following heuristics to make use of merging in the *propagate* and *laisser-faire* approaches:

- Propagate: For the *propagate* approach, we propose to make no use of *merging* and only consider the special case in which a region of the *to-do list* can be discarded because it is superseded by another region.

- Laisser-faire: For the *laisser-faire* approach, we propose to make use of *merging* whenever a duplicate is detected. That is, we only merge regions if they were derived from the same Skyline points. At the same time, we must be careful *not* to merge a region with one of its ancestor regions—this restriction is necessary in order to guarantee termination of the algorithm. (For brevity, we do not discuss this restriction and other heuristics in full detail and refer to a forth-coming technical report.)

3.4 Fine-grained Partitioning

Another option to avoid duplicates is to partition the regions in a way that they do not overlap. In Figure 5, for instance, we could partition into 8 non-overlapping regions of which 6 regions would be relevant for further processing. Implementing this approach, however, results in a sharp growth of the number of regions in the *to-do list*. Furthermore, this approach involves a complex post-filtering step in order to determine points of a region which are dominated by points of another region. Therefore, we did not pursue this approach any further.

3.5 Hybrid Approaches

Obviously, it is also possible to combine different approaches. As mentioned above, *merge* can be combined with both the *laisser-faire* and the *propagate* approach. Another option would be to start and *propagate* duplicates until the *to-do list* has reached a certain size; e.g., two hundred regions. Then, the algorithm switches to *laisser-faire* because propagation might be too expensive for a large *to-do list*. We will study such a hybrid approach in Section 4.4.

4 Performance Experiments

In this section, we study the performance tradeoffs of the NN algorithm as compared to existing algorithms to compute the Skyline. We use different kinds of synthetic databases, thereby varying the size of the database (number of points), the value distribution, and the number of dimensions of the points in the database. We study the running times of the algorithms to compute the full Skyline as well as the running times to compute the first results.

4.1 Experimental Environment

All experiments are carried out on a SUN Ultra 1 with a 167 MHz processor and 128 MB of main memory. The operating system is Solaris 8. The benchmark database, intermediate query results, and all software is stored on a 17.1 GB IBM disk drive. The programming language used to implement all algorithms is C++. In all experiments, the size of the main-memory buffer pool is constrained to be 10 percent of the size of the database[1].

As benchmark databases, we use the databases proposed in [BKS01]. In other words, we study databases of points with varying number of dimensions. Each point is 100 bytes long and composed of d doubles and a varchar field for padding (d being the number of dimensions). We study databases with 100,000 points and with 1 million points. Points are generated using one of the following three value distributions:

- **corr:** in a *correlated* database, points which are good in one dimension tend to be good in other dimensions, too. As a result, fairly few points dominate many other points and the Skyline of a *correlated* database is fairly small. As an example, consider a database of students: students who do well in *philosophy* often do well in other areas, too.

- **anti:** in an *anti-correlated* database, points which are good in one dimension are bad in at least one other dimension. As a result, the Skyline of an *anti-correlated* database is typically quite large. As an example, consider a database of hotels: hotels which are close to the beach tend to be expensive.

- **indep:** in an *independent* database, points are generated using a uniform distribution. The size of the Skyline of an *independent* data set is somewhere in between of that of a *correlated* and *anti-correlated* database.

Details of the three distributions can be found in [BKS01]. We also studied the benchmark databases that were used in the experiments presented in [TEO01]. These databases are based on the same three distributions, but they use different

[1] As observed in [BKS01], the size of the main-memory buffer pool impacts the performance of most Skyline algorithms only marginally.

domains. For brevity, we will discuss the results of those experiments only qualitatively in this paper.

In addition to the NN algorithm, we study the performance of the algorithms proposed in [BKS01] and [TEO01] as baselines. Specifically, we study the following algorithm variants:

- **NN:** We use an R*-tree [BKSS90] in order to carry out nearest neighbor search. If not stated otherwise, we use the *propagate* variant. We study the tradeoffs of alternative NN variants in a separate set of experiments (Section 4.4).
- **D&C:** Kung's divide & conquer algorithm extended by m-way partitioning and "Early Skyline", which has been shown to be the best variant of this algorithm in [BKS01].
- **BNL:** The block-nested-loops algorithm with a self-organizing list, as proposed in [BKS01]. Again, this is the best variant of this family of algorithms.
- **Bitmap:** The Bitmap algorithm as proposed in [TEO01]. This algorithm scans the database and uses bitmaps in order to detect whether a point is part of the Skyline. Unfortunately, this algorithm does not work for the benchmark databases used in our experiments: the size of the Bitmaps would be several hundreds of gigabytes large. This approach is only viable if *all* dimensions have a small domain (e.g. integers in the range of 0 to 100, as in the experiments of [TEO01]). Therefore, we chose to construct *approximate* bitmaps and implement the original Bitmap algorithm on top of these approximate bitmaps. These approximate bitmaps were constructed by mapping each double to an integer in the range of 0 to 100. As a result the approximate bitmaps are only a couple of hundred kilobytes large and the algorithm runs much faster in our experiments than it would do in practice. Due to the approximation, the algorithm returns false positives; i.e., points which are not part of the Skyline. We did not measure the running time to eliminate such false positives in our experiments.
- **B-tree:** This is the second algorithm proposed in [TEO01]. We used a light-weight implementation of this algorithm that does not require the use of extended B-trees. This light-weight implementation avoids overheads for traversing the B-tree and therefore runs slightly faster than the original algorithm. (In the original proposal of [TEO01], the special B-tree variants are only required to carry out updates efficiently, but not for calculating the Skyline [Tan01].) We would like to reiterate that the B-tree algorithm is not universal and requires a specific index for each kind of Skyline query. To carry out the full set of experiments (with 10-dimensional data) we had to construct 1023 different indexes in order to study the B-tree algorithm. On the other hand, one R*-tree is sufficient for the NN algorithm.

	100,000 Points			1,000,000 Points		
	Anti	Corr	Indep	Anti	Corr	Indep
Skyline	49	1	12	54	1	12
NN	0.57	0.02	0.20	0.69	0.02	0.50
BNL	1.77	1.65	1.68	17.16	16.24	16.07
D&C	2.63	2.56	2.63	28.65	28.53	28.50
Bitmap	6.09	0.84	1.40	57.12	12.23	17.90
B-tree	13.86	0.01	0.26	> 200	0.12	0.92

Table 1: Size of Skyline, Running Times [secs] 2-d, 100K and 1M points

4.2 Two-dimensional Skyline Queries

Table 1 shows the running times to compute the whole Skyline for the alternative algorithms for a two-dimensional Skyline query in databases with 100,000 and 1 million points. Since the Skyline is rather small, we measure the running times for retrieving the complete Skyline. The Skyline contains 49 points for the small anti-correlated database, 1 point for the small correlated database, and 12 points for the small independent database. For the large databases, the sizes of the Skyline are as follows: 54 (anti), 1 (corr), and 12 (indep). Note that the size of the Skyline does not necessarily grow with the size of the database [BKST78].

We can observe that the NN algorithm is the overall winner in this experiment. As mentioned in Section 2, 2-dimensional Skylines are a particularly good case for the NN algorithm. The BNL and D&C algorithms show relatively poor performance because both algorithms involve reading the whole database, whereas the NN algorithm can use the R*-tree in order to quickly retrieve Skyline points. Likewise, the Bitmap algorithm must consider all points of the database in order to compute the full Skyline. Due to its particular logic to test whether a point is part of the Skyline, the Bitmap algorithm performs sometimes better and sometimes worse than the BNL and D&C algorithms.

The B-tree algorithm performs well for the *correlated* and *independent* databases. However, it shows very poor performance for the *anti-correlated* databases. In this case, the B-tree algorithm must also read (almost) the whole database in order to compute the full Skyline because the termination condition, the special trick of this algorithm, does not apply until the very end in this particular data distribution. Furthermore, the B-tree algorithm has fairly high overheads for each point: it must compare each point with all Skyline points found so far.

Comparing the running times for the small and large databases, it can be seen that the NN algorithm scales best. Its running time stays almost constant, whereas the running times of the other algorithms increase sharply. For the large database, we had to stop the execution of the B-tree algorithm for the anti-correlated database: after 200 seconds, it had produced only 30 points (54 points are total).

We also carried out experiments with the databases that were used in the performance experiments of [TEO01]. In those experiments, the NN, Bitmap, and B-tree algorithms ran faster. However, the general trends were the same: (1) The NN algorithm was the clear overall winner. (2) The Bitmap algorithm was in the same ball park as the BNL and D&C algorithms. (3) The B-tree algorithm showed good performance for the *correlated* and *independent* databases, but terrible performance for the *anti-correlated* database.

Quality of Results

As mentioned earlier, the quality of the results is as important as the response time for an online algorithm. The user wants a *big picture*.

Figure 6a shows the full Skyline for the *anti-correlated* database for $d = 2$. Figure 6b shows the first 10 points returned by the NN algorithm (without user interaction) and Figure 6c shows the first 10 points returned by the B-tree algorithm. It can be seen that the NN algorithm gives a good *big picture* of the Skyline. It returns points which are fairly good in all dimensions first. As mentioned in Section 2.3, the NN algorithm could also adjust and produce *extreme* points which are very good in certain dimensions, if the user wishes.

Figure 6c shows that the B-tree algorithm does not help to produce a big picture quickly. The B-tree algorithm returns *extreme* points first which are good in one dimension. Adapting this experiment to our hotel example, the B-tree algorithm returns five hotels which are very cheap and five hotels which are very close to the beach. It fails, however, to give the user a good picture of the price/distance tradeoffs. In addition, the user has no control!

The Bitmap algorithm scans the database and uses bitmaps in order to detect whether a point is part of the Skyline. Therefore, the clustering of the database determines which points are returned first. Again, the user has no control.

4.3 High-dimensional Skyline Queries

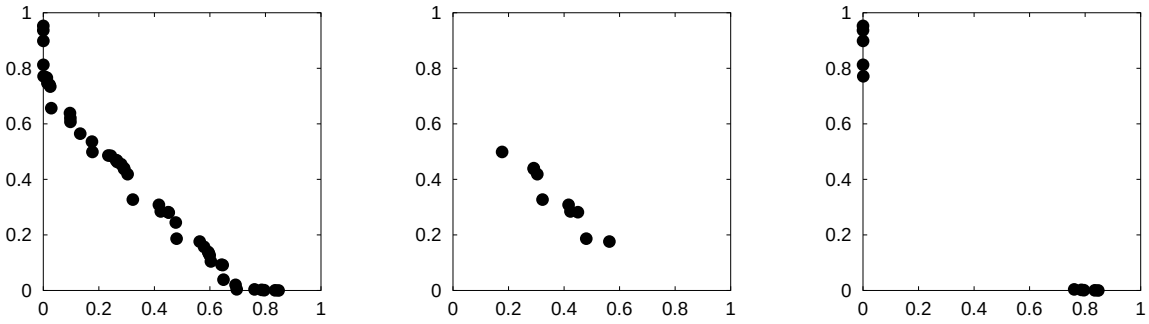
We now turn to an experiment that studies the behavior of the alternative algorithms for a five dimensional Skyline using an anti-correlated data set with 1 million points. The size of the Skyline is very large (35,947 points). Computing the full Skyline takes a long time, regardless what algorithm is used. As a result, an online algorithm is particularly important for interactive applications in this scenario: (1) to select relevant points from the Skyline (rather than flooding the user with results); (2) to give the first answers quickly.

	NN	B-Tree	Bitmap
Time for first 100 points [secs]	8.4	0.1	23.6
Response to User Interaction [secs]	0.1	~ 300	~ 87
Skyline points returned before user point found	0	15000	5117

Table 2: 5-d Anti-correlated, Large Database

Table 2 shows the running times of the alternative algorithms to produce the first 100 Skyline points. In addition, it shows how quickly the algorithms can return points based on user preferences. In this experiment, the B-tree algorithm produces the results very quickly: it takes less than a second to produce the first 100 Skyline points. This is not surprising because this algorithm simply needs to scan through the extended B-tree (that was specifically generated for this Skyline query, i.e. fixed dimensionality and a fixed origin) and return points that are extremely good in one dimension; the NN and Bitmap algorithms need to perform significantly more logic. So, the B-tree algorithm is good in order to return *some* results. However, if the user gives preferences, the B-tree algorithm cannot adapt well, even if the right index has been created. In this experiment, we simulated a user that is interested in a point that is good in all dimensions: using the B-tree algorithm, it takes 300 seconds to find such a point and more than 15,000 points need to be inspected before such a point is returned. (If the user gives more weight to certain dimensions, then the interactivity is slightly improved.) In terms of interactivity, the NN algorithm is the clear winner. Whenever a user gives a preference, the NN algorithm adjusts its distance function and returns the next point that matches these preferences immediately: it only takes 0.1 seconds for the NN algorithm to adapt. In some sense, the B-tree algorithm (blindly) scans through the Skyline whereas the NN algorithm is able to (selectively) pick points from the Skyline based on user preferences. (Naturally, blindly scanning has less overhead per point than selectively picking points.)

The Bitmap algorithm is not competitive in either aspect. (1) It produces Skyline points at approximately the same rate as the NN algorithm. (2) It is no match to the NN algorithm in terms of interactivity because it produces Skyline points in a somewhat random order (based on the clustering of the data). In this particular experiment, it happened to take 87 seconds before it was able to return a



(a) Full Skyline (b) First 10 points of NN alg. (c) First 10 points of B-tree alg.

Figure 6: Full and partial Skylines: 2-d, Anti-correlated 100K points

point that met the user preferences. For interactivity, the Bitmap algorithm could be improved using multi-dimensional indexes in addition to Bitmaps; however, the best way to do this extension would be to implement the NN algorithm!

We carried out experiments with databases and Skyline queries up to ten dimensions for all three distributions and with small and large databases. The results were always the same: For $d < 4$, the NN algorithm is typically the winner in all respects, regardless of the data distribution. For higher-dimensional Skyline queries, the B-tree algorithm produces results at a higher rate, but it always produces extreme points and it takes very long to produce points that are good in more than one dimension. The rate in which the Bitmap algorithm produces results is in the same ball park as that of the NN algorithm, but the Bitmap algorithm lacks the inter-activeness that the NN algorithm provides. For high-dimensional Skyline queries, the D&C and BNL algorithms are the best choices in order to compute the full Skyline, but they are not appropriate in interactive environments. For instance, it takes the D&C algorithm 150 seconds and the BNL algorithm even 3500 seconds before returning the first results in the scenario of Table 2.

4.4 Comparing Algorithm Variants

We now turn to a discussion of the performance tradeoffs of the variants of the NN algorithm for Skyline queries of more than two dimensions. These variants have been described in Section 3. Figure 7 shows the performance of four variants for a 5-dimensional Skyline with an *anti-correlated* database and 1 million points. We measured the following variants:

- **Laisser-faire:** see Section 3.1
- **Propagate:** see Section 3.2
- **Merge:** laisser-faire and merge regions whenever a duplicate is found, see Section 3.3
- **Hybrid:** propagate to the first 20 % entries of the *to-do list*. Duplicates that are not prevented by the reduced propagation are handled with the *laisser-faire* approach.

No index on the *to-do list* was used to find entries in the *to-do list* for the *propagate* and *hybrid* variants. Nevertheless, these two variants clearly outperform the other variants. The *merge* and *laisser-faire* variants spend too much time on duplicates. On an average, four nearest neighbor searches need to be carried out in order to find a new result. Clearly, the *propagate* and *hybrid* variants would benefit greatly from an index on the *to-do list*; however, our experiments (not shown) indicate that R*-trees or any other common multi-dimensional index structure do not work well for this purpose. Furthermore, the performance of the *hybrid* variant can be improved by tuning. In some experiments, the *hybrid* variant performed better if only 10 %, rather than 20 %, of the *to-do list* were scanned or if only a fixed number of entries of the *to-do list* was considered. All these tuning approaches are beyond the scope of this paper.

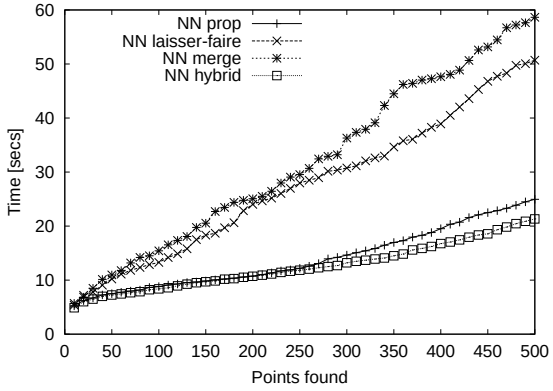


Figure 7: Running Times [secs]: NN Variants
5-d, Anti-correlated, 1M points

5 Conclusion

Skyline queries are important for several database applications, including customer information systems, decision support, and data visualization. In this paper, we studied a new *online* algorithm to compute Skyline queries. This algorithm is called the NN algorithm because it is based on nearest neighbor search, a well-studied database operation. We compared this new algorithm to existing algorithms that compute the Skyline in a batch and to existing algorithms that work progressively.

In our performance experiments, we could not identify a clear winner. All algorithms have their particular virtues. Our new algorithm, the NN algorithm, gives a *big picture* of the Skyline very quickly in all situations. However, it is not always the best choice if the full Skyline needs to be computed. In addition to the raw performance, the algorithms differ significantly in other criteria. The NN algorithm is the only algorithm that gives the user control over the process and allows the user to give preferences. The B-tree algorithm gives "extreme" points preference (i.e., points good in one dimension) and returns points which are good in many dimensions very late. The Bitmap algorithm scans the database and uses Bitmaps in order to detect whether a point is part of the Skyline. As a result, the order in which the points of the Skyline are returned is determined by the clustering of the database. Furthermore, the applicability of the B-tree and Bitmap algorithms is limited. Therefore, we strongly believe that the NN algorithm will be the Skyline algorithm of choice for interactive environments.

There are three main avenues for future work. First, we plan to further improve the performance of the NN algorithm by exploiting new techniques for nearest neighbor search. Second, we would like to investigate special-purpose main-memory index structures in order to manage the *to-do list* of the NN algorithm; initial experiments indicate that existing techniques are not appropriate for our purposes. Third, we would like to investigate how the NN algorithm can be combined with other algorithms. For instance, it would be possible to use nearest neighbor search in order to partition the data space and then use the block-nested-loops algorithm in order to process each region. Such a combined approach would give a big picture quickly and it would continue to compute the full Skyline efficiently.

Acknowledgments: We would like to thank Christian Böhm for many helpful discussions and for the R*-tree implementation. This research was partially supported by the German National Research Foundation (DFG) under contract Ko 1697/4-1.

References

[BD83] D. Bitton and D. J. DeWitt. Duplicate record elimination in large data files. *ACM Trans. on Database Systems*, 8(2):1983.

[BKS01] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proc. IEEE Conf. on Data Engineering*, Heidelberg, Germany, 2001.

[BKSS90] N. Beckmann and H.-P. Kriegel and R. Schneider and B. Seeger The R*-tree: An Efficient and Robust Access Method for Points and Rectangles In *Proc. of the ACM SIGMOD Conference*, Atlantic City, NJ, May 1990

[BKST78] J. L. Bently and H. T. Kung and M. Schkolnick and C. D. Thompson On the Avarage Number of Maxima in a Set of Vectors and Applications. In *JACM*, 25(4), 1978

[CK97] M. Carey and D. Kossmann. On saying "enough already!" in SQL. In *Proc. of the ACM SIGMOD Conference*, Tucson, AZ, May 1997.

[HAC+99] J. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, and P. Haas. Interactive data analysis: The control project. *IEEE Computer*, 32(8), September 1999.

[KLP75] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *Journal of the ACM*, 22(4), 1975.

[Lar97] P. A. Larson. Grouping and duplicate elimination: Benefits of early aggregation. Microsoft Technical Report, January 1997. `http://www.research.microsoft.com/~palarson/`.

[Mat91] J. Matoušek. Computing dominances in E^n. *Information Processing Letters*, 38(5), June 1991.

[McL74] D. H. McLain. Drawing contours from arbitrary data points. *The Computer Journal*, 17(4), November 1974.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, Berlin, etc., 1985.

[PY01] C. H. Papadimitriou and M. Yannakakis. Multiobjective query optimization. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, Santa Barbara, CA, USA, May 2001.

[RKV95] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proc. of the ACM SIGMOD Conference*, San Jose, CA, May 1995.

[SM88] I. Stojmenovic and M. Miyakawa. An optimal parallel algorithm for solving the maximal elements problem in the plane. *Parallel Computing*, 7(2),June 1988.

[Ste86] R. E. Steuer. *Multiple criteria optimization*. Wiley, New York, 1986.

[Tan01] K.-L. Tan. Personal communication, Sept. 2001.

[TEO01] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *Proc. of the Conf. on Very Large Data Bases*, Rome, Italy, Sept. 2001.