Dominant Skyline Query Processing

Zeng Yiming

Bachelor of Computing (First Class Honors)

National University of Singapore

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2006

To my parents.

Abstract

A skyline query retrieves from a given data set, a set of tuples that are not dominated by any other tuples with respect to a set of dimensions. Skyline computation has recently received a lot of attention from academia. In this thesis, we explored two skyline variants, which are meaningful and interesting when the skyline result set is either too large or too small. The first variant, called the *dominant skyline queries*, retrieves skyline tuples that dominate at least t other tuples. It is used to refine a large set of results to a smaller and more interesting set of tuples. The second variant, called the *tier-based skyline queries*, retrieves "skyline" points from tier 1 to tier k, where *tier-k* points are skyline points when *tier-1* to *tier-(k-1)* points are eliminated from the input data set. It is meaningful when the skyline result set is too small. We proposed several algorithms to solve these two variants respectively. We have also conducted extensive experiments to study the performance of various algorithms. Through the experiments, we identified some interesting trends and tradeoffs of these algorithms.

Acknowledgments

I would like to thank my research supervisor Dr. Chan Chee Yong for his invaluable guidance, suggestions, and support throughout the course of this thesis.

I also want to take this opportunity to thank my fellow lab mates. They have offered their generous help and support to my research.

I am deeply grateful for my parents. Their love accompanies and encourages me every moment. I would like to dedicate this work to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A skyline query finds, in a relation, all tuples that are not dominated by any other tuples in the same relation with respect to all the specified dimensions. As an example, assume in Figure 1.1 that we have a set of hotels and for each of them we record down its distance to downtown and rate. A user can ask for hotels that offer a good rate and close to downtown, which is a typical skyline query. Answering a skyline query is actually a multi-objective optimization problem. It is a useful class of queries with which users can specify multiple criteria (distance and rate in the example) for decision making. There may rarely be just a single optimal answer (or answer set) fulfilling a skyline query because a point optimal in every dimension rarely exists (e.g., hotels closer to city center are usually more expensive). In Figure 1.1, all the black points fulfill user's criteria because there exists no hotel with a shorter distance to downtown and offering lower rate, compared to any of the black points. Furthermore, these black points are incomparable with each other, because for any two of them, it is always the case that one point wins in one dimension and the other wins in the other dimension. Typically, skyline queries are formulated in the context of multi-dimensional Euclidean space where the dominance relationship is *minimum* or *maximum* (the dominance relationships in both dimensions of Figure 1.1 are *minimum*). Users can thus specify their preference on a set of dimensions, to minimize a subset of them and/or maximize the rest.

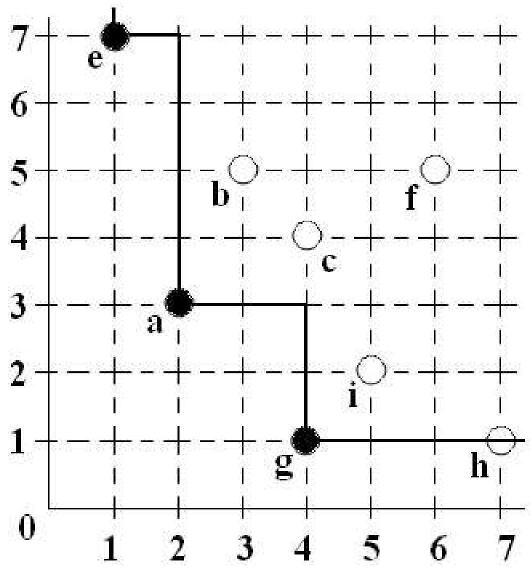


Figure 1.1: Example data set and skyline

1.1 Syntax and Semantics of Skyline Queries

The syntax and semantics of skyline queries were first formally presented in [3]. The basic syntax of skyline queries is defined using the following extension to SQL:

SELECT ... **FROM** ... **WHERE** ... **GROUP BY** ... **HAVING** ...

SKYLINE OF d_1 [***MIN***|***MAX***|***DIFF***] **...** d_n [***MIN***|***MAX***|***DIFF***].

The SKYLINE clause specifies the set of dimensions d_i's that a user wants to optimize, using three criteria. The *MIN* criterion indicates that the corresponding dimension should be minimized. The *MAX* criterion indicates that the corresponding dimension should be maximized. The *DIFF* criterion indicates that two tuples are not comparable if they have different values in the corresponding dimension.

Assuming no duplicate tuples with respect to the skyline dimensions, the "dominate" relation is defined as follows. A tuple p_i is said to *dominate* another tuple p_j if

1. p_i and p_j have the same values for the *DIFF* dimensions, and
2. p_i's *MIN* dimensions are not greater than the same dimensions of p_j's, and
3. p_i's *MAX* dimensions are not smaller than the same dimensions of p_j's.

All the tuples that are not dominated by any other tuples in the relation form the skyline result set. The corresponding skyline query of Figure 1.1 is **SELECT** * **FROM** *hotels* **SKYLINE OF** *rate* ***MIN*** *distance* ***MIN***
The result is the set of all the black points (tuples). For simplicity, all the discussions in this thesis will assume skyline computations using *MIN* conditions on the dimensions; however, all methods discussed can be applied

to any combination of conditions.

1.2 Motivations

With a large input data set, the answer set to a skyline query may include also a very large number of records. This is particularly the case when we have skyline queries involving many dimensions. Users would be overwhelmed if we dump all the skyline records to them without any further information. To avoid this scenario, it is desirable to have some ways to rank the skyline records according to certain criteria and return only the interesting skyline records (i.e., skyline points above a certain ranking threshold) or return all skyline records with their ranks.

There are many ways to define the ranking of points. One way to define the ranking is to associate a preference function with the dimensions of the data points, just as what is done in top-K queries [10, 4, 7, 16]. The difference is that the question now becomes computing top-K skyline points. BBS [17] can easily handle this by modifying the *mindist* definition to reflect the preference function (i.e., the *mindist* of a point equals to the value of the preference function applied to its dimensions).

Another way to define the ranking of a skyline point is based on the number of points it dominates. We call it the *dominating power* of a skyline point. Clearly, the value of the dominating power may range from zero to the size of the data set minus one. Intuitively, a skyline point with a high dominating power (i.e. dominates a large number of other points) is more interesting than a skyline point with a relatively low dominating power.

On the other hand, we may have a small skyline result set. Possible reasons are the input data set is small, or the data distribution is skewed. In this case, users may be interested in not only the conventional skyline tuples but also tuples that have properties similar to skyline tuples.

One way would be to retrieve tuples (not necessarily skyline) that are dominated by at most t_1 tuples, but dominate at least t_2 other tuples. This is a generalized problem of the skyline variant mentioned earlier based on *dominating power*.

We can also define the dominance relation among tuples in terms of *tier*. *Tier-1* tuples are the conventional skyline tuples. *Tier-2* tuples are skyline tuples when *tier-1* tuples are removed from the input data set. *Tier-k* tuples are skyline tuples when *tier-1* to *tier-(k-1)* tuples are removed from the input. *Tier-1* tuples are the most superior tuples. When such tuples are too few, we may be interested in tuples that belong to higher tiers.

1.3 Problem Definitions

Based on the above observations, we define the problems that we are going to solve formally. As mentioned earlier, we deal with two variants of the conventional skyline problem.

1.3.1 Dominant Skyline Queries

Given a set of data records S, a skyline query Q, and a dominating power threshold t, we want to retrieve all the records, each of which belongs to the result of Q and dominates at least t other records in S. We call the number of points dominated by a skyline point the *dominating power* of the skyline point.

As an example, consider the two-dimensional data set in Figure 1.2, the skyline points g, a, and e have dominating power 4, 3, and 0 respectively. With this data set and skyline query, when the dominating power threshold is set to 4, only point g will be returned as the answer. This problem is defined in [17], but the solution included there (to be discussed in Section 3.1) is naive.

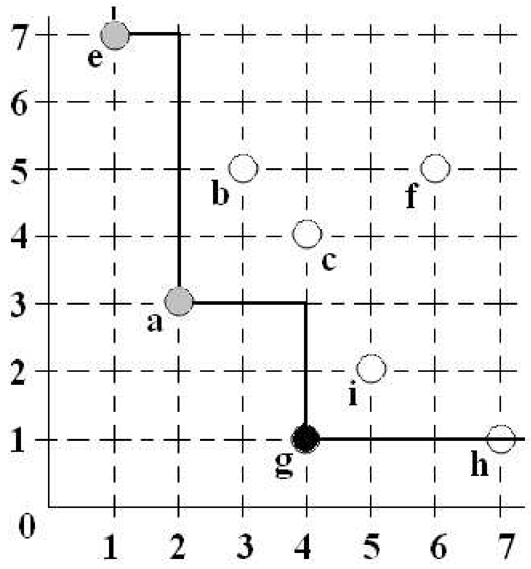


Figure 1.2: Dominant skyline query example data set

1.3.2 Tier-based Skyline Queries

Given a set of data records S, a skyline query Q, and a tier threshold k, we want to retrieve all the records that belong to any of *tier-i* where $1 \leq i \leq k$. *Tier-1* records are the standard skyline records. *Tier-i* records are the skyline records when the *tier-1* to *tier-(i-1)* records are removed from the input data set.

As an example, consider the data set in Figure 1.1. e, a, g, and h belong to tier 1; b, c, and i belong to tier 2; f belongs to tier 3.

1.4 Related Work

The first variant, the *dominant skyline queries*, was introduced in [17] which proposed the Branch and Bound algorithm for standard skyline computation. However, the problem cannot be solved using the technique proposed in an efficient manner. In this section, we give an overview of the algorithms to compute standard skyline queries and some skyline query variants.

Skyline query is a subclass of preference queries [6, 12]. It provides a means to compute preference queries efficiently.

The need for preference queries arises because traditional queries, which ask for results that *match users' criteria exactly*, cannot cope well with real users' demands. With all criteria specified, it is often the case that a query's result is empty as there is no exact match in the database. Leaving some criteria unspecified will lead to the other extreme where users are flooded with numerous irrelevant data [12]. Hence, we need a better query model. With preference queries, users can specify fuzzy criteria and their relative importance (i.e. prioritized preferences). The system is then expected to find results that *best match* with users' specifications. Consider the following scenario. A family wants to rent a flat. They want a flat around $100m^2$, preferably close to Suntec City, with rental between \$1,300 and \$1,500. The housing database may not have an entry that satisfies all the conditions, i.e. an empty result will be returned if the query is modeled as a traditional query, despite the difficulty of writing it (due to the fuzziness). On the contrary, if the query is modeled as a preference query, with extra specifications such as the relative importance of the conditions (e.g., among the three conditions, area is most important, price next, and location is least important), housing records that best match the conditions may be found.

Being a more realistic query model, preference queries have a wider range of applications such as personalized search engine and e-shopping ([1]). Unfortunately, existing query platforms (e.g. SQL) lack of direct support for preference queries. To catch up with the popularity, many researches ([5, 6, 10, 18]) try to extend the current query languages for preference query handling. Skyline query is one of the most extensively studied sub-problems of preference query. It corresponds to the *Pareto* preference constructor, where every criterion is equally important. Also, standard skyline query assumes that the records can be mapped to points in the Euclidean space, i.e., there is a total order in any single dimension.

The other subclass of preference queries, which is closely related to skyline query, is the top-K query [10, 4, 7, 16]. Top-K query retrieves the best K tuples that minimize a specific preference function. Each tuple is mapped to a numeric value (called *rank*) using a scoring function. The top K tuples with the highest ranks are included in the result. Top-K results may not be in the skyline, and it changes when the input function changes.

Skyline queries are also related to several well-known problems in Geometry, including convex hull and nearest neighbor search. Convex hull contains the subset of skyline points that may be optimal only for linear preference functions (as opposed to any monotone function for general skyline [3]). Several convex hull algorithms can be found in [2, 19]. Nearest neighbor queries retrieve the closest points to an input point. The depth-first algorithm of [20] branches down R-tree entries closest to the query point recursively. [13] presents a similar recursive algorithm to find skyline points using nearest neighbor search result.

The standard skyline computation has several important algorithms. Table 1.1 gives an overview of them based on the techniques used. Table 1.2 summarizes some of the skyline variants which are detailed in Section 2.2.

1.5 Contributions

In this thesis, we proposed several algorithms to compute two variants of the skyline query, the *dominant skyline queries* and the *tier-based skyline*

Algorithm	Technique
Block Nested Loop [3]	Pairwise comparisons
LESS [9]	Pairwise comparisons with pre-sorting
Divide and Conquer [3]	Chop up the data set into smaller enough ones that can fit into memory individually. Process each of them with in-memory algorithms and merge them to get final results.
Bitmap [21]	Encode every dimension of every tuple using bitmap. Get skylines using fast bitwise computations.
Index [21]	Group tuples according to their minimum dimensions. Sort each group and process top tuples of all groups.
Nearest Neighbor [13]	Use nearest neighbor search to find skyline points which further divide the space for recursive processing.
Branch and Bound [17]	Always branch down the most potential R-tree entries that may contain skyline points. At the same time, prune away dominated entries.

Table 1.1: Summary of skyline algorithms

Variant	Overview
Thick Skyline [11]	Retrieve skyline points and their ε-neighbors
Stable Skyline [8]	Extend the expressiveness of standard skyline using $EQUAL$ and BY
Streaming Skyline [15]	Compute skyline points in a streaming database

Table 1.2: Summary of existing skyline variants

queries.

A dominant skyline query, retrieves skyline tuples that dominate at least t other tuples. It refines the skyline result set to a smaller and more interesting set of tuples. We proposed several approaches to solve this variant effectively.

A tier-based skyline query retrieves "skyline" tuples within tier 1 to tier k. It extends the conventional skyline result set to a larger and meaningful set of tuples. We proposed three variants of algorithm based on Branch-and-Bound Skyline algorithm [17].

We conducted extensive experiments to study the algorithms we proposed. We investigated important parameters that affect the performance of the algorithms. Through these experiments, we identified the effects that various parameters have on evaluation time. We also identified some interesting tradeoffs among different approaches.

1.6 Organization of the Thesis

The rest of the thesis is organized as follows.

Chapter 2 gives an in-depth review of the existing algorithms for the standard skyline computation as well as some variants of the skyline problem. We analyze the limits and strengths of each algorithm. It is worthwhile to see how researchers re-formulate the problem to make it more interesting. Chapter 3 details the discussions of the dominant skyline queries. It also includes several algorithms to answer this type of queries. Chapter 4 provides the algorithms to answer tier-based skyline queries. It also discusses the differences among the algorithms and the possible impacts on performances. The experimental evaluation of various algorithms are also included in Chapter 3 and Chapter 4. Finally, we summarize the thesis in Chapter 5.

Chapter 2

Related Work

2.1 Existing Skyline Algorithms

Skyline query has been extensively studied over the past few years. Researchers have proposed various algorithms, ranging from those that do not need any index (e.g., Block Nested Loop [3], LESS [9]) to those utilize indexes such as Bitmap (e.g., Bitmap [21]), B+-tree (e.g., Index [21]), and R-tree (e.g., Nearest Neighbor [13], Branch and Bound [17]). Some algorithms need to read the entire data at least once before returning the first result (e.g., BNL, Divide-and-Conquer, Bitmap), others are able to start returning results without a complete view of the data set (e.g., Nearest Neighbor, Branch-and-Bound). Some algorithms can only answer skyline queries of a predefined subset of dimensions efficiently (e.g., Index), others can do so with respect to arbitrary dimensions (Nearest Neighbor, Branch-and-Bound).

2.1.1 Block Nested Loop

A straightforward way to compute the skyline points is to compare each point with every other point; points that are not dominated by any other points are in the skyline. Block Nested Loop ([3]) is built on this concept by scanning the data file and keeping a list of candidate skyline points in memory. The candidate list is initiated with the insertion of the first data point into it. For subsequent point p, there are three cases:

Case 1 If p is dominated by any other point in the list, it is discarded as it is not part of the skyline;

Case 2 If p dominates any point in the list, it is inserted into the list, and all those dominated by p are removed from the list;

Case 3 If p is neither dominated nor dominates any other points in the list, insert it into the list as it may be part of the skyline.

When the list keeps expanding, the memory may overflow. In that case, all points falling in the third case (the first two cases do not increase the list size) will go to a temporary file on disk. This fact necessitates the need of multiple passes of BNL when memory size is small. Actually, after the first pass, only points added to the candidate list before the creation of the temporary list are certain to be part of the skyline. Those added to the candidate list after the creation of the temporary list may not be skyline points since they are not compared against points in the temporary list yet.

In the next pass of the algorithm, these points together with the ones in the temporary list are treated as input and the above process starts all over again. One of the most expensive steps in the algorithm is to compare a point with the points in the candidate list. To reduce the number of comparisons, the list is organized as a self-organizing list. When a point is found dominating other points, it is moved to the top of the list. In this way, points with high dominating power will stay on the top, and subsequent points will be compared with them first.

The advantage of BNL is its wide applicability, since it can be used for any dimensionality without indexing and sorting the data file. Actually, it can be applied to other forms of preference constraints too as long as the preference relation is specified over two tuples. The deficiencies of the algorithm are the reliance on main memory and its inadequacy for progressive processing. If the memory size is small, for large input, it may need numerous iterations to compute the results. Also, it has to read the entire data file before it returns the first skyline point.

2.1.2 Linear Elimination Sort for Skyline

In [9], the authors proposed an improved algorithm, called Linear Elimination Sort for Skyline (or LESS for short), based on BNL. Prior to the computation of skyline points, all the points are sorted first, according to the entropy, which is $\sum_i \ln d_i$, where d_i's are the values of skyline dimensions. In this way, no point in the stream can be dominated by any point that comes after it. It also has the advantage of tending to push records that dominate many records towards the beginning of the stream, assuming uniform distribution of points in the space. In the sorting stage, it makes use of an *elimination-filter* (EF) window that keeps points with small entropies, to efficiently eliminate many dominated points. The EF window effectively reduces the size of the input for the actual skyline computation later.

LESS performs better than BNL in spite of the additional sorting stage. Once a point is put in the skyline-filter window, it is confirmed to be a skyline point. Also, the introduction of the EF window efficiently eliminates many points and hence reduce the input size for skyline computation.

2.1.3 Divide and Conquer

The Divide-and-Conquer algorithm [3] divides the data set into several partitions such that each partition fits in memory. Then any known in-memory algorithms can be used to compute the partial skyline of each partition. After that, the partial skyline results are merged to produce the final result. It is interesting to note that certain partitions P can be ignored during the merging process, as the partial skyline points in some other partitions already dominates all points in P. An example would be the upper right partition (dominated by the non-empty lower left partition) in Figure 2.1.

Divide-and-conquer algorithm is efficient only for small data sets that fit in memory. For large data sets, the partitioning process requires reading and writing the entire data set at least once, thus incurring significant I/O cost.

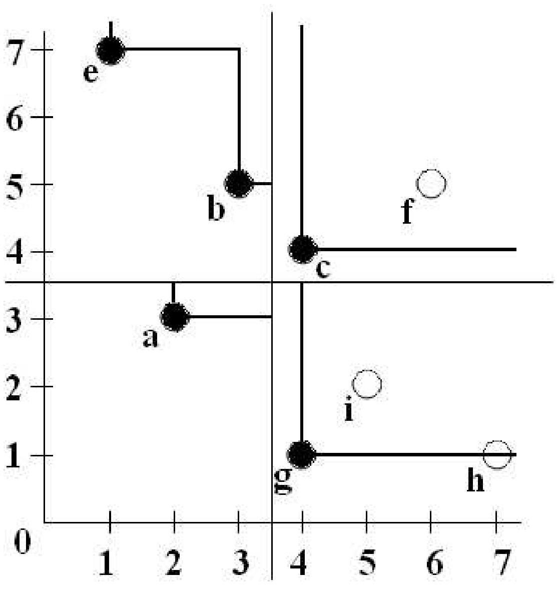


Figure 2.1: Divide and conquer

Also, like BNL, it is not suitable for on-line processing because it cannot report any skyline until the partitioning phase completes.

2.1.4 Bitmap

Bitmap technique [21] encodes every point into an n-bit binary vector, where n is the number of distinct values in all dimensions. Referring to the data set in Figure 1.1, in the x dimension, there are totally 7 distinct values; in the y dimension, there are totally 6 distinct values. So $n = 6 + 7 = 13$. Given a point p, suppose that it is the ith smallest point in the x dimension, and jth smallest point in the y dimension. Its bitmap representation would be $(\underbrace{11...1}_{7-i+1}\underbrace{00...0}_{i-1},\underbrace{11...1}_{6-j+1}\underbrace{00...0}_{j-1})$. Table 2.1 shows the bitmap representations of the set of data points in Figure 1.1. Now for a point, say c, we want to check if

id	coordinates	bitmap
a	(2, 3)	(1 1 1 **1** 1 1 0, 1 1 **1** 1 0 0)
b	(3, 5)	(1 1 1 **1** 1 0 0, 1 1 **0** 0 0 0)
c	**(4, 4)**	(1 1 1 **1** 0 0 0, 1 1 **1** 0 0 0)
e	(1, 7)	(1 1 1 **1** 1 1 1, 1 0 **0** 0 0 0)
f	(6, 5)	(1 1 0 **0** 0 0 0, 1 1 **0** 0 0 0)
g	(4, 1)	(1 1 1 **1** 0 0 0, 1 1 **1** 1 1 1)
h	(7, 1)	(1 0 0 **0** 0 0 0, 1 1 **1** 1 1 1)
i	(5, 2)	(1 1 1 **0** 0 0 0, 1 1 **1** 1 1 0)

Table 2.1: Bitmap approach

it is in the skyline. Note that in dimension 1, the least significant bit whose value is 1 is bit 4; in dimension 2, the least significant bit whose value is 1 is bit 4 too. We check whether point c is a skyline point using the following three steps.

Step 1 For each dimension, we search for the least significant bit whose value is 1 and get the *vertical* bit-slice of that bit position (e.g., $c_x = 11110100$ and $c_y = 10100111$ as highlighted in bold in Table 2.1). The we perform *and* operation of all the bit-slices. The result of this operation (e.g., $c_x \wedge c_y$=10100100) has the property that the nth bit is set to 1 if and only if the nth point has value in each dimension less or equal to the value of the corresponding dimension of the point under investigation (e.g., point c).

Step 2 For each dimension, we take the next bit-slice of each bit-slice we take in Step 1 (e.g.$c_x = 11010000$ and $c_y = 10000111$). Then we perform *or* operation of these bit-slices. The result of this operation (e.g., $c_x = 11010000 \vee c_y = 10000111 = 11010111$) has the property that the nth bit is set to 1 if and only if the nth point has some of its dimension's value less than the value of the corresponding dimension of the point under investigation (e.g., point c).

Step 3 We perform the *and* operation of the bit operation results from Step 1 and Step 2. The result (e.g., $10100100 \wedge 11010111 = 10000100$) has the property that the nth bit is set to 1 if and only if the nth point has each dimension's value less or equal to the corresponding dimension's value of the point under investigation (e.g., point c) and some of its dimension's value is strictly less than the corresponding dimension's value of the point under investigation. And the points these 1's corresponding to are the points that dominate the current point. Apparently, a skyline point should have a sequence of 0's in Step 3.

The Bitmap algorithm does not scale well when the data set size increases. In terms of time, it needs to scan the whole data set first before encoding each point using bitmap representation. In terms of space, it requires too many bits to encode just one point when we have many distinct values in each dimension. It is an I/O intensive algorithm.

2.1.5 Index

The Index method is proposed in [21]. It maintains d lists in which a point $p = (p_1, p_2, ..., p_d)$ is assigned to the ith list ($1 \leq i \leq d$), if and only if its

List 1		List 2	
$e(1, 7)$	$minC = 1$	$h(7, 1)$ $g(4,1)$	$minC = 1$
$a(2, 3)$	$minC = 2$	$i(5, 2)$	$minC = 2$
$b(3, 5)$	$minC = 3$	$f(6, 5)$	$minC = 5$
$c(4, 4)$	$minC = 4$		

Table 2.2: Index

coordinate p_i on the ith dimension is the minimum among all dimensions, i.e., $p_i \leq p_j$ for all $j \neq i$. Points of each list is sorted in ascending order of their minimum coordinate ($minC$, for short) and indexed by a B+-tree. A batch in the ith list consists of points that have the same ith coordinate (i.e., $minC$). The algorithm starts with loading the first batch of each list. It picks and processes the one with minimum $minC$. The processing of a batch involves computing the skyline points inside the batch, and, among the computed points, adding the ones not dominated by any of the already-found skyline points into the skyline list. After finishing processing a batch, it loads the next batch from the same list into memory. Then from the batches in memory, it again picks the one with the minimum $minC$ and processes it. The algorithm ends when all batches are processed or when the one of the already found skyline points has its coordinates all smaller than the $minC$'s of the next d batches. Table 2.2 shows the two lists for the two-dimensional data set in Figure 1.1.

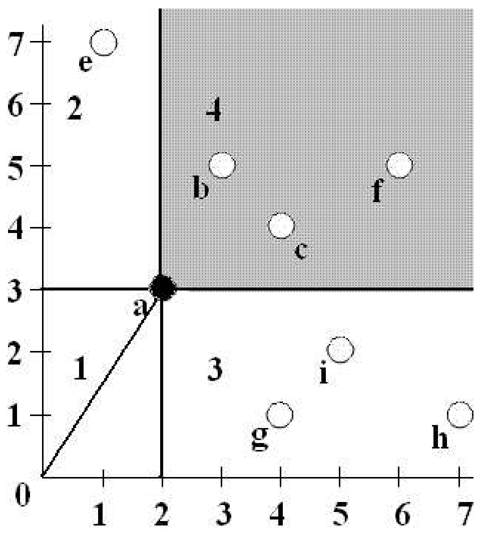
This method can return skyline points at the top of the lists fast, provided the pre-processing of the data points (i.e., distributing points to the right lists and building indexes for each list of points) can be done fast. However, it does not support retrieval of skyline points on arbitrary subset of the dimensions.

In general, in order to support queries for arbitrary dimensionality subsets, an exponential number of lists must be pre-computed.

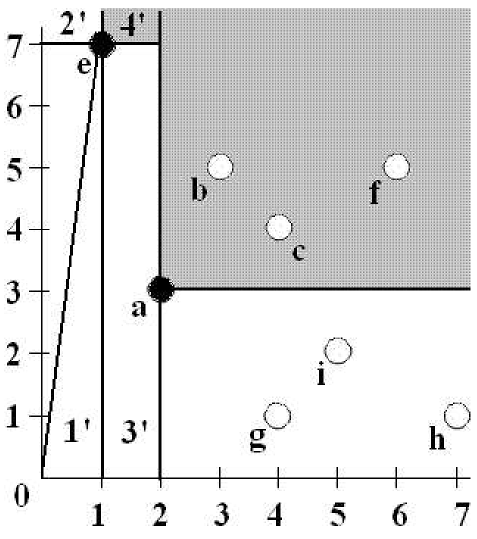
2.1.6 Nearest Neighbor

Nearest Neighbor (NN) [13] first finds the nearest neighbor point of the origin and partitions the space using that point. Then it inserts partitions that may contain skyline points into a *to-do list*. While the to-do list is not empty, it recursively does the same thing to every partition. As for the data set in Figure 1.1, Figure 2.2 illustrates the first two recursive calls to NN. Initially it finds the nearest neighbor (point a) to the origin, then divides the universe into three partitions: (i) $[0,2)[0,\infty)$ (i.e., subdivision 1 and 2), (ii) $[0,\infty)[0,3)$ (i.e., subdivision 1 and 3) and (iii) $(2,\infty)(3,\infty)$ (i.e., subdivision 4). Subdivision 4 can be pruned since it is dominated by point a. NN is applied on subdivision 1 and 2, followed by subdivision 1 and 3. For subdivision 1 and 2, the nearest neighbor is point e. Again, e divides this partition into subpartitions. Those that may contain skyline points (subdivision 1' and 2' and subdivision 1 and 3) will be explored using NN next.

For data with more than two dimensions, NN's performance is not satisfactory because there is a lot of overlapping among the partitions. Also the same skyline points may be found by some recursive applications of the algorithm. Another serious problem is regarding the size of the to-do list. It may exceed the size of the data set for as low as three dimensions.



(a) First call to partition the space



(b) Second call to partition the space

Figure 2.2: Nearest Neighbor example

2.1.7 Branch and Bound

The Branch-and-Bound Skyline (BBS) algorithm is proposed in [17]. This algorithm is able to output skyline points progressively. R-tree is used to index the multi-dimensional tuples, although other indexing techniques can be used too. Each intermediate entry (associated with Minimum Bounding Region, or MBR for short) and leaf (associated with actual data point) of the R-tree has a parameter called *mindist*, which represents the minimum distance from the origin to the entry/leaf. The *mindist* of a data point equals to the sum of all its coordinates and the *mindist* of an MBR equals to the sum of all the coordinates of its lower left point.

The algorithm, shown in Figure 2.3, starts from the root of the R-tree and insert the root entry to a heap (maintained according to *mindist* of all entries in ascending order). The algorithm always tries to expand the entry on top of the heap (i.e., the entry with smallest *mindist*) first and inserts its child nodes to the heap if they are not dominated by any skyline points discovered so far. On the other hand, if the top entry is found to be dominated by some already-discovered skyline point, the algorithm simply removes it from the heap without exploring it and goes to the next top entry on the heap. If the top entry is actually a leaf node (i.e., a data point) and not dominated by any skyline points obtained so far, that data point is a skyline point itself and the skyline list expands. In this way, the skyline points are obtained progressively in ascending order of their respective *mindist*. To speed up the dominance checking process, an in-memory R-tree is built for all the skyline points found so far. Whenever we need to check if an entry is dominated by some already-found skyline point p, we simply check whether the lower left corner of that entry falls in the dominance region of p, which is the rectangle defined by p and the edges of the universe.

It is proved that BBS is I/O optimal ($O(sh)$ where s is the size of the result and h is the height of the R-tree), meaning that it visits only the nodes that may contain skyline points and it does not access the same node twice. They also justified that the memory requirement of BBS is $\Theta(s)$ where s is the size of the skyline. The optimality of the algorithm lies in the ability to prune intermediate entries of the R-tree if they fall in the dominance regions of the already-found skyline points. These intermediate entries represent groups of points that are definitely not in skyline. Hence, there is no need to perform point-to-point comparison between skyline points and points in these groups.

```
Algorithm BBS($T$)
Input:   $T$ is an R-tree
Output:  a set $S$ of skyline points
1)  initialize heap $H$, set $S$ to be empty;
2)  insert the root entry of $T$ into heap $H$;
3)  while ($H$ is not empty) do
4)     remove top entry $e$ from $H$;
5)     if ($e$ is dominated by any point in $S$)
6)        discard $e$;
7)     else
8)        if ($e$ is an intermediate entry)
9)           for each child entry $e_i$ of $e$
10)             if $e_i$ is not dominated by some point in $S$
11)                insert $e_i$ into $H$;
12)       else
13)          insert $e$ into $S$;
14) return $S$;
```

Figure 2.3: BBS algorithm

2.2 Skyline Variants and Their Algorithms

More recently, the research community has focused on modified or extended definitions of skyline, or skyline computation in non-standard databases. In this section, we will see two variants of the skyline problems, namely the *thick skyline* and *stable skyline*. We will also review one interesting skyline computation algorithm, called *streaming skyline*, specifically applicable to streaming databases.

2.2.1 Thick Skyline

[11] proposed an extended definition of skyline, called *thick skyline*. A thick skyline includes not only the original skyline points, but also points within their ε-neighborhood. Such thick skyline points have applications in real life. For example, when a skyline hotel cannot be retrieved due to some reasons (e.g., the hotel is fully booked, although it is the "best" according to user specified criteria), users are usually willing to accept an alternative hotel that is just slightly worse. Three algorithms have been proposed for this problem. Sampling-and-Pruning algorithm tries to prune as many non-thick-skyline points as possible so that the actual computation only needs to consider a small amount of remaining points. The authors defined a strong dominating relationship–a point p strongly dominates another point q if $\forall$i, $1 \leq i \leq d$, $p_i + \varepsilon \leq q_i$ (where d is the number of dimensions) and $p_i + \varepsilon < q_i$ in at least one dimension. Firstly, it randomly samples k mutually indifferent points with high dominating capacity from the input. These k points are added to the thick skyline list S temporarily. Then, in the pruning process, if a point x is strongly dominated by a point s in S, it is removed. If it is not only a dominated point but also an ε-neighbor of s, it is added to the neighbor list of s. If x dominates s, s and x's strongly dominated neighbors are removed and x is added to the list. Finally, after the pruning process, the thick skyline of a small amount of remaining points can be computed using any method such as the Indexing-and-Estimating algorithm introduced below. Sampling-and-Pruning is not a very interesting algorithm and the experiment results showed its poor performance.

Indexing-and-Estimating algorithm is based on database indexes such as B-tree, and a smart range estimate method on the batches in the "minimum dimension" index used in [21]. The input points are partitioned into d lists such that a point $p = (p_1, p_2, ..., p_d)$ is assigned to the ith list $(1 \leq i \leq d)$ if and only if p_i is the minimum among all dimensions. Points in each list are sorted in ascending order of their minimum coordinate ($minC$, for short). It is proven in the paper that, if $p = (p_1, p_2, ..., p_d)$ is a skyline point in the batch $minC = p_i$ of the ith list, then p does not have any ε-neighbor in jth list $(j \neq i)$ if $(p_j - p_i) > \sqrt{2}\varepsilon$. It is also proven that the ε-neighbors of p can only exist in the batch range $[p_i - \varepsilon,\ p_i + \varepsilon]$ of the ith list; and the batch range $[p_j - \varepsilon,\ p_j + \frac{\varepsilon}{\sqrt{2}}]$ of the jth list $(j \neq i)$. As a direct result, if a skyline point p in the ith list is found, we only need to go back to find its ε-neighbors in the current batch of the jth list minus a sliding window of length ε. The algorithm initiates skyline list and ε-neighbors list, current batches, sliding windows and the upper bound range to scan in each list. Each point p in the minimum $minC_i$ is compared with the skyline list. If p is a skyline point, the corresponding upper bound range is updated, and part of p's ε-neighbor can be found in the sliding windows, while the others are left to the remaining accesses of the lists. When the skyline search finishes, the algorithm scans the upper bound ranges for any remaining ε-neighbors. Finally, the skyline points and their ε-neighbors are output as results.

The third algorithm called Microcluster-based algorithm partitions the database into microclusters based on CF-tree [22]. Microcluster is a technique for compressing and summarizing large amount of points. For mining of thick skyline, the database is partitioned into a set of microclusters with radius r_i (r_i can be around ε) in the leaf nodes of an extended CF-tree. Each non-leaf node represents a larger microcluster consisting of all its sub-microclusters. One microcluster A dominates another microcluster B, if the centroid of A dominates a virtual point in B whose coordinates are the minimum values of all the points in B in each dimension. The algorithm first identifies the microclusters that contain skyline points. These skyline microclusters are obtained by traversing the CF-tree in ascending order of *mdist* (the minimum distance from the microcluster to the origin), and then inserted into a heap according to *mdist*. When all skyline microclusters have been identified, the algorithm finds the skyline points in each microcluster. For all the skyline points found in one microcluster M, a group ε-neighbors search is launched by searching ε-neighboring microclusters. Points in the ε-neighboring microclusters are examined to see if they are ε-neighbors of skyline points in M. Experimental results show that the Indexing-and-Estimating and Microcluster-based algorithms outperform the Sampling-and-Pruning algorithm.

2.2.2 Stable Skyline

As another variant of the skyline definition, [8] proposed two extensions to the original definition. In addition to the existing *MIN*, *MAX*, and *DIFF* criterion directives, they introduced a new criterion directive *EQUAL* and a criterion modifier *BY*. The *EQUAL* criterion directive applied on some attribute a_i indicates that two tuples are not comparable if their a_i values are equal. This is just the opposite of the *DIFF* criterion directive which specifies that two tuples are not comparable if their a_i values are different. The BY criterion modifier allows us to enforce a stronger criterion in judging the dominance relation between two tuples. For example, the criterion *price MIN* BY $5,000$ means that tuple A is better than tuple B only if A's price is at least \$5,000 less expensive than B's. These extensions increase the expressiveness of the skyline operator. However, they also result in loss of transitivity in semantics (to be discussed later). In particular, the BY modifier even introduces cycles to the dominance relations.

How can $EQUAL$ affect transitivity? An $EQUAL$ operator prohibits tuples having same values on the $EQUAL$ dimensions to relate. In essence, it punches holes in the partial order of the preference relation that would be induced by the filter without its equal comparators, by making certain pairs to tuples incomparable which would have been comparable otherwise. These "holes" can violate transitivity. The other two properties of partial order, namely irreflexivity and asymmetry, are still preserved, so the preference relation is a DAG (Direct Acyclic Graph). As an example, let tuples A and C have the same value on dimension d_1, and tuple B have a different value on attribute d_1. Also $A.d_2 > B.d_2 > C.d_2$. For the skyline clause "skyline of d_1 $EQUAL$, d_2 *MAX*", A dominates B, B dominates C, but A and C are incomparable. Only A is in the skyline set. However, when B is removed from the input, C is also in the skyline set. In other words, the addition or deletion of non-skyline tuples from the input can affect what the skyline set

id	address	price	#bdrm	cond
1	32 Dover Rd	\$356 K	4	4
2	11 Linden Dr	\$353 K	2	5
3	27 West Coast Rd	\$350 K	3	3

Table 2.3: HouseListing

is. This situation is referred to as the instability of skyline. As a remedy, the authors redefined stability in the following way. A stable skyline set is obtained by including all skyline points in the set first, and then iteratively searching for points that are not dominated by any point already in the set. The authors proved that in a finite number of iterations, a fixed set of points will be found. This set is called the stable skyline set. The stable skyline set is a superset of the original skyline set. When the skyline query induces a partial order, the two sets are the same.

Cycles may be introduced in the dominance relations when we add the BY clause to the criterion. As an example, consider the following skyline query and the table $HouserListing$ as in Table 2.3.

SELECT *address*, *price*, *#bdrm*, *cond*
FROM $HouseListing$
SKYLINE OF *price* ***MIN BY*** **5000,** *#bdrm* ***MAX BY*** **2,** *cond* ***MAX BY*** **2**

Tuple 1 dominates tuple 2, tuple 2 dominates tuple 3, and tuple 3 dominates tuple 1. The preference relation is not even a DAG any more. It can be assumed that user does not really intend to specify cyclic preference relations, and there is no suitable semantics for preference relations with cycles.

The way to remedy it is to add in a judiciously chosen skyline *ground comparator*, which is comparator without the BY modifier. The skyline clause that contains a ground comparator, called a *ground filter* is guaranteed to be cycle free. This purposely added ground comparator should perturb the original preference relation as little as possible. It should, in essence, only affect the cycles. Such a comparator is not unique and the author gave one such comparator in the paper. The addition of the proper ground operator is to approximate user's intended preference relation by a cycle-free preference relation.

This paper extends the skyline definition from a rather theoretical angle. It enriches the semantics of skyline queries by introducing additional criterion directive and modifier. However, it is not clear how efficiently the new skyline query can be evaluated based on existing techniques.

2.2.3 Skyline Computation in Streaming Databases

An interesting algorithm to compute the skyline in a streaming database is proposed in [15]. In particular, the authors studied the problem of skyline computation with respect to the most recent N elements which can fit in the main memory. They investigated two types of stream computation models: *n-of-N* model and *(n_1, n_2)-of-N* model. The *n-of-N* model deals with the computation of skyline of any most recent n ($n \leq N$) elements. *(n_1, n_2)-of-N* model is a generalization of the *n-of-N* model: instead of dealing with skylines of the most recent n elements, it retrieves skylines between the most n_2-th recent element and the most n_1-th recent element (for any $n_1 \leq n_2 \leq N$).

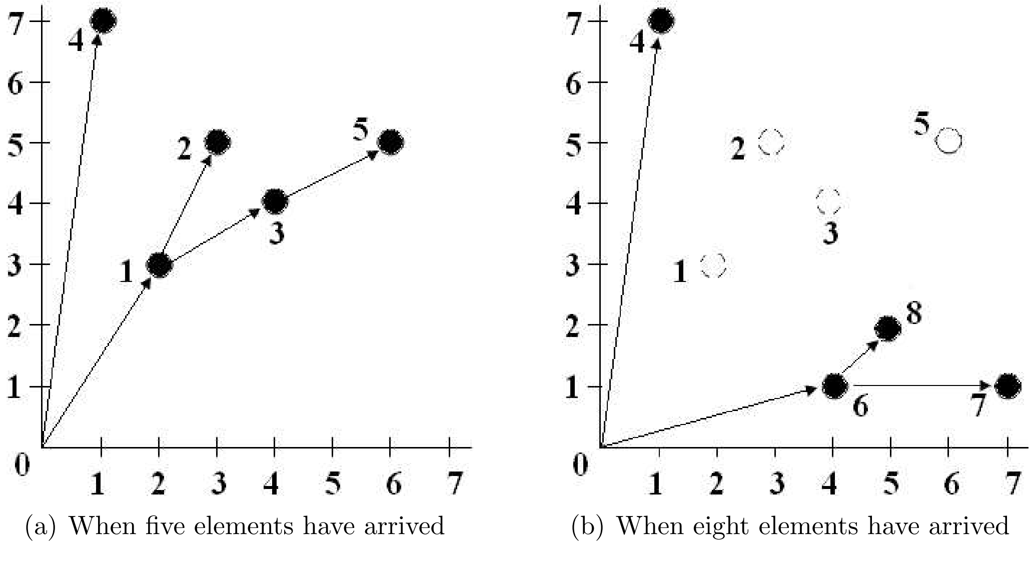
In the context of skyline computation in streaming database, a data element e is *redundant* with respect to the most recent N elements if e is expired (i.e. outside the most recent N elements) or is dominated by a younger element e'. The set of non-redundant elements R_N are the minimum set of elements that needs to be kept for *n-of-N* computation. An element e in R_N can be dominated by many other elements in R_N that arrive earlier than e. It is not necessary to keep all such dominance relations. Among these dominance relations, only a small number of critical relations are needed. In R_N, a dominance relation $e' \rightarrow e$ is critical if and only if e' is the youngest one (but older than e) that dominates e. Hence, the dominance graph (the graph that contains R_N as the vertex set and the dominance relation as the edge set) is a forest. To encode the graph is straightforward: every edge $e' \rightarrow e$ is represented by the interval $(k(e'), k(e)]$, and every root e is represented by the interval $(0, k(e)]$, where $k(e)$ means the element e arrives $k(e)$th in the data stream. Given n for the *n-of-N* query, an element e in R_N is in the answer if and only if $k(e)$ is the right end of an interval $(a, k(e)]$ that contains $M - n + 1$, where M is the number of elements seen so far. Because the data keeps streaming, the encoding scheme needs to be kept updated. When a new element e_{new} arrives, three steps are involved in maintaining the scheme.

1. If the oldest element e_{old} in R_N expires, remove it from R_N, and also remove the interval $(0, k(e_{old})]$. All intervals $(k(e_{old}), k(e)]$ need to be updated to $(0, k(e)]$.

2. Remove the intervals whose either end is dominated by e_{new}, R_N is also updated by removing the dominated elements and adding in e_{new}.

3. Find the element e that critically dominate e_{new}, add $(k(e), k(e_{new})]$, or $(0, k(e_{new})]$ if such an e does not exist, to the interval set.

Figure 2.4 shows the interval trees (in two different time instances) of the data set in Figure 1.1 when $N = 5$, assuming that the elements arrives in alphabetic order. We rename the elements using their arrival sequence for easy reference. When only five elements have arrived, the interval tree is shown on the left. When eight elements have arrived, earlier elements 1, 2 and 3 are expired and element 5 is dominated by younger elements. These four elements are hence redundant and not needed in R_N any more. The update interval tree is shown on the right. With the encoding scheme well



(a) When five elements have arrived (b) When eight elements have arrived

Figure 2.4: *n-of-5* encoding scheme of data set in Figure 1.1

maintained, the *n-of-N* queries can be answered efficiently because when a new element e_{new} arrives, only two types of changes may happen to the current result set S_n. A data element e is *deleted* from S_n if e_{new} dominates e or e is expired. A data element e in R_N is *added* to S_n if in the updated dominance graph after inserting e_{new}, one of the following two happens: 1) e is e_{new} and e' such that $k(e') \geq M - n + 1$ and $e' \rightarrow e_{new}$, or 2) e is critically dominated by the just expired element e'' in S_n and e'' is not dominated by e_{new}.

For *(n_1, n_2)-of-N* queries, similar algorithms apply. However, we need to keep all the most recent N elements P_N instead of just R_N. One extra information to maintain for each element e is the oldest element e' that dominates e and arrives after e. Such elements are denoted by b_e. As in the *n-of-N* query processing, the algorithm for *(n_1, n_2)-of-N* queries stabs the intervals by $M - n_2 + 1$. For the right end element e of each stabbed interval, they are the results for *n-of-N* queries. However, in *(n_1, n_2)-of-N* queries, we still need to check if $k(e) \leq M - n_1 + 1 < k(b_e)$. Only those e's that also satisfy this inequality are included in the result of *(n_1, n_2)-of-N* queries.

There are several problems concerning the efficiency and flexibility of the algorithms. Efficiency is not guaranteed by a theoretically proven bound for maintaining R_N and the encoding scheme. When a new element e_{new} arrives, we need to find within R_N those elements dominated by e_{new}. Also we need to find the element that critically dominates e_{new}. These two operations are done through building an in-memory R-tree. Due to the sophisticated cost model of R-tree, it is unrealistic to have a proper bound for maintenance of the encoding scheme. The other problem is that the current approach is not able to answer skyline queries on arbitrary subset of the dimensions. This is because the online determination of the critical dominance relationship and building of the interval tree are all based on the assumption that the skyline dimensions are known prior to data streaming. In fact, to handle skyline queries on arbitrary dimensions, the algorithms need to maintain a large number of interval trees, one for each possible subset of the dimensions. This is not a practical solution because of the high cost, in terms of both memory and time needed to maintain the structures.

Chapter 3

Dominant Skyline Queries

Dominant skyline queries are used to refine a large set of skyline points into a smaller and more interesting set of points. Given a set of data records S, a skyline query Q, and a dominating power[1] threshold t, we want to retrieve all the records, each of which belongs to the result of Q and dominates at least t other records in S.

In this chapter, we firstly provide the insights of the problem. Then, we propose and discuss in detail several two-step approaches based on pruning techniques and heuristic functions. Lastly, we present the experimental results of various algorithms.

[1]The *dominating power* of a skyline point is the actual number of points dominated by the skyline point.

3.1 Insights of the Problem

Dominant skyline computation is different from the standard skyline computation in the following way. In standard skyline computation, we only need to retrieve skyline points and the dominated points can be discarded as soon as possible. However, in this variant, we not only need to compute the skyline points but also their dominating powers (or, at least the lower bounds of their dominating powers). Dominated points cannot be discarded too soon because they may be dominated by multiple skyline points, some of which are yet to be discovered.

A naive way to compute this type of queries consists of two steps.

Step 1 Compute the skyline points with any of the known algorithms.

Step 2 Compute the dominating power of all skyline points by scanning the whole data set.

This naive approach was proposed in [17]. Without this naive approach, the existing algorithms cannot solve the dominant skyline queries efficiently.

Block Nested Loop

In Block Nested Loop approach, dominated points in earlier passes are discarded. If they are dominated by skyline points discovered in later passes, we have no way of counting the correct dominating powers of these skyline points unless we store the dominated points somewhere in main memory or disk.

LESS

Similar to BNL, LESS also needs to keep the dominated points for later processing, which requires more iterations for the dominant skyline points computation.

Bitmap

The Bitmap approach can handle the dominant skyline problem with a small modification. For example, in Figure 1.2, after we find out that point a is a skyline point, we want to know the dominating power of point a. Firstly, as we did previously, we get $a_x = 10010000$, $a_y = 10000111$. Now we compute $\neg(a_x \wedge a_y) = 01111111$. The result of this operation has the property that the nth bit is set to 1 if and only if the nth point has value in some dimension greater than the value of the corresponding dimension in point a. Secondly, we get the bit-slices following the bit-slices we get previously (i.e., $a_x = 00010000$, $a_y = 00000111$). Now we compute $\neg(a_x \vee a_y) = 11101000$. The result of this operation has the property that the nth bit is set to 1 if and only if the nth point has values in each dimension greater than or equal to the values of the corresponding dimension in point a. Lastly, we perform *and* operation of the two bit-slices (i.e., $01111111 \wedge 11101000 = 01101000$). The result of this operation has the property that the nth bit is set to 1 if and only if the nth point is dominated by point a. Hence, the number of 1's in the result is the dominating power of point a.

However, as mentioned in Section 2.1.4, the algorithm has several shortcomings (e.g., I/O intensiveness) that render it unsuitable for processing large input data. Also, the above mentioned technique can only be applied *after* the skyline points are discovered.

Index

Index method distributes a point to a dimension list according to its minimum dimension and computes skyline points from the top of all lists. Skyline points reside near the top of the lists. However, there may not be dominance relation between points on the top of one list and the points at the bottom. Pair wise comparisons are needed between skyline points and dominated points, to compute the dominating power of the skyline points.

Nearest Neighbor

When a skyline point is discovered, the dominance region of the point is pruned from further consideration. However, the dominance regions of multiple skyline points overlap with each other, i.e., a point may be dominated by several skyline points. If it is pruned early, we will overlook the possibility that it may be also dominated by some of the yet-to-be-discovered skyline points. Even if we do not prune them, the best we can do is still counting the dominated points for each skyline point.

Branch and Bound

In the BBS algorithm, an intermediate R-tree entry is discarded immediately after it is found dominated by some skyline point. This entry may be dominated by other skyline points that are yet to be discovered, similar to NN case. Therefore, for dominant skyline computation, if we want to employ

BBS, we cannot discard such an entry even if it is found dominated.

A nice property of BBS is that when an R-tree entry is found dominated, we do not have to branch down that entry further. We want to inherit this property for computing dominant skyline points. One idea is to pre-compute and store in each R-tree entry the number of points enclosed by that entry's MBR. We call it the *size* of an entry. When an entry is found dominated by a skyline point p, we can increase the dominating power of p by the size of the entry directly. By doing this, we can stop traversing down the subtrees rooted at one entry once it is found dominated by a skyline point. The saving is more significant if the entry is nearer to the root of the R-tree. However, the dominance region of a skyline point may not contain an R-tree entry completely, as illustrated in Figure 3.1. The dominance region of point g overlaps with the R-tree entry on the left. In such a case where an entry is not completely contained in the dominance region of a skyline point, we have to branch down the entry further. A similar idea was proposed in [14], which deals with answer approximation for aggregate queries (not skyline queries).

From the above discussion, it is clear that to solve the dominant skyline problem efficiently is not a trivial task. None of the existing algorithms can solve it without using the naive two-step approach. We want to improve the naive approach with some pruning and heuristic techniques. Section 3.2 presents the ideas of our algorithms.

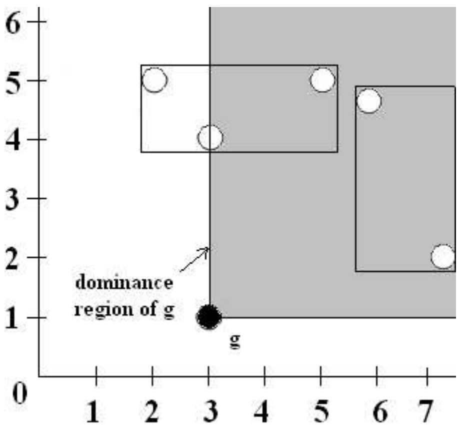


Figure 3.1: Overlapping between a dominance region and an R-tree entry

Naive two-step Approach	Enhanced Approach
Step 1: compute skylines	Step 1: compute skylines, at the same time, output definite dominant skyline points and prune definite non-dominant skyline points
Step 2: compute the dominating powers of *all* skyline points	Step 2: compute the dominating powers of the *remaining* candidate dominant skyline points

Table 3.1: Naive approach vs. enhanced approach

3.2 An Improved Two-step Approach

Table 3.1 compares the naive two-step approach with our proposed approach. In essence, we push part of the work from Step 2 to Step 1 by trying to determine as many dominant and non-dominant skyline points as possible in Step 1 and confirm the rest of the dominant skyline points in Step 2. Note that a dominant skyline point can be confirmed as long as the lower bound of its dominating power exceeds the specified threshold.

3.2.1 Step 1: Using BBS with Pruning

Step 1 is based on BBS for its nice property mentioned in Section 3.1. However, to adapt to the problem specifically, we need the following parameters maintained together with the the R-tree data structures.

Firstly, we associate a parameter called *size* with each R-tree entry. The *size* of an R-tree entry is the number of points enclosed by the MBR corresponding to the entry.

Secondly, we associate two parameters with a skyline point p.

LDP The lower bound of dominating power, calculated by summing the sizes of all R-tree entries (processed so far) completely contained by p's dominance region.

UDP The upper bound of dominating power, calculated by adding to LDP, the sizes of all R-tree entries (processed so far) that partially overlap with p's dominance region.

As an example, $g.LDP = 2$ and $g.UDP = 5$ in Figure 3.1.

In Step 1, we calculate the skyline points (making use of BBS), and at the same time, output definite dominant skyline points and prune definite non-dominant skyline points. Definite dominant skyline points are skyline points with $LDP \geq t$ and definite non-dominant skyline points are skyline points with $UDP < t$. Skyline points with $LDP < t$ but $UDP \geq t$ are called candidate dominant skyline points. After Step 1, all the candidate dominant skyline points are grouped into a set P. Besides P, we also maintain a set E of the R-tree entries that overlap[2] with some point(s) in P. Note that there is no parent-child or ancestor-descendant relations exist among all the R-tree entries in E. That is to say, if an R-tree entry is found overlapping with a candidate point, it will be inserted into E and not explored further until perhaps in Step 2 later. The overlapping relation between e_i's in E and p_j's in P can be modeled as a bipartite graph $BiGraph$. The two sets of vertices comprise of elements from E and P respectively. An edge connecting one e_i with one p_j represents their overlapping relation.

The pseudo code of Step 1 is depicted in Figure 3.2, Figure 3.3 and Figure 3.4.

In algorithm DomBBS (Figure 3.2), as in BBS, we initialize the heap H with the insertion of the R-tree root node into the heap (line 2). Set S contains all the skyline points found so far. Set $DefDom$ contains all the definite dominant skyline points (i.e., dominant skyline points confirmed so far). $DefDom$ is a subset of S. While the heap is not empty, we remove the top entry e (having the shortest *mindist*) from the heap (line 4) and examine it. Firstly, we get, from S, the set of skyline points that dominate e (line 5). If e is indeed dominated by some already found skyline point (i.e., Dom is not empty in line 6), we get from $S - DefDom$, the set of skyline points whose dominance regions overlap with e's MBR (line 7). Then, we update the LDP and UDP of the related skyline points. If e is an intermediate

[2]By "a point overlaps with an entry", we mean that the *dominance region* of the point partially overlaps with the entry.

```
Algorithm DomBBS($T$, $t$)
Input:    $T$ is an R-tree
          $t$ is a threshold
Output:   $DefDom$ is a set of definite dominant skyline points
          $BiGraph$ is a bipartite graph
1)  initialize heap $H$, bipartite graph $BiGraph$, set $S$, $DefDom$ to be empty;
2)  insert the root node of $T$ into heap $H$;
3)  while ($H$ is not empty)
4)    remove top entry $e$ from $H$;
5)    $Dom$=points in $S$ dominating $e$;
6)    if ($Dom$ is not empty)
7)      $Overlap$=points in $S - DefDom$ overlapping with $e$;
8)      $DefDom$=UpdateDP($Dom$, $Overlap$, $e.size$, $t$, $DefDom$);
9)      if ($e$ is an intermediate entry)
10)       UpdateBiGraph($Overlap$, $\{e\}$, $BiGraph$);
11)   else //$e$ is not dominated by any already found skyline points
12)     if ($e$ is an intermediate entry)
13)       for each child $e_i$ of e
14)         $Dom$=points in $S$ dominating $e_i$;
15)         if ($Dom$ is empty)
16)           insert $e_i$ to $H$;
17)         else
18)           $Overlap$=points in $S - DefDom$ overlapping with $e_i$;
19)           $DefDom$=UpdateDP($Dom$, $Overlap$, $e_i.size$, $t$, $DefDom$);
20)           UpdateBiGraph($Overlap$, $\{e_i\}$, $BiGraph$);
21)     else //$e$ is a data point
22)       insert $e$ to $S$;
23)       $DomEntries$=R-tree entries pruned earlier dominated by $e$;
24)       $OverlapEntries$=R-tree entries pruned earlier overlapping with $e$;
25)       for each entry $e_i$ in $OverlapEntries$
26)         $e.UDP+=e_i.size$;
27)       for each entry $e_i$ in $DomEntries$
28)         $e.LDP+=e_i.size$;
29)         $e.UDP+=e_i.size$;
30)       if ($e.LDP \geq t$)
31)         add $e$ to $DefDom$;
32)       else
33)         UpdateBiGraph($\{e\}$, $OverlapEntries$, $BiGraph$);
34) remove from $BiGraph$ points whose $UDP < t$;
35) return $< DefDom, BiGraph >$;
```

Figure 3.2: DomBBS

```
Algorithm UpdateDP ($Dom$, $Overlap$, $size$, $t$, $DefDom$)
Input:   $Dom$ is the set of points dominating an R-tree entry
         $Overlap$ is the set of points overlapping with the R-tree entry
         $size$ is the size of the R-tree entry
         $t$ is the threshold
         $DefDom$ is the set of current definite dominant skyline points
Output:  an updated $DefDom$
1)   for each point $p$ in $Dom$
2)     if ($p$ is not in $DefDom$)
3)       $p.LDP+ = size$;
4)       $p.UDP+ = size$;
5)       if ($p.LDP \geq t$)
6)         add $p$ to $DefDom$;
7)         remove $p$ from $BiGraph$;
8)   if ($Overlap$ is not empty)
9)     for each point $p$ in $Overlap$
10)      $p.UDP+ = size$;
11)  return $DefDom$;
```

Figure 3.3: UpdateDP

```
Algorithm UpdateBiGraph($Points$, $Entries$, $BiGraph$)
Input:   $Points$ is a set of skyline points
         $Entries$ is a set of R-tree entries
         $BiGraph$ is the current bipartite graph
Output:  an updated $BiGraph$
1)   for each point $p$ in $Points$
2)     if ($p$ is not in $BiGraph$)
3)       add $p$ to $BiGraph$;
4)   for each entry $e$ in $Entries$
6)     compress $e$ to $e'$;
7)     add e' to $InMemRtrees$;
8)     if ($e'$ is not in $BiGraph$ & e is not a point)
9)       add $e'$ to $BiGraph$;
10)  add edges between newly added $Points$ and $Entries$ in $BiGraph$;
11)  return $BiGraph$;
```

Figure 3.4: UpdateBiGraph

R-tree entry (line 9), we update the bipartite graph accordingly (line 10), as explained later in algorithm UpdateBiGraph. If, on the other hand, e is not dominated by any already found skyline points (line 11), we deal with it based on whether it is an intermediate R-tree entry. If e is an intermediate entry (line 12), we explore the entry. For each child entry, we insert it into the heap if it is also not dominated by any point in S (lines 15, 16). If a child entry e_i of e is dominated by some point in S, we get two sets of points which dominate e_i and overlap with e_i respectively (lines 14, 18); update the LDP and UDP for points in the two sets (line 19); and then update BiGraph (line 20). If e is not an intermediate entry but a data point (line 21), then it is confirmed to be a skyline point. We insert e into S (line 22). Note that e's dominance region may overlap with some entries pruned earlier. Also, e may dominate entries pruned earlier. So we need to update UDP and LDP of e (lines 25-29) with the $size$ of the related pruned R-tree entries. If e is hence confirmed dominant skyline point (line 30), e is included in $DefDom$ (line 31). Otherwise, e becomes one of the candidate dominant skyline points and is added to $BiGraph$ (line 33). Finally, after the heap is empty, we pruned away skyline points with $UDP < t$, and return points with $LDP \geq t$ (definite dominant skyline points), together with the bipartite graph $BiGraph$ (lines 34, 35). Note that $InMemRtrees$ are maintained for the pruned R-tree entries (or points). Indeed, two in-memory R-trees are maintained. One in-memory R-tree is built on all the lower left corner points of the pruned R-tree entries. The other in-memory R-tree is built on all the upper right corner points of the pruned R-tree entries. These two in-memory R-trees are kept for quick computation of the sets in line 23 and line 24.

To get the pruned entries (or points) dominated by a point e in line 23, we just need to get all the lower left points enclosed by the dominance region of e, which is a simple containment query on the first in-memory R-tree. Similarly, to get the pruned entries (or points) overlapping with a point e in line 24, we just need to get the set of all the upper right points enclosed by the dominance region of e and substract from it the points we found in line 23.

The UpdateDP method (Figure 3.3) updates the LDP and UDP for points in *Dom* unless they are already confirmed dominant, and UDP for points in *Overlap*. The set *Dom* keeps points found dominating an R-tree entry e. The set *Overlap* keeps points whose dominance regions are found overlapping with e's MBR. Finally, UpdateDP returns the updated list $DefDom$ of points with $LDP \geq t$.

The UpdateBiGraph method (Figure 3.4) updates the in-memory R-trees and the bipartite graph *BiGraph*. UpdateBiGraph adds a skyline point(or skyline points) and the overlapping R-tree entries(or an R-tree entry, respectively) into *BiGraph* when the overlapping relations are discovered in DomBBS. In line 6, to compress an entry e essentially means to compute a tuple <entry id, the lower left corner coordinates of e's MBR, upper right corner coordinates of e's MBR, the *size* of e > from the entry e. After this tuple is inserted into $InMemRtrees$ (and perhaps *BiGraph* if e is an intermediate entry), the actual entry can be removed from memory. Note that we will never update *BiGraph* with more than one skyline point and more than one R-tree entry at the same time.

Figure 3.5 shows the input to and output from Step 1.

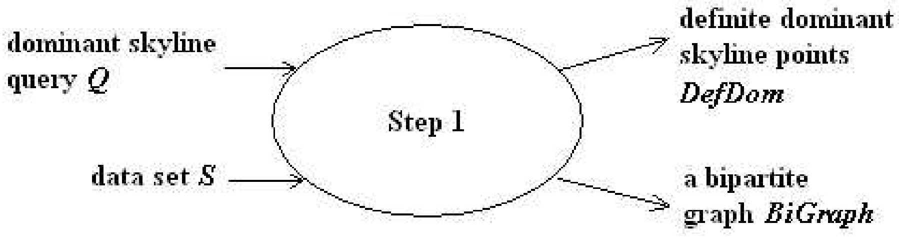

dominant skyline query Q
data set S
Step 1
definite dominant skyline points $DefDom$
a bipartite graph $BiGraph$

Figure 3.5: Input and output of Step 1 based on BBS

3.2.2 Step 2: Confirming Dominant Points with Heuristics

After Step 1, we have confirmed some definite dominant skyline points (with $LDP \geq t$) and pruned some definite non-dominant skyline points (with $UDP < t$). We are left with a bipartite graph *BiGraph* consisting of a set P of skyline points p_i's with $LDP < t$ but $UDP \geq t$ and a set E of compressed R-tree entries e_i's that overlap with the dominance regions of some p_i's. In Step 2, we want to explore the e_i's in E, to confirm the remaining dominant skyline points in P. We want to avoid exploring the same e_i twice if it overlaps with more than one candidate dominant skyline points.

We also hope to confirm the remaining definite dominant skyline points while exploring as few e_i's as possible. The size of E keeps changing while we are exploring the entries in E, because after exploring one entry e_i, we may add some/all of the child entries of e_i to E, and we may also eliminate some entries from E. How can exploring an entry e_i eliminate some other entries in E? An entry is eliminated from E if it is no longer needed for the purpose of confirming the rest of the dominant skyline points. This happens when all of the following three conditions are satisfied.

1. Exploring e_i will make some points overlapping with e_i definitely dominant or non-dominant;
2. Making these points, say p_j, definite will render exploring, for p_j, other entries that overlap with p_j no longer necessary;
3. Some of these entries only overlap with p_j.

If all of the three conditions are satisfied, then entries in condition 3 can be removed from E after exploring e_i.

With that in mind, it is obvious that a certain order of exploration of e_i's will incur fewer number of page accesses (i.e., exploring fewer e_i's in E) than other orders. We hope to weigh the e_i's so that always exploring the highest weighed entry first will give us a good order of exploration (in terms of the number of page visits). According to the observations mentioned earlier, an entry is of greater value if exploring it can remove more other entries from E and add fewer new entries to E. However, the latter is hard to predict unless we actually explore the entry. Hence, we will weigh each entry according to the former only. The ability of removing other entries relies on the "quality" of points p_j's that overlap with e_i. p_j is "good" if p_j overlaps with many entries, each of which only overlaps with p_j.

Let $degree_{e_i}$ of an e_i in E be the number of points in P overlapping with e_i. The weight of a point W_{p_j} should be inversely proportional to $degree_{e_i}$ for any e_i's overlapping with p_j. In other words, for an e_i overlapping with p_j, the smaller $degree_{e_i}$ is (i.e., fewer number of points overlapping with e_i), the better p_j is. The reason is that, confirming such an p_j is more likely to eliminate those e_i's overlapping with it. Also, the weight of a point W_{p_j} should be proportional to the number of e_i's overlapping with p_j. Hence, we use the following formula to calculate the weight of a point p_j. $W_{p_j} = \sum_{e_i' s\ connected\ with\ p_j} \frac{1}{degree_{e_i}}$

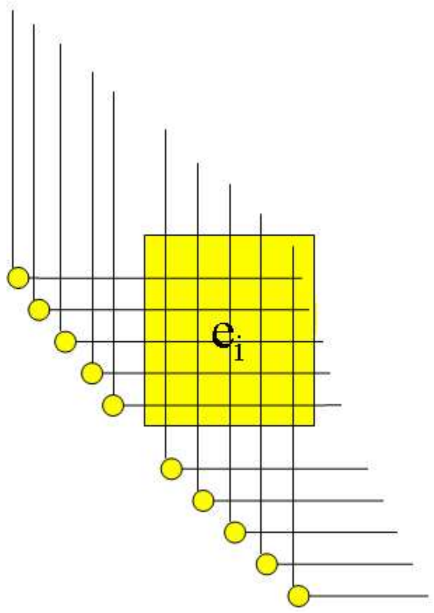


Figure 3.6: An extreme example showing that an entry should not receive *full* weights from every overlapping point

Now, an entry e_i is "good" if exploring it can confirm many overlapping points (being dominant or non-dominant) and these points are "good" points. It is natural to let e_i receive weights from all points overlapping with it. We need to address two questions here.

First question is that, does e_i receive the *full* weight of every point overlapping with it? Consider the example in Figure 3.6, where e_i overlaps with many points. These points are all very good points because every one of them only overlaps with e_i. If e_i receives full weights of all the points, e_i will be weighed very high, i.e., e_i is an entry worth exploration. However, exploring e_i will not remove any any other entries at all because all the points overlapping with e_i only overlap with e_i. To avoid mistakenly weighing such entries too high, the amount of weight that e_i receives from p_j should be at most $W_{p_j e_i}{=}W_{p_j} - \frac{1}{degree_{e_i}}$, which is the weight of p_j minus the portion that is contributed by e_i itself. Following this formula, the amount of weight that e_i receives from every point overlapping with e_i is zero.

The next question is that, does e_i receive the *full* portion of $W_{p_j e_i}$ for every p_j overlapping with e_i? We propose four ways (based on heuristics) to define the portion of weight $\sigma_{p_j e_i}$ that e_i receives from $W_{p_j e_i}$. These four ways are all based on one common observation as depicted in Figure 3.7. Recall that candidate skyline points have $LDP < t$ and $UDP \geq t$. As we explore entries in Step 2, skyline points' LDPs increase and UDPs decrease. If after exploring an entry, we can bring down UDP below t or lift up LDP above t, such an entry is "good" because by exploring it, we are able to confirm a dominant or non-dominant skyline point.

The four heuristic functions can be categorized as in Table 3.2. Heuristic Function 1 and 2 assume skewed distribution of points in MBRs while Heuristic Function 3 and 4 assume uniform distribution. Heuristic Function 1 and 3 make the portion $\sigma_{p_j e_i}$ larger if exploring e_i is likely to make p_j definitely

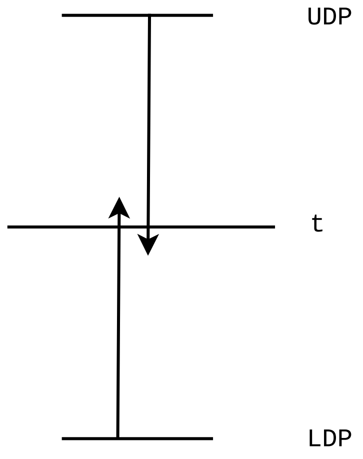


Figure 3.7: Effect of exploring entries in Step 2 for a candidate dominant skyline point

	Favoring entries exploring which may confirm dominant points	Favoring entries exploring which may confirm non-dominant points
Skewed distribution	Heuristic Function 1	Heuristic Function 2
Uniform distribution	Heuristic Function 3	Heuristic Function 4

Table 3.2: Categorization of four heuristic functions

dominant, estimated based on the respective distribution of points. On the contrary, Heuristic Function 2 and 4 make $\sigma_{p_j e_i}$ larger if exploring e_i is likely to make p_j definitely non-dominant.

Heuristic Function 1

We assume that *all* points enclosed by entry e_i's MBR are in the dominance region of point p_j, i.e., the points are in the shaded region of Figure 3.8.

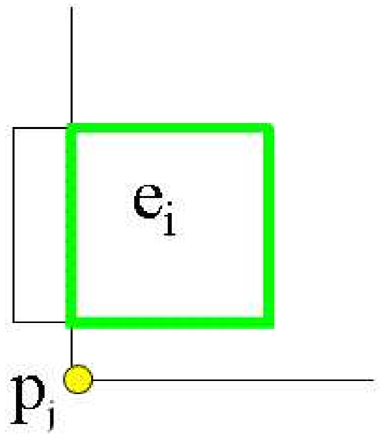


Figure 3.8: Heuristic Function 1 assumes all points of e_i are inside the framed region

Then

$$\sigma_{p_j e_i} = \begin{cases} 1, & \text{if } size_{e_i} \geq t - LDP_{p_j}; \\ 0, & \text{otherwise.} \end{cases}$$

i.e., e_i receives full weight of $W_{p_j e_i}$ from p_j if it can make p_j definitely dominant; zero otherwise. The intuition is that e_i is a good entry with respect to p_j if exploring it can make p_j definite dominant.

Heuristic Function 2

We assume that *all* points enclosed by entry e_i's MBR are not in the dominance region of point p_j, i.e., the points are in the shaded region of Figure 3.9. Then

$$\sigma_{p_j e_i} = \begin{cases} 1, & \text{if } size_{e_i} \geq UDP_{p_j} - t; \\ 0, & \text{otherwise.} \end{cases}$$

i.e., e_i receives full weight of $W_{p_j e_i}$ from p_j if it can make p_j definitely non-dominant; zero otherwise. The intuition is that e_i is a good entry w.r.t p_j if

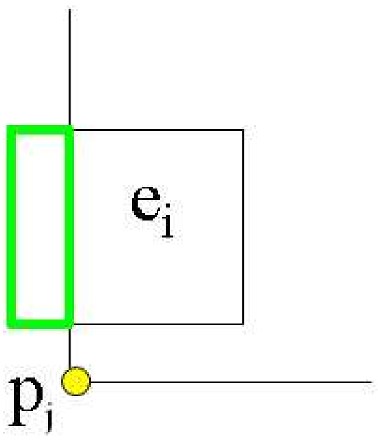


Figure 3.9: Heuristic Function 2 assumes all points in e_i are inside the framed region

exploring it can make p_j definite non-dominant.

Heuristic Function 3

The previous functions assume that the distribution of the points within an MBR is skewed. If we assume uniform distribution of points within an MBR, we can get different $\sigma_{p_j e_i}$. Heuristic Function 3 and 4 exploit this.

Consider the example in Figure 3.10. Suppose the overlapping region between e_i and p_1 contains enough points to make p_1 definitely dominant and the overlapping region between e_i and p_2 contains fewer points to make p_2 definitely dominant. e_i should receive a larger portion of weight from $W_{p_1 e_i}$ and a smaller portion of weight from $W_{p_2 e_i}$. However, without actually exploring e_i, we can only estimate the number of points contained in the overlapping region of a skyline point and an entry. The estimated number, assuming uniform distribution of points within one MBR, is given as

$N_{p_je_i}=\frac{S(Ovl_{p_je_i})}{S(e_i)}\times size_{e_i}$, where $S(Ovl_{p_je_i})$ is the volume of the overlapping region between p_j and e_i and $S(e_i)$ is the volume of the MBR corresponding to e_i. According to the example in Figure 3.10, a natural way to define the

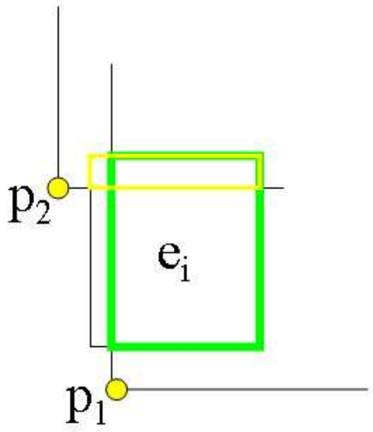


Figure 3.10: Heuristic Function 3 assumes uniform distribution

portion of weight that e_i receives from a point p_j is $\sigma_{p_je_i} = \frac{N_{p_je_i}}{t-LDP_{p_j}}$. In fact, $\sigma_{p_je_i}$ can exceed 1, which means the estimated number of points contained in the overlapping region is more than enough to make skyline point p_j dominant. This is one way to define the portion, which actually assumes that an entry is "good" if exploring it can confirm many *dominant* skyline points. It overlooks the fact that an entry is also "good" if exploring it can confirm many *non-dominant* skyline points.

Heuristic Function 4

Consider the example in Figure 3.11. Suppose that $size_{e_1}+size_{e_2} < t \leq UDP_{p_j}$, and the overlapping region of p_j and e_i contains fewer number of points to make p_j dominant. In this case, e_i is also worth exploring since p_j

can be confirmed non-dominant and therefore e_2 can be eliminated from E. The portion of weight that e_i receives from p_j in this case, can be defined as $\sigma_{p_j e_i} = \frac{size_{e_i} - N_{p_j e_i}}{UDP_{p_j} - t}$. As before, a good portion $\sigma_{p_j e_i}$ is expected to exceed 1.

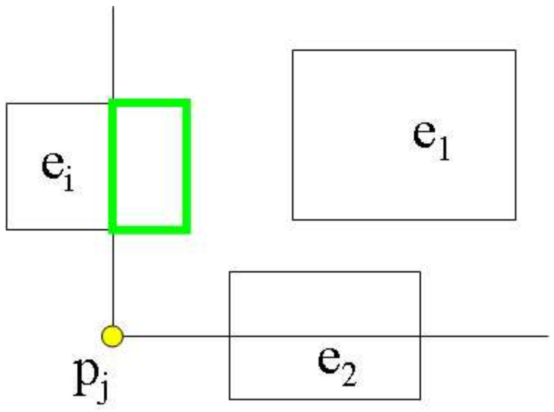
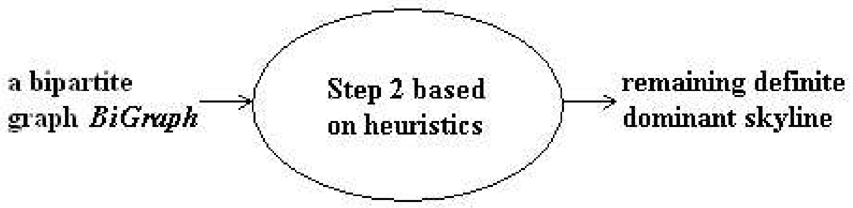


Figure 3.11: Heuristic Function 4: exploring e_i may make p_j non-dominant

Overall Weighing Function

Finally, we can weight an entry e_i according to the following formula: $W_{e_i} = \sum_{p_j^{\prime}s \text{ connected with } e_i} \sigma_{p_j e_i} W_{p_j e_i}$.

Figure 3.12 shows the input to and output from Step 2 based on heuristic functions.



a bipartite graph *BiGraph*
Step 2 based on heuristics
remaining definite dominant skyline

Figure 3.12: Input and output of Step 2 based on heuristic functions

Step 2 Using Scanning

Yet another alternative different from these heuristic functions based approaches is simply to use scanning in the second step. Remember after Step 1, we have confirmed some definite dominant skyline points and pruned away some definite non-dominant skyline points. So in Step 2, we can also scan the data set once more to confirm the rest of the dominant skyline points. Figure 3.13 shows the input to and output from Step 2 based on scanning.

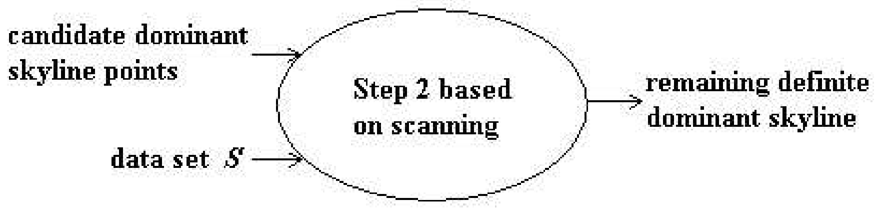

candidate dominant skyline points
data set S
Step 2 based on scanning
remaining definite dominant skyline

Figure 3.13: Input and output of Step 2 based on scanning

3.2.3 Discussions

In Step 1, when an entry is found dominated, it is compressed and then the real entry itself is deleted from memory. If the entry overlaps with some candidate dominant skyline point, we may need to bring it back to memory again in Step 2. To save these extra I/Os, one may think of caching all dominated entries in memory, which demands a large amount of main memory. One idea is to selectively caching some of them. However, it is hard to predict which entries will be needed in Step 2 because some of them may actually be removed by exploring other higher weighed entries. Note that, a lower weighed entry (and therefore more likely to be removed from consideration) in Step 2 overlaps with fewer number of points. Hence, we adopted a page eviction policy that kicks out entries that overlap with the least number of points first.

In Step 2, as explained earlier, there are four ways to define the portion of weights that an entry e_i receives from a point p_j. It is not clear whether one definition is more suitable than the rest for certain kinds of data distribution. We explored this in Section 3.3 through experiments.

3.3 Dominant Skyline Experiments

In this section, we present the experimental results of various algorithms for the dominant skyline computation. We used the generator in [3] to generate the input data files. Here are some common characteristics of the input data. Each tuple has d dimensions and one "bulk" attribute that is packed with garbage characters to ensure that the tuple is of 100 bytes long. Following the common methodology to study the performance of skyline query evaluation, three types of data sets are generated: (1) **Independent** where the attribute values of the tuples are generated using an uniform distribution; (2) **Coorelated** which contain tuples whose attribute values are good in one dimension and are also good in other dimensions; (3) **Anti-correlated** which contain tuples whose attribute values are good in one dimension but are bad in one or all of the other dimensions. The experiments were performed on a desktop PC running Fedora Core 4, with a Pentium IV 2.6 GHz CPU and 1

Parameter	Abbreviation
Number of Dimensions	d
Threshold Value	t

Table 3.3: Parameters of dominant skyline experiments and their abbreviations

GB memory.

"BNL" refers to Block-Nested-Loop algorithm followed by scanning. "Naive" refers to the naive two-step approach with BBS in Step 1 and scanning in Step 2. "RTree+Func1" refers to the improved two-step approach with heuristic function 1. "RTree+Func2" refers to the improved two-step approach with heuristic function 2. "RTree+Func3" refers to the improved two-step approach with heuristic function 3. "RTree+Func4" refers to the improved two-step approach with heuristic function 4. "RTree+Scan" refers to the improved two-step approach with scanning.

All the input data files, except the one used in Figure 3.17, contain 100,000 tuples. We investigated the performace impacts of dimensionality and threshold value. We also examined the progressiveness of various algorithms. We reserved 100 MB of the memory space for all the experiments in this section. We use the abbreviations in Table 3.3 for the parameters that we vary for different sets of experiments.

Dimension	Independent	Anti-correlated	Correlated
2	12/13	49/50	3/3
3	66/72	438/713	6/6
4	377/405	85/4069	14/14
5	876/1127	2/13102	23/25
6	1281/2361	0/26455	19/20
7	1473/5189	0/41965	92/99
8	1006/10020	0/56121	103/119

Table 3.4: Result summary of dominant skyline experiment with varying dimensionality

3.3.1 Impact of Dimensionality

In this set of experiments, $t = 5,000$. Figure 3.14, Figure 3.15, and Figure 3.16 show the results of varying dimensions for independent, anti-correlated, and correlated data respectively. Table 3.5 summarized the result size of this set of experiments. For example, when $d = 4$, independent data set has 12 dominant skyline points out of 13 skyline points.

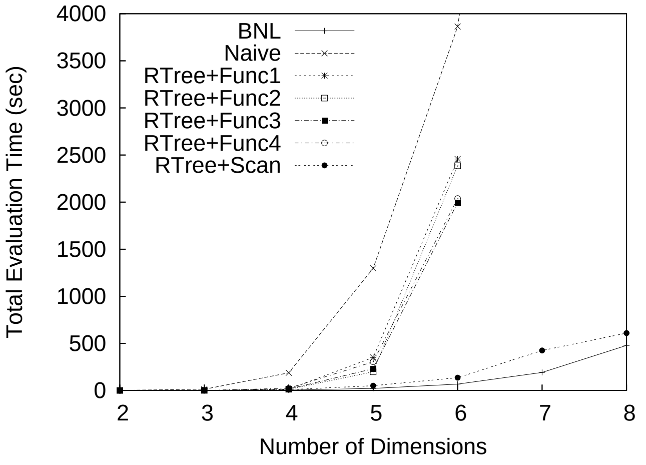


Figure 3.14: Total evaluation time vs. dimensionality for independent data

In Figure 3.14, we observed the following.

1. When $d > 5$, RTree+Func approaches started to lose to BNL because R-tree becomes inefficient as dimensionality increases.

2. RTree+Func approaches failed to finish evaluations when $d \geq 7$ because after Step 1, they all produced large bipartite graphs that could not fit in the pre-allocated memory space. These approaches are not suitable for processing of high dimensional data.

3. BNL won slightly over RTree+Scan when dimensionality was high. This is mainly due to the inefficiency of R-tree in high dimensional space.

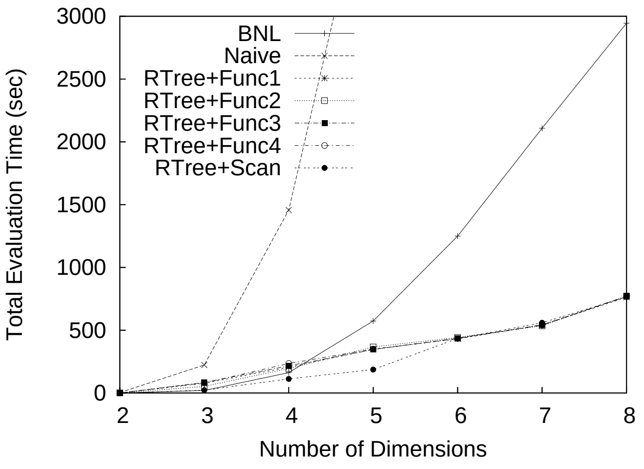


Figure 3.15: Total evaluation time vs. dimensionality for anti-correlated data

In Figure 3.15, we observed the following.

1. BNL is more sensitive to dimensionality increasing. It started to lose to R-tree based improved approaches when $d \geq 5$.

2. RTree+Scan won over RTree+Func approaches when $d < 6$. When $d \geq 6$, they had same evaluation time because there was no skyline

points left for Step 2 computations (i.e., all skyline points have been confirmed, either dominant or non-dominant, after Step 1).

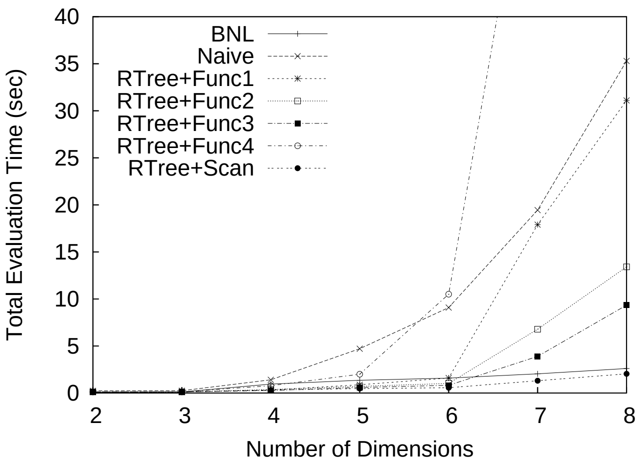


Figure 3.16: Total evaluation time vs. dimensionality for correlated data

In Figure 3.16, we observed the following.

1. RTree+Func4 performed worst when $d \geq 6$ because the heuristic function is more complicated and the function actually led to more entry explorations in Step 2 (twice as much as the other functions).

2. All 4 heuristic function lost to BNL, because for correlated data, skyline points usually have high dominating powers. BNL needs to scan only a small portion of it to confirm all dominant skyline points.

3. BNL and RTree+Scan had similar evaluation time.

Figure 3.17 repeated the same experiment as in Figure 3.14 with data set of 500,000 tuples. We observed similar trends except that 1)RTree+Func

Dimension	Independent
2	13/13
3	112/113
4	451/461
5	1629/1731
6	4027/4664
7	8374/11549
8	12203/23248

Table 3.5: Result summary of dominant skyline experiment with varying dimensionality and input size of 500k tuples

approaches ran out of memory earlier when $d = 5$; and 2)RTree+Scan performed far worse than BNL due to the inefficiency of R-Tree in high dimensional space. Table **??** summarized the result size of this set of experiments.

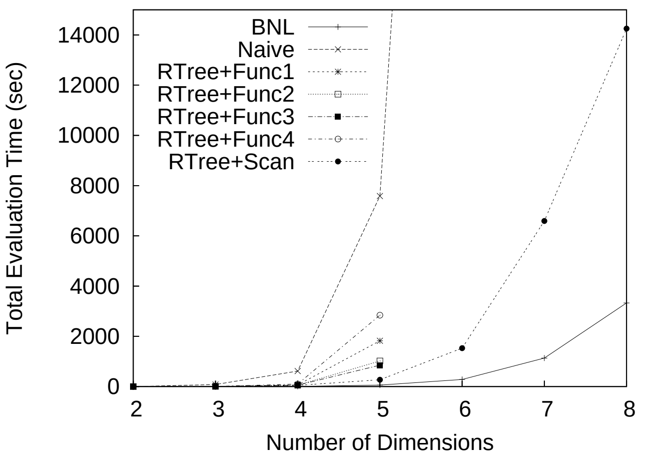


Figure 3.17: Total evaluation time vs. dimension for independent data of cardinality 500 K

3.3.2 Impact of Threshold

In this set of experiments, $d = 3$. Figure 3.18, Figure 3.19, and Figure 3.20 show the results of varying thresholds for independent, anti-correlated, and

Threshold	Independent	Anti-correlated	Correlated
0	72/72	713/713	6/6
20000	64/72	0/713	6/6
40000	53/72	0/713	6/6
60000	40/72	0/713	6/6
80000	24/72	0/713	6/6
100000	0/72	0/713	0/6

Table 3.6: Result summary of dominant skyline experiment with varying threshold

correlated data respectively. Table 3.6 summarized the result size of this set of experiments. For example, when $t = 0$, independent data set has 72 dominant skyline points out of 72 skyline points.

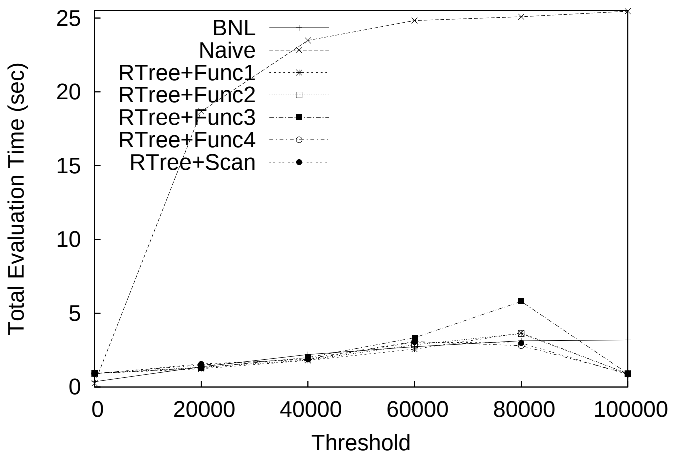


Figure 3.18: Total evaluation time vs. threshold for independent data

In Figure 3.18, we observed the following.

1. The evaluation time of BNL and Naive approaches both increased with threshold. When $t \geq 60$ K, the two approaches need to scan almost the entire data set to confirm dominant skyline points.

2. When $t = 80$ K, all the RTree+Func approaches have similar evaluation

time as BNL.

3. Function 3 turned out to be a slightly worse heuristic function than the rest when $t = 80$ K. This function introduced more page explorations than the rest.

4. For RTree-based improved approaches, when $t = 10$ K, Step 1 confirmed that all skyline points were non-dominant and Step 2 was not executed.

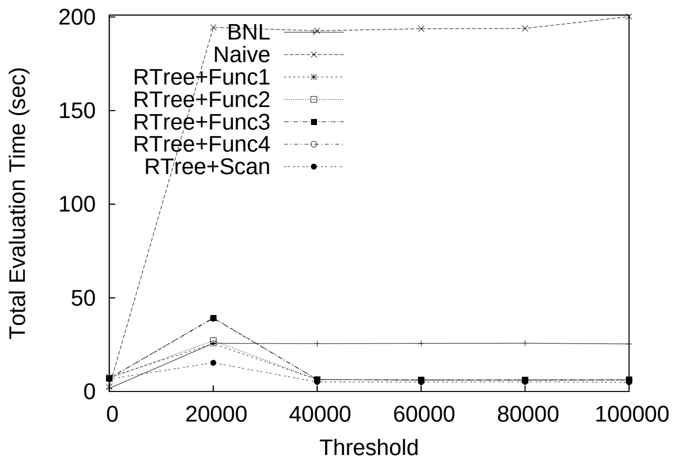


Figure 3.19: Total evaluation time vs. threshold for anti-correlated data

In Figure 3.19, we observed the following.

1. The evaluation time for Naive and BNL shot up sharply at $t = 20$ K. In all the algorithms, we try to report a dominant skyline point as soon as the lower bound of its power is above the specified threshold. For this anti-correlated data set, no skyline point has dominating power above 20 K, which means that both Naive and BNL need a complete scan of the data set to confirm the dominant skyline points.

2. When $t \neq 0$, for the rest approaches, almost all the skyline points can be confirmed non-dominant as soon as Step 1 finishes. The only exception happened with $t = 20$ K, where 40% of the skyline points were still needed to be confirmed after Step 1.

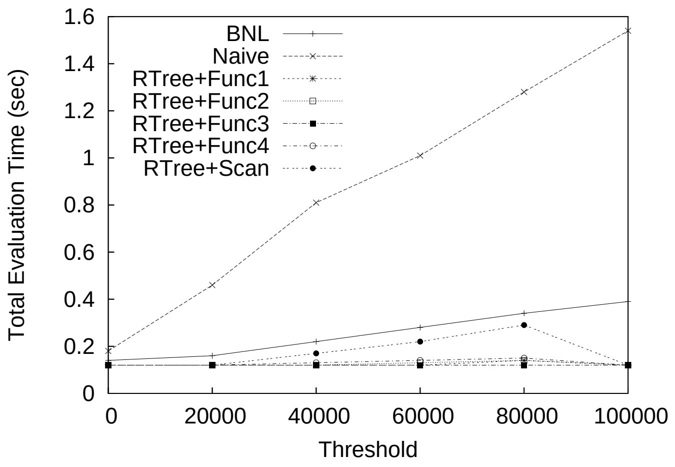


Figure 3.20: Total evaluation time vs. threshold for correlated data

In Figure 3.20, we observed the following.

1. Naive and BNL had increasing evaluation time as threshold increased. Remember that we report a dominant skyline as soon as the lower bound of its power is above the threshold. When threshold increases, dominant skyline points cannot be produced fast since we need to scan more data.

2. RTree+Func approaches all finished evaluation fast regardless of the threshold. It is because the number of skyline points is small (6 in total), so the number of R-tree entries needed to be explored in Step 2 is small too.

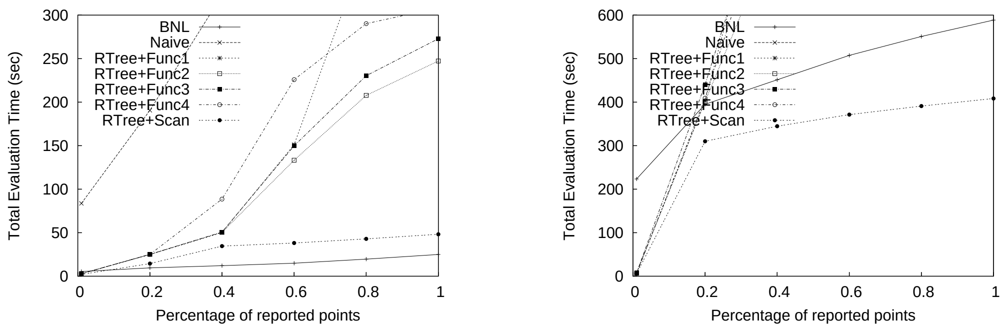
3. When $t \leq 80$ K, RTree+Scan had increasing evaluation time due to the same reason as explained in Point 1. The evaluation time dropped at $t = 100$ K because all skyline points were confirmed non-dominant (with respect to a threshold value of 100 K) immediately after Step 1. Scanning in Step 2 was skipped.

3.3.3 Progressive Behaviors

In this set of experiments, $d = 5$, and we use input data of 100,000 tuples. Figure 3.21 shows the progressiveness feature of various algorithms for independent and anti-correlated data. For independent data, $t = 6,000$. There are 842 dominant skylines out of 1127 skyline points. For anti-correlated data, $t = 1,500$ due to the skewed distribution of data points. There are 374 dominant skylines out of 13102 skyline points. We omitted the graph for correlated data because all algorithms run very fast to compute dominant skylines when the data distribution is correlated. All R-tree-based improved approaches are able to start confirming results earlier than BNL, thanks to the pruning technique used in Step 1.

3.3.4 Summary of Dominant Skyline Experiments

From the above experiment results, we see that Block Nested Loop approach performs best when the input data is independent. RTree+Scan approach works best when the input data is correlated or anti-correlated. There is no



(a) independent data

(b) anti-correlated data

Figure 3.21: Evaluation time vs. percentage of output for independent and anti-correlated data

consistent winner among the four heuristic functions. Recall that Heuristic functions 1 and 2 assume skewed distribution of data points within an MBR. However, we do not see better performance using these two heuristics for correlated or anti-correlated data sets. This is because data distribution in the Euclidean space does not necessarily say anything about the data distribution within a random R-tree MBR.

It is often the case that heuristics-based Step 2 does not yield a better performance than a simple scanning based Step 2. In terms of progressiveness, RTree-based improved approaches are able to start outputting results earlier.

Chapter 4

Tier-based Skyline Queries

This chapter provides detailed discussion on the second variant, i.e. tier-based skyline queries, as defined in Section 1.3.2. Such a query retrieves "skyline" points from *tier-1* to *tier-k*. *Tier-1* points are the traditional skyline points. *Tier-k* points are skyline points when *tier-1* to *tier-(k-1)* points are removed from the input. Tier-based skyline queries are useful when the traditional skyline result set is too small. Before we go into the details of this variant of skyline queries, let us see a *generalized dominant skyline problem* of Chapter 3. It also deals with the case where the skyline result size is too small.

When the result size of a skyline query is too small, we may want to retrieve all tuples (not necessarily skyline) that are dominated by at most t_1 tuples, but dominate at least t_2 tuples. It is a generalized definition of the *dominant skyline queries.* When $t_1 = 0$, it is indeed the definition of dominant skyline queries.

However, this type of queries can be easily answered with a simple modification of DomBBS (Figure 3.2) algorithm. Recall that in DomBBS, we keep all the already-found skyline tuples in a set S in memory. When the top entry e of the heap is removed, e is checked against all the tuples in S. If e is dominated by any tuple in S, it will not be explored in Step 1 any more. Otherwise, the child entries of e would be added to the heap for future processing. Now, to answer generalized dominant skyline queries, we keep a tuple (not necessarily skyline) in S *as long as it is dominated by no more than t_1 tuples already found in S*. When the top entry e is removed from the heap, it is checked against the tuples in S. If e is dominated by no more than t_1 tuples, e's child entries will be added to the heap for later processing.

Since the above solution is trivial, we focus our discussion on the tier-based skyline problem below.

4.1 Modifications of BBS

An obvious and naive approach to solve this tier-based queries is to compute skyline points tier by tier, starting from tier 1 all the way up to tier k. Most of the algorithms in Chapter 2 can be used to compute a tier of skyline tuples, only after the previous tiers are removed from input. However, BBS can be extended to solve this variant without using this naive approach.

Figure 4.1 shows the modified BBS algorithm to solve this tier-based

```
Algorithm TierBBS($T$, $k$)
Input:     $T$ is an R-tree
           $k$ is the maximal tier to be retrieved
Output:    a set $S$ of skyline points in tier 1 to tier k
1)   initialize heap $H$, set $S$ to be empty;
2)   insert the root entry of $T$ into heap $H$;
3)   while ($H$ is not empty) do
4)     remove top entry $e$ from $H$;
5)     $D = \{< p, tier(p) > | p \in S \wedge p \ dominates \ e\}$;
6)     $tier(e) = \max\{tier(p) | < p, tier(p) > \in D\} + 1$;
7)     if ($tier(e) > k$)
8)       discard $e$;
9)     else
10)      if ($e$ is an intermediate entry)
11)        for each child entry $e_i$ of $e$
12)          if $e_i$ is not dominated by any tier-$k$ point in $S$
13)            insert $e_i$ into $H$;
14)      else
15)        insert $< e, tier(e) >$ into $S$;
16)  return $S$;
```

Figure 4.1: BBS-based algorithm to answer tier queries

variant. We call the algorithm TierBBS. In line 4, when the top entry e is removed from the heap, it is checked against all the already-found *tier-i* ($1 \leq i \leq k$) skyline points in S, to determine (the lower bound of) the tier (i.e., $tier(e)$) that e belongs to (line 5 to 6). If $tier(e) \leq k$ (line 9), e will be explored further, similar to the BBS algorithm. If e is actually a point (line 14), then $tier(e)$ will be the actual tier that e belongs to. We can add the tuple $< e, tier(e) >$ into S (line 15).

If e is a point (line 14), why does $tier(e)$ computed in line 6 become the actual tier of e? Suppose that the actual tier of e, $tier'(e)$, is greater than $tier(e)$ which is computed in line 6. That is to say that there will be at least one point, say e', not yet found in S, but dominates e, with $tier(e') > tier(e)$. However, according to BBS, entries are explored in increasing order of their *mindist*. If e' dominates e, e' must be in S already, before e is explored.

4.1.1 Memory Management Issue with BBS

The modified algorithm in Figure 4.1 seems to be an easy solution to the tier-based skyline queries. However, one issue associated with it is the possibility of a large in-memory result set S. All the already found skyline points ranging from tier 1 to tier k are kept in S, which future candidate points will be compared against. When k, or the data set, or dimensions involved in the query is large, S can be very large. Furthermore, in the original BBS algorithm, S is managed using an in-memory R-tree, an index structure that requires a much larger pool of memory pages to maintain. Therefore, a practical solution must handle the memory overflow issue properly. That means when the memory limit is reached, we need a page replacement policy to decide which page to be removed from main memory first.

4.1.2 A Page Replacement Policy

With a large group of points scattered across different tiers, when a new point arrives, we may have run out of space to accommodate the new point. In this case, paging out some points is inevitable. A sensible guideline to decide which points to be paged out is to remove points less capable of confirming future points and their tiers. Because when a new point comes, if we have in-memory points that can confirm this new point's tier, we can immediately confirm whether it is part of the final results or not. Hence, points capable of confirming future points' tiers are more important and should stay in memory as long as possible. Intuitively, points in tier k should have the highest importance. This is because if an entry, removed from the heap, is dominated by any already-found point in tier k, then any points enclosed by this entry will be in tier $(k + 1)$ at least, so there is no need to explore this entry further. If the entry is not dominated by any already-found point in tier k, then some points enclosed by this entry may be in the result set, and we need to explore the entry further. Points in lower tiers are less critical in this sense. This is because whether or not an entry is dominated by such a point, this entry may still include points in the result and hence needs further exploration. However, *tier-(k-1)* points are intuitively more important than *tier-(k-2)* points because they directly confirm *tier-k* points. From these observations, a possible page eviction policy would be to page out points that belong to the lowest tier first.

4.1.3 TierBBS with In-memory R-tree

Recall that in the standard BBS algorithm, partial skyline results are kept in memory using an in-memory R-tree. The objective is to get a fast response when we need to know whether an entry or a point (off the heap) is dominated by any already-found skyline points. However, the page replacement policy based on tiers is hard to be executed efficiently with such an in-memory R-tree. This is because points are grouped into one R-tree leaf page based on the their dimensional values, not on which tier they belong to. To page out points in tier i may require a complete scan of the leaf pages to find out all such points, which could be inefficient. Therefore, for in-memory R-tree based memory management, we page out a random R-tree leaf when memory is full.

4.1.4 TierBBS with In-memory Linked-lists

A possible modification is to abandon the in-memory R-tree approach, and use a series of linked-lists to organize the in-memory points. Points belonging to the same tier are put into the same linked-list. When a page needs to be kicked out to disk, we always pick the list containing points in the lowest tier. Figure 4.2 depicted this data structure.

Compared to the original R-tree based memory management, list based memory management has its advantages and disadvantages. It is better not only because it is easier to find points in a particular tier, but also because it requires less memory pages to maintain the same number of points. Hence, it is a light-weighted index structure compared to R-tree. However, with points organized in lists, we may need to scan all the lists to decide the lower bound of tier that an entry belongs to, unlike the way we do it with an in-memory R-tree, where a containment query is all we need to find out all the points that dominate the entry. List structure may therefore be slower with this

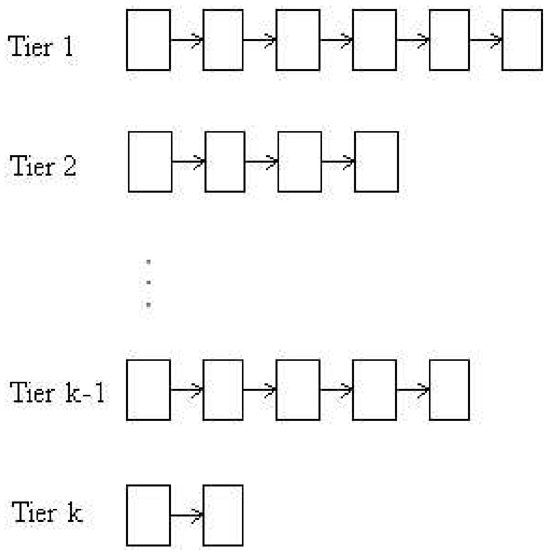


Figure 4.2: In-memory linked-lists to store the partial results

operation.

4.1.5 TierBBS with Sorted In-memory Linked-lists

We may also borrow the idea from LESS algorithm to maintain the lists sorted according to $\prod_i d_i$ where d_i are the values of skyline dimensions. In this way, a strong point (i.e., may be able to dominate more points) will "float" to the top of the list. It may accelerate the comparison. But maintaining such sorted lists also incurs cost. In Section 4.4, we present experimental results that explore this tradeoff.

4.2 Determining Tier Ranges for Points

When the main memory is large enough, no page eviction will occur, and the exact tier of each result point can be confirmed immediately, as in lines 6 and 15 of Figure 4.1. However, when the result size does not fit in memory, paging of result points occurs. Later points may hence have a range instead of an exact number as their tiers. It is easiest to explain this using an example. Let us say that we have some *tier-1* points paged out, when a new point comes, if it is not dominated by any in-memory points, we cannot say it is a *tier-1* point because it may or may not be dominated by the points paged out already. In this case, we assign a temporary tier range [1, 2] to this new point. Some later points dominated by this point may then have range [2, 3]. Therefore, after TierBBS finishes execution, we may have a set of points having tier ranges rather than exact tier numbers. This is a direct consequence of page evictions.

The range for one point, say p, is determined by the following two steps.

Step 1 Compare p against all the in-memory points. Let D be the set containing all the in-memory points that dominate p, i.e., $D = \{p' | p' \textit{ is in memory and } p' \textit{ dominates } p\}$. Let LB_p denote the lower bound of p's tier range and UB_p denote the upper bound of p's tier range[1]. Then $LB_p = \max\{LB_{p'} | p' \in D\} + 1$ and $UB_p = \max\{UB_{p'} | p' \in D\} + 1$.

Step 2 Let *Pruned* be the set of points already paged out of memory when

[1]If the exact tier of an in-memory point p' is known, then $LB_{p'} = UB_{p'}$

p's tier range is being determined. Then $UB_p = \max(UB_p{}^2, \max\{UB_{p'} | p' \in Pruned\} + 1)$.

4.3 Determing Exact Tiers for Points

After TierBBS finishes execution, the remaining task is to confirm the exact tiers of those points who have been assigned temporary tier ranges due to limited memory. Of course, points whose tier lower bounds are beyond k need no further processing.

We adopt the BNL algorithm to confirm the remaining points P. But first of all, we sort all the remaining points into ascending order according to their $\sum_i x_i$ where x_i's are the skyline dimensions. This is to ensure that a point will not be dominated by points appearing after it. The next step is to confirm *tier-i* points from $i = 1$ to $i = k$. For tier i, we start with the top point p from P, and compare it with the points already in tier i. There are three possible cases.

Case 1 If the tier lower bound of p is greater than i, we skip p for tier i and proceed with the point after p;

Case 2 If p is dominated by any point in tier i, we also skip p for tier i and proceed with the next point;

Case 3 If p is not dominated by any point in tier i, we remove p from P

[2]this UB_p is obtained in Step 1

and add it to tier i.

We are guaranteed in Case 3 that if p is not dominated by any point in tier i, p belongs to tier i. This is because of the order in which the points in P are preserved. Hence, the sorting stage of P is essential to ensure the correctness of results.

4.4 Tier-based Skyline Experiments

In this section, we present the experimental results of three variants of algorithm TierBBS as compared to algorithm BNL. "BBS-List" refers to the algorithm TierBBS using in-memory linked-lists. "BBS-List+" refers to the algorithm TierBBS using sorted in-memory linked-lists. "BBS-RTree" refers to the algorithm TierBBS using in-memory R-tree.

All the input data files contain 1,000,000 tuples. Every tuple in the input data is of 100 bytes long and two types of data sets, i.e. independent and correlated are generated. We investigated the performace impacts of dimensionality, maximum tier level, and memory size. The experiments were performed on a desktop PC with Fedora Core 4, a Pentium IV 2.6 GHz CPU and 1 GB memory. We use the abbreviations shown in Table 4.1 for the parameters that we vary for different sets of experiments.

Parameter	Abbreviation
Number of Dimensions	d
Maximal Tier Level	k
Main Memory Size	m

Table 4.1: Parameters of tier-based skyline experiments and their abbreviations

4.4.1 Impact of Dimensionality

In this set of experiments, $m = 1$ MB and $k = 4$.

Figure 4.3 shows the results of varying dimensions for independent data. When $d < 5$, BBS-List and BBS-List+ both have better performance than BNL. Actually when $d = 3$, BBS-List and BBS-List+ both run eight times faster than BNL; when $d = 4$, BBS-List and BBS-List+ run twice faster than BNL. However, when $d = 5$, both BBS based algorithms using in-memory linked-lists perform significantly worse than BNL. This is because 1 MB of memory size becomes too small for five-dimensional data, and excessive paging occurs. When $d = 2$, BBS-RTree performs similarly as the other two BBS based variants, and is better than BNL. However, it slows down drastically when $d \geq 4$. It is clear that for independent data, BBS based algorithms are more sensitive to increase of dimension. Table 4.2 summarized the result size of this set of experiments. For example, when $d = 2$, independent data set has 15 $tier - 1$ skyline points, 18 $tier - 2$ skyline points, 35 $tier - 3$ skyline points, and 41 $tier - 4$ skyline points.

Figure 4.4 shows the results of varying dimensions for correlated data. BBS-List and BBS-List+ have only a very small, almost negligible, increase

Dimension	Independent	Correlated
2	15/18/35/41	2/1/3/2
3	112/243/408/536	6/9/9/17
4	533/1484/2541/3325	12/22/39/40
5	2169/6724/12978/20489	50/82/151/213

Table 4.2: Result summary of tier-based skyline experiment with varying dimensionality

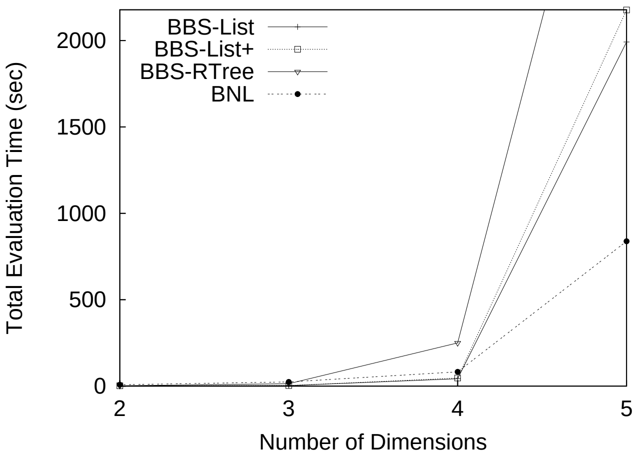


Figure 4.3: Total evaluation time vs. dimensionality for independent data

in evaluation time when dimension increases. This is because correlated data has very small number of skyline points in every tier. When dimension increases, this still holds. That results in very small increase of the number of in-memory comparisons for BBS-List and BBS-List+. BBS-RTree has noticable time increase when $d = 5$. This is the overhead cost due to the data insertions and index structure maintenance for R-tree. When $d < 5$, the in-memory R-tree has only 1 root page. When $d = 5$, the in-memory R-tree has 8 pages in total. BNL has constantly increasing response time as dimension increases.

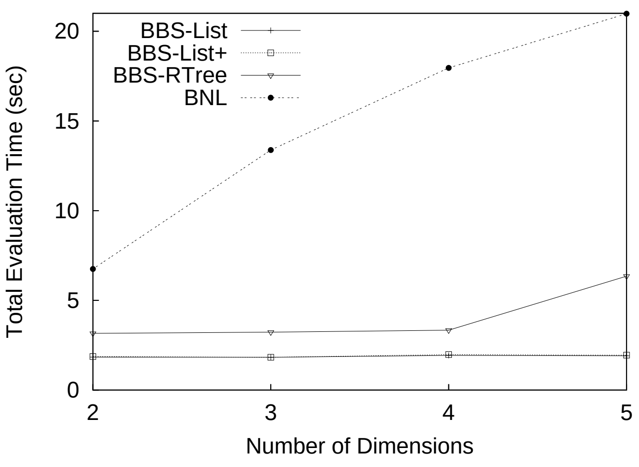


Figure 4.4: Total evaluation time vs. dimensionality for correlated data

number of tiers	Independent	Correlated
2	112/243	6/9
3	112/243/408	6/9/9
4	112/243/408/536	6/9/9/17
4	112/243/408/536/728	6/9/9/17/9

Table 4.3: Result summary of tier-based skyline experiment with varying number of tiers

4.4.2 Impact of Tier Level

In this set of experiments, $d = 3$ and $m = 1$ MB.

Figure 4.5 and Figure 4.6 show the results for independent and correlated data respectively. Table 4.3 summarized the result size of this set of experiments. For example, when $k = 2$, independent data set has 112 $tier - 1$ skyline points and 243 $tier - 2$ skyline points.

In Figure 4.5, BBS-List and BBS-List+ perform better than the rest two algorithms. They have small increases when tier number (k) increases. BBS-RTree is more sensitive to tier number increase than BNL. Initially, BBS-

RTree has shorter response time than BNL; but when $k \geq 4$, its evaluation speed starts to slow down.

In Figure 4.6, TierBBS algorithms have better performance than BNL. They show no significant increase in response time as tier number increases; because the number of in-memory results is very small for correlated data. BBS-RTree has higher index maintenance cost than BBS-List and BBS-List+.

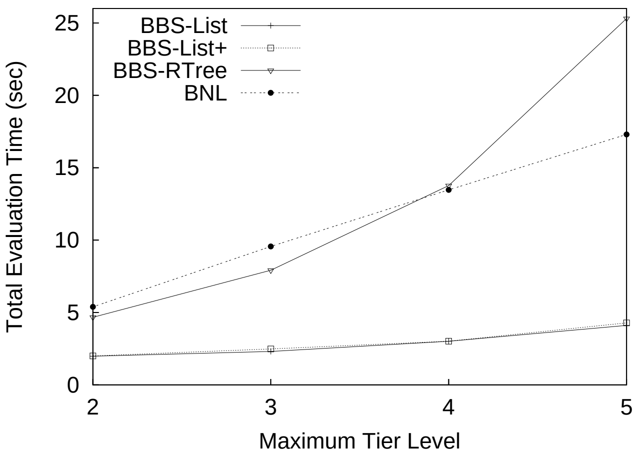


Figure 4.5: Total evaluation time vs. tier for independent data

4.4.3 Impact of Memory Size

In this set of experiments, $d = 3$ and $k = 4$. We used the same data set as before.

Figure 4.7 shows the results of varying main memory size for independent data. The subfigure on the right shows a zoomed-in view with the y-scale reduced. For BBS-List and BBS-List+, when the memory size is too small,

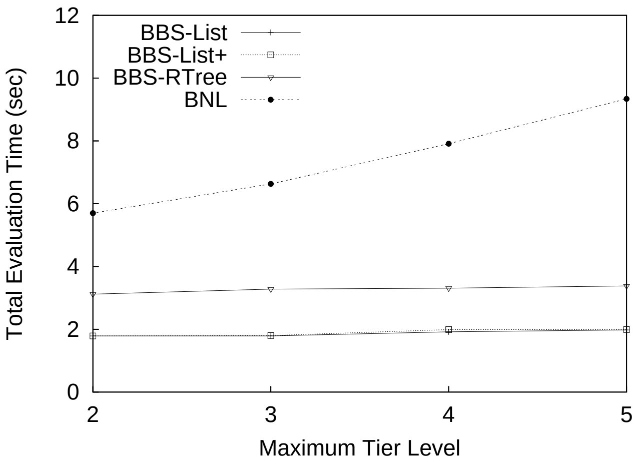
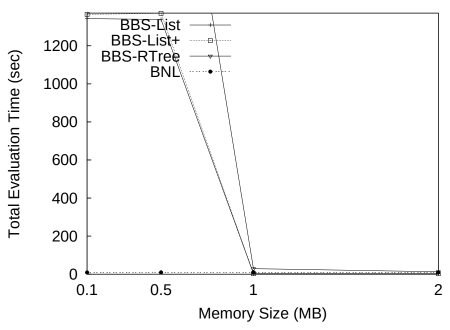


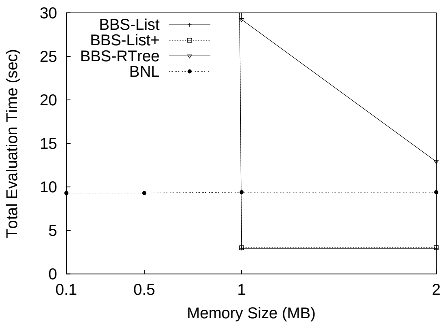
Figure 4.6: Total evaluation time vs. tier for correlated data

paging of in-memory results occurs excessively. Once the memory is big enough, paging frequency reduces significantly, and the response time drops. BBS-RTree needs a larger pool of memory pages to keep the same amount of in-memory points than BBS-List/BBS-List+; that means BBS-RTree may need more paging for the same amount of memory size. Also, R-tree has higher maintenance cost than linked-lists. These are the reasons why BBS-RTree performs worse than BBS-List and BBS-List+. On the contrary, BNL has similar evaluation speed when the memory changes from 0.1 MB to 2 MB. BNL does not rely on memory as heavily as BBS-based algorithms.

Figure 4.8 shows the results of varying main memory size for correlated data. We do not see the increase of memory size affect the response time for BBS-List and BBS-List+ in this case. Correlated data always have the smallest number of results hence it does not require as large memory as independent or anti-correlated data does. For BNL, larger memory again hurts the evaluation speed a bit.



(a) zoomed-out view



(b) zoomed-in view

Figure 4.7: Total evaluation time vs. memory size for independent data

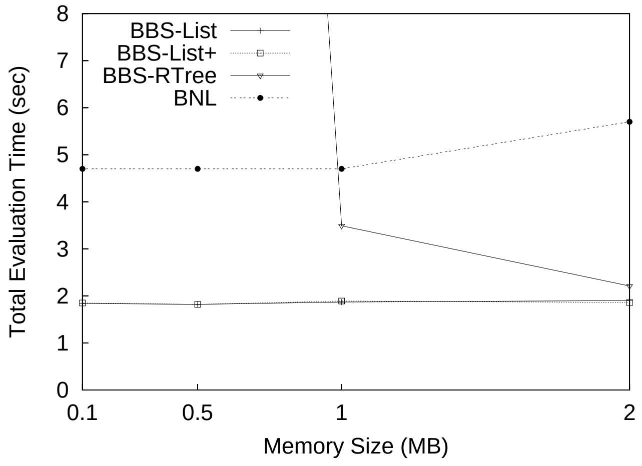


Figure 4.8: Total evaluation time vs. memory size for correlated data

4.4.4 Summary of Tier-based Skyline Experiments

From the above sets of experiments, we can draw the following conclusions. In most of the cases, algorithm TierBBS with in-memory linked lists has the best response time among all the algorithms. However, there is still a couple of cases where this does not hold. For independent data, when $d > 4$, BNL may perform better. The other case is when the memory size is too small, I/O cost may be too high for BBS-based approaches.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Skyline query is a sub-problem of the preference query. It provides a means to compute preference queries. By imposing *MIN*, *MAX*, *DIFF* conditions on a set of attributes, the query selects tuples that are indifferent to each other but dominate the rest of the tuples. It is important for several applications involving multi-criteria decision making. Recently, considerable attention is drawn to improving the efficiency of computing skyline points and proposing meaningful variants of and extensions to the conventional skyline queries.

In this thesis, we surveyed several important algorithms (i.e., Block Nested Loop, LESS, Divide and Conquer, Bitmap, Index, Nearest Neighbor, Branch and Bound) on the computation of conventional skyline points. We analyzed the strengths and limitations of each algorithm. We also reviewed three skyline variants, namely the thick skyline, the stable skyline, and the streaming skyline. It is worth studying the ways that researchers re-define the original problem to make it more interesting.

We also explored two variants of the conventional skyline problem. The first variant, called the *dominant skyline*, weighs a skyline tuple more superior if it dominates more other (non-skyline) tuples ([17]). Given a data set S, a skyline query Q, and a dominating power threshold t, a dominant skyline query asks for skyline tuples that dominate at least t other tuples. It is a useful way to summarize the skyline results when the size of the result set is large. Records with high dominating power are usually more interesting than those with relatively low dominating power. It turns out that this dominant skyline problem cannot be easily solved using existing algorithms in an efficient way, which makes the problem worth further exploration. We proposed several two-step algorithms based on R-tree.

The second variant, called the *tier-based skyline*, tries to retrieve "skyline" tuples from tier 1 up to tier k where k is a parameter from the query. Conventional skyline tuples are *tier-1* tuples. *Tier-k* tuples are skyline tuples when the *tier-1* to *tier-k* tuples are removed from the input. When the *tier-1* result is too small, users will probably be interested in tuples that belong to higher tiers. This variant is used to retrieve more interesting tuples which may not be in conventional skyline result. We proposed several algorithms based on BBS, with differences in the in-memory housekeeping.

We also conducted extensive experiments to study the performances of various algorithms. Through the experiments, we identified some interesting results and tradeoffs among the algorithms. These again shed light for possible future improvements and extensions.

5.2 Future Work

We proposed several two-step approaches for dominant skyline query processing. From the experimental results, we see that Step 1, which is based on some pruning techniques, showed definite better performance for earlier confirmation of partial results. Step 2 based on heuristics did not show good performance consistently. Possible reasons are: an in-memory bipartite graph is not the best way to organize the candidate dominant skyline points and their overlapping R-tree entries; heuristic-based approaches do not find the optimal exploration sequence of R-tree entries. In the future, we may explore more alternatives for the organization of candidate results and maybe other heuristics.

For tier-based skyline query processing, we see that TierBBS with in-memory linked-lists have the best performance in most of the cases. However, this approach needs enough main memory to ensure its fast running time. One possible alternative is to combine Block Nested Loop algorithm, which needs a smaller memory, with this algorithm, so that we can selectively run the better algorithm depending on the available memory size. And yet, we may explore other possibilities to reduce its reliance on memory.

Recall that the motivation of proposing the dominant skyline problem is to control the size of the result set. The DomBBS algorithm takes dominating power threshold as an input parameter to indirectly constrain the result size. Another direction to approach the problem is to construct an algorithm that takes the desired result size K as an input parameter and compute the top-K skyline points in terms of dominating power. However, this appears to be a harder problem. Unlike a traditional top-K ([10, 4, 7, 16]) query where an preference function exists, top-K dominant skyline query has no obvious function exists for us to optimize. If we were to think along the line of the DomBBS algorithm, we could use an arbitrary threshold to first determine the skyline points whose upper bounds of dominating powers are below the threshold. Excluding these points, if the number of the remaining skyline points is greater than K, we know the top-K dominant skyline points are among the remaining points. Otherwise, we need to use a smaller threshold. Either way, more iterations of the algorithm are needed to proceed and hence it could be too time-consuming. We need to think of a new algorithm for this top-K dominant skyline problem. This can be an interesting direction for future exploration. For tier-based skyline query processing, similarly, we can specify the result size K prior to query processing. The TierBBS algorithm computes skyline points tier by tier, so most probably when we finish processing a certain tier, the result size is already greater than K. We can then apply other criteria to filter the last tier of points.

Bibliography

[1] Rakesh Agrawal and Edward L. Wimmers. A framework for expressing and combining preferences. *SIGMOD Rec.*, 29(2):297–306, 2000.

[2] Christian Bhm and Hans-Peter Kriegel. Determining the convex hull in large multidimensional databases. In *DaWaK '00: Proceedings of the 2nd International Conference on Data Warehousing and Knowledge Discovery*, 2000.

[3] Stephan Brzsnyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In *ICDE '01: Proceedings of the 17th International Conference on Data Engineering*, pages 421–430, Washington, DC, USA, 2001. IEEE Computer Society.

[4] Yuan-Chi Chang, Lawrence Bergman, Vittorio Castelli, Chung-Sheng Li, Ming-Ling Lo, and John R. Smith. The onion technique: indexing for linear optimization queries. *SIGMOD Rec.*, 29(2):391–402, 2000.

[5] Jan Chomicki. Querying with intrinsic preferences. In *EDBT '02: Proceedings of the 8th International Conference on Extending Database Technology*, pages 34–51, London, UK, 2002. Springer-Verlag.

[6] Jan Chomicki. Preference formulas in relational queries. *ACM Trans. Database Syst.*, 28(4):427–466, 2003.

[7] Ronald Fagin. Fuzzy queries in multimedia database systems. In *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–10, Seattle, Washington, 1998.

[8] Parke Godfrey and Wei Ning. Relational preference queries via stable skyline. Technical report, York University, Canada, 2004.

[9] Parke Godfrey, Ryan Shipley, and Jarek Gryz. Maximal vector computation in large data sets. In *VLDB '05: Proceedings of the 31st International Conference on Very Large Data Bases*, pages 229–240. VLDB Endowment, 2005.

[10] Vagelis Hristidis, Nick Koudas, and Yannis Papakonstantinou. Prefer: a system for the efficient execution of multi-parametric ranked queries. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 259–270, New York, NY, USA, 2001. ACM Press.

[11] Wen Jin, Jiawei Han, and Martin Ester. Mining thick skylines over large databases. In *PKDD '04: Proceedings of the 8th European Conference on Principles and Practice of Knowledge Discovery in Databases*, pages 255–266, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[12] Werner Kieβling. Foundations of preferences in database systems. In *VLDB '02: Proceedings of the 28th International Conference on Very Large Data Bases*, Hong Kong, China, 2002.

[13] Donald Kossmann, Frank Ramsak, and Steffen Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB '02: Proceedings of the 28th International Conference on Very Large Data Bases*, Hong Kong, China, 2002.

[14] Iosif Lazaridis and Sharad Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 401–412, New York, NY, USA, 2001. ACM Press.

[15] Xuemin Lin, Yidong Yuan, Wei Wang, and Hongjun Lu. Stabbing the sky: Efficient skyline computation over sliding windows. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 502–513, Washington, DC, USA, 2005. IEEE Computer Society.

[16] Apostol Natsev, Yuan-Chi Chang, John R. Smith, Chung-Sheng Li, and Jeffrey Scott Vitter. Supporting incremental join queries on ranked inputs. *The VLDB Journal*, pages 281–290, 2001.

[17] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 467–478, New York, NY, USA, 2003. ACM Press.

[18] Christos H. Papadimitriou and Mihalis Yannakakis. Multiobjective query optimization. In *PODS '01: Proceedings of the twentieth ACM*

SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pages 52–59, New York, NY, USA, 2001. ACM Press.

[19] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry-An Introduction*. Springer-Verlag, New York, NY, USA, 1985.

[20] Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. Nearest neighbor queries. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, 1995.

[21] Kian-Lee Tan, Pin-Kwang Eng, and Beng Chin Ooi. Efficient progressive skyline computation. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 301–310, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[22] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: an efficient data clustering method for very large databases. *SIGMOD Rec.*, 25(2):103–114, 1996.