I N A U G U R A L - D I S S E R T A T I O N

zur

Erlangung der Doktorwürde

der

Naturwissenschaftlich-Mathematischen Gesamtfakultät

der

Ruprecht-Karls-Universität Heidelberg

vorgelegt von

Diplom-Informatiker (Univ.) Steffen Thomas Rost

aus Heilbronn

Tag der mündlichen Prüfung: 09. Oktober 2006

Skyline Query Processing

Gutachter: Prof. Dr. Donald Kossmann

Prof. Dr. Gerhard Reinelt

Acknowledgments

As with all big projects there were a lot of people who helped me in manifold ways to write this thesis. Drawing up a list and thanking them all would probably go beyond the scope of this chapter and in the end I would have forgotten someone, anyway. So to all who helped me and are not mentioned on this page, I say, thank you, thank you, thank you. Your help was much appreciated.

However there are some people I would like to thank for their outstanding help and support.

I would like to thank my supervisor Donald Kossmann who helped me to navigate through rough waters, never giving up hope on me, and always challenging me to give my best. Without him this thesis would not have been possible.

I would like to thank my colleagues, Andreas Grünhagen, Peter Fischer, Angelika Reiser, and Cristian Duda for their help. Showing up in their offices anytime, just asking a simple question and finding yourself in a fruitful discussion is the help I am most grateful for. I hope I have done the same for them.

I also would like to thank all the other authors who contributed to the Skyline. By publishing their thoughts on the Skyline it was possible for me to extend my knowledge on the Skyline.

And, last but not least, I would like to thank my parents whose love and support I can always count on. Without them neither my studies nor this thesis would have been possible. I love them from the bottom of my heart.

A five year project is now almost finished. Many things happened during these years. Looking back I see foremost the positive things: working together with colleagues, teaching students, and a great deal of personal development.

S.T.R.
June 2006

Abstract

This thesis deals with a special subset of multi-dimensional set of points, called the *Skyline*. These points are the maxima or minima of the complete set and are of special interest for the field of decision support. Coming from basic algorithms for computing the Skyline we will develop ideas and algorithms for "on-the-fly" or online computation of the Skyline. We will also extend the concept of Skyline with new application domains leading us to user profiling with the help of Skyline.

Zusammenfassung

Diese Arbeit behandelt eine spezielle Untermenge einer multi-dimensionalen Punktemenge, Skyline genannt. Diese Punkte sind die Maxima oder Minima der ganzen Menge und sind im Gebiet des Decision Support von besonderem Interesse. Wir werden grundlegende Skyline Algorithmen vorstellen und Ideen und Algorithmen, um die Skyline "on-the-fly" oder online zu berechnen. Wir werden außerdem das Skyline Konzept mit neuen Anwendungsgebieten erweitern wobei wir Benutzerpräferenzen mit Hilfe der Skyline berechnen wollen.

Contents

Part I

Introduction

CHAPTER 1

Motivation

It is impressive how the Internet, namely the World Wide Web, has become part of our daily life. Barely know 15 years ago, the Internet is now known to everyone. 15 years ago the Internet was used by a few people who used the WWW to interchange information with their kind. The information was well structured and concise in presentation. Now, 15 years later, the information on the WWW is overwhelming. The quantity of information has been greatly increased. This tremendous rise in the amount of information makes filtering vital for finding what one needs, because most information is not relevant for the individual person.
Not only has the quantity of information increased but also the pace the information arrives in, for example, RSS feeds, or online information systems such as online stock market systems. This again makes efficient information filtering necessary to find one's preferred information.
In a nutshell - filtering becomes more and more important for users to find the information they need.

These information filters can either be static, meaning they work on a static set of data, or dynamic, meaning they continuously scan an information stream and pick out data for the user.
Huge data sets that can be regarded as static are, for example, astronomical data, for example the SETI data [Set06] or the connection data of a telecommunications company, for example the connection data of all land lines of the Deutsche Telekom [Tel06] of last month. Both data sets are so big that before any analysis can take place the data has first to be reduced to an interesting subset. This reduction should not eliminate data that is interesting. So filtering, meaning reduction of data, is a tightrope walk.
Never ending dynamical data streams are, for example, data collected from sensors, click streams of WWW links, or online shops/auction updates. Here it should be possible to filter the data according to user profiles. The user chooses again facts that are interesting for him or her.

The Skyline is one way of doing filtering on static an dynamic data sets. It is not the magic bullet in reducing data. But it is certainly an interesting approach among others to handle the ever-growing volume of data.
This thesis is divided into 5 parts: Part I is the introduction to the topic. It includes the math-

ematical foundations of the Skyline, some application domains, some other explanation necessary to understand this thesis, and last but not least an overview of work related to the Skyline. Part II shows different Skyline algorithms for static data set. They all have been previously published and extensively discussed. Part III is the link between the static part and the dynamic part. It displays two Skyline algorithms for static data sets but also shows an alley for computing Skyline on dynamic data sets. All algorithms in Part III are fundamental to dynamic data sets. They also have been published previously. Part IV then deals with all aspects of changing data and shows Skyline algorithms and ideas to efficiently handle dynamic data sets and multiple queries. In Part IV we use the Skyline as a representation for user profiles. Part V finally concludes this thesis with an overall conclusion and an some closing words.

CHAPTER 2

Application Domains

This chapter gives an overview of applications we have in mind when we are talking about the use of the Skyline. Each of the following examples displays a possible application for the Skyline in different areas. Please note: Not all applications are possible for all Skyline algorithms. This chapter merely serves as an "appetizer" where we are addressing different problems that can be solved by Skyline processing.

2.1 Hotel Booking

Imagine a travel agency. You are planning a vacation in Lido di Jesolo, Italy [Jes06]. There are plenty of hotels there. The hotel of your choice should be close to the beach, the *Mare Adreatico* at your toe tips, and the rate for the room should also be not too expensive, that is, as cheap as possible. Since the price and the distance to the beach are closely related to each other, the closer to the beach the higher the price, the database system at your travel agents' cannot decide which hotel suits you the best, but it can present you all *interesting* hotels. Interesting hotels are not worse than any other hotel for both conditions, that is, no hotel is cheaper and closer to the beach at the same time. This set of interesting hotels we call the *Skyline* because of its graphical representation (see next section). Now it is up to you to make the final decision which hotel to book from the Skyline.
We will not dig any further here for this example because we will explain this particular example in greater detail later. It will be used throughout Parts II and III for illustrating various Skyline algorithms.

2.2 Visualization

Have you ever been to New York City [Nyc06]? Have you ever been walking the hustling, bustling streets of Manhattan down to Battery Park? Have you ever been on a ferry to Staten Island and looked at the skyline of southern Manhattan? Have you ever wondered what buildings can be seen from the ferry? It is easy, if you think for a moment: You can see buildings that are

either tall or near to the Hudson river. Those buildings form the skyline of southern Manhattan – buildings that are either tall or near to river form the *Skyline*.

2.3 Electronic Market Places

Consider Autoscout24 [Aut06], it is a German car Internet broker. One can sell and buy cars on this site. If you want to buy a car you register with a certain profile. Your profile contains a selection of criteria your "new" car should have, for example, *cheap cars that are not too old*, or *cheap cars that are fast*, additionally you further narrow your search space, for example by defining criteria like *with air-conditioning*, *with power locks*, or *with airbags*. Table 2.1 shows an example of the current database of available cars. For illustration purposes we left out any additional properties of the cars.

Model	Price	Age	Speed
BMW 330 xd	EUR 30,000	5 years	200 km/h
Ford Focus	EUR 8,000	3 years	150 km/h
Toyota Avensis	EUR 10,000	4 years	170 km/h

Table 2.1: *Used car market: offers*

Table 2.2 shows the preferences of two users. User 1 is interested in cheap cars and cars of low age, User 2 is interested in cheap cars that have a high speed. Table 2.2 shows also the cars that are interesting for each user taking Table 2.1 as data basis. For User 1 only the Ford is interesting since both the BMW and the Toyota are worse in terms of price and age than the Ford. For User 2 all three cars are of interest. These initial interesting cars represent the Skyline of the current car database (Table 2.1). Each user has a different Skyline.

	Preferences	Interesting Offers
User 1	{ price (min), age (min) }	{ Ford }
User 2	{ price (min), speed (max) }	{ BMW, Ford, Toyota }

Table 2.2: *Used car market: user preferences*

Both users do not want to by a their car right away but they both want to stay informed when new cars are offered or cars of their interest are sold. A new offer that is registered in the database might be the following:

(VW Golf, EUR 12,000, 2 years, 180 km/h)

User 1 must be informed about this new offer since it is interesting in terms of price and age, it is younger than the Ford. However, User 2 must not be informed about this new offer because the VW is worse in terms of price and speed compared to the Toyota. This way, the Skyline model serves as an information filter in this application scenario. Now let us assume that the Ford is sold to another user. Of course, both User 1 and User 2 must be informed because the Ford is no longer available and that offer was interesting for both users. In addition, User 1 should be informed about the Toyota because the Toyota now has become an interesting offer for this user.
We will use this example as basis for Part IV.

2.4 Stock Ticks

This is a similar application scenario as we have seen in the previous section. Consider a stock broker at the Deutsche Börse in Frankfurt [Boe06]. It is impossible to keep an eye on each and every stock that is traded there. As with the previous scenario the stock broker could have a Skyline of interesting companies, for example, rated by the growth, the earning per stock or the price of the stock, and would like to be informed if there are relevant changes in the business metrics she is interested in. In this application scenario an update of, for example the stock price, would mean a deletion and arrival of new stock price information. The only difference to the previous example is the rate at which updates of the data basis occur. The rate of updates for stock prices is probably much higher than registering a new offer at the Internet broker, for example, the main stock index at Deutsche Börse, the DAX, is computed every second.

2.5 Quality Assurance

Another scenario with slightly different requirements is depicted in Figure 2.1. This figure shows a curve of acceptable temperatures and pressures (dashed line) in a manufacturing process, for example, the production of polyethylene plastic bags. The dots and asterisks represent measurements from temperature and pressure sensors. For a complete monitoring the measurements are taken continuously at the same time for temperature and pressure. Both measurements are denoted as one point in the diagram. All points (dots) below the dashed line stand for "normal" production conditions. All points (asterisks) above the dashed line stand for critical production conditions. In these situation at least an alarm must be raised or if the condition do not drop below the dashed line the whole production process must be stopped.
Raising alarms in critical situations can be implemented using continuous Skyline queries. The set of interesting points (the Skyline) is initialized with the corner points of the curve of acceptable temperatures and pressures and the Skyline query defines temperature and pressure as interesting properties. Using this model, the critical points that raise alarms are those points which are either dominated by one or more Skyline points or which are incomparable to all Skyline points. Points that dominate any Skyline point are guaranteed to be acceptable.
A manufacturing process, for example in the chemical industry, may involve many dimensions representing conditions the manufacturing process must obey to be successful. These dimensions can be represented as different sets of corner points. While the Skyline model used here is the same as in the market place scenario, the requirements are slightly different: In the manufacturing scenario, the sets of interesting points, that is, the curves of acceptable measurements are static whereas these sets are dynamic in the market place scenario.

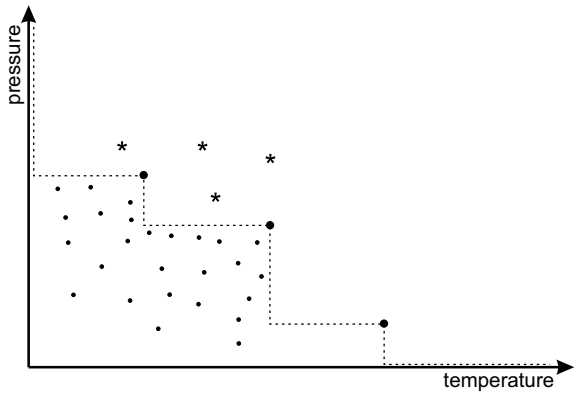


Figure 2.1: *Quality assurance: temperature / pressure curve*

CHAPTER 3

Mathematical Foundations

In the literature Skyline computation is known as the *maximum vector problem*. Sometimes it is referred to as *Pareto optimality* but in contrast to a Pareto probability distribution which can be expressed by a closed formula, the Skyline cannot be expressed by a formula. Connecting Skyline points by a line, as is is done in Figure 3.1(a), looks similar to a Pareto probability distribution curve.
We will first take a look at the definition of the problem itself and will then be describing the surrounding fields in a formal way. Note: The words *vector* and *point* are synonymous, and we will mostly use point instead of vector.

3.1 Maximum Vector Problem

The maximum vector problem was first discussed in [KLP75]. It describes the problem of finding the maximum of a set of vectors. What is the maximum of a set of vectors or points? For example

$A(5,5)$ and $B(2,2)$

If point A is greater in all dimensions than point B then A is clearly the maximum of the set of points $\{A,B\}$. But what happens if A is greater in all dimensions than B except for one dimension?

$A(5,2)$ and $B(2,3)$

Then a more sophisticated measure for comparing points has to be applied. The measure is called *dominance* and was defined in [PS85]. Since our goal in this thesis is finding the minimum of a set of points not the maximum, we formulate the definition of dominance in its original way and in parenthesis in the way we use it.

Dominance: Let D be a set of points

$$\left\{p^{(1)},...,p^{(N)}\right\} \subseteq \mathbb{R}^d,$$

where

$p^{(k)} = \left(p_1^{(k)}, ..., p_d^{(k)}\right)$ (representing each point) and d the number of dimensions.

A point p dominates another point q if

$p_i \geq q_i$ ($p_i \leq q_i$) for all $i = 1, ..., d$.

A point $p \in D$ is called a maximum (minimum) of D if there does not exist another point $q \in D$ that dominates p.

We denote dominance with the $\prec$-symbol where possible. If not possible we will write $<$ expressing our understanding of one point less than the other point.

The problem now consists of finding all the maxima (minima) of D. We look for minima because our application scenarios (see Chapter 2) mostly trying to minimize dimensional values instead of maximizing them. However, everything said in this thesis can be adapted to find the maxima of a set of points. Mostly only the comparison ($\geq$ instead $\leq$) has to be changed.

3.2 Maximum Vector Problem for "1-dimensional" Vectors

This is just a little shoot-out to demonstrate the consistency of the definition of dominance. Of course, the maximum vector problem for "1-dimensional" vectors, that is, numbers in $\mathbb{R}$, is also defined. In contrast to the maximum vector problem in $\mathbb{R}^d$ the result in a "1-dimensional" space is not a set of vectors. It is only one result. Finding the maximum or minimum vectors for $\mathbb{R}$ is finding the maximum or minimum of a set of numbers. This can be done, for example, by sorting. For sorting there are some well-known algorithms that work well in different application domains. For Sorting see [AHU74], for instance.

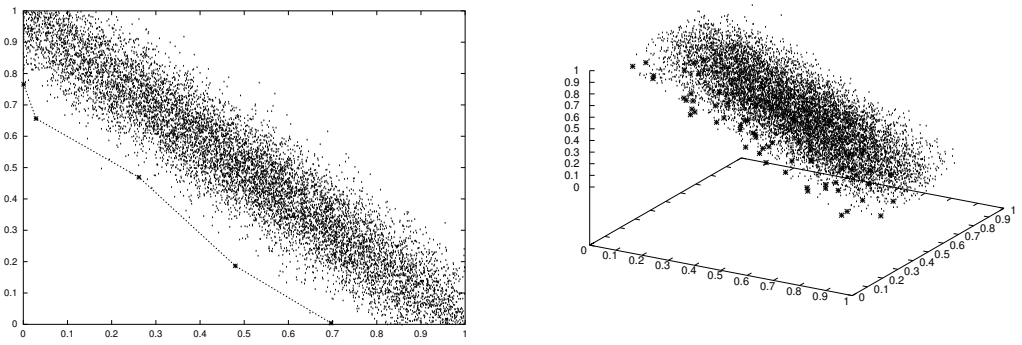
3.3 Skyline Properties

3.3.1 Number of Skyline Points

Two typical Skylines are depicted in Figure 3.1(a) and (b). Figure 3.1(a) shows a two-dimensional data set, Figure 3.1(b) a three-dimensional data set. We show these two picture just to give the reader the impression how the Skyline looks and what we are talking about. A detailed description of the data sets we used is given in Chapter 6.
The Skyline points in Figure 3.1(a) and (b) are printed in bold. In the 2-dimensional picture there are additionally connected with a dashed line. This is just for illustration purpose.
There are two important properties about the number of Skyline points ("Skyline size") we want to show here:

- The size of the Skyline increases with increasing dimensionality.
- The Skyline size increases also with increasing number of points in the data basis.

A theoretical model determining the number of Skyline points in a data set has been derived in [BKST78] and some techniques for cardinality and cost estimations related to Skyline processing haven been proposed in [CDK06]. We do not pursue this theoretical alley any further



(a) 2-dimensional (b) 3-dimensional

Figure 3.1: *Skylines*

since in this thesis we will concentrate on more pratical issues, that is. how we can compute the Skyline and how the Skyline can be used in different ways. Additionally to the above said, the Skyline size also depends of the character of the data set, that is, the value distribution of the points within the data set. So we postpone any further discussion about the number of Skyline points until Chapter 6.

3.3.2 Treatment of Duplicate Points

As a direct consequence of the definition of the Skyline in the previous section: Duplicates can occur. For example, if an algorithm, in some way, produces a superset of the Skyline, and further, this superset contains some duplicates points, then a Skyline algorithm will not find these duplicates. It will characterize all duplicates as Skyline points.

3.3.3 No Emphasis of Certain Dimensions

Another property of the Skyline of a set D is that for any monotonic scoring function

$$s : D \rightarrow \mathbb{R},$$

if $p \in D$ maximizes (minimizes) the scoring function s, then p belongs to the Skyline.
That means, no matter how you emphasize your preferences (a cheap hotel or near to the beach), your most favorite hotel will be in the Skyline.
Additionally, for every point p belonging to the Skyline, there exists a monotonic scoring function such that p maximizes (minimizes) that scoring function, That means, the Skyline does not contain any hotels which are nobody's favorite [BKS01].

3.3.4 Transitivity

Transitivity means that if one point dominates another point and this point dominates a third point then the third points is also dominated by the first point.
Consider three points, p, q, and r, point p dominates q and q dominates r then p also dominates r, that is,

$$p \prec q \quad \wedge \quad q \prec r \quad \Longrightarrow \quad p \prec r.$$

If p would not dominate r then r would be better or equal than p in at least one dimension. But since p dominates q point r would also be better than q in at least one dimension, hence q would not dominate r. This would be a contradiction to our first consideration □

CHAPTER 4

Taxonomy

4.1 Point Comparisons

Comparing two points for Skyline characteristics is slightly more complicated than a "normal" comparison. Each dimensions has to be considered and depending on the comparison of the previous dimension the complete Skyline comparison function is done or has to continue with the remaining dimensions. Figure 4.1 shows some pseudo-code for a Skyline comparison. But first we have a look at the outcomes a Skyline comparison can have. There are four states a Skyline comparison can return:

- **Equal**: Two points have the same values in each dimension, that is, for two points p and q having the same dimensionality d the following holds $\forall i, 1 \leq i \leq d, p_i = q_i$.
- **Greater**: Two points have the same value in each dimension, but one point has a greater value in at least one dimension, that is, for two points p and q having the same dimensionality d and p greater as q the following holds $\forall i, 1 \leq i \leq d, p_i = q_i \ \wedge \ p_k > q_k, 1 \leq k \leq d$.
- **Less**: Two points have the same value in each dimension, but one point has a smaller value in at least one dimension, that is, for two points p and q having the same dimensionality d and p less than q the following holds $\forall i, 1 \leq i \leq d, p_i = q_i \ \wedge \ p_k < q_k, 1 \leq k \leq d$.
- **Incomparable**: Two points have the same value in each dimension, but one point has a greater value in at least one dimension and a smaller value in at least another dimension, that is, for two points p and q having the same dimensionality d and p incomparable to q the following holds $\forall i, 1 \leq i \leq d, p_i = q_i \ \wedge \ p_k > q_k, 1 \leq k \leq d \ \wedge \ p_l < q_l, 1 \leq l \leq d$.

Figure 4.1 shows the pseudo-code for the Skyline comparison function as it will be used throughout this thesis. The comparison function gets two points p and q and returns the on of the four possible states discussed above. The following variables are used: p.dimension holds the number of dimensions of the points, of course, q.dimension would also possible; p.data[i] and q.data[i] refer to the value of the point in each dimension. The key word *continue* (line 7) skips the remaining 4 if-clauses and continues with the next dimension.

```
1  enumeration {INCOMPARABLE, LESS, EQUAL, GREATER};
2
3  function SkylineComparison(Point p, Point q)
4
5    int state = EQUAL;
6    for (int i := 1 to p.dimension) do
7      if (p.data[i] = q.data[i])) continue;
8      if (p.data[i] < q.data[i] && state == GREATER)
9        return INCOMPARABLE;
10     if (p.data[i] < q.data[i]) state = LESS;
11     if (p.data[i] > q.data[i] && state == LESS)
12       return INCOMPARABLE;
13     if (p.data[i] > q.data[i]) state = GREATER;
14   end;
15   return state;
16
17 end;
```

Figure 4.1: *Skyline comparison function*

As one can easily see, for two points to be *incomparable* it is sufficient for the first dimension to be *greater* (*less*) and the second dimension to be *less* (*greater*). In these cases it is not necessary to look at the remaining dimensions and the Skyline comparisons is done with the two points being *incomparable*. This is not true for the *less*, *greater*, or *equal* outcome. For those three states all dimensions of the points have to be considered resulting in more time spent in the Skyline comparison. Figure 4.2 shows the cumulative times for the different outcomes. We measured 10 million comparisons for 2-dimensional point and 5-dimensional points. For 2-dimensional points we found almost no difference between the four outcomes. *Equal* was the fastest, *less* and *greater* the slowest, and *incomparable* in between. The same is true for 5-dimensional comparisons. But here the differences between the fastest and the slowest comparisons is much more observable. The reason for *equal* being the fastest comparisons, even though all dimensions have to be considered, is that of all if-clauses only the first has to be executed. The remaining ones are skipped, that is what the key word *continue* does.

4.2 Domination

A point dominates another point if it is as good or better in all dimensions and better in at least one dimension. Depending on the definition of *better* one is looking for all maximum points (*better* means *greater*) or all minimum points (*better* means *less*). Throughout this thesis we say: A point dominates another point if the points comparison returns *less*. Of course, all definitions, proofs, and algorithms also work if domination is defined as the point comparison returning *greater*. If a point p dominates point q, we write $p \prec q$. If it is not possible to use the $\prec$-sign, for example in pseudo-code, we use the normal "less than" $<$. This endorses our thinking of domination that a point is "less than" another point.

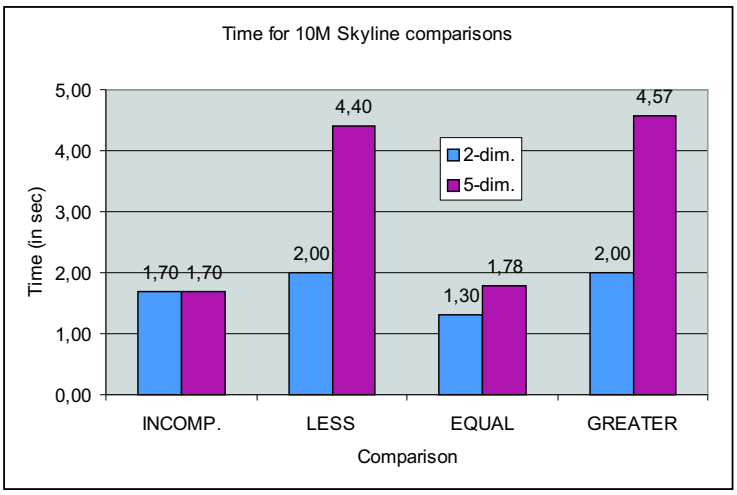


Figure 4.2: *Time for 10 million Skyline comparisons and different dimensionalities*

4.3 Skyline Queries

[BKS01] introduced an extension to SQL which we want to use as well. The SQL `SELECT` statement is extended by an optional `SKYLINE OF` clause describing the attributes (dimensions) that should be considered for Skyline computation:

`SELECT` *projection*
`FROM` *collection*
`WHERE` *predicates*
`SKYLINE OF` $\dim_1$ [`MIN`|`MAX`|`DIFF`], ..., $\dim_n$ [`MIN`|`MAX`|`DIFF`];

Projection, *collection*, and *predicates* have the same semantics as in standard SQL. If not otherwise said, there is no explicit projection and only one collection, the data set the Skyline is computed of.

```
SELECT *
FROM data_set
...
```

The `SKYLINE OF` clause describes for each dimension if this particular dimension should be minimized (`MIN`), maximized (`MAX`), or simply be different (`DIFF`). The `SKYLINE OF` clause is computed after the `SELECT`, `FROM`, `WHERE`, `GROUP BY`, and `HAVING` part of the SQL statement but before an `ORDER BY` part.
For example, consider two points with dimensionality n

$p = (p_1, ..., p_k, p_{k+1}, ..., p_l, p_{l+1}, ..., p_m, p_{m+1}, ..., p_n)$ and
$q = (q_1, ..., q_k, q_{k+1}, ..., q_l, q_{l+1}, ..., q_m, q_{m+1}, ..., q_n)$.

Point p dominates point q for a Skyline query having the following `SKYLINE OF` clause

`SKYLINE OF` $\dim_1$ `MIN`, ..., $\dim_k$ `MIN`,
$\dim_{k+1}$ `MAX`, ..., $\dim_l$ `MAX`,
$\dim_{l+1}$ `DIFF`, ..., $\dim_m$ `DIFF`

if the following conditions hold

$$\begin{array}{lll} p_i \leq q_i & \forall & i = 1,...,k \quad (\texttt{MIN}s) \\ p_i \geq q_i & \forall & i = (k+1),...,l \quad (\texttt{MAX}s) \\ p_i \neq q_i & \forall & i = (l+1),...,m \quad (\texttt{DIFF}s). \end{array}$$

Dimensions $\dim_{m+1},...,\dim_n$ are irrelevant for Skyline computation, but are, of course, part of the point and are output to the user as well as all other dimensions ($\dim_1,...,\dim_m$).

4.4 Skyline Algorithms

In this section we want to give the reader a quick overview of different ways to compute the Skyline without giving particular Skyline algorithms. This overview concentrates on three distinguishing marks: User interaction, display Skyline points before end of computation, and one pass over data to complete Skyline computation. With user interaction we mean a possibility for the user to control the sequence of returning Skyline points, displaying Skyline points before the end of computation is of particular interest for quick overviews of how the Skyline looks like, and one pass over data is important in the field of database systems. The given classification of Skyline algorithms will be used throughout this thesis. An overview is given in Table 4.1. This is, however, only a quick overview of different Skyline algorithms. A more detailed description of batch, progressive, non-online, and online algorithms is given in Chapter 8 and Chapter 16.

	User interaction possible	Display Skyline points before end of computation	One pass over data for complete computation
Online algorithms	yes	yes	yes
Progressive algorithms	partly	yes	yes
Batch algorithms	no	no	no

Table 4.1: *Classification of Skyline algorithms*

A short explanation for each class of algorithm is given in the following sections.

4.4.1 Batch Algorithms

In Part II we will take a look at, what we call, *batch algorithms*. Those algorithms take a set of points and compute the Skyline of this set of points in a batch sort of way, that is, without any user interaction and without displaying found Skyline points before the end of the complete computation. Characteristic to all of these algorithms is that on average they require more than on pass over the data set to completely compute the Skyline. More than one pass means that points of the data set might be examined twice or more times during computation.
As an add-on to the batch algorithms we present some early ideas to pre-filter the data set in order to reduce the number of points a later Skyline algorithm has to look at to determine the Skyline. One could think of the reduced data set as a "candidate Skyline" that contains definitely all points that belong to Skyline but also non-Skyline points that the filter could not eliminate. The pre-filtering should be adequate to support a later Skyline algorithm.

This pre-filter should not be confused with our information pre-filter which is described below and has a totally different application scenario. The data set reduction algorithms are described in Chapter 14.

4.4.2 Progressive Algorithms

Progressive algorithms are also discussed in Part II. They also compute the Skyline in a batch sort of way. However, they mostly require only one pass over the data set to compute the complete Skyline and they can display found Skyline points before the the whole computation is finished. More to these algorithms is said in Chapter 8.

4.4.3 Online Algorithms

These algorithms do not compute the Skyline of a set of points in batched or detached way. Their unique property of using Nearest Neighbor queries to find Skyline points makes it possible to iteratively and interactively compute the Skyline, that is, as soon as a Skyline point is found it is output to the user and the user can choose the area he or she wants to see the next Skyline points. This adjustment is made without changing basic parameters of the computation, thus preserving the results computed so far. These algorithms also require only on pass over the data set to compute the complete Skyline, no point is looked at twice. Part III, is concerned with these algorithms.

4.5 Information Pre-filter

In Part IV we take a different view of the Skyline. In this part we do not strive to compute the Skyline in elegant ways as we did the parts before. Instead we use the Skyline, that is, the points that form the Skyline as filter points for arriving and leaving points of our data set. The data set is not static anymore. However, the ultimate goal remains the same: Have the current Skyline of the data set prepared for the user. We even take this scenario one step ahead and say that there are multiple user registered with a query system, each user having his or her own *profile*, that is, a query asking for the Skyline on specified dimensions and having further predicates restricting the points that are of interest for the user. In this part, we want to efficiently compute multiple Skyline queries in the face of arriving and leaving points of the data set. The computation will be continuous, meaning that the Skyline query willbe computed over and over again.

CHAPTER 5

Related Work

We divide this section into different parts. The first part shows Skyline papers and different aspects of Skyline processing and related areas of multi-dimensional ranking. The second part is concerned with different ways of user profiling and user preferences. The third part deals with streaming and continuous query processing. The fourth part shows some aspects of multiple query optimization and finally the fifth part is a collection of various aspects more or less closely related to our Skyline story.

5.1 Skyline, Convex Hull and Top-K Processing

In the literature computing the Skyline is known as the maximum vector problem. Kung et al. [KLP75] first discussed the problem of finding the maximum or minimum of a set of vectors. They introduced an optimal recursive algorithm and showed a lower bound for the number of comparisons for this algorithm. Preparata et al. restated the idea in [PS85]. Other approaches to the maximum or minimum vector problem are, for example, Stojmenovic et al. [IS88], Matoušek [Mat91] and Rhee et al. [RDL95]. All algorithms depicted in these papers assume a complete main memory representation of the data set.

Börzsönyi et al. [BKS01] were the first (to the best of our knowledge) who discussed the idea of the maximum or minimum vector problem in the view of database systems. They introduced two algorithms for Skyline computing. One, an enhancement of the straightforward way to compute the Skyline and two, an adaption of Kung's recursive algorithm. Furthermore, they set the use cases for Skyline computing in database systems and discussed various alternative ideas to compute Skyline using means database systems provide.

Chomicki et al. [CGGL03] introduced a batch Skyline algorithm which uses a topological sort to speed up Skyline processing. Godfrey, one of the co-authors of this paper, discussed cardinality estimations for Skyline processing in [God04]. Some more theory on sorting based Skyline processing is given in Chomicki et al. [CGGL05]. Some sorting based Skyline processing ideas were also formulated in [Ros01].

Progressive Skyline algorithms were first introduced by Tan et al. in [TEO01]. The authors showed two algorithms which outperformed previous batch Skyline algorithms for some use

cases. In [EOT03] the authors generalized their ideas from [TEO01] and showed a few enhancements for their algorithms.

Kossmann et al. introduced the concept of online Skyline algorithms in [KRR02]. They described a Skyline algorithm using a multi-dimensional indexing structure and Nearest Neighbor Search. Papadias et al. [PTFS03] used the Nearest Neighbor idea to devise an optimal Skyline algorithm with respect to R-tree node accesses.

Balke et al. [BGZ04] introduced concepts to compute the Skyline of data that is distributed over the web at different data sites and Lo et al. [LYLC04] added progressiveness to Balke's algorithm.

Another concept for Skyline is the processing of partially-ordered domains, that is, computing the Skyline of domains which include interval data (for example, temporal data) categorical data (for example, type or class hierarchies). Some work has been done that addresses these kind of problems of partially-ordered domains, for example, [CET05a] and [CET05b].

There has been some attention on sub-space Skyline processing. The question is, what is the relationship between the Skylines in the subspaces and those in the super-spaces? Work done in this area include Pei et al. [PJET05], Tao et al. [TXP06], and Yuan et al. [YLL+05].

Computing the Skyline on a sliding window of streamning data has been subject to many recent research work, including Lin et al. [LYWL05], Morse et al. [MPG06], and Tao et al. [TP06].

Closely related to Skyline computing is the problem of computing the convex hull of a data set. The convex hull is a subset of the Skyline. The convex hull optimizes the data set for a linear scoring function whereas the Skyline optimizes the data set for any monotonic scoring function. In theory the convex hull has been discussed, for example, by Preparata et al. in [PS85]. There exist also lot of algorithmic approaches to compute the convex hull, for example, Akl et al. [AT78] and Green et al. [GS79], only to name two of them. However, most of these approaches assume the data set to be in main memory. A newer paper tackling the problem in a database context is Böhm et al. [BK01].

Another related field to Skyline is the Top-K query processing. Those queries return the best k objects that maximize or minimize a certain preference function. Top-K processing was subject of, for example, Carey et al. [CK97a] and [CK97b]. However Top-K processing would certainly find some interesting hotels but would also return non-interesting hotels. Top-K processing is, however, a good way to post-filter the Skyline further reducing the result size. A newer paper discussing Top-K processing in a streaming data environment has been published by Babcock et al. [BO03].

An interesting approach of combining Top-K retrieval and Skyline processing is presented in Balke et al. [BZG05].

Closely related to Skyline is the topic of Nearest Neighbor queries. A Nearest Neighbor queries looking for an optimal hotel with "zero cost" and "zero distance" would return interesting hotels but would also ignore a hotel fare away from the beach but extremely cheap (this hotel is certainly part of the Skyline). Nearest Neighbor queries has been discussed in various papers, for example, Roussopoulos et al. [RKV95] or Berchthold et al. [BBKK97]. A query returning k Nearest Neighbors, so called kNN-Queries would return interesting points but also points that are dominated by interesting points. These kNN-Queries have been, for instance, the topic of Seidl et al. [SK98] or Yu et al. [YOTJ01].

5.2 User Preferences and User Profiling

Defining queries with user preferences is the topic of Kießling [Kie02]. The paper presents a model for the formulation of user preferences and shows how to decompose complex preferences into simpler ones for efficient computation. Skylining and Pareto preferences are part of these decompositions.
Some newer work about preferencing includes Balke [Bal06].
User preferences and user profiling has been subject to many paper recent paper, primarily in the XML area. With being far from complete we try to point out some projects in the area XML and user preferences. These systems are mostly called Publish and Subscribe systems. Indexing predicates for faster trigger processing has been the idea in Hanson et al. [HCH+99]. Fabret et al. [FJL+01] also uses indices on predicates and a special clustering method in oder to reduce the number of required subscription checks. Gupta et al. [GS03] studied the evaluation of a large number of XPath filters, that is, a way to describe predicates in XML, by constructing a single deterministic pushdown automata. The Stanford Information Filtering Tool (SIFT) project, for example Yan et al. [YGM99] proposed various techniques for publish and subscribe systems. Routing and information filtering for XML messages has been subject in the XFilter project, Altinel et al. [AF00] and the YFilter project, Diao et al. [DFZF03].

5.3 Streaming and Continuous Query Processing

In Part IV we will use the Skyline as pre-filter for incoming points. Pre-filtering incoming points means continuously processing Skyline queries over continuously arriving points, that is, streaming data. The term *continuous query* has been formed by Terry et al. [TGNO92]. Other early work on processing continuous queries was carried out by Liu et al. [LPBZ96, LPT99] and recently, there has been significant work done in the area of streaming processing including continuous query processing.

There is the Stanford Stream Data Management (STREAM) Project [Sta04]. The group published various papers about data stream management, for example, Babcock et al. [BBD+02] being a good overview article about different aspects in stream processing, or Babu et al. [BW01].

Streaming data processing in sensor networks has been studied in the COUGAR project [Cou05] at Cornell. Papers the group published include Bonnet et al. [BGS01] and Yao et al. [YG03].

Continuous Query processing over XML data has been subject of the NIAGARA project [Nia04] at the University of Wisconsin-Madison and the Oregon Health & Science University. Publications like Naughton et al. [NDM+00] or Chen et al. [CDTW00] laid out the case.

Another data streaming, or data-flow engine has been developed within the Telegraph project at UC Berkeley. Their seminal paper, Avnur et al. [AH00], describes an adaptive query processing engine. Other basic papers of this group include Hellerstein et al [HFC+00] and Madden et al. [MSHR02]. This group also studied sensor networks and how sensor with limited capability can be queries continuously, Madden et al. [MF02].

Last but not least, there is the Aurora project at Brown University and Brandeis University in

combination with the Medusa project from the Massachusetts Institute of Technology. Aurora being a streaming processing engine and Medusa providing a special network infrastructure for Aurora operation. An overview of both projects is given in Carney et al. [CÇC+02] or Zdonik et al. [ZSC+03].

5.4 Multiple Query Optimization and View Maintenance

Also in Part IV we will consider multiple Skyline queries and how we can efficiently answer a group of Skyline queries. This touches the topic of multiple query optimizing. Sellis [Sel88] laid the basis for grouping queries in his work about multiple query optimization.

Another related topic is the incremental maintenance of materialize views. This topic has been subject of numerous research work, for example, Gupta et al. [GMS93] or Abiteboul et al. [AMR+98]. Both papers propose only a partial recomputing of the materialized view instead of recomputing the complete view.

5.5 Continuous Skyline Processing

For continuous Skyline processing there has been some research done for continous Skyline compuation over sliding windows. A Skyline algorithm working on sliding windows considers only the last n points for Skyline computation that have arrived at the systems.

5.6 Miscellany

Some algorithms in Part II and Part III make use of common indexing structure in database systems. Those indexing structures include B-trees and their variations and R-trees and their variations. Comer [Com79] discussed various aspects of the B-tree which was introduced by Bayer et al [BM72]. Beckmann et al. [BKSS90] extended the R-tree, called R*-tree, first introduced by Gutman [Gut84]. A good survey of multidimensional indexing tructures was published by Gaede et al. [GG98].
Algorithms in Part IV make use of Nearest Neighbor queries that search for Nearest Neighbor only in a certain area and Nearest Neighbor queries that are issued multiple times. A commonly used Nearest Neighbor algorithm in combination with R-trees is the Branch-and-Bound algorithm by Rossopolous et al. [RKV95]. Nearest Neighbor queries within a certain area, so called constrained Nearest Neighbor queries, have discussed in Ferhatosmanoglu et al. [FSAA01]. Continuously issued Nearest Neighbor queries have recently been studied by Tao et al. [TPS02].

CHAPTER 6

Settings for Performance Measurements

These sections deal with the kind of data sets and the environment that was used to conduct performance measurements. The sections give a general overview of data sets and computers that were used. Not all data sets were used for all measurements and computers could be changed for single measurements. The corresponding measurement environment is given with each performance results.

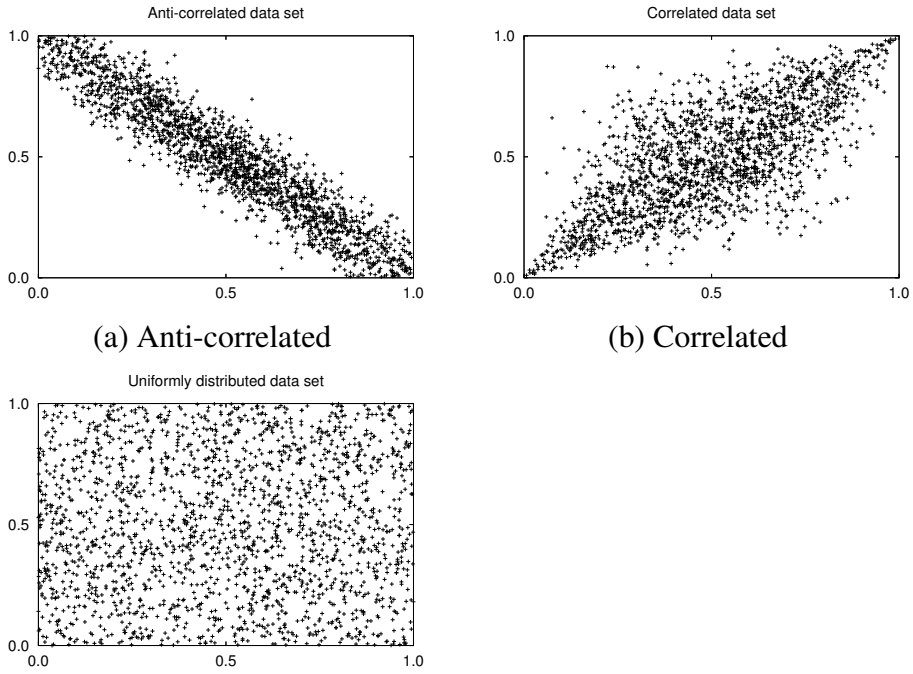
6.1 Multi-Dimensional Data Sets

The data sets we used are the same [BKS01] proposed in their seminal Skyline paper. Those data sets include anti-correlated, correlated and uniformly distributed data value distributions. We will describe the generation process for those distributions here.
We also introduce a new data set, Correlation Groups, which resembles one of our application domains from Chapter 2, namely the car Internet broker (Section 2.3). The generation of this data set is also described here.

We used floating point numbers for each dimension. The range of values for each dimensions was $[0.0; 1.0)$ with 6 decimal digits. That means that the value ranged from 0.000000 to 0.999999. The dimensionality of the points in the set ranged from two dimensions up to 10 dimensions. Mostly, data sets with 100,000 points were used. Some measurements included data sets containing 1,000,000 points. Figure 6.1 shows 2-dimensional samples of the three value distributions anti-correlated, correlated, and uniformly distributed. The correlation groups value distribution is not shown here (see later for explanation).

Data Set: Anti-correlated

An anti-correlated data set represents data in which that points are good in one dimension tend to be bad in one or all other dimensions. Our hotel example (Section 2.1) falls into this category. For this distribution we first choose a plane perpendicular to the line from $(0, ..., 0)$ to $(1, ..., 1)$ using a normal distribution. The variance used for the normal distribution was very small so that points are placed close to the point $(0.5, ..., 0.5)$. Within the plane, the attribute values



(a) Anti-correlated (b) Correlated

(c) Uniformly distributed

Figure 6.1: *Visualization of the data sets*

are generated using a uniform distribution. Figure 6.1(a) shows a 2-dimensional anti-correlated data set with 2000 points.

Data Set: Correlated

Correlated points tend to be good in all dimensions. Certain aspects of our car Internet broker example fall into this category: speed and horse power are certainly correlated. A random point for this data value distribution is generate as follows. First, a plane perpendicular to the line from the origin to $(1,...,1)$ is selected using a normal distribution. A normal distribution is used in order to get more points in the middle than at the ends. The new point will be in that selected plane. Within the plane, all attribute values of the point are generated using a normal distribution. This ensures that most points are placed close to the diagonal from $(0,...,0)$ to $(1,...,1)$. Figure 6.1(b) shows a 2-dimensional data set containing 2000 points.

Data Set: Uniformly Distributed

Uniformly distributed means, that in each dimension a value is equally likely. For this kind of data set all attribute values are generated independently using a uniform distribution. Figure 6.1(c) displays a 2-dimensional data set with 2000 points.

Data Set: Correlation Groups

This data set consists of two or more correlation groups. A correlation group consists of values that are correlated within the group but have no observable correlation to the other correlation groups. For example, consider a 5-dimensional data set describing the significant attributes of a car. We can identify two correlation groups, that is, {*price, insurance, taxes*} and {*speed, horse*

power}. The dimensions are grouped into two correlation groups. Within a group, values are correlated, that is, normally a higher price results in higher a higher insurance rate and speed and horse power are also correlated. On the other hand horse power and price have not necessarily a correlation (OK, OK, more expensive cars normally have more horse power, but there are also cheaper cars having more horse power than you might expect - a *Rennsemmel*). We generate a correlation groups data set by taking two correlated data sets with lower dimensionality. For the example given above we used a 2-dimensional correlated data set concatenated to a 3-dimensional correlated data set. Depending on which two dimensions you would choose the picture of a correlation groups data set would look like Figure 6.1.

Number of Skyline Points

As we already said we used mostly data sets containing 100,000 points. Table 6.1 gives the number of Skyline points for an anti-correlated, correlated, and uniformly distributed containing 100,000 points. The correlation groups data set is left out here because it serves a special purpose. We only used a 5-dimensional correlation groups data set.

Dimensions	Anti-correlated	Correlated	Uniformly distributed
2	49	1	12
3	632	3	69
4	4239	11	267
5	12615	17	1032
6	26843	21	1986
7	41484	43	5560
8	55691	121	9662
9	67101	243	16847
10	75028	378	26047

Table 6.1: *Skyline sizes for a 100,000 points data sets*

The following comparison hold true for the data set and any subset that was constructed out of the original data set:

$$\begin{aligned}\text{SkylineSize}(\text{anti-correlated}) > \text{SkylineSize}(\text{uniformly distributed}) > \\ \text{SkylineSize}(\text{correlation groups}) > \text{SkylineSize}(\text{correlated})\end{aligned}$$

The percentage of Skyline points of a data set ranges from 0.001 % (2-dimensional correlated data set) to 75 % (10-dimensional anti-correlated data set).

6.2 Conducting Measurements

6.2.1 Distinguishing

There are two kinds of algorithms: Algorithms designed by us, and algorithms designed by others. That leads to different ways of reporting measurement results. As for Part II and Part III some algorithms are designed by us and some are not. For these two parts our performance reporting completely relies upon results published in papers, either by us or by other authors.

Mostly, the underlying data sets are the same as reported in the previous section but the computers the measurements are conducted on vary. That is no big deal, since results always compare either our algorithms to other authors' algorithms or vice versa. The reader should get the picture even if the result resembles the picture of computers that were used 2 or 3 years ago. In Part IV, the (historically) latest part, we show extensive measurement on todays' available computers. See next section for more details.

6.2.2 Machine Data

Most measurements were conducted on a Pentium 4 processor at 3.2 GHz. with 2000 MB of main memory. The operation system was SuSE Linux 9.0 (Kernel 2.6). The data was hosted on 596 GB SCSI RAID systems. Almost all algorithms worked solely in main memory and were CPU bound. The loading times for the data were neglected, that is, the running time measurements started after the data have been loaded.

CHAPTER 7

Hotel Example and Pseudo Code

7.1 Example

We want to introduce a short example. This example is used throughout this thesis to illustrate the techniques of the different Skyline algorithms. For some algorithms it can be taken as is, for some Skyline algorithms it has to be adapted. The example resembles an application domain given in the introduction: Hotels with price and distance to the beach. For each Skyline algorithm the example is repeated and adaptions, if necessary are shown.
There are, for now and for ease of presentation only three dimensions describing the characteristics of hotels, These correspond to attributes of a table. The first dimension is the name of the hotel, the second dimension is the price for a double bedroom per night and the third dimension is the distance to the beach. A short example occurrence is given in Table 7.1. Hotel names in bold are Skyline hotels.

Name	Price [EUR]	Distance [m]
Hotel Arena	45	100
Hotel Aden	40	200
Hotel International	42	300
Hotel Aurora	35	400
Hotel Majestic Toscanelli	50	280
Hotel Monaco & Quisisana	60	150
Hotel Elpiro	55	50
Hotel Marlisapier	65	250
Hotel Al Gambero	72	40
Hotel Rex	40	500
Hotel Heron	68	100

Table 7.1: *Hotels in Lido di Jesolo*

Figure 7.1 shows a graphical representation of Table 7.1. The hotel example would be characterized as an anti-correlated data set as denoted in Chapter 6.

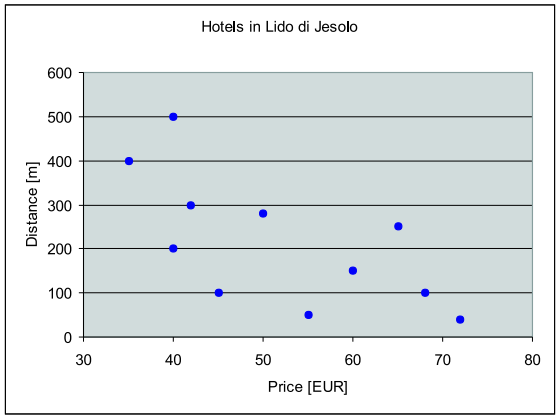


Figure 7.1: *Hotel in Lido di Jesolo, graphical representation*

All hotels are all located in Lido di Jesolo, Italy [Jes06]. Please note: Hotel names correspond to real hotels in Lido di Jesolo. Prices and distances, however, are totally fictitious. They are made up to to illustrate the facts in the following parts and sections. Of course the distance to the beach should be as short as possible since no one wants to walk too far before he or she can jump into the *Mare Adreatico*. And, of course, the price should be as low as possible. Who wants to pay more than necessary?

7.2 Pseudo Code

All algorithms that are shown in this thesis are displayed in pseudo code. The main keywords of the pseudo code are described here. The pseudo code is Pascal-like. Blocks normally start with **begin** and end with **end**. However, if the start of a block is clear, the **begin** will be omitted. This is true for the **if**-, **while**-, and **for**-statement. We also add some object-oriented constructs to our pseudo code. The members and methods of the objects should be self-explaining. If not, they are explained somewhere in the code or in the description of the code.
The following describes the basic statements used in the pseudo code, keywords are printed in **bold**, statements are concluded by a semi-colon.

- Comment: // ... or /* ... */.
- Negation: **!***bool-expression*, negates a boolean expression.
- Assignment: *variable* := *expression*, assigns the right-hand *expression* to the left-hand *variable*.
- Comparisons: *expression1* = *expression2*, tests if *expression1* equals *expression2*. Other comparison operators are $<$, $>$, $<=$, $>=$ with the usual meaning.
- Block: **begin** [*Name*] *statements* **end**: Describes a block of statements, **begin** can be omitted (see above).
- Loop: **for** (*loop-index begin* **to** *loop-index end*) **do** *statements* **end**: Normal for-loop. Loop is executed until *loop-index end* is reached.

- Loop: **while** (*loop-condition*) **do** *statements* **end**: Normal while-loop. Loop is executed until *loop-condition* evaluates to false.
- Conditional execution: **if** (*condition*) **then** *statements* [**else** *statements*] **end**: If *condition* evaluates to true the *statements* after **then** are executed, if *condition* evaluates to false the *statements* after **else** are executed.

At the beginning of each algorithm input and output variables are declared. These variables behave like global variables, that is, they are visible throughout the program code and also in sub-functions. All other variables are visible in the block they are declared in and all subordinate blocks. Variables can have a variety of types:

- Simple data types
  - **int**: Integer numbers, a subset of the natural numbers (0, 1, 2, ...) and their negatives (-1, -2, ...). In mathematics denoted with $\mathbb{Z}$. Integers number are limited in range due to a limited number of bits in their internal representation, mostly 32 bits.
  - **float**: Floating point numbers, a subset of the real numbers and their negatives. In mathematics denoted with $\mathbb{R}$. Floating point numbers are limited in range and precision due to a limited number of bits in their representation, mostly 32 or 64 bits.
  - **bool**: Boolean variable, a variable that is either *true* or *false*.
- Complex data types:
  - **Point**: Describing a point with a **float** value for each dimension. A point also has a method *dim* that returns the number of dimensions.
  - **Dataset**: Set of **Points**. Typical methods are *insert(Point)*, *delete(Point)* with the usual meaning, and *length* returning the number of elements in the data set.
  - **Skyline**: Set of **Points**, subset of a **Dataset** with Skyline property. Typical methods are *insert(Point)*, *delete(Point)* with the usual meaning, and *length* returning the number of elements in the Skyline.
  - **File**: Pointer to a file on disk containing information needed.
  - **Todolist**: Set of **Points** describing minimum bounding rectangles in an R-tree. Since our minimum bounding rectangles always have the origin as lower left point, another point is sufficient (see Chapter 17) to describe a d-dimensional minimum bounding rectangle.

The purpose of the pseudo code is to describe the algorithms more formally than it is possible with text only. Readers having some knowledge in a modern programming language will not have any problems understanding the pseudo code.

Part II

Batch and Progressive Skyline Computation

CHAPTER 8

Classification

The following pages describe what we call batch and progressive Skyline algorithms. Both kinds of algorithms treat the Skyline computation as a detached operation with no user interaction. The described algorithms can be easily incorporated into existing DBMS. There is almost no change required to the logical data model, that is, the relational schema of the data.

8.1 Batch Algorithms

We classify the Skyline algorithms published in [Bör99] and [BKS01] as batch algorithms, meaning that they take a set of points and compute the Skyline of this set of points. There is no user interaction and the output of the computed Skyline takes place at the very end of the computation. Like in the "old" days of main frame computing, the program code was submitted (via punched card) and sometime later the program was finished and the results were picked up by the user who wrote the program.
Common to all of these algorithms is that they do multiple passes over the data they compute the Skyline of. Even stronger, the require at least one complete pass of the data to output a Skyline point. The following algorithms fall into this group:

- **Standard/naive algorithm**: This algorithm is described in Chapter 9.
- **Block-Nested-Loops algorithm**: This algorithm is described in Chapter 10.
- **Divide-and-Conquer algorithm**: This algorithm is described in Chapter 11.

8.2 Progressive Algorithms

Skyline algorithms we classify as progressive algorithms were published in [TEO01]. They also compute the Skyline in a batch sort of way as described in the previous section. They require, however, only one pass of the data and output Skyline points continuously during Skyline computation. Some Skyline points can be output with one pass of the data. We shall see the main difference to other continuously outputting algorithms in Part III. Progressive algorithms are:

- **Bitmap based algorithm**: This algorithm is described in Chapter 12.
- **Partition-Index algorithm**: This algorithm is described in Chapter 13.

8.3 Candidate Skyline Computation - Skyline Pre-filters

In Section 14 we show two ideas to pre-filter the data set the Skyline is computed on, that is, the data set is run trough a filter in order to eliminate points that definitely cannot belong to the Skyline. The resulting reduced data set is then run through a normal batch or progressive Skyline algorithm to determine the Skyline.

CHAPTER 9

The Standard Algorithm

9.1 Algorithm Description

It is quite clear how to compute the Skyline of a set of points: Compare each point with every other point in the data set. The only thing which one has to check, is that identical points are not compared to each other. That would yield false positives since according to the definition in Chapter 4 two identical points are both part of the Skyline. To avoid these false positives simply omit the points with the same index in the data set.

Name	Short	Price [EUR]	Distance [m]
Hotel Arena	**a**	45	100
Hotel Aden	**b**	40	200
Hotel International	c	42	300
Hotel Aurora	**d**	35	400
Hotel Majestic Toscanelli	e	50	280
Hotel Monaco & Quisisana	f	60	150
Hotel Elpiro	**g**	55	50
Hotel Marlisapier	h	65	250
Hotel Al Gambero	**i**	72	40
Hotel Rex	j	40	500
Hotel Heron	k	68	100

Table 9.1: *Hotel example for the Standard algorithm*

Applying the Standard algorithm to out hotel example is straightforward. Table 9.1 shows the hotel example. No changes need to be applied to the data. The algorithm simply takes the first point a and compares it with all other points (b through k). It does not compare a with itself. Having looked at all points the algorithm can say that a is a Skyline point. Now the algorithm continues with b and looks at all other points starting from a through k skipping itself. This continues until all points have been compared with each other.
The Standard algorithm in pseudo code is displayed in Figure 9.1. The $<$ (line 20) denotes domination as defined in Chapter 3. The algorithm simply makes a copy of the data set the

```
1  Input:     Dataset D
2  Output:    Skyline S
3
4  begin StandardAlgorithm
5
6    // Walk through outer dataset
7    for (int i := 1 to D.size()) do
8      Point p := D[i];
9
10     // Dataset needed a second time
11     Dataset D_temp := D;
12
13     // Control Skyline
14     bool hit := true;
15
16     // Walk through inner Dataset
17     for (int j:= 1 to D_temp.size()) do
18       Point q := D_temp[j];
19
20       if(i != j && q < p) then
21         hit := false;
22         break;
23       end;
24     end;
25
26     if(hit) then
27       // p survived one pass
28       S.insert(p);
29     end;
30   end;
31
32 end;
```

Figure 9.1: *Standard algorithm for Skyline computing*

Skyline is computed on and compares each point in the outer data set with each point in the inner data set. Note: The algorithm does not make a complete copy of the data set. It simply opens up another pointer to the file or table in the database system.

9.2 Discussion

It is by fare the easiest algorithm to compute the Skyline. The easiness of the algorithm is its main drawback. Going through tho whole data set for every point is not efficient enough for modern application requirements. But, nevertheless, for small data sets it is fast enough and implementing does not require in-depth knowledge of Skyline computation. The Standard algorithm is mostly used as “worst case” example, but still performs well for low dimensionality or small data sets. Some algorithms discussed later require the computation of a pre-Skyline,

that is, the Skyline of a small subset of the whole data set. The Standard algorithm is suitable for these kinds of computations. Moreover, it does not favor any data distribution. As we will see in further chapters, some algorithms - batch, progressive, or online algorithms - perform better for certain data value distributions. The Standard algorithm is "immune" to changes in data value distributions which makes it applicable for small Skyline problems with possibly different data value distributions. It is also suitable when memory usage is of importance since it does not require any additional memory except for comparing two points and storing the Skyline. As we will see in later Chapters there are algorithms which need additional memory.

CHAPTER 10

The Block-Nested-Loops Algorithm

10.1 Description

The Block-Nested-Loops algorithm is based on the Standard algorithm from Chapter 9. It compares each point in the data set to a *window* of incomparable points, that is, points that represent the Skyline of the points in the data set seen so far, we call it the candidate Skyline. If a new point of the data set is dominated by any point in the window, it is discarded. In this case it has not to be compared to all points in the data set which marks the main difference to the Standard algorithm. If a new point of the data set dominates one or more points in the window, these points are deleted from the window and the new point is inserted into the window. If a new point is not dominated by any point of the window it is a potential new Skyline point and hence inserted into the window. The Block-Nested-Loops algorithm in pseudo code is displayed in Figure 10.1.
The algorithm pays particular attention to the fact when the candidate Skyline points, the window, does not fit into main memory anymore (indicate by a computer flag *memAvailable*. Then the newly read point is swapped to a temporary file on disk. The time when a point was swapped out is denoted by a timestamp (*timestampOut*). Before an new point is read from the data set the candidate points are checked for any points that have already been identified as Skyline points, that is, all points that have been compared to all points in the data set and are still in the candidate Skyline. That is true for all points in the candidate Skyline that have the same input timestamp (*timestampIn*) than the current timestamp. After that check a new point is read from the data set and assigned the current timestamp. It is put into the window and compared to all points in there (except itself, of course). If it is dominated it is deleted from the window and the algorithm continues with the next point from the data set. If it dominates candidate Skyline points all dominated points are delete from the window and the new point stays in the window. After the algorithm is done with the data set it checks if there are any points swapped out on disk. If this is true the temporary file is loaded and the algorithm finishes up the points from the temporary file points. At the end all points that are still in the window are copied to the Skyline. Let us take a look at the first few steps of the Block-Nested-Loops algorithm considering our example in Table 9.1. We assume a window size, that is, memory limitations, of three points. Table 10.1 displays the value of important variables after each step (lines 43).

```
1  Input:     Dataset D
2  Output:    Skyline S
3
4  begin BNLAlgorithm
5    Dataset candSkyline; File temp;
6    int timestampIn := 0; int timestampOut := 0;
7
8    for (int i := 1 to D.size()) do // Walk through dataset
9
10     // all points that survived one pass
11     for (int t := 1 to candSkyline.size()) do
12       Point q := candSkyline[t];
13       if(q.timestamp() = timestampIn) then
14         S.insert(q); candSkyline.delete(q);
15       end;
16     end;
17
18     // Read point from dataset
19     Point p := D[i];
20     p.timestamp := timestampOut;
21     candSkyline.insert(p); timestampIn++;
22
23     // Compare to "window" of points, skip "last" point (p)
24     for (int t := 1 to candSkyline.size()-1) do
25       Point q := candSkyline[t];
26       if(p > q) then
27         candSkyline.delete(p); break; // p is dominated by q
28       else
29         candSkyline.insert(p); candSkyline.delete(q); // p dominates q
30       end;
31       if (!memAvailable) then
32         temp.insert(p); candSkyline.delete(p);
33         timestampOut++;
34       end;
35     end;
36
37     // algorithm done with D
38     if(!temp.empty() && i = D.size()) then
39       D.load(temp); temp.clear();
40       timestampIn := 0; timestampOut: = 0;
41     end;
42
43   end;
44
45   for (int t := 1 to temp.size()) do
46     Point q := temp[t];
47     S.insert(q); temp.delete(q);
48   end;
49 end;
```

Figure 10.1: *Block-Nested-Loops algorithm for Skyline computing*

Step	Read	candS	tsIn	tsOut	temp	Skyline
Load D		-	0	0	-	-
1	*a*	*a* (0)	1	0	-	-
2	*b*	*a* (0), *b* (0)	2	0	-	-
3	*c*	*a* (0), *b* (0), *c* (0)	3	0	-	-
4	*d*	*a* (0), *b* (0), *c* (0)	4	1	*d* (0)	-
5	*e*	*a* (0), *b* (0), *c* (0)	5	1	*d* (0)	-
6	*f*	*a* (0), *b* (0), *c* (0)	6	1	*d* (0)	-
7	*g*	*a* (0), *b* (0), *c* (0)	7	2	*d* (0), *g* (1)	-
8	*h*	*a* (0), *b* (0), *c* (0)	8	2	*d* (0), *g* (1)	-
9	*i*	*a* (0), *b* (0), *c* (0)	9	2	*d* (0), *g* (1), *h* (2)	-
10	*j*	*a* (0), *b* (0), *c* (0)	10	2	*d* (0), *g* (1), *h* (2)	-
11	*k*	*a* (0), *b* (0), *c* (0)	11	2	*d* (0), *g* (1), *h* (2)	-
Load temp		-	0	0	-	-
12	*d* (0)	*d* (0)	1	0	-	*a* (0), *b* (0), *d* (0)
13	*g* (1)	*d* (0), *g* (0)	2	0	-	*a* (0), *b* (0), *d* (0)
14	*h* (2)	*d* (0), *g* (0), *h* (0)	3	0	-	*a* (0), *b* (0), *d* (0)

Table 10.1: *Steps of the Block-Nested-Loops algorithm*

The number in parentheses after each point shows the timestamp of the point. The last step is omitted, that is, taking all remaining points from the window and transferring it to the Skyline.

A variation of the Block-Nested-Loops algorithm organizes the window of candidate Skyline points as a self-organizing list, that is, a point which dominates another point from the data set is moved to the beginning of the window. That way *most* dominant points are located at the beginning of the window and hopefully incoming points are kicked out by these points without having to look at large portions of the window. This variant is particular effective for skewed data, that is, if there are some points that dominate many other points.

Additional variants of the Block-Nested-Loops algorithm are depicted in [BKS01]. The authors played with different variations of the Block-Nested-Loops algorithm and showed an extensive performance study.

10.2 Discussion

The Block-Nested-Loops algorithm works particular well if the Skyline is small, that is, during computation the candidate Skyline points fit into the window and main memory. This algorithm needs additional memory, namely the window. If the window is too small the algorithm has to reload parts of the window from the temporary disk storage and perform multiple runs. The best case for the Block-Nested-Loops algorithm is, obviously, when all (candidate) Skyline points fit into the window. This is true for data sets having a small dimensionality and higher-dimensional data sets having a correlated or uniformly distributed data value distribution (see Chapter 6 for details). For more details on performance of the Block-Nested-Loops algorithm the interested reader is referred to [BKS01].

CHAPTER 11

The Divide-and-Conquer Algorithm

11.1 Description

The Divide-and-Conquer algorithm was first described in [KLP75, PS85]. This basic Divide-and-Conquer algorithm works by recursively partitioning the data set along a pre-determined median value of one dimension. See Figure 11.1(a). It displays the hotel example from Table 9.1:

1. Choose an arbitrary dimension d_p for partitioning, for example *price*. Compute the median or some approximate median m_{price}.

2. Divide the data set into two partitions P_1 and P_2 such that P_1 contains all points whose value of dimension d_{price} is better than m_{price}, that is, $p \in P_1 \iff p_{\text{price}} \leq m_{\text{price}}$ and P_2 contains all points that have a worse value in dimension d_{price}, that is $p \in P_2 \iff p_{\text{price}} > m_{\text{price}}$.

3. Compute the Skylines S_1 of P_1 and S_2 of P_2. In order to do so, apply the algorithm recursively on P_1 and P_2.

4. The recursion stops if a partition contains only one or a few points. The Skyline for one point is trivial, the Skyline of a few points can, for example, be computed with the Standard algorithm from Chapter 9.

5. Compute the overall Skyline by merging S_1 and S_2. While merging those points of S_2 are eliminated that are dominated by points in S_1.

Step 5 means basically computing the Skyline of $S_1 \cup S_2$ using the Standard algorithm (Chapter 9) or the Block-Nested-Loops algorithm (Chapter 10). This can be quite time consuming and memory intensive, that is, if $S_1 \cup S_2$ does not fit into main memory. Another idea is described in [PS85] and is depicted in Figure 11.1(b). The idea is to partition S_1 and S_2 using a median of another dimension d_m with $d_p \neq d_m$. In our case we use the dimension *distance* with m_{distance}. As a result we get 4 partitions, $S_{1.1}$, $S_{1.2}$, $S_{2.1}$ and $S_{2.2}$. The following observations facilitate the merging process ($i \in \{1,2\}$):

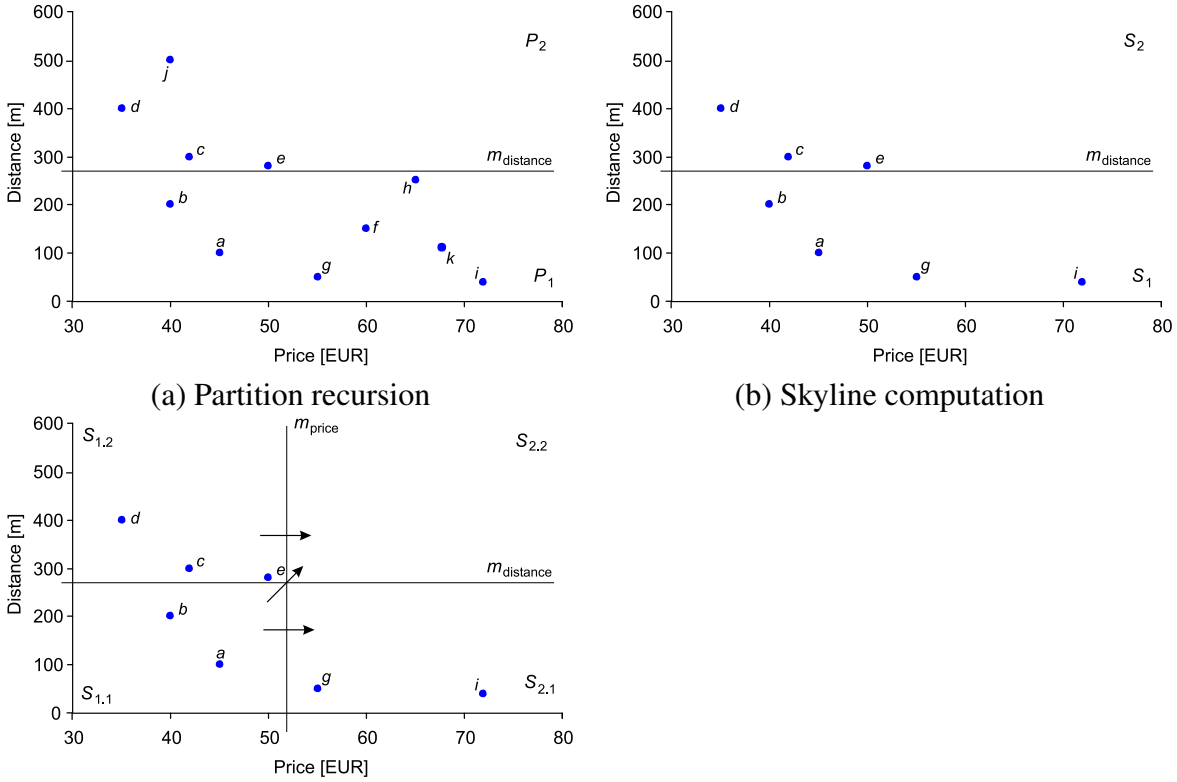
- Points in $S_{1.i}$ are better in dimension d_p than points in $S_{2.i}$.
- Points in $S_{i.1}$ are better in dimension d_m than points in $S_{i.2}$

Merging has to be done between $S_{1.1}$ and $S_{2.1}$, $S_{1.2}$ and $S_{2.2}$, and $S_{1.1}$ and $S_{2.2}$. $S_{1.2}$ and $S_{2.1}$ need not be merged since points of both partition are definitely incomparable, that is, $\forall p \in S_{1.2} \wedge \forall q \in S_{2.1} \mid p_x < q_x \wedge p_y > q_y$. Merging of all other pairs is done recursively by partitioning all pairs again. The recursion stops if all dimensions have be considered or if one partition is empty. Merging is then trivial.
As an example consider again Figure 11.1. We stop the partition recursion after one step, as it is depicted in Figure 11.1(a) and compute the Skyline of P_1 and P_2 by the Standard algorithm, $S_1 = \{c,d,e\}$ and $S_2 = \{a,b,g,i\}$ (see Figure 11.1(b)). During the merging phase we also stop the recursion after on step. This is depicted in Figure 11.1(c). Now we perform the merging. As said above merging has to be done between $S_{1.1}$ and $S_{2.1}$, the resulting Skyline here is $r_1 = \{a,b,g,i\}$; between $S_{1.2}$ and $S_{2.2}$ resulting in $r_2 = \{c,d,e\}$ (note: $S_{2.2} = \{\}$); and $S_{1.1}$ and $S_{2.2}$ resulting in $r = \{a,b\}$. Instead of merging $S_{1.1}$ and $S_{2.2}$ to r $S_{1.1}$ can be immediately merged with r_2 yielding $r_3 = \{a,b,d\}$. The resulting overall Skyline is then $r_1 \cup r_3 = \{a,b,d,g,i\}$.

In [BKS01] two variants of the Divide-and-Conquer algorithm are given which enhance the basic Divide-and-Conquer algorithm. We want to quickly introduce the two variants here since they mark the best performing variants of the Divide-and-Conquer algorithm.
The first variant is called *m-way partitioning*. The idea is to divide into m partitions so that each partition is expected to fit into main memory. The m-way partitioning can be applied to the partitioning step to produce partitions $P_1, ...P_m$ so that each partition fits into main memory as well as the merging step to produce partitions in a way that all partitions necessary for one merging step fit into main memory.
The second variant is called *early Skyline*. This variant changes the partitioning step where the data set is partitioned into m partitions. This variant simply computes a candidate Skyline before the partitioning step takes places, that is, load as many points as fit into main memory and compute a candidate Skyline of these points by applying the basic Divide-and-Conquer algorithm. Points that are already being dominated are deleted from main memory. Divide the remaining points into m partitions and proceed with the m-way Divide-and-Conquer algorithm. The pseudo code of the basic Divide-and-Conquer algorithm is given in Figure 11.2 and Figure 11.3.

11.2 Discussion

If the input data set does not fit into main memory the performance of the Divide-and-Conquer algorithm degrades heavily. The reason for this can easily be seen: the data set is read from disk, partitioned, partly written to disk, read again from disk, and so on. This is done until a partition fits into main memory. The two proposed variants relieve this problem a bit.
The Divide-and-Conquer algorithm has the same complexity in the worst and in the best case. In bad cases where the data set contains many Skyline points, it outperforms the Block-Nested-Loops algorithm. In good cases, it is outperformed by the Block-Nested-Loops algorithm.



(a) Partition recursion

(b) Skyline computation

(c) Merging recursion

Figure 11.1: *Partitioning for Divide-and-Conquer algorithm*

```
1  Input:     Dataset D
2  Output:    Skyline S
3
4  begin DCAlgorithm
5    SkylineBasic(D, dim);
6  end;
7
8  function SkylineBasic(Dataset D, int dim)
9
10   if (D.size() = 1)
11     return D;
12   end;
13
14   // Partition dataset D using dimension dim
15   float pivot := median(D, dim);
16   Dataset p1, p2;
17   (p1, p2) = partition(D, dim, pivot);
18
19   // Recursive
20   Skyline s1 := SkylineBasic(p1, dim);
21   Skyline s2 := SkylineBasic(p2, dim);
22
23   // Merge
24   return union(s1, MergeBasic(s1, s2, dim));
25 end;
```

Figure 11.2: *Divide-and-Conquer algorithm for Skyline computing*

```
1  function MergeBasic(Skyline s1, Skyline s2, int dim)
2
3    Skyline r;
4    if(s1.size() = 1) then
5      Point p := s1[1];
6      for int i := 1 to s2.size() do
7        Point q := s2[i];
8        if(q < p) then
9          r.insert(q);
10       end;
11     end;
12   else if(s2.size() = 1) then
13     r := s2;
14     for int i := 1 to s1.size() do
15       if(p < q) then
16         r := NIL;
17       end;
18     end;
19   else if(dim = 2) then
20     Point min := minimum(s1);
21     for int i := 1 to s2.size() do
22       Point q := s2[i];
23       if(q < min) then
24         r.insert(q);
25       end;
26     end;
27   else
28     float pivot = median(s2, dim - 1);
29     Skyline s11, s12, s21, s22;
30     (s11, s12) := partition(s1, dim - 1, pivot);
31     (s21, s22) := partition(s2, dim - 1, pivot);
32     Skyline r1 := MergeBasic(s11, s21, dim);
33     Skyline r2 := MergeBasic(s12, s22, dim);
34     Skyline r3 := MergeBasic(s11, r2, dim - 1);
35     r := union(r1, r3);
36   end;
37
38   return r;
39 end;
```

Figure 11.3: *Merge algorithm for D&C Skyline computing*

CHAPTER 12

Bitmap Based Algorithm

This algorithm has first been published in [TEO01]. Some enhancement has been published in [EOT03]. We explain the original Bitmap algorithm by applying it to our hotel example.

12.1 Algorithm Description

Previous algorithms had to look at the points in the data set at least once to determine all Skyline points (most algorithms required multiple passes over the data set). In this section we describe an algorithm that requires exactly one pass over the data set to compute all Skyline points. This algorithm uses bitmaps to encode all information that is needed to decide whether a point is part of the Skyline or not. A point $p = (p_1, p_2, ..., p_d)$ with d being the number of dimensions is mapped to a bit vector. This bit vector holds information about the rank of each value $p_1, p_2, ..., p_d$ compared to other values of the same dimension. The length of the bit vector is determined by the number of distinct values over all dimensions. Let k_i the number of distinct values in the ith dimension then m is the sum of all k_is, that is, $m = \sum_{i=1}^{d} k_i$. For our hotel example see Table 12.1. This table shows all distinct values and their rank of the hotel example. The number of distinct values in the first dimension is $k_1 = 10$ and the number of distinct values in the second dimension is $k_2 = 10$. This means $m = k_1 + k_2 = 10 + 10 = 20$.
The bitmaps encode the rank or position of the value of the point in each dimension. Assume that p_i, the value of point p in the ith dimension, is j_ith smallest value in dimension i, that means that $j-1$ values in dimension i are smaller than this value. This value p_i will be encoded with k_i bit (see above) where the $k_i - j_i + 1$ most significant bits are set to 1 and the remaining ones are set to 0. Take a look at Table 12.2. "Hotel Heron" (the last one) has a price of 68 EUR which is the 9th smallest value of all points. That means that the $10-9+1=2$ most significant bits of the bitmap representing dimension 1 will be set to 1 and the remaining bits will be set to 0, `1100000000`. Similarly, the distance of "Hotel Heron" is 100 m which is the 3rd smallest value of all points, the bitmap representation will contain $10-3+1=8$ 1s at the beginning, `1111111100`.
Now let us take a look at two example points. By accessing the point we want to decide whether it is part of the Skyline or not. We first take a look at a Skyline point, "Hotel Elpiro" with

Rank	Value	
	Dimension 1 (price)	Dimension 2 (distance)
1	35	40
2	40	50
3	42	100
4	45	150
5	50	200
6	55	250
7	60	280
8	65	300
9	68	400
10	72	500

Table 12.1: *Number of distinct values*

Name	Price [EUR]	Distance [m]	Bitmap	
			Dimension 1	Dimension 2
Hotel Arena	45	100	1111111000	1111111100
Hotel Aden	40	200	1111111110	1111110000
Hotel International	42	300	1111111100	1110000000
Hotel Aurora	35	400	1111111111	1100000000
Hotel Majestic Toscanelli	50	280	1111110000	1111000000
Hotel Monaco & Quisisana	60	150	1111000000	1111111000
Hotel Elpiro	55	50	1111100000	1111111110
Hotel Marlisapier	65	250	1110000000	1111100000
Hotel Al Gambero	72	40	1000000000	1111111111
Hotel Rex	40	500	1111111110	1000000000
Hotel Heron	68	100	1100000000	1111111100

Table 12.2: *Hotel example for Bitmap algorithm*

the bitmap representation (1111100000, 1111111110). The algorithm creates two bit-slice a_1=11111010010 and a_2=00000010100. These bit-slice contain 11 bits, one entry for each hotel. The bit-slice are obtained by "reading" the bitmaps in Table 12.2 column wise. The column is determined by the point that is to be checked for Skyline property. It is the least significant column that contains a 1, that is for a_1 the 6th column (counted from the end of bitmap for dimension 1), and the 2nd column for a_2. The result of a bitwise AND operation is

$$A = a_1 \wedge a_2 = 11111010010 \wedge 00000010100 = 00000010000$$

The bit-slice A has the property that the nth bit is set to 1 if the nth point has dimension values less or equal to the dimension values of the point to be checked for Skyline property. Now the algorithm creates a second bit-slice for each dimensions. This second bit-slice is the preceding bit-slice of a. This yields another two bit-slices b_1=11111000010 and b_1=00000000100. The results of a bitwise OR operation is

$$B = b_1 \vee b_2 = 11111000010 \vee 00000000100 = 11111000110$$

Bit-slice B has the property that the nth bit is set to 1 if the nth point has some dimension values less than the value of the corresponding dimension of the point in turn. If there is no preceding

bit-slice the result of this operation is set to 0. The final bit operation determines the Skyline property of "Hotel Elpiro"

$$C = A \wedge B = 00000010000 \wedge 11111000110 = 0000000000$$

Since the final bit operation yields a bit-slice that does not contain 1s, that is, it is 0, "Hotel Elpiro" belongs to the Skyline.
Now let us try this with an non-Skyline hotel. We choose "Hotel Heron". The algorithm again creates two column wise-bitmaps a_1=11111111011 and a_2=10000010101. The result of the first bitwise AND operation is

$$A = a_1 \wedge a_2 = 11111111011 \wedge 10000010101 = 10000010001$$

The bitwise OR operation yields

$$B = b_1 \vee b_2 = 11111111010 \vee 00000010100 = 11111111110$$

And the final bitwise AND yields

$$C = A \wedge B = 10000010001 \wedge 11111111110 = 10000010000$$

It does not belong to the Skyline since the resulting bit-slice contains 1s. Moreover, we can see that "Hotel Heron" is dominated by the 1st and the 7th hotel, because the resulting bit-slice contains the 1 at the first and 7th position. Figure 12.1 shows the pseudo-code of the Bitmap algorithm. The function *determineIndex* computes the number of the bit-slice to be taken (see above). The function *getBitSlice* gets two input parameters, the first one is the index of the bit-slice, the second one the dimension to be taken.

12.2 Discussion

The Bitmap algorithm can return Skyline points by scanning the whole data set once. By using bitmap operation it decides on the Skyline property. The efficiency of the algorithm relies totally on the speed of bitwise operations. A major drawback of this algorithm lies in the use bitmaps and bit-slices in order to determine if a point is in the Skyline or not. This drawback is two-fold. First, for each point inspected the algorithm must retrieve the bitmaps of all dimensions of all points in order to get the bit-slices. Second, if the number of distinct values is large, in our example we only had 10 different values per dimension, the space consumption of the bitmaps may be prohibitive for the algorithm. Remember that the length of the bitmaps over all dimensions is the sum of distinct values in each dimension (see above). Consider a 5-dimensional data set with 100,000 points. We assume, we have 1000 distinct value for each dimensions. By choosing randomly generated floating point values for each dimension, this seems not too much. That means the total length of the bitmap is $l_{\text{Bitmap}} = 5 \cdot 1000$ Bits $= 5000$ Bits. The space consumption is $s = \frac{100000 \cdot 5000 \text{ Bits}}{8 \text{ Bits/Byte}} = 62500000$ Bytes ≈ 62.5 MBytes. These bitmaps are needed in addition to the data set. Compared to the size of the data set $s_d = 100000 \cdot 5 \cdot 4$ Bytes $=$ 2000000 Bytes ≈ 2 MBytes this seems too much. The Bitmap algorithm can only be applied if the number of distinct values is small. For a concise performance discussion and some ideas to overcome the space overhead, please see [TEO01] or [EOT03].

```
1  Input:     Dataset D
2  Input:     Bitmap[] BM // as seen in Table
3  Output:    Skyline S // Point by point
4
5  begin BitmapAlgorithm
6
7    for (int i := 1 to D.size()) do
8
9      // finds the index for the bitslice
10     int k := BM[i].determineIndex();
11
12     // get kth bitslice for first dimension
13     // and perform first AND
14     Bitmap A := BM[i].getBitSlice(k, 1);
15     for int j := 2 to dim do
16       A := A & BM[i].getBitSlice(k, j);
17     end;
18
19     // get preceesing bitslice
20     Bitmap B := BM[i].getBitSlice(k-1, 1);
21     for int j := 2 to dim do
22       A := A | BM[i].getBitSlice(k-1, j);
23     end;
24
25     Bitmap C := A & B;
26
27     if (C = 0) then
28       S.insert(D[i]);
29     end;
30
31   end;
32
33 end;
```

Figure 12.1: *Bitmap algorithm*

CHAPTER 13

Partition-Index Algorithm

This algorithm has been originally published in [TEO01]. Some enhancement has been published in [EOT03]. We explain the original Partition-Index algorithm by applying it to our hotel example.

13.1 Algorithm Description

Before we can apply the Partition-Index algorithm to our hotel example we need to normalize the values in each dimension. Explanation why we need to normalize is given later on. For normalization we use the ranks of each dimension value as given in Table 12.1. The dimension values are transformed by substituting the real value by its rank among all values. Table 13.1 shows the result of this transformation.

Name	Short	Price [EUR]	Distance [m]	Price [rank]	Distance [rank]
Hotel Arena	**a**	45	100	4	3
Hotel Aden	**b**	40	200	2	5
Hotel International	c	42	300	3	8
Hotel Aurora	**d**	35	400	1	9
Hotel Majestic Toscanelli	e	50	280	5	7
Hotel Monaco & Quisisana	f	60	150	7	4
Hotel Elpiro	**g**	55	50	6	2
Hotel Marlisapier	h	65	250	8	6
Hotel Al Gambero	**i**	72	40	10	1
Hotel Rex	j	40	500	2	10
Hotel Heron	k	68	100	9	3

Table 13.1: *Hotel example for Partition-Index algorithm*

The Partition-Index algorithm partitions the d-dimensional points in the data set into d lists in a way that point $p = (p_1, p_2, ..., p_d)$ is put into list i if the value of point p in the ith dimension is the minimum value among all dimensions of p, that is, $p_i \leq p_j \ \forall\, 1 \leq i, j \leq d \ \wedge \ i \neq j$. For

instance, point d ("Hotel Aurora") is put into list 1, because its normalized dimension values are $(1,9)$, whereas point i ("Hotel Al Gambero") is put into list 2, since its values are $(10,1)$. The two lists are shown in Table 13.2. The points in the two lists can be indexed by any one-dimensional indexing structure. The authors of [TEO01] used a B+-tree to index the points. An example of such a B+-tree is shown in Figure 13.1.

List 1		List 2	
Point	Minimum distance	Point	Minimum distance
d	1	i	1
b, j	2	g	2
c	3	a, k	3
e	5	f	4
		h	6

Table 13.2: *Example lists for Partition-Index algorithm*

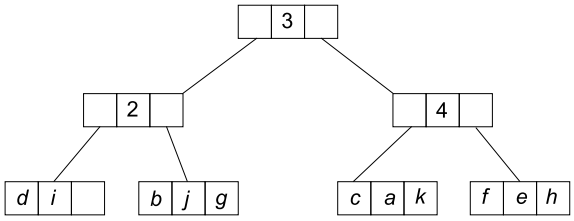


Figure 13.1: *Example B+-tree for Partition-Index algorithm*

A *batch* in the ith list in Table 13.2 consists of points that have the same minimum distance. There is a batch containing points $\{b, j\}$ in list 1 or a batch containing point $\{i\}$ in list 2, for instance. For our example all batches contain a single point except batch number 2 in list 1, this batch contains two points, and batch number 3 in list 2, this batch also contains two points. The algorithm starts with the first batch of each list and handles the one with minimum distance. Since both (first) batches in our example have the same minimum distance, the algorithm starts with the batch from list 1, $\{d\}$. Processing a batch has two steps. First, compute the Skyline inside the batch. The size of a batch is small compared to the number of total points, so the Standard algorithm (Chapter 9) should suffice. Second, among the computed (pre-) Skyline points from the batch the algorithm adds those points to the Skyline that are not dominated by any already found Skyline points. The first batch from list 1, $\{d\}$, contains only one point. So d is added to the Skyline, since it is not dominated by any Skyline point (the Skyline is still empty). The next batch in list 1 has a minimum distance of 2, so the algorithm switches to list 2. Processing batch 1 from list 2 results in another Skyline point, i is the only point in the batch and is not dominated by $S = \{d\}$. Now, batch 2 from list 1 is handle, $\{b,\ j\}$, the pre-Skyline is point b. It is also not dominated by any Skyline point and thus added to the Skyline, $S = \{d,i,b\}$. Batch 2 from list 2, $\{g\}$, is also a Skyline point. Batch 3 from list 1 does not add a Skyline point since c is dominated by Skyline point b. Batch 3 from list 2 adds a point to the Skyline. The pre-Skyline of this batch is point a which is not dominated by any other Skyline point. The Skyline now consists of 5 point, $S = \{d,i,b,g,a\}$. The algorithm terminates since both dimension values of a (4, 3) are smaller than or equal to the minimum distance of the next batches from both lists, that is, batch $\{e\}$ has minimum distance 5 and batch $\{f\}$ has minimum distance 4.

```
1  Input:     Dataset D
2  Input:     B+-tree B
3  Output:    Skyline S
4
5  begin IndexPartitionAlgorithm
6
7    bool search_partition[];
8    float max_value[];
9    float min_value[];
10   Point p[];
11
12   for (int i := 1 to d) do
13     search_partition[i] = true;
14     p[i] = B.getPointMinDimensionValue(i);
15     max_value[i] = p[i].getMaxDimValue();
16     min_value[i] = p[i].getMinDimValue();
17   end;
18
19   float mn = min(max_value);
20   float mx = min(min_value);
21
22   for int i := 1 to d do
23     if(mn < max_value[i]) then
24       search_partition[i] = false;
25     end;
26   end;
27
28   <continued in part 2>
29
30 end;
```

Figure 13.2: *Partition-Index algorithm for Skyline computing, part 1*

Figure 13.2 and Figure 13.3 show some pseudo code for the Partition-Index algorithm. The algorithm contains some functions that need to be explained.

The function *getPointMinDimensionValue* traverses the B+-tree until it finds the point with the smallest value in dimension *i*. Routines *getMaxDimValue* and *getMinDimValue* return the maximal and the minimal value of a point. A simple function, *done* checks the boolean vector *search_partition* if there are still partitions left that need to be processed. To retrieve the next point from the B+-tree the *getNextPoint* function is used. This function retrieves the next point according to the algorithm description given above, that is, the function chooses the correct list and retrieves the next point or retrieves the next point from a batch. Finally, the *computeSkyline* function computes the Skyline of a partition or merges Skylines.

Is this algorithm correct? The authors of [TEO01] give three Lemmata that show that the algorithm works correctly. The first lemma is needed to prune some points, the second lemma says that Skyline points contain minimal values, and the third lemma is necessary to compute

```
1   while(!done(search_partition)) do
2     Skyline s[];
3     Partition P[];
4     int j := 1;
5
6     for (int i := 1 to d) do
7       if(min[i] = mx) then
8         P[j].addPoint(p[i]);
9         p[i] := B.getNextPoint(p[i]);
10
11        while(p[i].getMinDimValue() = mx) do
12          mn := min(mn, p[i].getMaxDimValue());
13          P[j].addPoint(p[i]);
14          p[i] := B.getNextPoint(p[i]);
15        end;
16
17        min[i] := p[i].p[i].getMinDimValue();
18      end;
19    end;
20
21    s[j] = P[j].computeSkyline();
22    S = S.computeSkyline(s[j]);
23    j := j + 1;
24
25    mx := min(min_value);
26
27    for (int i := 1 to d) do
28      if(mn < max_value[i]) then
29        search_partition[i] = false;
30      end;
31    end;
32  end;
```

Figure 13.3: *Partition-Index algorithm for Skyline computing, part 2*

the Skyline partition-wise by looking at partitions in ascending order. All lemmata are proven in [EOT03], but the authors use the max-notation. In order to give the reader an idea, we note the lemmata in min-notation and the ideas for proving them.

Lemma 1: Consider two d-dimensional points $p = (p_1, p_2, ..., p_d)$ and $q = (q_1, q_2, ..., q_d)$. Let $p_{\max} = \max_{j=1}^{d}(p_j)$ and $q_{\min} = \min_{j=1}^{d}(q_j)$. Let $p_{\max}$ occur at dimension $d_{\max}$ and $q_{\min}$ occur at $d_{\min}$. Then if $p_{\max} < q_{\min}$, p dominates q.

Lemma 2: Let D be a data set containing $|D|$ d-dimensional points. We define m as

$$m = \min_{i=1}^{|D|}\left(\min_{j=1}^{d} p_{ij}\right)$$

(m is the minimal value of all dimensions of all points) where p_{ij} corresponds to the value of the jth dimension of the ith point of D. We define M as

$$M = \left\{(p_1, p_2, ..., p_d) | (p_1, p_2, ..., p_d) \in D \wedge \min_{j=1}^{d} p_j = m\right\}$$

(M is set of points of D that have m as their minimum value). Let S_D be the Skyline of D and S_M the Skyline of M. Then $S_M \subseteq S_D$.

Lemma 3: Let D be a data set containing $|D|$ d-dimensional points. Let there be k distinct values in the dimensions of the points in D. Let m_1 denote the minimal value, m_2 the second smallest value, and so on, and finally m_k the maximal value. Moreover, let us split D into k partitions P_1, ... P_k, such that

$$P_i = \left\{(p_1, p_2, ..., p_d) | (p_1, p_2, ..., p_d) \in D \wedge \min_{i=j}^{d} p_j = m_i\right\}$$

Let S_D be the Skyline of D, and S_i the Skyline of P_i. Let us compute S_D by examining partitions in the order P_1, P_2, ..., P_k. Then, when we are examining P_l, we can determine whether points in S_l are in S_D without having to look at $Pl + 1$, ..., P_k.

Lemma 1 is obvious to proof: $p_{\max}$ is the maximum value of all p_j, and $q_{\min}$ is the minimum value of all q_j, since $p_{\max} < q_{\min}$ all values q_j are larger than values p_j. Hence, p dominates q.
For Lemma 2 one has to show that none of the points in $D - M$ dominate a point $p \in S_M$: With m' being the minimal value of all dimensions of all points in $|D - M|$ on can see that $m' > m$. Thus, no point in $|D - M|$ dominates p.
Finally, Lemma 3 is proven by induction over the partitions P_1, ..., P_k.

13.2 Discussion

One advantage of the Partition-Index algorithm can be seen by looking at Table 13.2. At the top of each list Skyline points occur with high probability. Hence, the Partition-Index algorithm can quickly return Skyline points and produce Skyline points at a high rate. But its main advantage is its disadvantage as well. The Skyline points in each list are sorted by extreme values first. This means the Partition-Index algorithm will return extreme Skyline points, that is, points that are extremely good in one dimension, first. This is OK, when computing the complete

Skyline. It does not matter when Skyline points are produced and eventually the Partition-Index algorithm will find all Skyline points, in particular those that do not have extreme values in the dimensions. But returning extreme points first is not very helpful in online scenarios as described in Part III where a fast computed "big picture" is necessary. In such a scenario it is neither helpful to compute the whole Skyline. A comparison between this algorithm and true online algorithms is given in Chapter 17. Another disadvantage of the Partition-Index algorithm is that the lists (and the corresponding index structure) have to be pre-computed for each possible combination of Skyline attributes. Supporting $m - d$-Skyline queries would require to pre-compute all combinations of dimensions, that means, that 2^d combinations would have to be pre-computed. Another disadvantage is seen in the first section of this chapter. The Partition-Index algorithm is not directly applicable to dimensions having different domains. In our example the price and the distance had to transformed to integer values between 1 and 10 in order to make the algorithm work. The list approach expects similar dimension domains. Nevertheless, the Partition-Index algorithm is probably the fastest known algorithm to compute the complete Skyline in a batch / progressive sort of way.

CHAPTER 14

Candidate Skyline Computation - Skyline Pre-filters

In this chapter we present two ideas for reducing the data set so that the subsequent batch Skyline algorithm finds an already pre-aggregated data set for its Skyline computation. First, we show a filter algorithm that uses a sorted data set and, second, we show a filter algorithm that uses a multi-dimensional indexing structure such as the R-tree [Gut84] (and its derivatives) or the UB-tree [Bay97].

Common to both pre-filter algorithms are the following steps:

- Run data set through filter (either sorted or indexing structure). This results in a smaller data set.
- Run smaller data set through Skyline algorithm. This results in the Skyline of the original data set.

Both ideas have previously been published in [Ros01].

14.1 Pre-filtering a Sorted Data Set

14.1.1 Scoring

In [Ros01] this way of pre-filtering the data set is described with the term *Scoring*. By scoring we mean converting a point, that is, a multi-dimensional object, to a one-dimensional object, the *score*. This conversion can be done in different ways. Common to all ways is the use of a monotonic increasing function applied to all dimensions of a point. For example, one could sum up all dimensions of a point and use the resulting value as the score. We define the score of a point p as

$$\text{score}(p) = F_{i=1}^{d}\ p_i$$

with d being the number of dimensions and p_i the value of point p in dimension i ($i = 1..d$). The function F is a monotonic increasing function such as sum or multiplication. The use of a

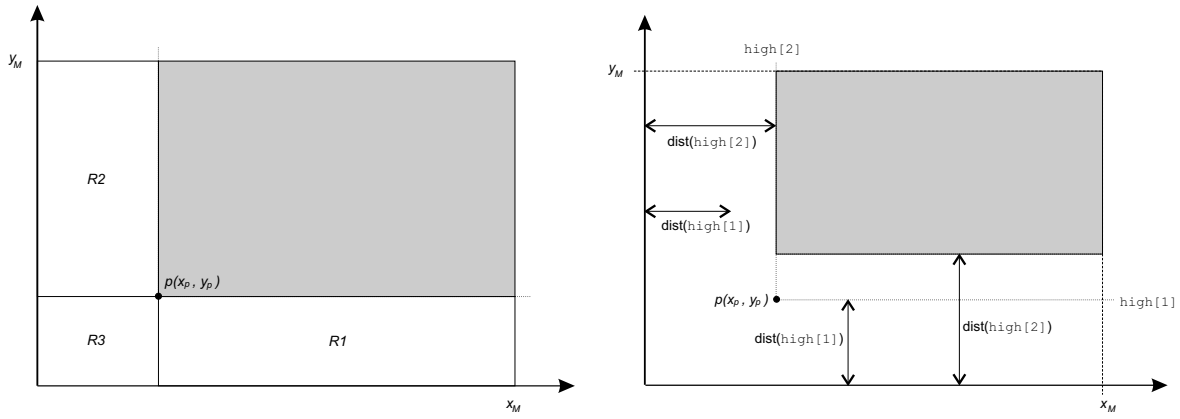
monotonic increasing function is necessary, because the following requirement must hold when using the score for Skyline computation: If point p is greater than point q, that is p is greater in all dimensions than q, the score of point p must also be greater than the score of point q. There are two ways each dimension can be treated by F. One, all dimensions are treated equally by F or, two, function F could put emphasis on certain dimensions. Both ways are feasible for Skyline computation. We choose the first way, that is, not putting an emphasis on certain dimensions.
After the scoring has taken place, the data set is sorted by ascending score. This results in a data set which is most likely to have the Skyline points at the beginning, since small dimension values result in a small score value.
The whole scoring procedure comprehends three steps. These are:

1. computing the score for each point,
2. extracting the maximum value of each dimension for later purpose,
3. and sorting the points by ascending score.

Computing the score and extracting the maxima can be done in one step. For sorting one has to wait until scoring and extracting the maxima are finished. These steps could be done while gathering statistics of the data set which is done regularly by a Database Management System.

14.1.2 Filtering

For computing the Skyline of a sorted data set one needs to first look at an example to understand how the algorithm works. Figure 14.1(a) shows a 2-dimensional scenario. We chose an arbitrary point p for illustration. The purpose of filtering is to determine points that definitely cannot belong to the Skyline. Since our Skyline tries to minimize, these points lie, for a 2-dimensional set of points, in the right-upper rectangle of point p. These lie within the gray shaded rectangle in Figure 14.1(a). Assume p belongs to the Skyline. The minimum x-value of the rectangle is the x-value of p and the minimum y-value of the rectangle is the minimum y-value of p. So, points that still can belong to the Skyline lie within two rectangles, *R1* with the lower point $(x_p, 0)$ and upper point (x_M, y_p) and *R2* with the lower point $(0, y_p)$ and the upper point (x_p, y_M). Rectangle *R3*, with the lower point $(0,0)$ and the upper point (x_p, y_p), contains no points. If *R3* would contain any points p would not be in the Skyline, since p then would be dominated by the points in *R3* (the points in *R3* are better in every dimension than p). Note that even for higher dimensionality we need only two points to describe the "rectangle". These two points must be diagonal within the n-dimensional space. All hyper-planes of the object must be parallel to the axes.
The Skyline, of course, is unknown while filtering. But it is not necessary to know if point p belongs to the Skyline. If we read a point p from the data set we can discard all the points that are dominated by this point, even if the point we read is not in the Skyline. If the point p is not in the Skyline it will be discarded later, but all the points that were dominated by p are also be dominated by the newly found point q that dominates p. This property we described in Chapter 3 as transitivity. In order to get results quickly and not having to read all the points in the data set, the points should be ordered in some way. The simplest way is an ordering that yields a data set that is most likely to have the Skyline points at the beginning. This is accomplished by the scoring and sorting procedure described in the previous section.



(a) Algorithm introduction (b) Algorithm explanation

Figure 14.1: *Sorted Skyline computation*

The filter algorithm basically scans through the whole data set, that is reading the points including the score, one by one. It determines by the means of the score if a point can belong to the Skyline or not. If the score of the point last read is greater than some minimum score, which is explained later, then the point cannot belong to the Skyline and is discarded. Since all subsequent points have scores greater than the score of the point last read, all subsequent points can be discarded. The algorithm terminates with a superset of the Skyline, the candidate Skyline. The pseudo-code of the algorithm is shown in Figure 14.2. Also the function calculating the score is shown (line 36).
The function `maxOfVector` determines the maximum value of a vector, here the maximum of the array high[]. The algorithm has two data structures which are used to determine whether the point last read could belong to the Skyline or not. The first one, alltimehigh[], is computed at the very beginning. It contains the highest possible score values without the dimension i. For a d-dimensional data set this is

$$\text{alltimehigh}[i] = \text{calcScore}(\max(\text{dim}_1), ..., \max(\text{dim}_{i-1}), \max(\text{dim}_{i+1}), ..., \max(\text{dim}_d)).$$

with dim_i being the value in the ith dimensions. The values of alltimehigh[] do not change during the filtering, the maximum value of each dimension was extracted during pre-processing (see previous section).
The second data structure is called high[]. It holds the score of the ith dimension of the point and alltimehigh[i], that is,

$$\text{high}[i] = \text{calcScore}(\text{alltimehigh}[i], \text{p}[i]).$$

The score, which is used to determine if a point can belong to the Skyline or not, is the maximum of high[i]. We call it abortScore. If the score of a point becomes larger than the abortScore, the filtering is done. See Figure 14.1(b). This maximum corresponds to the minimal possible rectangle, for 2-dimensional points, that can be neglected for Skyline computation. The maximum of high[] has to be taken, because otherwise one would discard points that could still belong to the Skyline. For example, in Figure 14.1(b) if not the maximum of high[], that is high[2], but the smaller value high[1] would be taken instead, then one would discard points that lie between the y-axis and the high[2]-line. These points can still be part of the Skyline.
For a proof of correctness of this method, results of how effective the filter can reduce the number of points that still have to be check by a Skyline algorithm, and some variations of the described filtering method we refer the reader to [Ros01].

```
1  Input:     sorted Dataset D  // sorted data set incl. statistics
2  Input:     int dim           // number of dimensions
3  Input:     float[] alltimehigh // pre-calc. from statistics
4  Output:    candidate Skyline S
5
6  begin SortedSkylineFilter
7
8    float abortScore := 1000000.0;
9    float[] high;
10
11   for (int i := 1 to D.size()) do
12     Point p := D[i];
13
14     if (p.score <= abortScore) then
15
16       for (int j := 1 to dim) do
17         high[j] := calcScore(alltimehigh[j], p[j]);
18       end;
19
20       if (abortScore > maxOfVector(high))
21         abortScore := maxOfVector(high);
22       end;
23
24     else
25       // filtering is done
26       exit(0);
27     end;
28
29     S.insert(p);
30
31   end;
32
33 end;
34
35
36 function calcScore(float a, float b)
37   return a {+, *} b;
38 end;
```

Figure 14.2: *Pseudo-code for sorted Skyline pre-filtering*

14.2 Pre-filtering with a Multi-dimensional Indexing Structure

Instead of representing the data set for Skyline computation by a flat file as it is true for all previously described algorithms, the data set can also be represented by an indexing structure. As a multi-dimensional indexing structure we use an R*-tree [BKSS90]. All points are read into the R*-tree. Instead of having row identifiers in the leaf node of the R*-tree the points directly lie within the leaf nodes. So the R*-tree represents the complete data set.

A naive way of Skyline pre-filtering would be

- Traverse to each leaf node (only leaf nodes carry points) the R*-tree,
- calculate the candidate Skyline of each leaf node and
- concatenate leaf candidate Skylines to the overall candidate Skyline of the complete data set (represented by the R*-tree).

The resulting candidate Skyline must be run through a normal Skyline algorithm to determine the real Skyline. This naive R*-tree algorithm works astounding well as is can be seen in [Ros01]. It serves also as the first step to a smarter way of using the R*-tree in terms of Skyline computation, the Branch-and-Bound Skyline pre-filter. The Branch-and-Bound Skyline pre-filter algorithm is an optimization of the naive algorithm. Instead of reading all leaf nodes (if all leaf nodes are visited all intermediate nodes have also be visited) and computing the candidate Skyline out of every leaf node, the Branch-and-Bound algorithm visits only those nodes which definitely contain points that could be part of the Skyline according to some intermediate condition. This intermediate condition is derived from an already computed candidate Skyline. For example, if we read the points of Node 1 in Figure 14.3 first and compute a candidate Skyline, then we do not have to access the remaining nodes. This is because any point of the candidate Skyline computed from Node 1 dominates every point contained in nodes 2 through 4. The rectangles correspond to the minimum bounding rectangles (MBR) of each node.

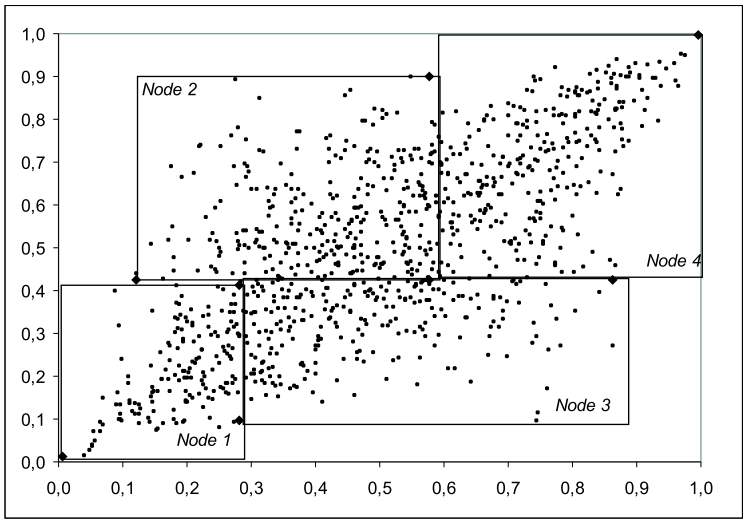


Figure 14.3: *Explanation for the Branch-and-Bound R*-Tree filter algorithm*

In other words, if the lower left corner of an MBR is dominated by one or more candidate Skyline points then this leaf node cannot contain any Skyline points. Therefore, this node (for leaf node) or this subtree (for an internal node) can be discarded. Note that even for higher dimensions only two points are needed to describe the MBR. These points must be diagonal points, the lower point and the upper point of the MBR.
The Branch-and-Bound Skyline pre-filter algorithm works like following:

- Traverse the R*-Tree in a depth-first way. In every internal node take the first entry and descend down to the leaf (think of the leftmost leaf).

- Calculate the Skyline of this leaf. This candidate Skyline will be the *pruning condition* for the following subtrees.
- Test the remaining subtrees if they can contain Skyline points: Read the MBR of the subtree in turn and check if any point in the candidate Skyline dominates the lower point of the MBR. If yes, ignore the subtree, it contains no points that could participate in the Skyline; if no, traverse the subtree and start the test again.
- If in leaf node, compute the Skyline of this leaf node, concatenate it to the already existing candidate Skyline.

The selection of the first subtree and eventually the first leaf, which gives the first pruning condition is vital for this algorithm. For example, if one would find the one Skyline point of a 2-dimensional correlated data set in the first leaf, the algorithm would terminate very quickly since all subsequent test would prune all the remaining subtrees.
For brevity reason we do not show the results of the Branch-and-Bound Skyline pre-filter algorithm here. The interested reader is forwarded to [Ros01]. The use of another multi-dimensional indexing structure, the UB-tree [Bay97], is also not discussed here.

CHAPTER 15

Summary: Skyline Computation

In Chapters 8 through 14 we gave an introduction to 5 different Skyline algorithms that - in our terms - qualify to be either a batch or a progressive algorithms and showed an idea how to pre-filter the Skyline. We also laid out our classification for batch and progressive algorithms and discussed why we classified each algorithm into the particular group.

All Skyline algorithms have their particular advantages and disadvantages, the Standard algorithm, for example, is good for small amounts of data points but has a major drawback when it comes to larger data sets. But its particular simplicity of implementation makes this algorithm worth considering when it comes to choosing a Skyline algorithm. All other algorithms are more complex to implement and more sophisticated when computing the Skyline.

The decision what Skyline algorithm to use requires a thorough analysis of the situation the algorithm should be used for and a thorough consideration of the pros and cons of each algorithm.

Since all these algorithms and ideas have been published previously this part should be seen as a concise compendium on Skyline algorithms. For a thorough discussion the reader is forwarded to the particular paper given at the beginning of each section.

Part III

Online Skyline Computation

CHAPTER 16

Scenario and Requirements

16.1 Scenario

The previous chapters dealt with mostly detached Skyline computations, meaning that some query on the database wants to compute the Skyline, but the user who submitted the query does not care when the query is done. The user "picks up" the result sometime later.
Now the scenario changes: Consider our hotel example (Chapter 7) in an online booking system for hotels. The user submits the online query. With our previously described algorithms the user waits a long time before any result is presented. Then the user gets the whole Skyline, meaning all hotels in Lido di Jesolo with price and distance to the beach. But does the user need to see all the results before he or she decides that, for example, the distance to the beach is more important to him or her than the price? In most cases it is sufficient to give the user a "big picture" of the Skyline, meaning some points having a low price and a high distance, some points having a high price and a low distance, and some points in the middle. Now the user can decide what is more important to him or her since he or she has a first impression what the results will look like. Assuming the results are presented in a graphical user interface, the user can click on a region which is interesting to him or her, telling the computer, that the user's preferences lie in the area he or she just clicked on. The computer should now return results in that area without changing the overall Skyline query, that means, not suddenly weighing one dimension more than the other. Simply the order the Skyline points are returned should change. The generality of the Skyline points returned so far and all future Skyline points should not be lost. Furthermore, the user does not want to wait long for first results, he or she should be able to look a the graphical user interface and see points appearing incrementally starting right after issuing the query.

For the previously described scenario the batch and progressive algorithms depicted in Part II are only partly applicable or not applicable at all. We will see a comparison between the fastest algorithm from Part II and the Nearest Neighbor algorithm from this part in Section 17. The comparison focuses on the quality of the returned results stressing the things we said previously about the "big picture".
Before we come to describing two variations of the Nearest Neighbor algorithm we set up some

requirements a Skyline algorithm has to fulfill to be called an online algorithm.

16.2 Requirements for an Online Algorithm

The previous section has already shown some demands we have in order to call an Skyline algorithm an online Skyline algorithm. In this section, we will highlight some additional requirements for an online algorithm. Adopting the criteria set from the Control project [HAC+99], we demand the following properties from an online Skyline algorithm:

1. The *first results* should be returned *almost instantaneously*. It should be possible to give guarantees that constrain the running time to produce the first few results. This is particularly important for an only Skyline algorithms since patience is probably not the user's most distinctive talent.

2. The algorithm should produce *more and more results* the longer it runs. Eventually, if given enough time, the algorithm should produce the full Skyline. The user looks at the initial results and while he or she watches more and more points appear on the screen.

3. The algorithm should only return points which are part of the Skyline. In other words, *false-positives are not allowed*. For instance, the algorithm should not return good hotels at the beginning and then replace these good hotels with better hotels, that is, Skyline hotels. Most users will be interested in stable answers: They do not want answers to appear and then disappear on their screen.

4. The algorithm should be *fair*. In other words, the algorithm should not favor points that are particularly good in one dimension, instead it should continuously compute Skyline points from the whole range. This is an important property to guarantee the "big picture".

5. The user should be in *control of the computation*. In other words, it should be possible for the user to make preferences while the algorithm is running. For example, if the algorithm has run for a while and returned mostly cheap hotels, then the user should be given the opportunity to specify that he or she is more interested in hotels which are near the beach. The algorithm should adapt itself and subsequently look for interesting hotels which are close to the beach. Using a graphical user interface, the user should be able to click on the screen and the algorithm will return the next points of the Skyline which are near the point that the user has clicked on. The overall Skyline query should not be altered.

6. The algorithm should be universal with respect to the type of Skyline queries and data sets. It should also be based on standard technology, that is, indices, and it should be easy to integrate the algorithm into existing database systems. For a given data set, for example, hotels one index should be enough to consider all dimensions that a user might find *interesting*. The algorithm should also be universal with respect to:

   - dynamic data: independent of changes to the data set
   - scalability: both dimensionality and data size
   - data value distribution

- type of Skyline query: if the data set has d dimensions, it should be possible to ask for Skyline queries that involve only m dimensions ($m < d$); the other dimensions can be part of predicates in the WHERE clause of the query. We call such queries m-d-Skyline queries. We believe that such m-d-Skyline queries will be very frequent in practice because not every user is interested in all aspects.

With these requirements in mind we now can describe the Nearest Neighbor online algorithm. The first algorithm in Chapter 17 introduces the ideas and relations that lie behind Nearest Neighbor computation and Skyline computation. We describe the original Nearest Neighbor Skyline algorithm. This algorithm was published in [KRR02]. Chapter 18 focuses on an improved version of the original algorithm, published in [PTFS03]. Unfortunately, the authors of [PTFS03] call their algorithm "progressive". We used "progressive" as an enhancement of batch algorithms. However, their algorithm qualifies as an online algorithm, as we will see.

CHAPTER 17

Nearest Neighbor Algorithm

17.1 Relationship Between Nearest Neighbors and the Skyline

What is Nearest Neighbor computing? In a nutshell - the Nearest Neighbor is the point closest to another point. Considering a set of point D, a point $n \in D$ is the Nearest Neighbor of a point $q \in D$ if no other point is closer to q than n according to some distance function f. Nearest Neighbor computation has been subject of various papers, for example, [RKV95] or [BBKK97].

The relationship between Skyline points and Nearest Neighbors and the resulting algorithm has first been described in [Ros01].

Before we start describing the algorithm, let us look at two basic observations that constitute the foundation of our Nearest Neighbor algorithm for computing the Skyline of a data set. The first observation describes a direct relationship between a Nearest Neighbor and a Skyline point, the second observation deals with the transitivity of the first observation. For a graphical representation take a look at Figure 17.1, the table displaying names and values in each dimensions (Table 7.1) is left out here for brevity reasons. Both observations use the following parameters:

- n is the Nearest Neighbor.
- f is any monotonic distance function, for example the Euclidean distance.
- D is a two-dimensional data set containing positive floating point values, that is each point $p \in D$ has an x and a y value, denoted as p_x and p_y, with $p_x \geq 0.0$ and $p_y \geq 0.0$.
- q is the query point the Nearest Neighbor is computed to. In our case the query point is always the origin, $q = (0,0)$.

A Nearest Neighbor n of a point q is then defined as

$$\{n \in D \mid \forall\, x \in D,\, n \neq x \,:\, f(n,q) \leq f(x,q)\}$$

We formulate our two observations as lemmata:

Lemma 1: Let f, D and q be defined as given above, and let n be the Nearest Neighbor of q. Then, n is in the Skyline of D.

Proof (Lemma 1): This lemma can be proven by contradiction. Let $n = (n_x, n_y)$. Furthermore, we assume that n is not part of the Skyline. As a result, there must be a point $b = (b_x, b_y)$ that dominates n, otherwise n would be a Skyline point. In other words, $b_x < n_x$ and $b_y \leq n_y$ or $b_x \leq n_x$ and $b_y < n_y$. Under these circumstances, however, $f(b,q)$ must be smaller than $f(n,q)$ because f is a monotonic distance function. This is a contradiction to n being the Nearest Neighbor of q. As a result, n must be part of the Skyline. □

Going back to Figure 17.1 it can easily be seen that "Hotel Arena" (45 EUR, 100 m) is the Nearest Neighbor to the origin and is part of the Skyline. Obviously this lemma also holds for higher-dimensional data sets. Simply extend above given proof for higher dimensions.

The other observation is an extension of the first. It says that any Nearest Neighbor n to the origin q within a certain region that is constrained by the origin and an upper right corner point m is part of the Skyline of the subset D_m of D and, furthermore, is part of the Skyline of the whole data set D. A region is given by two diagonal points, a lower left corner point and an upper right corner point. In our case the lower left point is always the origin. So it suffices to specify a region by its upper right point.

Lemma 2: Let f, D and q be defined as previously, let $m = (m_x, m_y)$ be a point defining the region $(0,0;m_x,m_y)$ and let D_m be a subset of D such that D_m contains all points of D with $x < m_x$ and $y < m_y$. Let $n \in D_m$ be a Nearest Neighbor of the point q according to f. Then, n is in the Skyline of D (naturally, n is also in the Skyline of D_m, using Lemma 1).

Proof (Lemma 2): The proof of this lemma is almost identical with the one of Lemma 1. The only additional step is to prove that the (imaginary) point b must also be in the given region, which is trivial based on the transitivity of the $<$-relation: If n is not the Nearest Neighbor of q then $b_x < n_x$ and $b_y \leq n_y$ or $b_x \leq n_x$ and $b_y < n_y$ (1). Since $n \in D_m$, $n_x \leq m_x$ and $n_y \leq m_y$. With (1) it follows that $b_x < n_x \leq n_x$ and $b_y \leq n_y \leq m_y$ or $b_x \leq n_x \leq m_x$ and $b_y < n_y \leq m_y$. Hence, $b \in D_m$. □

Also this lemma can be extended for higher-dimensional data sets. Essentially, the second lemma means that if we partition the data set with region described above, then it is sufficient to look for Skyline points using Nearest Neighbor search in each region separately. These two lemmata lead to a divide & conquer algorithm using Nearest Neighbor search.

17.2 Nearest Neighbor Algorithm for 2-dimensional Skylines

First, let us take a look at an example. This example is based on Figure 17.1 but contains more points to illustrate how the algorithm works. This will be the first part of this section. The second part of this section contains an in-depth description of the Nearest Neighbor algorithm and some pseudo-code of the algorithm.

17.2.1 Example

The following example illustrates the Nearest Neighbor algorithm for 2-dimensional Skyline queries. We use the hotel example from Figure 7.1 but add a few more points not shown in the table.

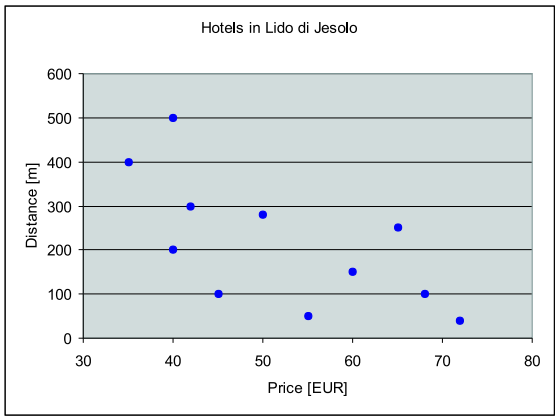


Figure 17.1: *Hotel example for Nearest Neighbor algorithm*

We describe the first 6 steps of the algorithm. These steps are also depicted in Figure 17.2. As distance function we use $f = \text{price} + \text{distance}$.

- **Step 1 and 2**: The algorithm starts off with a Nearest Neighbor search within the whole data set, that is, not restricted by any region. If a Nearest Neighbor is found, $n_0 = (x_{n_0}, y_{n_0})$, the data set is partitioned into three regions. The first region has the coordinates $\text{region}_1 = (0,0;\infty,y_{n_0})$. It is positioned "along" the x-axis. The second region has the coordinates $\text{region}_2 = (0,0;x_{n_0},\infty)$ and is positioned "along" the y-axis. Both regions are labeled in Figure 17.2(a) and (b). The third region is not labeled. It has the coordinates $(x_{n_0},y_{n_0};\infty,\infty)$. This region contains all points that are dominated by the just found Nearest Neighbor n_0. This region is of no interest anymore, the points can be discarded.

- **Step 3 and 4**: After the first and second step the algorithm only needs to investigate previously defined regions (region_1 and region_2). The algorithm continues in the same ways as for step 1. A new Nearest Neighbor search is issued but now the Nearest Neighbor search is limited by the boundaries of region_1. The newly found Nearest Neighbor, $n_1 = (x_{n_1}, y_{n_1})$, further divides region_1 into $\text{region}_{1.1}$ and $\text{region}_{1.2}$ with the coordinates $\text{region}_{1.1} = (0,0;\infty,y_{n_1})$ and $\text{region}_{1.2} = (0,0;x_{n_1},x_{n_0})$. Points not within these regions are discarded. Note: The four points along the y-axis are not discarded since the are located in region_2 which is dealt with separately. Steps 2 and 3 are depicted in Figure 17.2(c) and (d).

- **Step 5 and 6**: Now the algorithm deals with region_2. A Nearest Neighbor search bounded by region 2 is issued. With the Nearest Neighbor found region_2 is divided into $\text{region}_{2.1}$ and $\text{region}_{2.2}$ with coordinates $\text{region}_{2.1} = (0,0;x_{n_0},y_{n_1})$ and $\text{region}_{2.2} = (0,0;x_{n_0},\infty)$. Points not in these two regions are again discarded.

- **Further steps**: The algorithm continues with all found regions that have not been searched for a Nearest Neighbor, that are, $\text{region}_{1.1}$, $\text{region}_{1.2}$, $\text{region}_{2.1}$ and $\text{region}_{2.2}$. For $\text{region}_{1.1}$

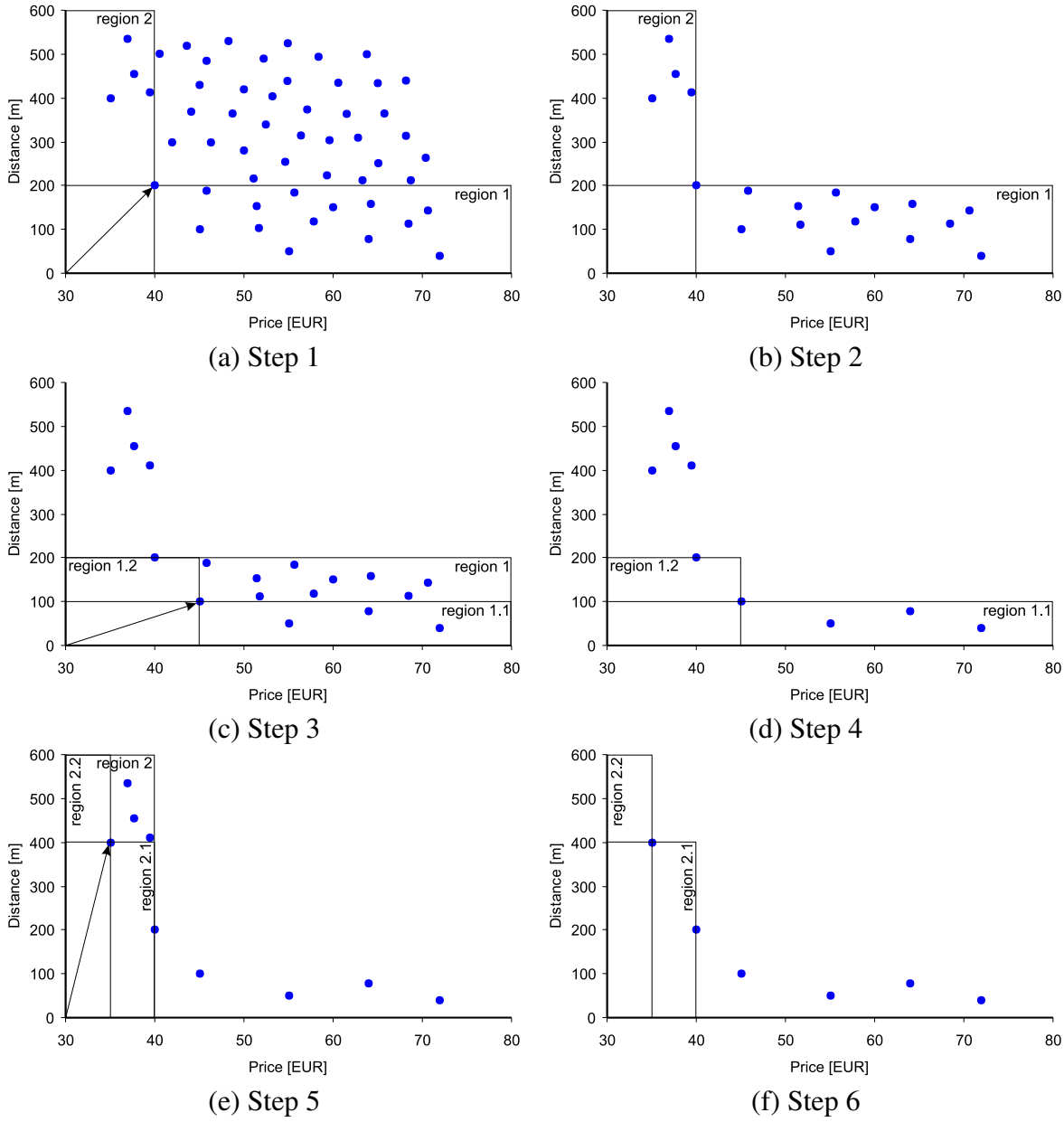
the Nearest Neighbor search and partitioning steps continue. For all other regions the Nearest Neighbor search does not return any points, the regions are empty. So for those regions no partitioning is done and the algorithm terminates for those regions. For $region_{1.1}$ the algorithm will do two more Nearest Neighbor searches and partitioning steps. Then all regions that are produced will be empty and the algorithm terminates completely having found all Skyline points for the data set.

There are some additional facts that should be mentioned here. First, the Nearest Neighbor dividing a region is not part of the newly produced region. If it would be part of a produced region, our algorithm would find this Nearest Neighbor over and over again, not making any progress at all. Second, the order in which regions are processed does not matter at all. This can be easily proven by Lemma 1, any Nearest Neighbor to the origin is a Skyline point. Here we can see the possible user interaction, the user clicks on the screen an the algorithm chooses the region closest to the point the user has clicked on and starts returning Skyline points waning from this region.

17.2.2 Algorithm Description

For the algorithmic description, please take a look at Figure 17.3, the pseudo-code. We assume the availability of a function that computes the Nearest Neighbor to the origin with certain boundaries. When using a multi-dimensional indexing structure such as an R*-tree [BKSS90] such a function can easily be implemented by slightly changing the Branch-and-Bound Nearest Neighbor algorithm by [RKV95]. The B&B Nearest Neighbor algorithm eliminates branches of the R*-tree which cannot contain the Nearest Neighbor since their distance to the query point (the one the Nearest Neighbor is computed to) is too high. We additionally eliminate those branches of the R*-tree that are "out of bounds", that is, branches where the corresponding region (or MBR, that is minimum bounding rectangle) lies out of the given boundaries. Conflicts are solved by accessing branches that are partly out of bounds, possibly returning a Nearest Neighbor and then checking if the Nearest Neighbor lies within the given boundaries. If this is not the case, the branch is eliminated and the Nearest Neighbor search continues regularly with the next branch. These are only a few adjustments to the original Branch-and-Bound Nearest Neighbor algorithm by [RKV95]. In the following, when we speak about Nearest Neighbor algorithm, we mean the Nearest Neighbor algorithm with boundary restrictions.

We introduce a *ToDoList*. The ToDoList stores regions that still have to be processed, that is, regions that were produced by previous steps but have not been searched for Nearest Neighbors yet. At the beginning the ToDoList contains one region, the whole data set without restriction (line 10). Note: Since the regions always have the origin as lower left point and since the edges of the regions are parallel to the x- and y-axis, we only need one point to fully describe a region, that is the upper right point. The algorithm runs until the ToDoList is empty (lines 12). Each step of the algorithm takes another region from the ToDoList. The function *nextMBR* also deletes the region from the ToDoList since after processing the region is not needed anymore. The *boundedNNsearch* gets the query point (in our case always the origin), the data set D, the boundaries of the region (represented by the upper right point) and a monotonic distance function f for the Nearest Neighbor computation (in our case the Euclidean distance). It performs the Nearest Neighbor search on the R-tree R. If the Nearest Neighbor search returns a point this point is used to produce new regions according to the previous section. This is done by the



(a) Step 1 (b) Step 2

(c) Step 3 (d) Step 4

(e) Step 5 (f) Step 6

Figure 17.2: *Nearest Neighbor algorithm, first 6 steps*

function *setMBR*. In each step the Nearest Neighbor that has been found is added to the Skyline and output to the user.

The algorithm employs an R*-tree by [BKSS90] and uses a slightly changed Nearest Neighbor algorithm by [RKV95]. However, other multi-dimensional indexing structures, such as the UB-tree [Mar99], can also be used. In general, any indexing structure can be used that bears the possibility of searching for Nearest Neighbors. In order not to violate the online requirements from Chapter 16, one must take care that the Nearest Neighbor algorithm used can return points almost instantaneously or has a fixed (time) constraint when first Nearest Neighbors are returned.

```
1  Input:    R–Tree R // Dataset D represented by index structure
2  Input:    function f // Distance function
3  Output:   Skyline S // Point by point
4
5  begin NNAlgorithm
6
7    Todolist T; // stores MBRs for bounded NN computation
8    Point qp := (0, 0); // point where the NN is computed to
9
10   T.addMBR((inf, inf)); // add "whole universe" as search region
11
12   while(!T.empty()) do
13
14     Point m := T.nextMBR(); // lower bound of MBR is always (0, 0)
15
16     if(R.boundedNNsearch(qp, D, m, f) != NULL) then
17       Point n := R.boundedNNsearch(qp, D, m, f);
18       T.setMBR(n);
19       S.add(n); // or output Point n
20     end;
21
22   end;
23
24 end;
```

Figure 17.3: *Nearest Neighbor algorithm for 2-dimensional Skylines*

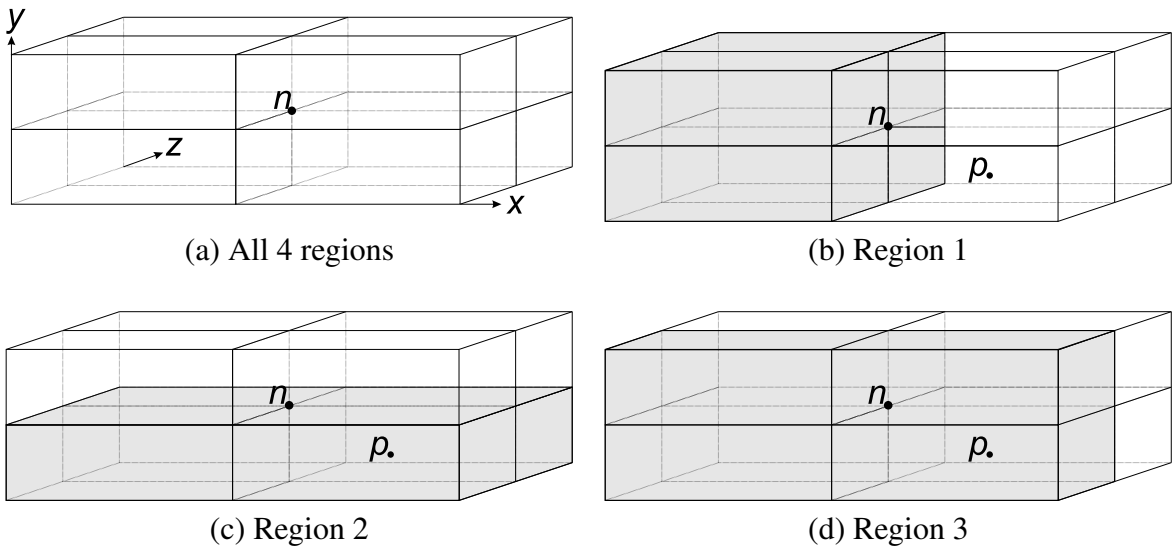
17.3 Nearest Neighbor Algorithm for d-dimensional Skylines

As we have seen in the previous section, the Nearest Neighbor algorithm for 2-dimensional Skyline problems works fine. For 3- and higher-dimensional Skyline problems the algorithm needs to be slightly adapted. These adaptions do not change the basic character of the Nearest Neighbor algorithm. In particular, they do not violate the online requirements from hapter 16. They arise from particularities only observable with more than two dimensions. Those particularities, namely the chance to encounter the same Skyline point more than once and why this can happen, are dealt with in the first part of this section. The second part of this section discusses different possibilities of dealing with those duplicates. Also we need to explain if and how the two basic observations can be extrapolated for d- dimensional Skyline problems in order to apply our Nearest Neighbor algorithms for those kind of Skyline problems.

17.3.1 Particularities

Before we start talking about specialties of the d-dimensional Skyline algorithm we should first talk about the applicability of the 2-dimensional Skyline algorithm of Figure 17.3 for higher dimensions. Figure 17.4 shows a 3-dimensional data space and a Nearest Neighbor $n = (x_n, y_n, z_n)$. Following the ideas of the previous section the data space can be partitioned into four regions. The regions have the following coordinates $\text{region}_1 = (0,0,0; x_n, \infty, \infty)$,

region$_2 = (0,0,0;\infty,y_n,\infty)$ and region$_3 = (0,0,0;\infty,\infty,z_n)$. Region 1 to 3 have to be further processed by our algorithm. The fourth region with the coordinates $(x_n,y_n,z_n;\infty,\infty,\infty)$ does not need to be considered any further since it only contains points that are dominated by the Nearest Neighbor n. Lemmata 1 and 2 can be applied. Both proofs are almost identical, just use 3- instead of 2- dimensional points and regions. That means that n can be given to the user as a Skyline point and region 1 to 3 can be processed separately for further Skyline points.



(a) All 4 regions (b) Region 1

(c) Region 2 (d) Region 3

Figure 17.4: *3-dimensional regions for Skyline computation*

Moreover, lemmata 1 and 2 can be generalized to d dimensional Skyline queries with $d > 2$ leaving us with the benefit that the algorithm of Figure 17.3 is also applicable for d dimensions if it is adjusted to work with $d+1$ regions after each partitioning step instead of only 3 regions as in the 2-dimensional case. The ToDoList grows by d regions in each step, the region which contains dominated points is, of course, not added to the ToDoList, hence, only d regions are added in each step. Table 17.1 shows the growth of the ToDoList for some dimensions.

Step	Dimension			
	2	3	5	10
Step 1	1	1	1	1
Step 5	5	9	17	37
Step 10	10	19	37	82
Step 20	20	39	77	172
Step 30	30	59	117	262
Step 40	40	79	157	352
Step 50	50	99	197	442
Step 100	100	199	397	892

Table 17.1: *Growth of ToDoList*

As it can be seen, the ToDoList grows somewhat fast for higher dimensionality. However, we expect the ToDoList to fit into main memory. In the case when it does not fit into main memory anymore, parts of it can be swapped out to disk and later, when needed, reloaded into main memory.

Now we have to talk about the particularities of higher-dimensional Skyline queries. For that, we should go back to Figure 17.4. We can observe that the regions in 3-dimensional (and higher-dimensional) space overlap in contrast to a 2-dimensional space. They overlap in such a way that the same Skyline points can be found twice or even more often. Take a look at point p in Figure 17.4. Say, that point p is a Skyline point, that means that p is not dominated by n because it is better than n in the y and z dimensions. As it can be seen, point p can be found when processing region_2 or region_3. Such duplicates could not occur with the Nearest Neighbor algorithm in 2-dimensional space. Duplicates seem to be a subtlety of high-dimensional data sets and there are many ways to carry out duplicate elimination, for example, as described in [BD83] or [Lar99]. However, as we will see, such duplicates can influence the performance of the Nearest Neighbor algorithm severely. In the next section, we will present alternative ways to extend the Nearest Neighbor algorithm in order to deal with such duplicates.

17.3.2 Duplicate Treatment

In this section we will describe how duplicates as they occur in the Nearest Neighbor Skyline computation in high-dimensional environments can be either eliminated or avoided. Each technique has its advantages and trade offs. We will also study these. The end of this section also gives a performance comparison of the different techniques.

Laisser-faire Method

The first technique is called Laisser-faire. This approach eliminates duplicates in a post-processing step. Each Nearest Neighbor found is checked against a main memory hash table. If the current Nearest Neighbor is found in the hash table, it is a duplicate and thus it is not output to the user. Nevertheless, the region the Nearest Neighbor is found in has to be subdivided according to the Nearest Neighbor (the duplicate) and the emerging regions are put into the ToDoList for further inspection. If the Nearest Neighbor is not found in the hash table it is a new Skyline point. It is then recorded in the hash table and output to the user.
The approach with a main memory hash table is simple to implement and sufficient to eliminate duplicates [Lar99]. If a main memory hash table is not usable any more, that is, the hash table grows too much, an index on second storage, for example, a B-tree [Com79], can be used. One disadvantage of this algorithm is that this approach results in a great deal of wasted work. Using this approach, it is possible that the Nearest algorithm does not make any or only little progress because it finds the same Skyline point several times. In the worst case, a point can be found $d-1$ times.
There is another disadvantage. According to the original Skyline definition in [BKS01], duplicates that occur in the input data set, so-called real duplicates, are eliminated as well by this approach. Those duplicates, however, should be part of the Skyline. In order to overcome this shortcoming, we suggest the use of a unique identifier for each point and take this identifier into account when eliminating duplicates.

Propagation Method

Instead of eliminating duplicates after they occur, it is possible to prevent duplicates *before* the occur. A Nearest Neighbor can be found again if it is contained in another region in the ToDoList. So whenever the Nearest Neighbor algorithm finds a Skyline point we scan the

whole ToDoList. If the Nearest Neighbor is contained in one ore more regions, these regions are subdivided by the Nearest Neighbor and then deleted from the ToDoList. Thus, we avoid the Nearest Neighbor search for this region. This is correct since only points that are dominated by a Skyline point are discarded by this operation. It is called the Propagation approach. The big advantage of this approach is that it completely avoids wasted work to find duplicates. However, the big disadvantage is that scanning the whole ToDoList for every Nearest Neighbor found implies significant overhead. This overhead can be partly remedied when using a main memory indexing structure for indexing the regions in the ToDoList. Instead of scanning the ToDoList, an index access quickly finds regions that can be eliminated.

Merge Method

We now turn to a technique that can be used to improve both the Laisser-faire and Propagate approaches. The idea is to merge (or even eliminate) regions of the ToDoList under certain circumstances. The basic idea of merging is quite simple. Assume that two regions $a = (a_1, a_2, \ldots, a_d)$ and $b = (b_1, b_2, \ldots, b_d)$, d being the number of dimensions, are in the ToDoList. As said before, we denote only the upper right point for a region. Essentially, this means that we still need to look in both of these areas for more Skyline points. Now, we can merge these two regions into a single region: $a \oplus b = (\max(a_1, b_1), \max(a_2, b_2), \ldots, \max(a_d, b_d))$. Region $a \oplus b$ supersedes both regions; thus, we are sure not to miss any points if we investigate $a \oplus b$ only. A particular situation arises, if a supersedes b; that is, if $a_1 \geq b_1$, $a_2 \geq b_2$, $\ldots$, and $a_d \geq b_d$. In this particular situation, $a = a \oplus b$ and thus b can be simply discarded from the ToDoList. In this situation, b can be discarded from the ToDoList even if a has already been processed and is not part of the ToDoList anymore. Merging reduces the size of the ToDoList. On the negative side, merging increases the size of the regions. Furthermore, finding good candidates to merge can become expensive. In addition, eliminating regions which are superseded by other regions involves remembering regions after they have been processed. Therefore, merging must be employed with care. We propose the following heuristics to make use of merging:

- Propagate: For the propagate approach, we propose making no use of merging and only consider the special case in which a region of the ToDoList can be discarded because it is superseded by another region.
- Laisser-faire: For the laisser-faire approach, we propose to make use of merging whenever a duplicate is detected. That is, we only merge regions if they were derived from the same Skyline points. At the same time, we must be careful not to merge a region with one of its ancestor regions - this restriction is necessary in order to guarantee termination of the algorithm. If a region was merged with its ancestor region the algorithm would be trapped in an endless loop since the same regions would be produced over and over again.

Fine-grained Partitioning

Another option to avoid duplicates is to partition the regions in a fine-grained way so that they do not overlap. However, overlap-free regions mean that not only d regions have to be further processed but $2^d - 2$ regions (see Figure 17.4). Table 17.2 shows the growth of the ToDoList (please see also Table 17.1 for comparison).
This exponential increase of regions is a major drawback of this strategy. In Figure 17.4, for instance, we could partition into 8 non-overlapping regions of which 6 regions would be relevant for further processing. Implementing this approach, however, results in a sharp growth of

Step	Dimension			
	2	3	5	10
Step 1	1	1	1	1
Step 5	5	21	117	4085
Step 10	10	46	262	9190
Step 20	20	96	552	19400
Step 30	30	146	842	29610
Step 40	40	196	1132	39820
Step 50	50	246	1422	50030
Step 100	100	496	2872	101080

Table 17.2: *Growth of ToDoList with fine-grained partitioning*

the number of regions in the ToDoList. Furthermore, this approach involves a complex post-filtering step in order to determine points of a region which are dominated by points of another region. Therefore, we did not pursue this approach any further.

Hybrid Method

Finally, we propose a hybrid technique to deal with duplicates. As mentioned above, merging can be combined with both the laisser-faire and the propagate approach. Another option would be to start and propagate duplicates until the ToDoList has reached a certain size. Then the algorithm switches to laisser-faire because propagation might be too expensive for a larger ToDoList.

17.4 Online Algorithm?

This section recalls the requirements set forth in Chapter 16. We want to discuss if the previously described algorithm fulfills these requirements. Only if it fulfills all of these requirements can we call it an online algorithm. Let us take a look at each requirement:

1. First few results: With the help of a multi-dimensional index structure such as an R-tree, Nearest Neighbor search is a cheap operation so that the Nearest Neighbor algorithm returns the first results almost instantaneously. Even more, not only the first result can produced almost instantaneously but the first few results can be returned in less than a second even for high-dimensional data sets (we will see this in a later section). Due to the curse of dimensionality, Nearest Neighbor search can become an expensive operation for very high-dimensional data. In practice, however, we do not expect users to specify more than, say, five dimensions as part of their Skyline queries, particularly, in interactive environments.

2. Compute the complete Skyline: The algorithm will explore all regions and will find all points of the Skyline.

3. No false-positives: Considering the two lemmata mentioned at the beginning of this chapter, all Nearest Neighbors found by our Nearest Neighbor algorithm are part of the Skyline.

4. Fairness: The Nearest Neighbor algorithm produces results from the whole range of results very quickly. We will further discuss this property in a later section.

5. Control: There are two ways in which the Nearest Neighbor algorithm can react to preferences specified by the user, that is, change the order in which query results are returned:

   - First, if a user clicks on a particular point in the graphical user interface during the running time of the algorithm, the algorithm can process those regions of the ToDoList next that are located near the point the user has clicked on.
   - Second, the distance function f is a parameter of the Nearest Neighbor algorithm and can be changed any time during the execution of the algorithm. This property can be exploited in the following way. At the beginning, the algorithm starts with some default distance function, for example, $f = \text{price} + \text{distance}$. If the user clicks on the point (price = 40, distance = 200), the user puts more emphasis on a cheap hotel rather than a short distance to the beach. The distance function can now be adjusted to $f = 2 \cdot \text{price} + \text{distance}$. With every interaction of the user, the distance function can be adjusted accordingly. We have to stress here that changing the distance function does not change the Skyline. From Lemma 1 we can see that the Skyline is independent from the distance function. Only the order in which the Skyline points are output is changed when the distance function is changed. The Skyline points returned before changing the distance function remain valid as well as the ToDoList. As a result, the Nearest Neighbor algorithm can continue to produce Skyline points using the new distance function without any adaptions.

6. Universal: As we have seen, the algorithm can be extended so that it works for Skyline queries that involve more than two dimensions. It can be applied if the query involves additional predicates (indexed and not indexed) and it can also be applied in a mobile scenarios (more of that in Part IV). It is based on multi-dimensional index structures (R-tree) that support Nearest Neighbor search. Such index structures are well known and provide scalability, dynamic updates, and independence of the data value distribution. If the data set involves, say, five dimensions which are potential criteria for Skyline queries, then a single five-dimensional R-tree will be sufficient to execute all Skyline queries.

The Nearest Neighbor algorithm fulfills all online requirements. Only now is it possible to think about mobile applications of this algorithm. Mobile environments contain continuous queries which should be answered immediately, or in other words, which should be answered *online*. More of this will be discussed in Part IV.

17.5 Discussion

This section presents some running time results of the Nearest Neighbor algorithm. The first part of this section compares the Nearest Neighbor algorithm with batch algorithms from Part II. Those results were published in [KRR02]. The second part of this section takes the progressive algorithms from Part II and compares them to the Nearest Neighbor algorithm. These results were also published in [KRR02]. This section concludes with a discussion about the quality of Skyline results when computed in an online fashion.

All experiments were carried out on a SUN Ultra 1 with a 167 MHz processor and 128 MB

of main memory. The operating system was Solaris 8. Data sets we used were anti-correlated, correlated, and uniformly distributed (see Chapter 6 for details).

17.5.1 Comparisons with Batch Algorithms

For these experiments we used the Nearest Neighbor algorithm as described previously. As duplicate treatment strategy we used the Propagate approach. The data sets contained 100,000 and 1 million points. As value distributions we used an anti-correlated data set, a correlated data set, and a uniformly distributed data set. Detailed descriptions of the three types of data sets can be found in Chapter 6. As Batch algorithms we deployed a variation of the Block-Nested-Loops algorithm depicted in Chapter 10 and a variation of the Divide-and-Conquer algorithm depicted in Chapter 11. Table 17.3 displays the results for 2-dimensional Skyline queries.

	100,000 Points			1,000,000 Points		
	Anti.	Corr.	Unif. Dist.	Anti.	Corr.	Unif. Dist.
NN	0.57	0.02	0.20	0.69	0.02	0.50
BNL	1.77	1.65	1.68	17.16	16.24	16.07
D&C	2.63	2.56	2.63	28.65	28.53	28.50

Table 17.3: *Batch vs. NN [KRR02], size of Skyline, running times [secs] for 2-dimensional Skyline*

We can observe that the Nearest Neighbor algorithm (NN) is the overall winner in this experiment. As mentioned before, 2-dimensional Skylines are a particularly good case for the Nearest Neighbor algorithm. The Block-Nested-Loops (BNL) and the Divide-and-Conquer (D&C) algorithms show relatively poor performance because both algorithms involve reading the whole data set, whereas the Nearest Neighbor algorithm can use the R*-tree in order to quickly retrieve Skyline points. When comparing the running times for the small and large data set, it can be seen that the Nearest Neighbor algorithm scales best. Its running time stays almost constant, whereas the running times of the other algorithms increase sharply. To be more precise, the running times of the Nearest Neighbor algorithm do not increase even when the data set size is increased by a factor of ten. (100K points versus 1M points). For the Block-Nested-Loops and Divide-and-Conquer algorithms the running times increase approximately by a factor of ten when comparing the 100K and 1M data sets. It is quite clear why. Both Batch algorithms need to scan the whole data set at least once to be able to decide on the Skyline membership or not. The Nearest Neighbor algorithm draws advantage from its unique property that is does not have to look at the whole data set to decide Skyline membership.

17.5.2 Comparison with Progressive Algorithms

Now we want to compare the Nearest Neighbor algorithm to the Bitmap algorithm and the B-tree algorithm published in [TEO01], described in Chapter 12 and Chapter 13, respectively. Table 17.4 is displays the results of our Nearest Neighbor algorithm, the Bitmap algorithm and the B-tree algorithm. Again, 100,000 and 1 million points with anti-correlated, correlated, and uniformly distributed data values were used.
Again, the Nearest Neighbor algorithm is the winner in this contest. The Bitmap algorithm must consider all points in the data set. This leads to the same observation as previously for the Block-Nested-Loops and Divide-and-Conquer algorithm: The running times for the large data sets is about 10 times higher than for the small data set. The B-tree algorithm performs

	100,000 Points			1,000,000 Points		
	Anti.	Corr.	Unif. Dist.	Anti.	Corr.	Unif. Dist.
Skyline size	49	1	12	54	1	12
NN	0.57	0.02	0.20	0.69	0.02	0.50
Bitmap	6.09	0.84	1.40	57.12	12.23	17.90
B-tree	13.86	0.01	0.26	> 200	0.12	0.92

Table 17.4: *Progressive vs. NN, size of Skyline, running times [secs] for 2-dimensional Skyline* [KRR02]

well for the correlated and uniformly distributed data sets. However, it shows poor performance for the anti-correlated data sets. In this case, the B-tree algorithm must also read (almost) the whole data set in order to compute the full Skyline because the termination condition, the special trick of this algorithm, does not apply until the very end in this particular data value distribution. Furthermore, the B-tree algorithm has fairly high overhead for each point: it must compare each point with all Skyline points found so far. For the large data sets, we had to stop the execution of the B-tree algorithm for the anti-correlated database after 200 seconds, it had produced only 30 Skyline points up to then.

Although we believe that for this particular Skyline setup the 2-dimensional problems are by far the most important problems, we also want to take a look at higher dimensional data sets, here a 5-dimensional anti-correlated data set with 1 million points. The size of the Skyline is fairly large. The Skyline contains about 36,000 points. Computing the full Skyline takes a long time, regardless what algorithm is used. As a result, online characteristics of an algorithm are of particular interest for interactive applications in this scenario. Rather than flooding the user with results, first results should be returned immediately and the user should be able to pick his or her choice from relevant points.

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

Table 17.5: *Progressive vs. NN , first results [KRR02]*

Table 17.5 shows the running times of the alternative Skyline algorithms to produce the first 100 Skyline points. In addition, it shows how quickly the algorithms can return points based on user preferences. In this experiment, the B-tree algorithm produces the results very quickly: it takes less than a second to produce the first 100 Skyline points. This is not surprising because this algorithm simply needs to scan through the extended B-tree and return points that are extremely good in one dimension. The Nearest Neighbor and Bitmap algorithms need to perform significantly more logic. So the B-tree algorithm is good in order to return *some* results. However, if the user gives preferences, the B-tree algorithm cannot adapt well. In this experiment, we simulated a user that is interested in a point that is good in all dimensions: If one uses the B-tree algorithm it takes 300 seconds to find such a point and more than 15,000 points need

to be inspected before such a point is returned. In terms of interactivity, the Nearest Neighbor algorithm is the clear winner. Whenever a user gives a preference, the Nearest Neighbor algorithm adjusts its distance function and returns the next point that matches these preferences immediately: it only takes 0.1 seconds for the Nearest Neighbor algorithm to adapt. In some sense, the B-tree algorithm (blindly) scans through the Skyline whereas the Nearest algorithm is able to (selectively) pick points from the Skyline based on user preferences. Naturally, blindly scanning has less overhead per point than selectively picking points. We will stress this fact in a later section where we take a look at the first results returned by different Skyline algorithms in the 2-dimensional case, a quality check.

17.5.3 Comparisons of Different Duplicate Treatments of the Nearest Neighbor Algorithm

We now turn to a discussion of the performance trade offs of the variants of the Nearest Neighbor algorithm for Skyline queries of more than two dimensions. These variants have been previously described. Figure 17.5 shows the performance of four variants for a 5-dimensional Skyline with an anti-correlated data set and 1 million points.

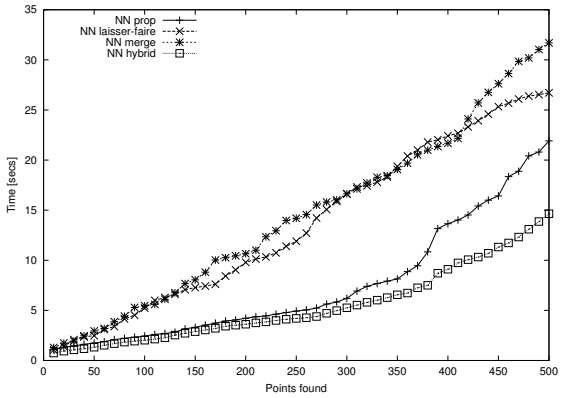


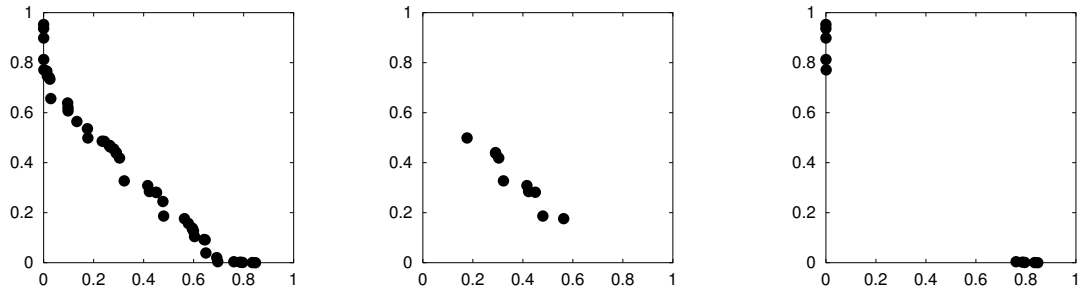
Figure 17.5: *Comparison of Nearest Neighbor variants*

We measured the following variants: Laisser-faire; Laisser-faire and merge regions whenever a duplicate is found; Propagate; Hybrid, that is, propagate to the first 20 % entries of the ToDoList, duplicates that are not prevented by the reduced propagation are handled with the laisser-faire approach. No index on the ToDoList was used to find entries in the ToDoList. Nevertheless, the last two variants (Propagate and Hybrid) clearly outperform the other variants. The merge and laisser-faire variants spend too much time on duplicates. On average four nearest neighbor searches needed to be carried out for these two variants in order to find a new result. Furthermore, the performance of the hybrid variant can be improved by tuning. In some experiments, it performed better when only 10 %, rather than 20 %, of the ToDoList were scanned or when only a fixed number of entries of the ToDoList was considered.

17.5.4 Quality of Results

As said in Chapter 16, mostly users are not probably interested in *all* Skyline points (for example, all hotels in Lido di Jesolo). They want a good overview of their choices in order to pick their most favorite hotel. The user wants a “big picture” rather than being struck by thousands

of results. In this section we compare the B-tree algorithm, that produces Skyline points at very high rates, and the Nearest Neighbor algorithm, that has a good interactive performance. Figure 17.6 shows a 2-dimensional Skyline anti-correlated Skyline problem. Figure 17.6(a) shows all Skyline points in a coordinate system. Figure 17.6(b) shows the first 10 points returned by the Nearest Neighbor algorithm and Figure 17.6(c) shows the first 10 points returned by the B-tree algorithm.



(a) Full Skyline (b) First 10 points NN algo. (b) First 10 points B-tree algo.

Figure 17.6: *Quality of results*

The Nearest Neighbor algorithm returns points that are good in all dimensions, giving the user an excellent "big picture" of, for example, the hotel situation. It returns some hotels that are cheaper and have a longer distance to the sea and it returns some hotels that are more expensive but have a shorter distance to the sea. Now, the user can decide what is more important to him or her and click on one of the hotels or select an area which is of more interest. The Nearest Neighbor algorithm adapts to this new situation and starts returning points that lie in the area the user has clicked on. If the user chooses an area located at the x- or y-axis the Nearest Neighbor algorithm will start returning extreme points, that are either very cheap or very close to the beach.

The B-tree algorithm fails to give the user a "big picture" of the hotel situation. It returns extreme points first, that is five hotels that are cheap and five hotel that are close to the beach. After these 10 points, however, the user does not get a good picture of the price/distance trade-offs. The users has to wait until the B-tree algorithm starts returning points "in the middle" since they cannot control the behavior of the B-tree algorithm. When the B-tree algorithm starts returning even-tempered points the screen will be filled with extreme points preventing the user from getting the big picture due to an overwhelming amount of points.

CHAPTER 18

Branch-and-Bound Skyline Algorithm

This algorithm has been published in [PTFS03]. We will explain the main algorithm by applying it to our hotel example. Some newer work about this algorithm has been published in [PTFS05].

18.1 Algorithm Description

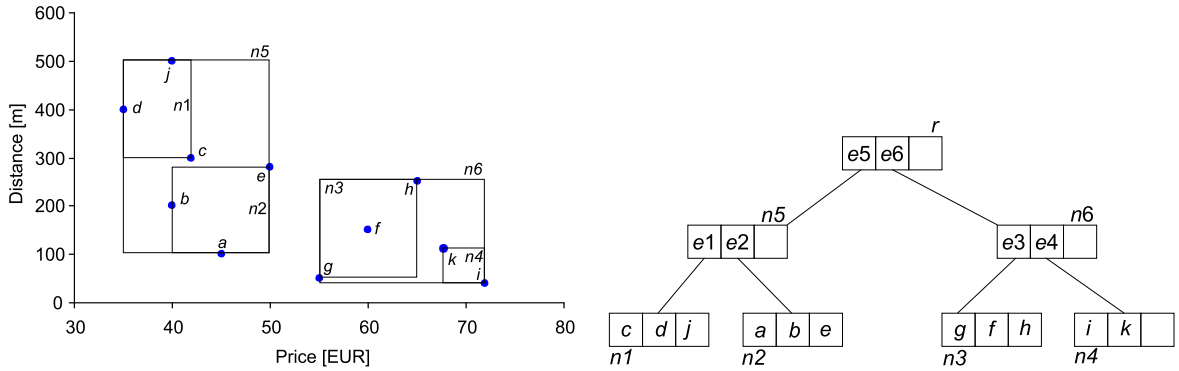
Table 18.1 recalls our hotel example from Chapter 7. We add two more columns to the table. The first added column abbreviates the hotel names with characters from a to k. The second added column displays the value of the distance function for each hotel. As distance function we use $f = \text{price} + \text{distance}$. These values are later necessary to perform a sorted processing of the entries. Please note that these values do not have to be precomputed when using an R-tree as indexing structure for the points. The distance can be computed "on the fly" when accessing the R-tree nodes.

Name	Short	Price [EUR]	Distance [m]	Distance f
Hotel Arena	**a**	45	100	145
Hotel Aden	**b**	40	200	240
Hotel International	c	42	300	342
Hotel Aurora	**d**	35	400	435
Hotel Majestic Toscanelli	e	50	280	330
Hotel Monaco & Quisisana	f	60	150	210
Hotel Elpiro	**g**	55	50	105
Hotel Marlisapier	h	65	250	315
Hotel Al Gambero	**i**	72	40	112
Hotel Rex	j	40	500	540
Hotel Heron	k	68	100	168

Table 18.1: *Hotel example for Branch-and-Bound algorithm*

As previously mentioned it is handy to use an R-tree as indexing structure for the hotels. Figure 18.1 shows one possible R-tree with a node capacity of 3 points per node. In Figure 18.1(a) the

points (hotels, *a* to *k*), the leaf nodes (*n*1 to *n*4), and the intermediate node (*n*5, *n*6) are shown with their orientation in a 2-dimensional plain. The root node *r* is not shown. It contains nodes *n*5 and *n*6. Figure 18.1(b) displays the R-tree.



(a) Orientation (b) Nodes

Figure 18.1: *R-tree layout*

For all leaf nodes and intermediate nodes the distance is also needed. Table 18.2 shows those distances. The distance of a node is defined by its lower left corner. Again, these distances are inherent information when using an R-tree. If they are not computed and stored when building the R-tree, they can be computed when accessing the specific node. So almost no overhead is implied when computing these distances. The distance function is, of course, the same function as used for the distance of each point.

Node	Distance f
$n1$	335
$n2$	140
$n3$	105
$n4$	108
$n5$	135
$n6$	95

Table 18.2: *R-tree, distance of nodes*

With this additional information, we can now describe the Branch-and-Bound Skyline algorithm. Table 18.3 displays each step to compute the Skyline of the example. The Branch-and-Bound Skyline algorithm is an adaption of the "Incremental Nearest Neighbor" algorithm from [HS99]. It starts at the root node *r* and inserts all entries ($e5$, $e6$) into a heap sorted according to their distances. Each entry *e* corresponds to a node *n*. The the first entry, that is, the one with the minimum distance ($e6$) is *expanded*. Expanding means retrieving all entries from lower nodes that the entry is pointing to. Here, $e6$ expands to $e3$ and $e4$. These two entries are also inserted into the heap according to their distance. Again the entry with the lowest distance is is expanded ($e3$). This yields the first three points (g, f, h) in the heap, inserted in order of distance. The algorithm discovers the first Nearest Neighbor (g, in bold). It is the first entry in the heap and it is not dominated by any Skyline point since the Skyline *S*is still empty. The Nearest Neighbor (g) is inserted into the Skyline (or output to the user) and removed from the heap. Now the expansion procedure continues with the next entry in the heap. The only differences now is that all new entries of the heap are *pruned* (in square brackets) with the Skyline (so far only one

point, g). If the entry is dominated by this point (later by any Skyline point) it is removed from the heap without looking at it. This is the case for the next expansion step ($e4$) which would yield two new entries (i and k). The entry i is inserted into the heap whereas entry k is pruned since it is dominated by g. With i the algorithm finds its second Nearest Neighbor, hence i is inserted into Skyline S. This procedure continues until the heap is empty. Whenever a point (as opposed to an intermediate node entry) is at the top of the heap it is a Nearest Neighbor. The last step here does not execute expansions any more since only points are still in the heap. These points are pruned with the Skyline.

Action	Heap contents	S
Access root	$e6$ (95), $e5$ (135)	
Expand $e6$	$e3$ (105), $e4$ (108), $e5$ (135)	
Expand $e3$	g **(105)**, $e4$ (108), $e5$ (135), f (210), h (315)	g
Expand $e4$ and prune with S	i **(112)**, [k (168)], $e5$ (135), f (210), h (315)	g, i
Expand $e5$ and prune with S	$e2$ (140), $e1$ (335), f (210), h (315)	g, i
	a **(145)**, b **(240)**, [e (330)], $e1$ (335), f (210), h (315)	g, i, a, b
Expand $e1$ and prune with S	[c (342)], d **(435)**, [j (540)], f (210), h (315)	g, i, a, b, d
Prune with S	[f (210)], [h (315)]	g, i, a, b, d

Table 18.3: *Brand-and-Bound Skyline algorithm steps*

Figure 18.2 displays the pseudo-code for the Branch-and-Bound algorithm. The domination-function returns true if the entry in turn is dominated by a Skyline point and false if no Skyline point dominates the entry. Each entry is checked twice for Skyline dominance: before it is inserted into the heap and before it is expanded. This is necessary because an entry in the heap might become dominated by a Skyline point discovered after it has been inserted. In this case the entry can be discarded right away and does not have to be expanded any further.

Is this algorithm correct? The authors of [PTFS03] give three lemmata that show that the Branch-and-Bound Skyline algorithm works correctly. We want to briefly discuss these lemmata and their proofs here.

Lemma 1: The Branch-and-Bound algorithm visits (leaf and intermediate) entries of an R-tree in ascending order of their distance to the origin.

Lemma 2: Any data point added to the Skyline during execution of the algorithm is a Skyline point.

Lemma 3: Every data point will be examined unless one of its ancestor nodes has been pruned.

Lemma 1 is proven straightforward: A heap is used to preserve a sorted order of leaf and intermediate tree entries. Lemma 2 can be shown by contradiction: Assume that p was added to S, but p is not a Skyline point. Then p must be dominated by (at least) one Skyline points s which is (at least) better in one dimension (see definition of domination in Chapter 4). This means that $\text{distance}(s) < \text{distance}(p)$. With Lemma 1, s must be have been visited before p and hence, p should be pruned by s. Finally the proof of Lemma 3 is obvious, since all entries that are not pruned by a Skyline point are inserted into the heap and later examined.

In [PTFS03] the authors also discuss the I/0 optimality of this algorithm, which we omit here,

```
1  Input:    R-Tree R // Dataset D represented by index structure
2  Input:    Distance function f
3  Output:   Skyline S // Point by point
4
5  begin BBSAlgorithm
6
7    Heap H(f);     // sort the heap according to f
8    Node n := R.getRoot();
9
10   for (int i := 1 to n.getFanout()) do
11     H.insert(n.getNextNode());
12   end;
13
14   while(!H.empty()) do
15
16     Entry e = H.getTop()
17
18     if(domination(e, S)) then
19       // discard e
20     else
21       if(e.instanceof() = Node) then
22         // intermediate node: insert all children
23
24         for int i := 1 to e.getFanout() do
25           Entry k := e.getNextNode();
26           if(domination(k, S)) then
27             // discard k
28           else
29             H.insert(k);
30           end;
31         end;
32
33       else
34         // e is data point
35
36         S.insert(e);
37
38       end;
39     end;
40   end;
41
42 end;
```

Figure 18.2: *Branch-and-Bound algorithm*

and they describe some variants of Skyline queries, which we also omit. The interested reader is forwarded to [PTFS03]. However, in a later section we briefly show some running time results that compare the Nearest Neighbor Skyline algorithm to the Branch-and-Bound Skyline algorithm.

18.2 Online Algorithm?

As we previously said, the Branch-and-Bound Skyline algorithm qualifies as an online algorithm. We want to quickly scan through the requirement from Chapter 16 to check each requirement separately.

1. First few results: As soon as the algorithm accesses a leaf node it finds the first Skyline point. With the characteristics of an R-tree this requires not more than a few page accesses on disk which normally is very quick. Consecutive points are also found very quickly since only a few more page accesses are required.

2. Compute the complete Skyline: Lemma 3 ensures this property. Every point will be examined unless it is dominated or one of its ancestor nodes is dominated. In this case the point is also dominated.

3. No false-positives: Lemma 2 guarantees that only real Skyline points are inserted. into the Skyline.

4. Fairness: The Branch-and-Bound algorithm finds the Skyline points according to their distance. This distance involves all dimensions equally balanced.

5. Control: By changing the distance function the user can control the order in which Skyline points are output. However, changing the distance function during computation is not possible, since then all heap entries would have to be re-sorted. In that case Lemma 2 does not hold anymore.

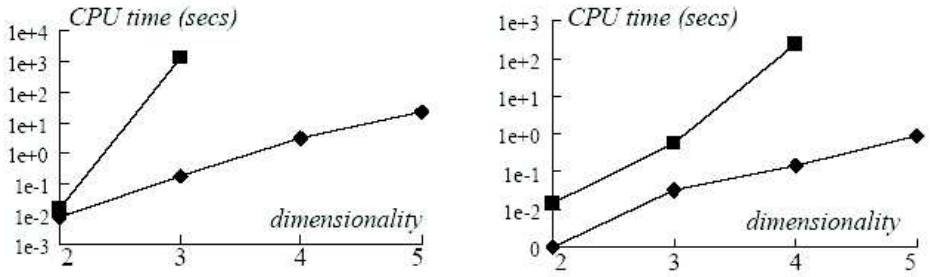
6. Universal: This property is also fulfilled. The Branch-and-Bound algorithm can be used with with different indexing structures. The same indexing structure can be used for *m-d*-Skyline queries. No data value distribution is favored.

Besides the control property, the Branch-and-Bound algorithm completely fulfills all requirements. The control property is partly fulfilled. Nevertheless, we call the Branch-and-Bound Skyline algorithm an online algorithm.

18.3 Discussion

Let us take a short look at some running time experiments which compare the Nearest Neighbor Skyline algorithm with the Branch-and-Bound Skyline algorithm. Those results were published in [PTFS03]. The results were produced on Pentium 4 computer running at 2.4 GHz with 512 MB of main memory. The authors used anti-correlated and uniformly distributed data sets as described in Chapter 6 with 100,000 and 10 million points. They used an R*-tree by [BKSS90]. In Figure 18.3 the Branch-and-Bound algorithm is compared to the Nearest Neighbor algorithm with 1 million points of 2, 3, 4, and 5 dimensions. The data sets used had an anti-correlated and

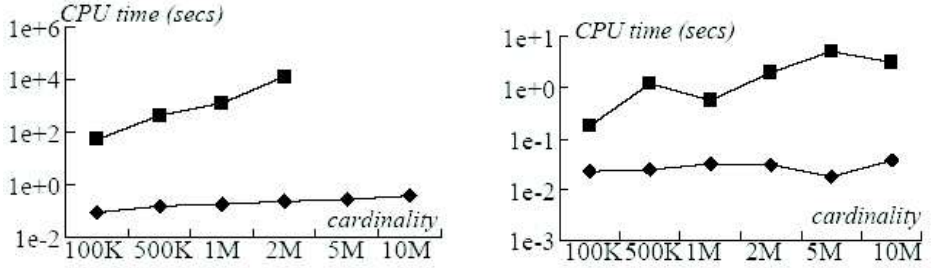
uniformly distributed data value distribution. The full Skyline was computed. It can be seen that the Branch-and-Bound algorithm outperforms the Nearest Neighbor algorithm for all dimensions. Even more, for 5 (uniformly distributed) and for 4 and 5 (anti-correlated) dimensions the Nearest Neighbor algorithm did not come to an end. The gain of the Branch-and-Bound algorithm was at least 10 times as fast as the Nearest Neighbor algorithm.



(a) Anti-correlated (b) Uniformly distributed

Figure 18.3: *Running time BBS vs. NN, 1M points, dim 2 to 5*
[PTFS03]

Figure 18.4 again compares the Branch-and-Bound algorithm to the Nearest Neighbor algorithm in terms of data set sizes. The data set sizes ranked from 100,000 to 10 million points. Again, the Nearest Neighbor algorithm is no match for the Branch-and-Bound algorithm.



(a) Anti-correlated (b) Uniformly distributed

Figure 18.4: *Running time BBS vs. NN, dim 3, 100K to 10M points*
[PTFS03]

The Branch-and-Bound Skyline algorithm is superior to the Nearest Neighbor Skyline algorithm in all cases. The only drawback is its lack of user control possibilities discussed in Chapter 18. Since the running time of the Branch-and-Bound algorithm is much lower than the running time of the Nearest Neighbor algorithm the lack of user control could be overcome be simply recomputing the Skyline from scratch.

CHAPTER 19

Summary: Online Skyline Computation

In Chapters 16 through 18 we showed two Skyline algorithms that qualify as online Skyline algorithms. We also presented requirements for an algorithm to be called an online Skyline algorithm. Also we laid out a scenario for which online algorithms can be used and showed why we think this scenario will happen in daily life.

The Nearest Neighbor Algorithm was the first algorithm to be suitable for our online scenario exploiting the relationship between Skyline and Nearest Neighbors as discussed in [Ros01]. It is superior to most Skyline algorithms given in Part II especially for data sets with low and medium dimensionality. It is also the algorithm of choice if you want to have a quick glance at the shape of the Skyline since it produces the first Skyline points quickly with no particular emphasis to any dimension. A fine add-on is also that the user can control the sequence the Skyline points are computed.

The Branch-and-Bound Skyline Algorithm is an adaption and extension of the Nearest Neighbor Algorithm. Its particular strength lies in the overall speed of computing the Skyline, being less affected by data sets with high dimensionality. A minor drawback of the algorithm is that due to its construction there is no user control like for the Nearest Neighbor Algorithm.

Part IV

Continous Skyline Computation

CHAPTER 20

Layout

Part IV takes Skyline computation a leap ahead. Instead of trying to compute the Skyline in efficient ways we will use the Skyline, the set of incomparable points, as an information pre-filter as described in Chapter 4.
There are two implications with this: First, an information pre-filter usually works on changing data, for example, stock ticks quoting new prices for stocks are sent to each stock broker. The information pre-filter has to decide if, according to some user preferences, if the new quote should be displayed to the stock broker or not. Certainly, each stock broker has his or her own the of stocks of interest. So in order to use the Skyline as an information pre-filter we have to enable the Skyline computation to deal with changing data. We call this scenario *continuous Skyline computation* since changing data certainly involves recomputing the Skyline in some way. Depending of the rate the changes of the data occur this Skyline re-computation becomes a continuous computation. Chapter 21 deepens this scenario and Chapter 22 explains two basic operations that are needed for this scenario.
And second, when using the Skyline as an information pre-filter, then certainly there is not only one pre-filter but thousands, each having a slightly different appearance. For example, one stock broker is interested in petro-chemical companies, the other in companies which sells consumer goods. We have to make sure that each pre-filter is efficiently recomputed when data changes occur. Again Chapter 21 deepens this scenario and Chapter 23 depicts a way to efficiently recompute many Skyline queries.
Specifying user preferences can be done in many ways. We choose to use an adaption of SQL as proposed in Chapter 4. So user express their preferences in an SQL notation, a Skyline query.

CHAPTER 21

Continuous Skyline Scenario

21.1 Continuous Skyline Queries

Continuous Skyline queries are Skyline queries that are recomputed on a regular basis. Recomputing a Skyline query means recomputing the set of incomparable points, the Skyline. On a regular basis means, whenever an event occurs, the Skyline query is recomputed. An event for recomputing the Skyline query might be the insertion or deletion of a point into or from the underlying data set. Then definitely the Skyline query has to be recomputed since an insertion or deletion might influence the Skyline.
Before we dwell into details about what is needed to react to changing data sets, we want to restrict the Skyline query as given in Chapter 4. The Skyline queries in the following have the form given below:

SELECT *projection*
FROM *collection*
WHERE *predicates*
SKYLINE OF $\dim_1$ [MIN|MAX], ..., $\dim_n$ [MIN|MAX];

Projection and *collection* have the same meaning as previously said. *Predicates* are restricted to range predicates on n dimensions concatenated by AND, for example,

SELECT *
FROM data_set
WHERE $0.1 < \dim_1 < 0.2$ AND $0.1 < \dim_4 < 0.2$
SKYLINE OF $\dim_1$ MIN, $\dim_2$ MIN;

We do not consider queries with joins or group by. Also we do not consider Skyline queries with diff annotations as described in Chapter 4. This is done to facilitate the handling of queries, especially when multiple continuous queries are computed. Figure 21.1 shows the pseudo-code representation of a query.
A query consists of five parts: The first part holds the Skyline S, that is, all incomparable points at present time. For instance, the Skyline can be an array of points. The second part

```
1  structure Query
2
3    // holds the current Skyline of query
4    Skyline S;
5
6    // representation of SKYLINE OF clause (comparison function)
7    // returns: p {LESS | GREATER | EQUAL | INCOMPARABLE} q
8    function compare(Point p, Point q, bool dims);
9
10   // representation of WHERE clause
11   // returns: p fulfills predicates or not {TRUE | FALSE}
12   function predicates(Point p);
13
14   // everything that has to be done when a point is
15   // inserted into the data set
16   function insert(Point p);
17
18   // everything that has to be done when a point is
19   // delete from the data set
20   function delete(Point p);
21
22 end;
```

Figure 21.1: *Pseudo-code of a query*

(*compare*) is an internal representation of the `SKYLINE OF` clause. This is a modification of the comparison function given in Figure 4.1. The modifications include the possibility to compute *m-d*-Skylines, that is, only a subset m of the d dimensions ($m < d$) is used to compute the Skyline on. The comparison function gets two points p and q and a representation of the dimensions the Skyline is computed on, *dims*, and returns p is less than, greater than, equal to, or incomparable to q with respect to *dims*. This comparison function is denoted as c_Q in subsequent chapters.
The third part (*predicates*) is an internal representation of the `WHERE` clause holding all the predicates. The function gets a points and checks if the point fulfills all predicates. It, therefore, returns true or false.
The functions *insert* and *delete* represent the operations that have to be done with respect to the Skyline query when a new point is inserted into the data set or deleted from the data set except the insertion or deletion of the point from the data set itself. This is done in a separate step.
Later we will take a closer look at the functions *compare* and *predicates* and the functions *insert* and *delete*. For now they serve as a placeholder for the further discussion of continuous Skyline queries.

Please note: The internal representation of a Skyline query as given in Figure 21.1 is an extended representation of a Skyline query arranged for continuous Skyline queries. The *insert* and *delete* functions are, of course, not part of SQL.

Now the questions arises if the Skyline query has to be completely recomputed for the complete data set dropping the Skyline computed so far or if, for example, the Skyline of the query

can be used to determine if the change in the data set has an effect on the specific query and the resulting Skyline points. Those questions are answered in Chapter 22. We will give our considerations, algorithms and operations, and proofs for certain details in this chapter. We will also present measurements for each operation necessary for continuous query processing helping the reader to assess the complexity of the operations.

21.2 Skyline Queries and User Profiles

Three of our given application domains from Chapter 2, the Electronic Market Places, the Stock Ticks, and the Quality Assurance examples use a Skyline query in a continuous way. This means, that in these application scenarios a Skyline query is continuously submitted to check whether the underlying data set has been changed or not and if the Skyline has to be recomputed. A change of the data set occurs if, for example, a new point is inserted into the data set or a point is deleted from the data set.
The Electronic Market Places example is also the scenario which our special 5-dimensional data set, the correlation groups data set is made for. Chapter 6 explains how the data set is constructed.

The Skyline query and its associated points, the Skyline, represent all points that the user who specified the query is interested in. Looking at it from this point of view a Skyline query can be seen as a user profile. The profile describes preferences a user has. For example, Table 21.1 displays the offer table from our application domain "used car market" (Chapter 2).

Model	Price	Age	Speed
BMW 330 xd	EUR 30,000	5 years	200 km/h
Ford Focus	EUR 8,000	3 years	150 km/h
Toyota Avensis	EUR 10,000	4 years	170 km/h

Table 21.1: *Used car market: offers*

A user preference *cheap cars that are fast and price* $<$ *20,000 EUR* results in the following Skyline query:

```
SELECT *
FROM offers
WHERE 0 < price < 20000
SKYLINE OF price MIN, speed MAX;
```

The associated Skyline S, the points of interest, contains two point, the Ford and the Toyota, $S = \{\text{Ford}, \text{Toyota}\}$.
Now, the following offer is inserted into the data set:

(VW Golf, EUR 9,900, 2 years, 180 km/h)

This new offer changes the points of interest for the user who specified the above query, since now the Toyota is dominated by the VW. The new Skyline consists of the Ford and the VW, $S = \{\text{Ford}, \text{VW}\}$. The Skyline query and its associated points serve as a user profile in this application scenario.

21.3 Multiple Continuous Skyline Queries

As you surely have noticed in the previous section our "user profile" scenarios will not have only one registered Skyline query with the system but many, possibly thousands of Skyline queries which all have to be answered continuously, that is if new points arrive or points are deleted from the data set. One possibility of answering all those queries is in a sequential way. This means each query is inspected separately for each new point that is inserted or each point that is deleted. Figure 21.2 shows this approach.

```
1  Input:     Query Q[] // array of registered queries
2
3  procedure SequentialChecking(Point p, bool inserted)
4
5    if(inserted)
6      for (int i:= 1 to Q.length) do
7        Q[i].insert(p);
8      end
9    end
10   else
11     for (int i:= 1 to Q.length) do
12       Q[i].delete(p);
13     end
14   end;
15
16 end;
```

Figure 21.2: *Sequential checking of all queries*

The procedure *SequentialChecking* is called each time a point is inserted into the data set or deleted from the data set. It has two input parameters, the point which was inserted or deleted and a switch determining if the point was inserted or deleted. Global parameter is an array holding the queries in our internal representation (see Figure 21.1). A *system* as we speak of will be explained in the next section. At each a call the procedure loops through every query and depending on insertion or deletion of the point a function *insert* respectively *delete* is called. These two function are also part of our query representation in Figure 21.1. They make sure that, after *insert* or *delete* is called, the Skyline remains the set of incomparable points according to the user preferences (represented by *compare* and *predicates*) and, of course the changed underlying data set. The steps that have to be done if a point is inserted or deleted from the data set will be explained later.

Obviously with thousands of queries registered, looping through all queries may become quite time consuming. In Chapter 23 we will present efficient methods to check queries in a "grouped" sort of way so that "similar" queries can be checked together. We will present extensive measurements comparing, for example, the time needed for sequentially checking all registered queries and the time needed for checking all registered queries in an "intelligent" way.

21.4 Taxonomy

Figure 21.3 shows a *query system* as we think of. The data set is the one which is update, either points are inserted there or points are deleted from there. The query engine makes sure that each registered query, $Q_1, Q_2, ..., Q_n$, is executed after an update of the data set has occurred. The query engine could be the function given in Figure 21.2, checking each Skyline query in an naive, sequential way after an update has occurred.

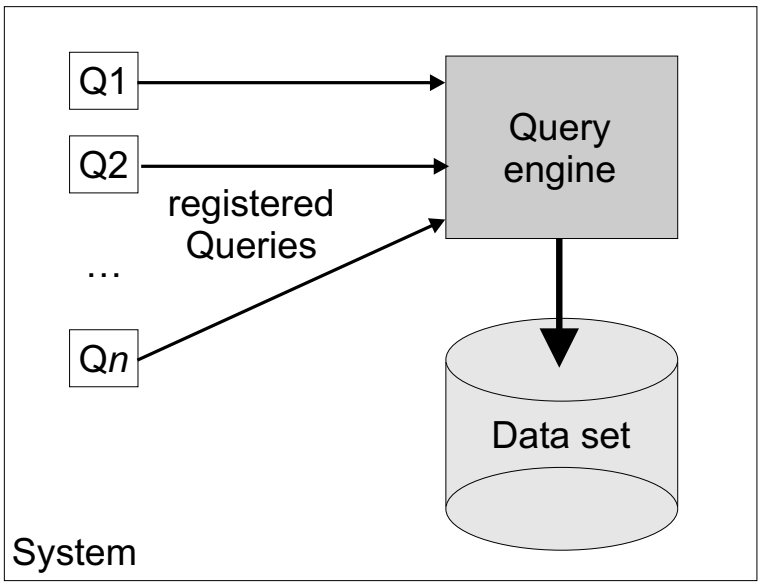


Figure 21.3: *Query system*

Skyline queries $Q_1, Q_2, ..., Q_n$ are queries as depicted in Figure 21.1 holding all necessary descriptions of a Skyline query including the set of Skyline points. Each query was registered by a user at some point in time. Registering means: The user submits the query to the system and the query engine computes the Skyline of the query using the current data set. After that, the query "waits" in the query system for new points to arrive or for points to be deleted.

CHAPTER 22

Operations for Continuous Skyline Queries

In this chapter we want to describe two basic operations that are necessary to handle continuous Skyline queries over changing data sets. These operations are an *insert* operation and a *delete* operation. Points can be inserted into the data set or deleted from the data set. Updating a point, for example, adjusting one dimension value, can be realized with first deleting the point and then inserting a "new" point.
This chapter establishes the operations that we introduced in Figure 21.1 and used in Figure 21.2.

22.1 Insert Operation

For describing the insert operation we look at the following setup: The data sets we use here are similar to those we used for batch, progressive, and online Skyline algorithms from Part II and Part III. The difference here is that the data set is not static, points arrive at the system and are inserted into the data set.
Recomputing the complete Skyline query seems not an option since we learned from previous chapters (especially those in Part II and Part III) that computing the complete Skyline of a data set can be quite time-consuming. Nevertheless, after an insert has occurred the Skyline S (in Figure 21.1) must remain the set incomparable points of data set D with respect to the particular Skyline query Q.
The idea is to use the already computed Skyline itself to check if the recently occurred insert has any effect on the Skyline query. When a new point is inserted into the data set the following observations can be made:

- New point **is dominated** by a Skyline point: By the transitivity property of the Skyline (see Chapter 3) it can be concluded that the new point cannot be part of the Skyline, thus it is discarded. It is inserted into the data set but the Skyline of the query stays unchanged.

- New point **dominates** one or more Skyline points: Again, with the transitivity property it can be said that the new point is part of the Skyline, thus all points that are dominated by the new point are deleted from the Skyline and the new point is inserted into the Skyline.

The deleted (previous) Skyline points stay in the data set and, of course, the new point is also inserted into the data set.

- New point **is incomparable** to all Skyline points: This means that a new Skyline point was inserted. The new point is inserted into the Skyline and also inserted into the data set.

If the new points is equal to a Skyline point, for example, a second car with similar properties is offered, it is also inserted into the Skyline. The user has now two choices of cars with the same properties. This is again a direct consequence of the definition of the dominance in Chapter 3.

22.1.1 Algorithmic Description

What was said above is formulated as pseudo-code in Figure 22.1. As one might notice, there is no access to the data set necessary since all decisions can be made with information held by the Skyline query itself, namely the Skyline of the query. Since this is the case, we do not display the insertion of the new point into the data set.
First, it is checked if the new point fulfills the predicates of the the query Q. If this is not the case the new point is not relevant for this particular query and is discarded. If the point fulfills the predicates of the query the further processing of the point is initiated. By walking through the Skyline the new point is compared against all Skyline points. Depending on the outcome of the compare function, the new point is either discarded (if a Skyline point is less than the new point) or inserted (if one or more Skyline points are greater than the new point, if a Skyline point is equal to the new point, or if all Skyline points are incomparable to the new point).

22.1.2 Performance of Insert Operation

Table 22.1 show the running times for the insert operation for 2- and 5-dimensional Skyline queries. All inserts were carried out sequentially, meaning a new point was inserted after a previous point was fully processed. It can be observed that the running times are higher for the uniformly distributed and anti-correlated data value distributions; in particular, for the 5-dimensional queries. The reason is that the sizes of the Skyline S became very large in these cases. Please note: In this special case we report the time for 10,000 inserts instead of a 1000 inserts (as it is done later on). This is in order to get stable, comparable results.

	Time [s]
Anti-corr.	0.0010
Corr.	0.0008
Unif.	0.0090
Corr.-groups	0.0008

(a) 2-dimensional Skyline

	Time [s]
Anti-corr.	0.200
Corr.	0.002
Unif.	0.020
Corr.-groups	0.003

(b) 5-dimensional Skyline

Table 22.1: *Running time for a 10,000 inserts*

The running times for all four data sets are low considering the number of points that were inserted. The reason for this is the beauty of the insert operation: only the Skyline has to be checked when a new point is inserted, no access of the data set is necessary.
Still the difference between the anti-correlated, correlated, uniformly distributed, and correlation groups data sets can be seen. For the anti-correlated data set it takes the longest to insert

```
1  function insert(Point ip, Query Q)
2
3    // check predicates
4    if (!Q.predicates(ip)) then
5      return false;
6    end;
7
8    // walk through Skyline S
9    for (int i:= 1 to Q.S.size()) do
10
11     switch (Q.compare(Q.S[i], ip)) do
12       case LESS:
13         // Skyline point dominates new point
14         return false;
15       case GREATER:
16         // one or more Skyline point(s) are dominated by new point,
17         // delete Skyline point(s), insert new point
18         Q.S.delete(Q.S[i]);
19         Q.S.insert(ip);
20         return true;
21       case EQUAL:
22         // new point is equal to a Skyline point
23         Q.S.insert(ip);
24         return true;
25       case INCOMPARABLE
26         // all Skyline points are incomparable to new point
27         // nothing to do here
28     end;
29
30   end;
31
32   // Skyline points were dominated (greater) or all Skyline points
33   // are incomparable
34   // insert new point
35   Q.S.insert(ip);
36   return true;
37
38 end;
```

Figure 22.1: *Pseudo-code for insert operation*

10,000 points, that is, because of the number of points already in the Skyline which is significantly higher compared to, for example, a correlated data set. These points have to be checked when an new point is inserted. The correlated data set is the fastest when it comes to inserting points. The Skyline for this data value distribution is the smallest compared to other data value distributions. So only a few Skyline points have to be checked when a new point is inserted into the data set.

22.2 Delete Operation

The setup for this operation is almost the same as described in the previous section, except that points are not inserted into the data set but delete from the data set. However, the objective stays the same. The Skyline S must remain the set of incomparable points of data set D and query Q. There are two possible case that can occur:

- Deleted point **is not part** of the Skyline: The point is delete from the data set D. The Skyline does not have to be adjusted.
- Deleted point **is part** of the Skyline: This is, obviously, the more interesting case. First, the point is deleted from the data set D and the Skyline S. Then, the data set D has to be searched for new Skyline points that were previously dominated by the just deleted point. Those points may now be part of the Skyline.

Searching for new Skyline points can be accomplished be recomputing the complete Skyline of the data set D. But the objective is to re-use the remaining Skyline points to save work that has been done previously.
Figure 22.2 illustrates the first step of our approach. Points s_1, d, s_2 constitutes a section of a Skyline before the deletion of point d. The dominance area of each Skyline point is also given in the figure (the lines starting from each Skyline point). Points p_1, ..., p_5 are sample points for illustration purposes. Points p_5 and p_4 are dominated by Skyline points s_1 and s_2, respectively. These points remain dominated when point d is deleted. Point $p1, p_2, p_3$ are all dominated by point d and become potential new Skyline points - candidate points - when d is deleted since they are only dominated bey d and not by any other Skyline points.

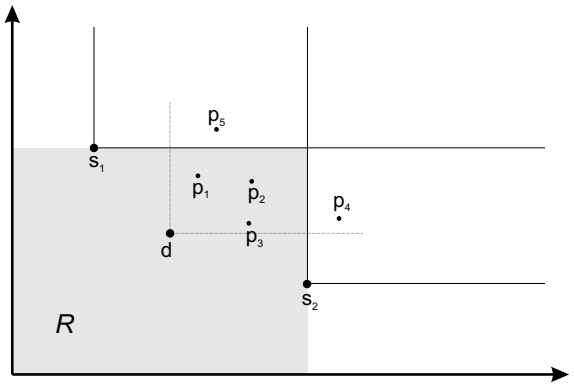


Figure 22.2: *Deleting a point from the Skyline*

As one can easily see, the gray shaded range R is the area that has to be searched for new Skyline points when point d has been deleted. It is limited by the origin and the remaining Skyline points s_1 and s_2. New Skyline points can only be found in this area. So the existing Skyline can be used to limit the search area and one has not to recomputed the Skyline of the complete data set. The range R has coordinates $(0,0;x_{s_2},y_{s_y})$. Unfortunately, points s_1 and s_2 are unknown in the sense of not knowing which Skyline points limit the range R. Any two points in the remaining Skyline could be the ones that are needed. This is the reason we have to take an iterative approach of finding these two limiting points. This iterative approach is shown in Figure 22.4.

22.2.1 Algorithmic Description

First, we want to describe the delete algorithm as given in Figure 22.3. The function to compute the candidate points is described later.

```
1  Input:     Dataset D
2
3  function delete(Point p, Query Q)
4
5    // walk through Skyline S
6    for (int i:= 1 to Q.S.size()) do
7
8      // deleted point belonged to the Skyline
9      if (Q.compare(Q.S[i], p) = EQUAL) then
10
11       // delete point from Skyline
12       Q.S.delete(Q.S[i]);
13
14       // compute candidate points taking other Skyline points
15       // into account
16       Dataset candSky := candidatePoints(Q, D, p);
17
18       // compute new Skyline points from candidate points
19       Skyline addSky := skyline(candSky);
20
21       // add new Skyline points
22       Q.S.insert(addSky);
23
24       return true;
25
26     end;
27
28   end;
29
30   // deleted point did not belong to the Skyline
31   return false;
32
33 end;
```

Figure 22.3: *Pseudo-code for the delete operation*

The delete algorithm is simple, walks through the Skyline and check if the point that is going to be deleted is part of Skyline or not. If the point is not part of the Skyline it is just deleted from the data set (not shown here). If it is part of the Skyline it is deleted from the Skyline. Then the candidate Skyline points are computed with the help of the remaining Skyline points, this is done in the function *candidatePoints*. After the candidate points have been computed the Skyline of the candidate points are computed, denoted by *skyline*. This can be done with any normal Skyline algorithm as they are described in previous parts. Finally, the Skyline of the candidate points are added to the Skyline of the query. Since the candidate points are restricted

to an area that is not influenced by any other Skyline point, except the outer bounds of the area, the Skyline points of the candidate points can just be added the the Skyline of the query without any further processing.

Figure 22.4 depicts the algorithm for candidate computation. The algorithm starts with the whole universe as the area for candidate points. With each new Skyline point it refines that area to be smaller than the previous area. If a Skyline points produces an area that is larger than the previous one that area is not taken into account. The only thing the algorithm has to make sure is that the deleted point is contained in the produced area.

```
1  function candidatePoints(Query Q, Dataset D, Point p)
2
3    // array describing the range, that is,
4    // the upper right corner of a rectangle
5    // with lower left corner (0, 0, ..., 0)
6    float range[];
7
8    // range is the cmplete data set
9    for (int j:= 1 to p.dim) do
10     range[j] = 1000000.00;
11   end;
12
13   // narrow range with each Skyline point
14   for (int i:= 1 to S.size()) do
15
16     Point q = S[i];
17
18     for (int j:= 1 to p.dim) do
19       range[j] = min(q[j], range[j])
20     end;
21
22   end;
23
24   // fetch points from the data set
25   Dataset cand = D.getPoints(range);
26
27   // check predicates of Query on retrieved points
28   for (int i:= 1 to cand.size()) do
29     if (!Q.predicates(cand[i])) then
30       cand.delete(cand[i]);
31     end;
32   end;
33
34   return cand;
35
36 end;
```

Figure 22.4: *Pseudo-code for the candidatePoints operation*

Initially, the whole data set is considered. With every Skyline point in the remaining Skyline the range R is narrowed, thereby eliminating all points from the data set that are dominated by the Skyline point in term. As a result R contains all points that are dominated by d, but which are not dominated by any other point in the Skyline. Retrieving the range R is done by native methods of the data set. In our case we used an R*-tree with a given method for retrieving points within an MBR.

22.2.2 Performance of Delete Operation

Table 22.2 shows the running times for the delete operation for 2- and 5-dimensional Skyline queries. We carried out 1000 deletes. Furthermore, Table 22.2 shows how many deletes (in percent) caused a reorganization of the Skyline S of the query, that is, the deleted point was part of the Skyline and after the deletion the Skyline had to be partly recomputed. All deletes were carried out sequentially. It can be observed that the running times are significantly higher for the independent and anti-correlated data value distributions; in particular, for the 5-dimensional queries. The reason is that the sizes of the Skyline S became very large in these cases.

	Time [s]	Reorgs [%]
Anti-corr.	0,07	1,6%
Corr.	0,02	0,1%
Unif.	0,03	0,7%
Corr.-groups	0,02	0,1%

(a) 2-dimensional Skyline

	Time [s]	Reorgs [%]
Anti-corr.	8,74	37,3%
Corr.	0,11	0,8%
Unif.	0,59	4,3%
Corr.-groups	0,04	0,4%

(b) 5-dimensional Skyline

Table 22.2: *Running time for a 1000 deletes, complete algorithm*

Table 22.3(a) shows the running times of the algorithm that recomputes the Skyline from candidate points if a point of the Skyline has been deleted. Comparing tables 22.2 and 22.3(a), it becomes clear that the cost of carrying out the delete operation is dominated by the cost to compute the Skyline of the candidates. Only for 5-dimensional Skyline queries with an independent or anti-correlated data set, the costs of the candidate algorithm are significant. The effectiveness of the candidate algorithm depends on the number of dimensions, the value distribution in the data set and on the specific Skyline point that is deleted. In many cases, the candidate algorithm restricts the set of candidates to a few points only. In (rare) extreme cases (anti-correlated, 5 dimensions, certain deleted points), the candidate algorithm returns a range R that contains 25 percent of the points in the data set as candidates.

	Time [s]	Reorgs [%]
Anti-corr.	6,00	37,3%
Corr.	0,09	0,8%
Unif. dist.	0,51	4,3%
Corr.-groups	0,02	0,4%

(a) with candidates

	Time [s]	Reorgs [%]
Anti-corr.	40,15	37,3%
Corr.	0,19	0,8%
Unif. dist.	1,72	4,3%
Corr.-groups	0,10	0,4%

(b) naive way

Table 22.3: *1000 deletes: Running time for recomputing the Skyline after a delete (5-dimensional)*

Table 22.3(b) shows the running times of a naive algorithm to recompute the Skyline after a delete has occurred. This algorithm recomputes the Skyline using the whole data set without

making use of the candidate algorithm in order to limit the number of candidate points to which the Skyline algorithm is applied. Comparing tables 22.3(a) and 22.3(b), it becomes clear that this naive algorithm has unacceptably high performance compared to the algorithm proposed in the previous section. For the tough cases (anti-correlated database, 5 dimensional queries), the naive algorithm is outperformed by our algorithm by a factor of 7.

CHAPTER 23

Multiple Continuous Skyline Queries

As we already have pointed out, there will not only one continuous Skyline query registered at the system, but many. And as we have seen in the previous chapter the cost for adjusting the Skyline of a query can be time-consuming in some cases. So the questions arises, if there is a more intelligent way to compute the residue for many queries of a changed data set, after either insertion or deletion of points.

23.1 Basic Considerations

Consider two Skyline queries. Query Q_1 asks for fast cars that are not too expensive and additionally restricts the price of the car to be not more than EUR 25,000, that is

```
SELECT *
FROM offer
WHERE price < 25000
SKYLINE OF speed MAX, price MIN;
```

or (in a different representation)

$$Q_1\,(\{\max(\text{speed}), \min(\text{price})\}, \text{price} < 25,000)$$

A second query Q_2 asks for all cars that are fast and cheap with a price not more than EUR 30,000. That is

```
SELECT *
FROM offer
WHERE price < 30000
SKYLINE OF speed MAX, price MIN;
```

or

$$Q_2\,(\{\max(\text{speed}), \min(\text{price})\}, \text{price} < 30,000).$$

Both queries are very similar. The only difference is that the second query is less restrictive to the price. The Skyline conditions are the same. The points that are interesting for the users that issued Q_1 and Q_2 to the system, the Skyline, will have many points in common. Only cars that obey the Skyline restrictions with a price higher than EUR 25,000 will shown to user Q_2 and not to both users.
If we could answer both queries "as one" for each update of the data set we certainly would save a lot of work. The only difference in the treatment of inserted or deleted points has to be made is for points $25,000 \leq \text{price} < 30,000$.

Now consider hundreds of queries differing only a "penny". Those queries could be answered in a sequential way, meaning each query is treated separately for each update of the data set (see Chapter 21 with Figure 21.2) or we could answer those queries "as one query" by making sure that the actual differences of the queries are handled properly saving possibly hundreds of similar operations.

23.1.1 Grouping Skyline Queries

The idea we will pursue in this part is to group similar Skyline queries into a hierarchy or tree. The grouping will be done in a way so that decisions made on the top level of the hierarchy will hold for all branches below. With individual Skyline queries at the branches it will be possible to discard, for example, a point that is inserted into the data set, right at the top level node, saying that this particular point is of no interest for all underlying Skyline queries. One decision for a number of Skyline queries.

23.1.2 Further Sections

Grouping will be done on similarities of Skyline queries, meaning that only similar Skyline queries will be grouped together. When it comes to comparing similarities, one should first take a look on how similarity can be described. So before we discuss the idea of grouping Skyline queries in detail we first take a look at measuring the differences between Skyline queries. Measuring distances and looking for similarities are two sides of the same coin. Chapter 23.3 deals with distances between Skyline queries.
Chapter 23.4 deals with speeding up the *insert* operation given in Figure 22.1. It is divided into two sections, one dealing with Skyline restrictions in the `SKYLINE OF` clause and one dealing with predicate restrictions in the `WHERE` clause. In another chapter (Chapter 23.5) we will combine the consideration on distances between Skyline queries and the consideration on speeding up the insert operation from Chapter 23.4 discussing contingent adjustments.
Grouping Skyline queries on similarities can be done in various ways, Chapter 23.6 introduces two simple approaches how to group queries into hierarchies, a first fit approach and a best fit approach.
A benchmark design section and a section with performance measurements are given in Chapter 23.7 and Chapter 23.8. A concluding sections (Chapter 23.9) depicts some possible enhancements to the delete operation.

Before we start with our discussion, the next section is a short shoot-out to recapture an observation which was mentioned a few times ago, the concept of *correlation groups*.

23.2 Correlation Groups

Our construct called *correlation group* arises from prior observations that differences in correlated dimensions of a data set have a lesser effect on the distance between two queries than differences in other dimensions.
A correlation group is simply a set of dimensions that have correlated data values, that is, if dimension one has a small value then dimension 2 must also have a small value. For our four data sets displayed in Chapter 6 this means:

- Anti-correlated data set: The dimensions have no correlation between each other. This means each dimension is its own correlation group. We say that the data set consists of $n = d$ correlation groups, d being the number of dimensions
- Correlated data set: All dimensions are correlated between each other. The data set consists of only one correlation group comprehending all dimensions.
- Uniformly distributed data set: The dimension values of this data set are uniformly distributed. This means that no assertion can be made about the correlation between dimensions. This data set also consists of $n = d$ correlation groups with d being the number of dimensions.
- Correlation groups data set: This kind of data set is designed to consist of exactly the number of correlation groups as we wanted if to have. The number of correlation groups depends on the building process of the data set. For the special 5-dimensional data set representing car offerings, this means that we have two correlation groups, a correlation group consisting of 2 dimensions and a correlation group consisting of 3 dimensions.

In general, we divide the dimensions of the data set into n correlation groups. A correlation group is a group of dimensions that have correlated data values. If no dimensions are correlated the data set contains $n = d$ correlation groups, that is, each dimension is its own correlation group. If all dimensions are correlated there is only one correlation group comprehending all dimensions of the data set.

23.3 Distance of Skyline Queries

In this section we will describe two approaches for measuring the distance between Skyline queries. The first one is an analytical approach, the second one is an empirical approach.

23.3.1 Analytical Distance Between two Skyline Queries

A Skyline query as depicted in the previous Section has four query parameters. These are

- the set of Skyline dimensions: The dimensions stated in the `SKYLINE OF` clause. These are the dimensions the Skyline is computed on. Short *sd* - skyline dimension.
- the set of predicate dimensions: The dimensions stated in the `WHERE` clause. Each dimension is restricted by a following range predicate. Short *pd* - predicate dimension.
- the lower bounds of the predicate dimensions, also stated in the `WHERE` clause. Short lpv - lower predicate value

- the upper bounds of the predicate dimensions, also stated in the `WHERE` clause. Short *upv* - upper predicate value.

A Skyline query having k range predicates can therefore be written as

$$Q(\{sd\},(lpv,pd,upv)_1,...,(lpv,pd,upv)_k)$$

Examples of Skyline queries are given in the above section, Q_1 and Q_2. In the following, however, we will use numbered dimensions, that is, dim1, dim2, ..., dim$n$, for Skyline dimensions and predicate dimensions. The data values in each dimensions are in the interval $[0.0,1.0[$. Please see Chapter 6 for details about the data sets we used throughout this thesis.

Q denotes a Skyline query of the set of all Skyline queries $\mathscr{Q}$. The distance function for the distance between two Skyline queries $\Delta(Q_1,Q_2)$ we define as:

$$\begin{aligned}\Delta(Q_1,Q_2) &= \delta_{\text{sky}}(Q_1,Q_2)+\delta_{\text{pred}}(Q_1,Q_2)+\\ &\quad \delta_{\text{lpv}}(Q_1,Q_2)+\delta_{\text{upv}}(Q_1,Q_2)\end{aligned} \tag{23.1}$$

It consists of 4 sub-distances. Each sub-distance is concerned with one part of a Skyline query.

Distance in Skyline Dimensions

The distance between Skyline dimensions, $\delta_{\text{sky}}(Q_1,Q_2)$ in Formula 23.1, is not quite obvious. What is the difference between two queries with Skyline dimensions *dim123* and *dim45* or *dim12* and *dim123*? The distance should take into account that certain dimensions are correlated and certain are not correlated. Distances in correlated dimensions are of less significance than differences in non-correlated dimensions. The distance between Skyline dimensions of two queries Q_1 and Q_2 is defined as

$$\delta_{\text{sky}}(Q_1,Q_2)=\begin{cases} 0 : sd_{Q_1}=sd_{Q_2} \\ 0 : sd_{Q_1}\simeq sd_{Q_2} \\ f_{\text{dim}} : \text{else}\end{cases} \tag{23.2}$$

Formula 23.2 determines the distance in Skyline dimensions to be zero if either the Skyline dimensions of both queries are equal or the Skyline dimensions of both queries are in the same correlation group. Both Skyline dimensions being in the same correlation group is denoted by $\simeq$. The distance is f_{dim} if the Skyline dimensions of both queries do not correspond. The penalty factor f_{dim} is explained later.
Looking at an example, we consider a 5-dimensional data set with correlation groups $c_1=\{\text{dim1},\text{dim2},\text{dim3}\}$ and $c_2=\{\text{dim4},\text{dim5}\}$. This could be the example given previously: {*price, insurance, taxes*} and {*speed, horse power*}. Considering only 2-dimensional Skyline definitions, that is, $sd=\{\text{dim12},\text{dim13},...,\text{dim45}\}$, there are all in all 10 different possibilities, Table 23.1 shows which Skyline queries have - Skyline dimensions wise - distance 0.
Looking at Table 23.1, two Skyline queries with have distance 0 if the two queries have the same Skyline dimensions (diagonal entries) or if they are within the set $\{\text{dim12},\text{dim13},\text{dim23}\}$.
For a correlated data set the table contains only 0, for anti-correlated and uniformly distributed data sets the values of the table are f_{dim}.
In a later section we will evaluate the possibilities of combining Skyline queries according to Table 23.1.

	dim12	dim13	dim23	dim14	dim24	dim34	dim15	dim25	dim35	dim45
dim12	**0**	**0**	**0**	f_{dim}	f_{dim}	f_{dim}	f_{dim}	f_{dim}	f_{dim}	f_{dim}
dim13	**0**	**0**	**0**	f_{dim}	f_{dim}	f_{dim}	f_{dim}	f_{dim}	f_{dim}	f_{dim}
dim23	**0**	**0**	**0**	f_{dim}	f_{dim}	f_{dim}	f_{dim}	f_{dim}	f_{dim}	f_{dim}
dim14	f_{dim}	f_{dim}	f_{dim}	**0**	f_{dim}	f_{dim}	f_{dim}	f_{dim}	f_{dim}	f_{dim}
dim24	f_{dim}	f_{dim}	f_{dim}	f_{dim}	**0**	f_{dim}	f_{dim}	f_{dim}	f_{dim}	f_{dim}
dim34	f_{dim}	f_{dim}	f_{dim}	f_{dim}	f_{dim}	**0**	f_{dim}	f_{dim}	f_{dim}	f_{dim}
dim15	f_{dim}	f_{dim}	f_{dim}	f_{dim}	f_{dim}	f_{dim}	**0**	f_{dim}	f_{dim}	f_{dim}
dim25	f_{dim}	f_{dim}	f_{dim}	f_{dim}	f_{dim}	f_{dim}	f_{dim}	**0**	f_{dim}	f_{dim}
dim35	f_{dim}	f_{dim}	f_{dim}	f_{dim}	f_{dim}	f_{dim}	f_{dim}	f_{dim}	**0**	f_{dim}
dim45	f_{dim}	f_{dim}	f_{dim}	f_{dim}	f_{dim}	f_{dim}	f_{dim}	f_{dim}	f_{dim}	**0**

Table 23.1: *Grouping matrix for 2-dimensional Skyline queries and 5-dimensional data set*

Distance in Predicate Dimensions

The distance between predicate dimensions, δ_{pred}, is defined in the same way as δ_{sky}. For computing the distance between two Skyline queries all dimensions that occur in the predicates of one query are treated together. For example, consider two queries,

$$Q_1\,(\{\min(\text{dim1}),\min(\text{dim2})\}, 0.00 < \text{dim1} < 0.75, 0.00 < \text{dim3} \leq 0.50)$$
$$Q_2\,(\{\min(\text{dim1}),\min(\text{dim2})\}, 0.00 \leq \text{dim2} < 0.50, 0.00 < \text{dim3} \leq 0.40)$$

the first query has range restrictions on dimensions dim1 and dim3, the second query on dimensions dim2 and dim3. The difference δ_{pred} is computed using the same formula as for computing the distance δ_{sky}.

Distance in Lower Values of Predicates

The distance between all lower values of the predicates, δ_{lpv}, is defined as:

$$\delta_{\text{lpv}}(Q_1, Q_2) = \sum_{i=1}^{p} |Q_1.\text{lv}_i - Q_2.\text{lv}_i| \tag{23.3}$$

with Q.lv meaning the lower value of the predicate bound of a query Q and p being the number of the corresponding predicates. Corresponding means that only predicates are taken into account that have the same predicate dimensions in both queries. For example, consider again the two queries given above. The first range restriction of each query is on a different dimension, dim1 for Q_1 and dim2 for Q_2, the second range restriction of each query is on the same dimension, dim3 for both queries. Only the second range restriction of both queries is taken into account for the difference in the lower value of the predicates. The difference in the lower value of the first range restrictions is overwritten by the difference in the predicate dimensions.

Distance in Upper Values of Predicates

Almost the same definition is used for the distance between all upper values of the predicates, δ_{upv}:

$$\delta_{\text{upv}}(Q_1, Q_2) = \sum_{i=1}^{p} |Q_1.\text{uv}_i - Q_2.\text{uv}_i| \tag{23.4}$$

with Q.uv meaning the upper value of the predicate bound of a query Q and p being the number of the corresponding predicates. Again, only predicates are taken into account that have the same predicate dimensions in both queries.

Weight of Sub-distances

Table 23.2 originates from our empirical testing. What we found out there, in a nutshell, results in a weight list of sub-distances.

Weight	Distance
δ_{sky}	3
δ_{pred}	2
δ_{lpv}	1
δ_{upv}	1

Table 23.2: *Weight of distances*

The table should be read as follows: If two queries differ in Skyline dimensions, δ_{sky}, then all other dimensions normally are of lesser importance. This is also the reason why we compute the distances in lower and upper predicate values only on corresponding dimensions. Differences in lower and upper predicate values of non-corresponding dimensions are taken care of with the distance in predicate dimensions.

This approach of determining the distance of Skyline queries is called the Query Analyzing approach.

23.3.2 Empirical Distance Between two Skyline Queries

This is another approach of determining the distance between two Skyline queries. This approach does not check query parameters to determine the distance but checks the queries for similarities after a certain number of points have been inserted. There are different ways to check similarities of Skyline queries. One could monitor incoming points and note how each query treats the points. Treating means if the query discards the point or accepts the point into its Skyline. Each query needs a structure to hold the information about each points, for example, a boolean vector where “true” means that point was accepted into the Skyline and “false” meaning that the point was discarded. Entry 1 in each structure stands for the treatment of point 1 and so on. After a certain number of inserts the structures of all queries are taken into account when it comes to determining the distance between the Skyline queries.

Instead of keeping track what each query is doing with the inserted points, another way would be looking at the intersection of Skyline points from the different queries after a certain number of inserts. If the intersection contains more points than a certain limit, for example, more than 30% of the points of the smallest Skyline is in the intersection of the two queries, Skyline queries can be grouped together. The advantage of this second approach is that no additional structure is needed for determining the distance between the two queries. As we will see later on, intersections between Skylines have to be computed anyway, so this second approach uses techniques which will be used later again. So we use the second approach to determine the distance between Skyline queries.

This approach requires a "learning step". We define the learning step as a detached step that takes places before any "real" points arrive at the system. It consists of two parts. In the first part the learning points are inserted into n queries, yielding Skyline points for each query. In the second part, the comparison step, queries are checked by their Skyline intersection.

The distance $\Delta(Q_1, Q_2)$ between two queries Q_1 and Q_2 is defined as

$$\Delta(Q_1, Q_2) = 1 - \frac{|Q_1 \cap Q_2|}{\min(|Q_1|, |Q_2|)}$$

with $|Q|$ meaning the number of points in the Skyline. The distance takes into account the number of points in the intersection of Skyline points and the minimum number of points in each individual Skyline.
For this approach the distance between two queries is measured in the number of points in the Skyline intersection.

23.4 Speeding up the Insert Operation

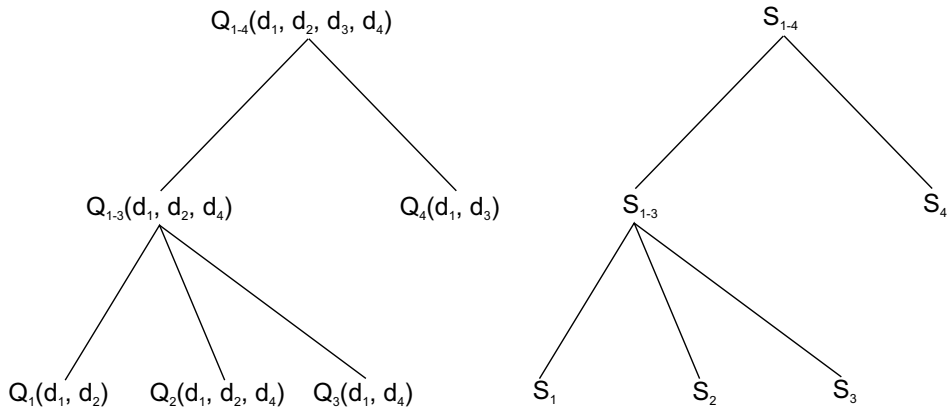
Now that we have talked about distances between queries, we want to show an idea how to answer many Skyline queries "as on query".
The idea is to group queries and their Skyline points (S_Q) in order to answer queries together. In other words, given queries Q_1, Q_2, ..., Q_m and corresponding sets of Skyline points S_{Q_1}, S_{Q_2}, ..., S_{Q_m}, and a point p that is inserted, we construct a query $Q_{1\ldots m}$ and a set of Skyline points $S_{Q_{1\ldots m}}$ with the following property:

$$\begin{aligned} \text{insert}(p, Q_{1\ldots m}) &= \text{false} \Longrightarrow \\ \forall\, i : \text{insert}(p, Q_i) &= \text{false} \end{aligned}$$

With *insert* we mean the insert operation depicted in Figure 22.1. This way, we only need to investigate Query $Q_{1\ldots m}$ rather than inspecting each of the m queries individually in order to filter out a new data point p.

The idea of grouping queries can be applied in different ways to speed up the insert operation for sets of queries by using adaptions of the same techniques proposed in the following section. However, we will concentrate on quickly detecting for which queries a new point is *not* relevant because we believe that this case will be the most important case in practice. Another way, for example, would be applying the grouping in a way to find out if the newly inserted point is relevant.
First, we will focus on indexing queries that have an empty `WHERE` clause. After we have developed an idea for grouping queries without predicates we will take a look at processing predicates for grouped queries. As we will see in a later section, processing predicates is somewhat orthogonal to processing Skyline conditions, meaning both computations can be split into two separated steps.

Note: The terms "indexing queries" and "grouping queries" can and will be used synonymously throughout this thesis. The reason for this is that we arrange the group of queries that can be answered simultaneously in a tree structure. This tree structure is then used to quickly classify subtrees as important or unimportant for further processing. This is exactly how indices work.



(a) Query definitions (b) Set of points

Figure 23.1: *Hierarchy of filters*

23.4.1 Grouping Skyline Queries

Let Q_1, ..., Q_m be a set of queries. Let Q_i be characterized by S_{Q_i}, the Skyline for Q_i, and sd_{Q_i}, the set of dimensions specified in the `SKYLINE OF` clause of query Q_i. Then, $Q_{1...m}$ can be characterized as follows:

$$S_{Q_{1...m}} := \bigcap_{i=1,...,m} S_{Q_i} \tag{23.5}$$

$$sd_{Q_{1...m}} := \bigcup_{i=1,...,m} sd_{Q_i} \tag{23.6}$$

Recall the car market example from Chapter 2 with used cars in the data set and users interested in different car properties.

Model	Price	Age	Speed
BMW 330 xd	EUR 30,000	5 years	200 km/h
Ford Focus	EUR 8,000	3 years	150 km/h
Toyota Avensis	EUR 10,000	4 years	170 km/h

Table 23.3: *Used car market: data set*

Table 23.3 shows the data set with three cars and Table 23.4 shows two users with the dimensions they are interested in and the Skyline for these dimensions.

	Preferences	Interesting Offers - Skyline
User 1	{ price, age }	{ Ford }
User 2	{ price, speed }	{ BMW, Ford, Toyota }

Table 23.4: *Used car market: interesting dimensions*

We can group the two (base) query, Q_1 for user 1 and Q_2 for user 2, and construct a query Q_{1-2} for both users. This query would be characterized with the Skyline dimensions the query is defined on, $sd_{Q_{1-2}}$, and the resulting set of interesting points, the Skyline $S_{Q_{1-2}}$:

- $sd_{Q_{1-2}}$:= {price, age, speed}

- $S_{Q_{1-2}}$:= {(Ford Focus, EUR 8,000, 3 yrs, 150 km/h)}

Now consider a new car offer arriving at the system

(Alfa Spider, EUR 12,000, 4 years, 140 km/h)

Using the constructed query Q_{1-2}, its Skyline $S = \{\text{Ford}\}$, and its comparison function $c_{Q_{1-2}}$ that considers all three dimensions), it is possible to immediately find out that the offer for the Alfa is irrelevant for both users because the Alfa costs more, is older and slower than the Ford. Thus saving one Skyline comparison in this example.
If a new offer is posted that is incomparable to the Ford in terms of price, age, and speed, then this offer must be considered by the base queries, Q_1 and Q_2, individually in order to find out for which users this offer is relevant. In this case, the grouping is not effective.

Just from this short example, it is quite clear, the larger the root node Skyline ($S_{Q_{1-2}}$) is the greater the chance to discard a new point. In Chapter 23.5 we will explain the relationship between the previous distance discussion and the discussion of grouping here. In a nutshell, the lesser the distance of Skyline queries, the larger the root Skyline.

The correctness of filtering with query Q_{1-2} and S_{1-2} can be proven considering the following observation:

$$sd_{Q_i} \subset sd_{Q_j} \wedge c_{Q_j}(p,q) \in \{less, equal\} \Longrightarrow$$
$$c_{Q_i}(p,q) \in \{less, equal\}$$

In other words, if a point p dominates q or is equal to q according to Q_{1-2}, point p can be used to filter out q for queries Q_1 and Q_2. The proof for this observation is straightforward and uses the definitions of c_{Q_j} and c_{Q_i} which depend on sd_{Q_j} and sd_{Q_i} □

Just like Skyline queries, groups of Skyline queries can be combined, using the same rules. This way, it is possible to construct hierarchies of filters. Figure 23.1(a) shows four continuous Skyline queries $Q_1, ..., Q_4$ in a 4-dimensional data space. Note: For ease of presentation dimension dim_1 is shorten to d_1, dim_2 to d_2 and so on. Q_1 involves dimensions dim_1 and dim_2 (denoted as $Q_1(\text{dim}_1, \text{dim}_2)$), Q_2 involves dimensions dim_1, dim_2, and dim_4, etc. Q_1, Q_2, and Q_3 are grouped to $Q_{1-3}(\text{dim}_1, \text{dim}_2, \text{dim}_4)$. Q_{1-3} and Q_4 are in turn grouped to $Q_{1-4}(\text{dim}_1, \text{dim}_2, \text{dim}_3, \text{dim}_4)$. Figure 23.1(b) shows the corresponding tree that represents the sets of points associated to each Skyline query and each group of Skyline queries. S_{1-4} contains all points of the intersection of $S_{Q_1} \cap ... \cap S_{Q_4}$. S_{1-3} contains all points of the intersection of $S_{Q_1} \cap S_{Q_2} \cap S_{Q_3}$ that are not in S_{1-4}. S_1 contains all points of S_{Q_1} that are not in $S_{Q_{1-3}}$ or $S_{Q_{1-4}}$, and so on. Thus, the Skyline of Q_1 is $S_{1-4} \cup S_{1-3} \cup S_1$, a walk along the tree branches from the root to the leaf.

Using the trees of Figure 23.1, the insert operation is executed for a new data point p as follows. First, the comparison function of Q_{1-4}, $c_{Q_{1-4}}$, is used to compare p with all points of S_{1-4}. If one of the points of S_{1-4} dominates p according to $c_{Q_{1-4}}$, then p is not relevant for any query and the execution terminates. Otherwise, the comparison function of Q_{1-3} is used in order to compare p with all points of $S_{1-4} \cup S_{1-3}$. Again, if p is dominated by any point in this step, we do not need to consider p for any of the first three queries. In the worst case, we need to go down to the leaf level where each query is processed individually. In this case, the additional effort to consider grouped Skyline queries is wasted work.

The constructed filters do not only serve for discarding new points, they can also be used to quickly identify points that are interesting for a group of queries. If p dominates one point of S_{1-4} then it is guaranteed that p is a Skyline point for all Skyline queries $Q_1, ..., Q_4$. Again, this assertion holds at every level of the filter hierarchy. Consequently, we have two stop conditions for our algorithm and only in the case of p being incomparable we have to recursively go through the complete hierarchy. These two stop conditions can be summed up with

- If a newly inserted point p is dominated by a point in one of the Skylines of the hierarchy, it can be discarded for all Skylines that are below the Skyline which discards the point, that is, the subtree in the hierarchy.
- If a newly inserted point is interesting for on of the Skylines of the hierarchy, it is also interesting for all Skylines that are below the Skyline for which the inserted point is interesting.

For the insert operation, the sets of points associated with each Skyline query, that is, the Skyline of each query, and each group of queries must be updated when a new interesting point is inserted. In particular, the intersection property must be maintained. Maintaining this property, however, is straightforward and cheap.
However, not all points need to be considered using the comparison functions of the grouped queries. Using grouped Skyline queries can be seen as a pre-filtering step and limiting the amount of work done in such a pre-filtering step, limits the overhead carried out in bad cases, that is, when the pre-filter fails.

23.4.2 Predicate Processing for Grouped Skyline Queries

After having looked at processing restrictions given in the `SKYLINE OF` clause of a Skyline query, we want to take a look at restrictions given in the `WHERE` clause of a Skyline query. Any predicate given in the `WHERE` clause restricts the underlying data set, for example, if

Q_1(*some Skyline conditions*, model = 'Ford').

only Ford cars are of interest to the user and all other brands are filtered out. The number of data points is (possibly) greatly reduced.

For simplification, we allow only one predicate per Skyline query, that is, a range restriction on one dimension of the data set. However, everything said here, works also for multiple range restrictions on different dimensions and the theory in the previous sections is also laid out with more than one predicate per Skyline query. As previously said, a restriction in the `WHERE` clause can greatly reduce the amount of data to be checked for Skyline affiliation. For an inserted point this means, if it is discarded by the predicate the rest of the insert operation can be skipped. Testing a range predicate takes certainly less time than executing Skyline comparison functions, so for single queries predicates should be checked before any Skyline operation. This is depicted in Figure 22.1 (insert operation). However, checking many predicates for grouped queries can be time consuming and some predicates might be unnecessary to check, for example, when the insert operation at Q_{1-4} (Figure 23.1) discards an incoming point then the predicates of $Q_1, ..., Q_4$ do not have to be checked.
We propose a simple strategy to evaluate predicates of grouped Skyline queries: For each incoming point we perform the insert operation on the root node, for example, node Q_{1-4}, without

checking the predicates. If it returns "less" the point is discarded. No query predicates have to be evaluated, we are done for this point. If the insert operation on the root node returns "greater", "equivalent" or "incomparable" we check the predicates of all leaf-level queries and keep track of each predicate that was checked so we do not evaluate a predicate twice.
If a predicate of a Skyline query returns false that particular Skyline query needs not to be check by a later insert operation. If all predicates of Skyline queries of a subtree evaluate to false that particular subtree needs not to be checked by a later insert operation. All other queries are checked by the method proposed in the previous section. If all predicates evaluate to false, the incoming point is also discarded. No other operation has to be performed for that point.
Treating predicates has been the subject of numerous work. One can think of many way to evaluate predicates of Skyline queries, for example, indexing predicates would also be possible. Methods to index predicates of continuous queries has been the subject of earlier work (for example, [HCH+99] and [FJL+01]). Those techniques are directly applicable to continuous Skyline queries.

23.5 Finding Good Groups for the Insert Operation

23.5.1 Empirical Testing

Extensive empirical tests preceded the formulation of above facts. Our testing included the variation of all query parameters, that is,

- the Skyline dimensions,
- the predicate dimensions,
- the lower bound of the range predicates, and
- the upper bound of the range predicates,

one at a time, leaving all other query parameters untouched. By doing that we developed an intuition how sensible the grouping reacts to changes in query parameters. The previous "distance" sections and also the following sections are the results derived from these tests.

23.5.2 Evaluation of Skyline Distances

As one can easily see, the grouping idea from the previous section works best the more Skyline points reside in the root node of the Skyline query tree. The bigger the Skyline of the root node the better the filtering, the bigger the chance to discard a new point because of it is dominated by s Skyline point.
To makes this clear, consider the 2-dimensional correlated data set (Chapter 6). The Skyline of this particular data set contains only one Skyline point, *the* Skyline point. Now think a Skyline query hierarchy where this one Skyline point resides in the root Skyline. All incoming points would be discarded right away since there is no second Skyline point. All Skyline queries that were grouped together in this query hierarchy would be answered by a single insert operation per incoming point on the root node.
So grouping works well if there is a sufficient number of Skyline points in the root Skyline of the tree of grouped queries. The root Skyline contains all points that are in common to all

queries grouped together in the tree. To end up with a sufficient number of Skyline points in the root Skyline the underlying queries need to be similar. Which queries are similar or have a small distance was discussed in Chapter 23.3.

We take a look at a correlated data set. According to what we said previously, all queries can be - Skyline wise - grouped together since all dimensions are correlated (hence a correlated data set). We checked the intersection of Skyline points for a couple of combinations of Skyline queries without any predicates. Table 23.5 shows the size of the intersection of 10,000 points.

Skyline dims	⋂	Skyline dims	⋂	Skyline dims	⋂
dim12/dim13	**1**	**dim13/dim23**	**1**	dim23/dim24	3
dim12/dim14	1	dim13/dim24	1	dim23/dim25	4
dim12/dim15	2	dim13/dim25	0	dim23/dim34	2
dim12/dim23	**5**	dim13/dim34	2	dim23/dim35	2
dim12/dim25	4	dim13/dim35	1	dim23/dim45	1
dim12/dim34	2	dim13/dim45	2	...	...
dim12/dim35	1	dim13/dim14	1		
dim12/dim45	2	dim13/dim15	3		

Table 23.5: *Intersection size of combinations of Skyline queries*

For example, if we group two Skyline queries with Skyline dimensions sd_1 = dim12 and sd_2 = dim23 we have 5 Skyline points in the intersection of Skyline points (remember the whole correlated data set consists of 100,000). As long as we only group Skyline queries having sd = dim12 or sd = dim23 these 5 Skyline points will "survive". However if we group a Skyline query having sd = dim13 these 5 Skyline points will be reduced to one Skyline point at the most since Skyline queries with sd_1 = dim12 and sd_2 = dim13 have only one Skyline point in common. The more different Skyline queries we group together the less the chance to have a sufficient number of Skyline points in the Root-Skyline for efficient filtering. Therefore we decided to allow only two different Skyline queries - Skyline wise - to be grouped together. After a different Skyline query is grouped to a tree, the penalty factor, f_{dim}, is altered to a higher value preventing a second different Skyline query to be grouped to the tree. For example, the first query in the tree t has $sd_1 = \text{dim12}$. The penalty factor $f_{\text{dim}} = a$ allowing a Skyline query with $sd_2 = \text{dim23}$ to be grouped to the tree t. Now, since $sd_1 \neq sd_2$ the the penalty factor f_{dim} is altered to be $f_{\text{dim}} = 2 \cdot a$, preventing a third Skyline query with $sd_3 = \text{dim34}$ being grouped to the tree t.

23.5.3 Evaluation of Predicate Distances

Following our prior empirical testing it seems not a good idea to group Skyline queries which differ in predicate dimensions. Differences in predicate dimensions might yield similar Skyline points, but initial tests showed that the Skyline points are not similar. So, for our performance experiments we forbid grouping with different predicate dimensions.

Also big differences in the lower bound of a predicate seems not a good idea since changes here mean changing the origin of the Skyline computation and thus yielding different Skyline points. For our performance experiments only small distances for δ_{lpv} where allowed.

23.6 First Fit and Best Fit Grouping

After we have discussed how to speed up the insert operation, that is, building a hierarchy or tree of Skyline queries and their associated Skylines, we want to take a look at how queries can be grouped to a tree for processing insert operations. We study two different approaches to assign queries to trees. Both approaches can be classified as greedy algorithms.
We take a look at a FirstFit approach that assigns a query to the first tree that fulfills certain rules and we take a look at a BestFit approach that assigns a query to the tree where (a) certain rules are fulfilled and (b) the distance between the new query Q and the tree T is minimal, that is, $\Delta(Q, \rho_T)$ is minimal. Figure 23.2 depicts the FirstFit approach, ρ_T specifies the representative of the tree T. This representative is also a Skyline query, for example, the first Skyline query that was associated with the tree.
Note: In the figure ρ_T is depicted as T[j]. For each query $Q \in \mathcal{Q}$ ($\mathcal{Q}$, the set of all queries, is represented by the array Q[]) the algorithm looks at all trees $T \in \mathcal{T}$ and computes the distance between Q and the representative of the tree. If the distance is less than $|\varepsilon|$ the query is assigned to the tree. This repeats until all queries are assigned. If a query does not fit to any tree, that is, $\forall\, T \in \mathcal{T}, \Delta(Q, \rho_T) > |\varepsilon|$, it is made he initial query of a new tree.

```
1  Input:     Query Q[]
2  Input:     int no_of_queries
3
4  procedure FirstFitGrouping(float epsilon)
5
6    Tree T[];
7    int no_of_trees = 0;
8
9    for (int i:= 1 to no_of_queries) do
10     for (int j:= 1 to no_of_trees) do
11
12       if (distance(Q[i], T[j]) <= abs(epsilon)) then
13         T[j].assign([i])
14       end;
15
16       break;
17
18     end;
19
20     T.newTree(Q[i]);
21
22   end;
23
24 end;
```

Figure 23.2: *FirstFit grouping algorithm*

Figure 23.3 shows the BestFit approach. Instead of assigning the query Q to the first tree that meets the requirements, all trees in $\mathcal{T}$ are checked for the best fitting tree. The query is either assigned to the best tree or it is the initial query of a new tree.

```
1  Input:     Query Q[]
2  Input:     int no_of_queries
3
4  procedure BestFitGrouping(float epsilon)
5
6    Tree T[];
7    Tree t_min;
8    float epsilon_min;
9    int no_of_trees = 0;
10   bool found;
11
12   for (int i:= 1 to no_of_queries) do
13     for (int j:= 1 to no_of_trees) do
14
15       found = false;
16
17       if (distance(Q[i], T[j]) <= abs(epsilon)) then
18         (epsilon_min, t_min) = min((epsilon_min, t_min),
19                                    (distance(Q[i], T[j]), T[j]));
20         found = true;
21       end;
22
23     end;
24
25     if(found) then
26       t_min.assign(Q[i]);
27     else
28       T.newTree(Q[i]);
29     end;
30   end;
31
32 end;
```

Figure 23.3: *BestFit grouping algorithm*

Note: The number of Skyline queries does not change in our scenario, that is, the set of Skyline queries within the system is static.

23.7 Benchmark Design

23.7.1 Questions to be Answered

There is one questions which arises immediately when talking about grouping Skyline queries: Does grouping work? Meaning, can we benefit from grouping by saving time to process the Skyline queries? Along with this question, there are more questions that should be answered by a benchmark. These questions include:

- How much improvement is made through grouping concerning the running time against a sequential answering of queries?
- How much improvement is made through grouping concerning other parameters, memory usage, for example?
- How well do the alternative approaches, analyzing the Skyline query parameter or learning similarities, to group queries work?
- Does grouping scale for large numbers of queries and high volumes of data?

In order to answer these questions, we carried out a series of comprehensive performance experiments using the techniques described in the previous sections with a varying number of queries and different data value distributions distributions. This section describes the benchmark and experimental environment used. The following section presents the results of the performance experiments.

23.7.2 Query Workloads

The data that had to be filtered (described in Chapter 6) consisted of multi-dimensional floating point values with 6 significant digits after the decimal point. Correspondingly, the range predicates in the Skyline queries also involved floating point values as their upper and lower bounds.
Query generation involved the generation of the `WHERE` and `SKYLINE OF` clauses of a query.

The `SKYLINE OF` clause is generated as follows: First, the number of Skyline dimensions is chosen, for example, Skyline queries involving two dimensions in the `SKYLINE OF` should be generated. Then all possible combinations of Skyline dimensions are generated according to the number of dimensions contained in the data set that should be used for the measurement, for example, a five-dimensional data set should be used. Those combinations of Skyline dimensions we call candidate Skyline combinations. Then the Skyline dimensions are chosen randomly, using a Zipf distribution, among the candidate Skyline combinations.

For example, consider a five-dimensional data set with all two-dimensional combinations of Skyline dimensions, the following were the generated candidate combinations of Skyline dimensions: dim12, dim13, dim14, dim15, dim23, dim24, dim25, dim34, dim35, and dim45.

In the experiments reported in the next section, two-dimensional and 5-dimensional Skyline queries were studied in 5-dimensional and 10-dimensional data sets, respectively. For the two-dimensional Skyline queries, all ten dimension combinations were candidate combinations. For the five-dimensional Skyline queries all 252 dimensions combinations were candidates for the `SKYLINE OF` clause. The selection of dimension combinations was carried out randomly using a Zipf distribution; as a result, certain dimensions were more popular than others (for example, the price of a car is usually relevant for all potential buyers).

In all experiments in this chapter the `WHERE` clause contained only one range predicate on one dimension. Again, the dimension of that predicate was chosen randomly among the 5 or 10 dimensions possible according to the data set using a Zipf distribution. The upper and lower bounds of these predicates were generated randomly using a uniform or Gaussian distribution

	Range of values	Distribution
Skyline dimensions	All 2-dimensional combinations in a 5-dimensional data set (10 possibilities): dim12, ..., dim45	Zipf distribution with $\theta = 2$. Probability for the i value is $P_i = \frac{1}{i^2}$
	All 5-dimensional combinations in a 10-dimensional data set (252 possibilities): dim12345, ..., dim678910	Zipf distribution with $\theta = 2$. Probability for the i value is $P_i = \frac{1}{i^2}$
Predicate dimensions	dim1 to dim5	Zipf distribution with $\theta = 2$. Probability for the i value is $P_i = \frac{1}{i^2}$
	dim1 to dim10	Zipf distribution with $\theta = 2$. Probability for the i value is $P_i = \frac{1}{i^2}$
Lower predicate values	Between 0.2500 and 0.5000. Significance $s = 4$, that is, four digits after the decimal point	Uniform
	Between 0.1500 and 0.3500. Significance $s = 4$, that is, four digits after the decimal point	Gaussian
Upper predicate values	Between 0.5000 and 1.0000. Significance $s = 4$, that is, four digits after the decimal point	Uniform
	Between 0.6500 and 0.8500. Significance $s = 4$, that is, four digits after the decimal point	Gaussian

Table 23.6: *Overview of benchmark parameters*

with mean, deviation, and significance as shown in Table 23.6. That table also summarizes the other benchmark parameters. In all, the different query workloads study a wide range of possible workloads: workloads in which many queries can be grouped because they have similar filtering effects and workloads in which grouping is not beneficial because the queries have highly varying filtering characteristics.

23.7.3 Data Set Generation

The data generation follows the rules given earlier in Chapter 6. Our special 5-dimensional data set - the correlation groups data set - for the Electronic Market Place scenario (Chapter 2) will also be used here. This data set resembles the fact that some dimensions in a multi-dimensional data set are correlated and some are not. For example, consider a 5-dimensional data set describing the significant attributes of a car. We can identify two correlation groups,

that is, {*price, insurance, taxes*} and {*speed, horse power*}. The dimensions are split into two correlation groups. Within a group, values are correlated, that is, normally a higher price results in higher a higher insurance rate and speed and horse power are also correlated. On the other hand horse power and price have not necessarily a correlation.

23.7.4 Generating Query Sets and Benchmark Execution

As mentioned in a previous section, there is a set of points, S_Q associated to each Query Q. A new point is compared to the points in S_Q according to c_Q in order to find out whether the new point is relevant or not. This set of points is determined for each query and experiment by generating a database of 10,000 points. For Query Q, S_Q is set as the result of applying Q to that database of 10,000 points as if Q were a regular (rather than a continuous) Skyline query simulating an already running query system.
After that, another 90,000 points are generated and these 90,000 points are filtered using all queries and their associated set of points and the time to filter these 90,000 points is measured. In all, an experiment involved carrying out the following steps:

- Generate queries.
- Compute the Skyline of the 10,000 initial data set points for each query.
- Analyzing approach: Group queries into trees and compute the intersection of Skyline points where necessary.
- Learning approach: Generate additional sample points for filtering and group queries into trees and compute the intersection of Skyline points where necessary.
- Start time measurement.
- Test the filtering of 90,000 points.
- Stop time measurement.

This procedure is repeated several times and the overall average time is reported in order to get stable results.

23.7.5 Experimental Environment

All measurements are carried out on an Intel Pentium 4 machine at 3.2 GHz with 2 GB of main memory. If not stated otherwise, the parameters for the Learning and query Analyzing approach are set in all experiments as shown in Table 23.7. In one experiment, the number of sample points for the Learning approach is varied in order to study the sensitivity of this approach towards that parameter. We do not show experiments that study the sensitivity of the approaches to the other parameters because this sensitivity was very low in general.

	Analysis	Learning
f_{dim}	0.2	n.a.
δ_{sky}	0.1	n.a.
ε	0.41 ($\delta_{\text{sky}} = 0.20$, $\delta_{\text{lpv}} + \delta_{\text{upv}} = 0.21$)	0.33
Sample points	n.a.	10000

Table 23.7: *Default Parameter Settings*

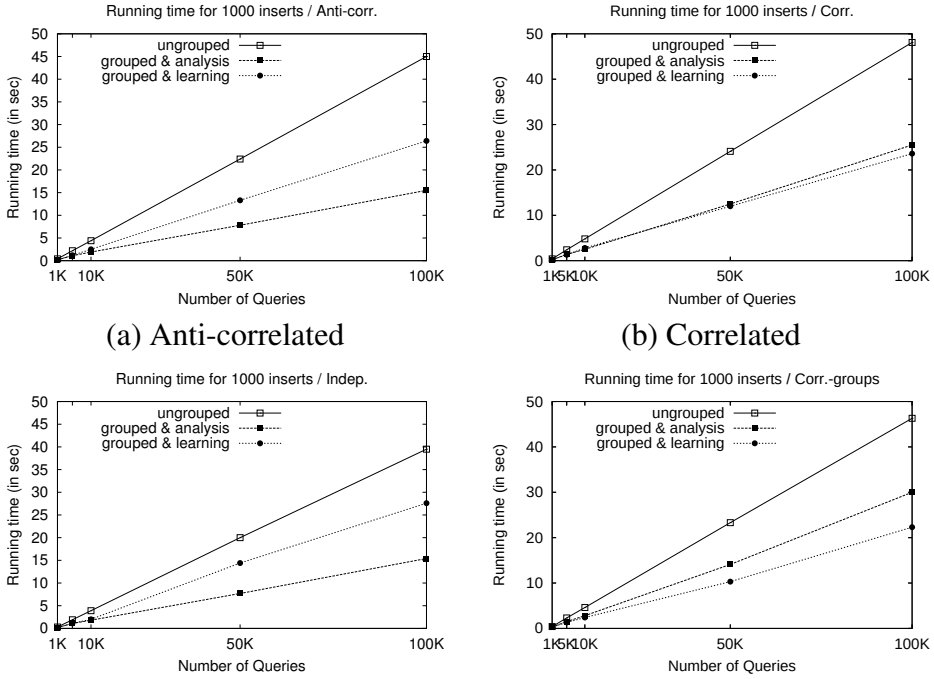
23.8 Performance of the Insert Operation

In this performance section we take a closer look at our grouping approaches. These two grouping approaches arise from our discussion about determining the distance between two Skyline queries. We discussed an analytical approach for determining the distance between two Skyline queries, in the following this approach is called the *Analyzing approach* or *grouping by analyzing*; and an approach which looks for intersections between Skylines, an empirical approach for determining the distance between two Skyline queries, the *Learning approach* or *grouping by learning*. Both approaches were discussed in Chapter 23.3.
Our performance measurements showed no difference between a FirstFit grouping and a BestFit grouping (as described in Chapter 23.6). So we only report the running times for the FirstFit approach here.

23.8.1 Two-dimensional Query Workload

Uniform Predicate Bound Distribution

We start our performance section with 2-dimensional Skyline queries and uniform predicate bound distribution. We used 5-dimensional data sets with anti-correlated, correlated, and uniformly distributed data value distribution. We also used our special correlation groups data set. We varied the number of queries from a 1000 queries to 100,000 queries. Figure 23.4 displays the results for ungrouped computation, grouped computation using the Analyzing approach and grouped computation using the Learning approach.
For the Analyzing approach we used a rule which allowed the following deviations: a maximal deviation of ± 0.01 for the lower predicate bound and a maximal deviation of ± 0.20 for the upper predicate bound, predicate dimensions had to be equal. For Skyline dimensions queries with two different combinations of Skyline dimensions were allowed, that is, the distance in Skyline dimensions was set to $\delta_{\text{sky}} = 0.2$ and $f_{\text{dim}} = 0.1$. After a query with a different combination of Skyline dimensions was assigned to the tree f_{dim} was set to 0.4, preventing a third different combination of Skyline dimensions be present in the tree. The overall distance of two Skyline queries was therefore $\Delta(Q_1, Q_2) \leq |0.41|$ and $\Delta(Q_1, Q_2) \leq |0.61|$.
For the Learning approach the number of learning points was 10,000 and 66 percent of the Skyline points of the minimal Skyline had to be in the intersection, that is, $\Delta(Q_1, Q_2) \leq 0.33$.
Grouping is the clear winner for this setup. No matter what data set was used grouped computation wins by a factor between 1.5 and 2.5 compared to the ungrouped computation. Looking at the correlated and correlation-groups measurements one can see that grouping by learning is slightly better than grouping by analyzing. A tree built by the learning approach always has a sufficient number of Skyline points in the top node of the tree to do the pre-filtering. The Analyzing approach sometimes shoots itself in the foot by putting a query into a tree that eliminates



(a) Anti-correlated (b) Correlated

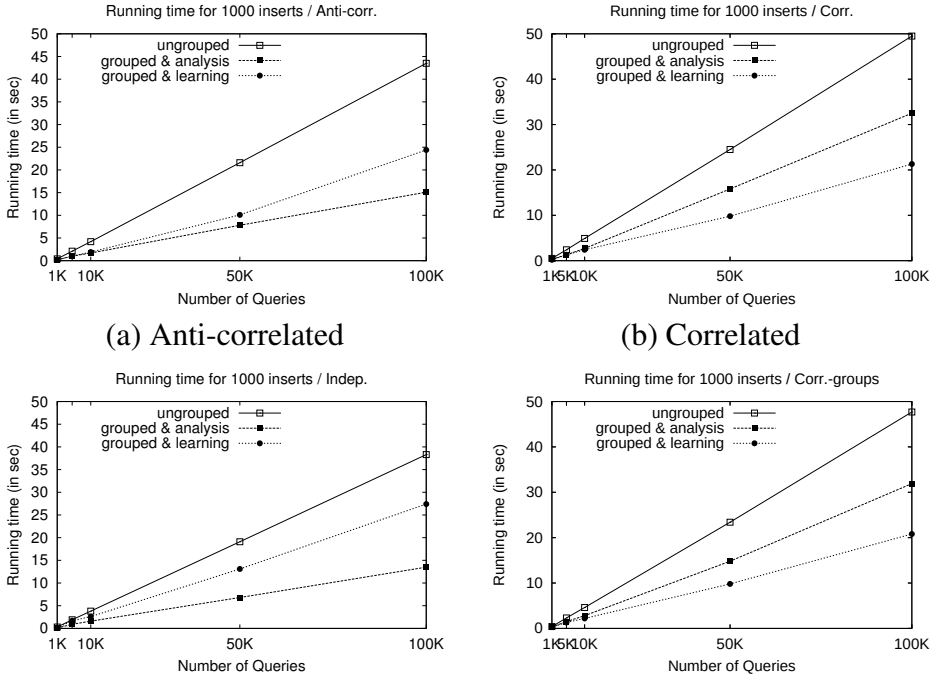
(c) Independent (d) Corr.-groups

Figure 23.4: *Running time for up to 100K queries, uniform predicate bound distribution*

most of the top node Skyline points.
Nevertheless both approaches yield similar tree structures which can be seen by looking at the number of trees: for example, 2276 trees (100,000 queries, correlated data set, Analyzing approach); 2214 (100,000 queries, correlated data set, Learning approach).
Looking at the anti-correlated and uniformly distributed data set, grouping by analyzing wins even against the Learning approach. For these two data sets no difference in Skyline dimensions were allowed in trees. This brings some advantages for the Analyzing approach since it is now more efficient than the Learning approach.

Gaussian Predicate Bound Distribution

The second measurement has almost the same setup as the first one. The only difference is that instead of a uniform predicate bound distribution a Gaussian predicate bound distribution is used. Again, for the Analyzing approach a maximal deviation of ± 0.01 for the lower predicate bound and a maximal deviation of ± 0.20 for the upper predicate bound was allowed, predicate dimensions had to be equal; Skyline dimensions had the same restrictions as given above, that is, the distance in Skyline dimensions was set to $\delta_{\text{sky}} = 0.2$ and $f_{\text{dim}} = 0.1$. After a query with a different combination of Skyline dimensions was assigned to the tree f_{dim} was set to 0.4, preventing a third different combination of Skyline dimensions be present in the tree. The overall distance of two Skyline queries is therefore $\Delta(Q_1, Q_2) \leq |0.41|$ and $\Delta(Q_1, Q_2) \leq |0.61|$. Figure 23.5 shows the results for all four data sets (anti-correlated, correlated, uniformly distributed, and correlation groups).
The Learning was done with the same parameters as above: number of learning points was 10,000 and 66 percent of the Skyline points of the minimal Skyline had to be in the intersection, that is, $\Delta(Q_1, Q_2) \leq |0.33|$.



(a) Anti-correlated (b) Correlated

(c) Independent (d) Corr.-groups

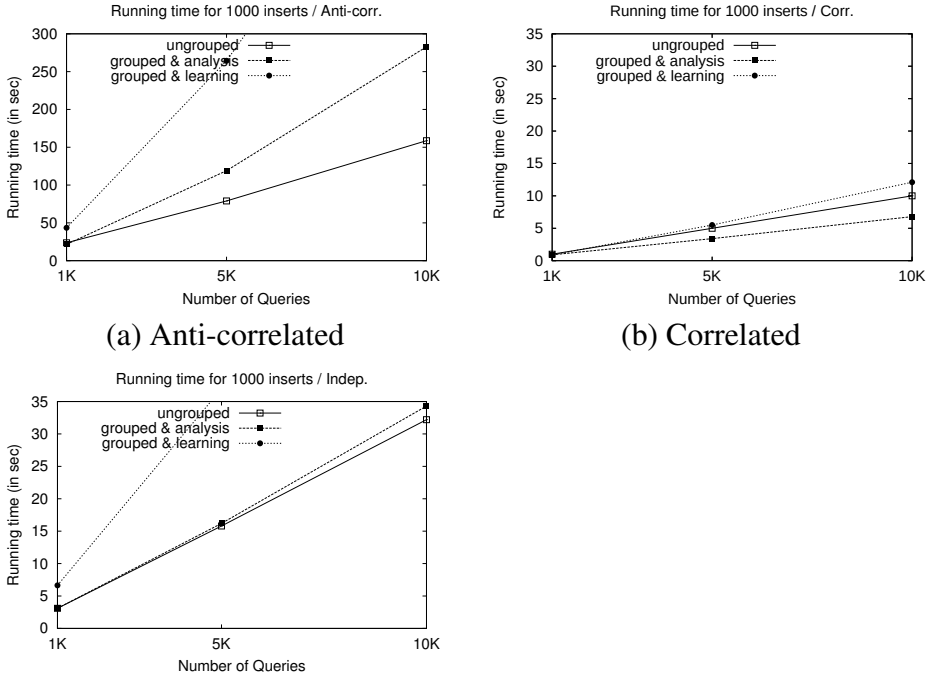
Figure 23.5: *Running time for up to 100K queries, Gaussian predicate bound distribution*

The change in predicate bound distribution has a small effect on all grouped running times. They become slightly better. This expresses itself in a slight increase of the factor grouped computation is gaining compared to ungrouped computation. The number of constructed trees was in the same ball park than reported previously.

23.8.2 Five-dimensional Query Workload

The next measurement shows the performance of the grouping algorithms for all 5-dimensional Skyline combinations in a 10-dimensional data set. Deriving from our previous experiment we used only uniformly distributed predicate bound distribution. Figure 23.6 shows the result. Data sets used were anti-correlated, correlated, and uniformly distributed. Our special data set, correlation groups, was solely designed for a two-dimensional query workload in a 5-dimensional data set, so we did not used it here in this high-dimensional scenario.

In this high-dimensional example there are two things to observe. First, for this setup, grouping by analyzing wins only for the correlated data set (factor of approximately 1.4 compared to the ungrouped computation). For the anti-correlated and uniformly distributed data sets grouping by analyzing looses. The reason for this can be see in Table 23.8. This table shows the insert selectivity of our Skyline pre-filter, that is, the number (or percentage) of points that when run through the filter qualify as new Skyline points and hence have to be inserted into the Skyline. For the anti-correlated and uniformly distributed data sets approximately each 14th points respectively each 33rd point is inserted into the Skyline. This decision is made by the new point not dominating any Skyline point but by being incomparable to all other Skyline points resulting in a vast number of Skyline comparisons and therefore in an increased overhead for the grouped computation. The decision of a point being incomparable to all other Skyline points is done at the very end of the comparison process. So all Skyline points so far have to be looked



(a) Anti-correlated

(b) Correlated

(c) Independent

Figure 23.6: *Running times for up to 10K queries, 5-dimensional Skylines, uniform predicate bound distribution*

at before this decision can be made and the Skylines for these two data sets are extremely large (for a 100,000 points data sets the Skylines for anti-correlated data or uniformly distributed sets contain approximately 75,000 points respectively 26,000 points). For the correlated data set only each 100th point is inserted (Skyline size for a 100,000 points data set is approximately 400).

To ease up this situation we could change the semantic of the Skyline comparison so that points are “less” incomparable. A short example of this idea is given in the next section.

Distribution	Selectivity (2 dim)	Selectivity (5 dim)
Anti-corr.	0.08%	6.82%
Corr.	0.06%	0.74%
Unif. dist..	0.07%	2.62%
Corr.-groups	0.04%	0.85%

Table 23.8: *Insert selectivities for 2- and 5-dimensional Skylines, 10K queries, 10000 inserts*

The second observation is that the grouping by learning degrades heavily. The reason for this degrading of the Learning approach is that the Learning approach groups too much. For the anti-correlated data set it produces about 350 trees whereas the Analyzing approach produces about 2200 trees. Grouping too much means less points in the Skyline intersection residing in the root node and hence less decision possibilities for incoming points.

23.8.3 Changing Skyline Comparison for 5-dimensional Skylines

Making the Skyline comparison "less" incomparable should yield a better performance for the grouped computation in a high-dimensional example. We changed the Skyline comparison in a way that points are treated as "incomparable" only if more than 2 dimensions (out of 5 dimension) differ more than 25%. Otherwise the new point is considered to be close to the point it is compared to and therefore discarded. This should speed up the comparison process since more points should be discarded. Table 23.9 shows the running times for a 1000 queries and 1000 inserted points for the unchanged Skyline comparison on the one hand and on the other hand the running times for the changed Skyline comparison.

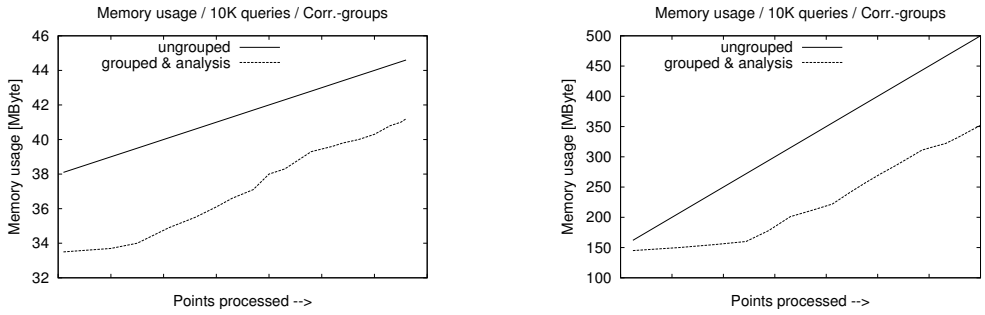
	Normal Compare			Changed Compare		
	ungroup.	group.	Select.	ungroup.	group.	Select.
Anti-corr.	33.6	39.5	6.82%	8.3	6.6	0.33%
Corr.	1.5	1.3	0.74%	0.9	0.5	0.03%
Uniform.	6.2	6.2	2.62%	4.4	4.1	1.20%

Table 23.9: *Running times (in secs) for 1000 queries and 1000 inserts, normal and changed Skyline compare (5-dimensional)*

As it can be seen, the changes in Skyline comparison particularly support a faster decision for anti-correlated data set, that is, the selectivity drops from almost 7% to 0.3%. The running times for the ungrouped and grouped computation also decrease but the gain for the grouped computation is higher. The same is true for the uniformly distributed data set. However, the decrease is less since the selectivity decreases less. Also the correlated data set is affected. Changing the Skyline comparison in the proposed way especially helps the grouped computation.

23.8.4 Memory Usage

This section shows another advantage of grouping queries: Because of the fact that grouping queries means less duplicate points have to be stored (for ungrouped computation each query has its own Skyline), the grouping approach in general has the advantage that it uses less memory than the ungrouped approach. This is particular interesting for higher-dimensional Skyline queries (the higher the Skyline dimensionality the more Skyline points are found).
Figure 23.7 displays this advantage for a 2-dimensional and a 5-dimensional example. Grouping needs less memory. Ungrouped queries do not "know" each other, each query holds its own Skyline points. This means that possibly many points are stored multiple times as some queries have Skyline points in common (remember: that is one of the ideas for grouping). By grouping those queries which have points in common, either by analyzing query parameters or by learning, we reduce the number of points stored multiple times and therefore save memory. This effect can be seen through all grouping algorithms and all data sets. The gain in saving memory depends on the number of Skyline points per query. This depends heavily on the type of data set that is used, for example, for an anti-correlated data set the gain is more than for a correlation groups data set and, of course, it depends also on the number of Skyline dimensions. In Figure 23.7 we do not show the memory usage over running time but over number of points processed. This way we can compare the ungrouped computation with the grouped computation.



(a) 2-dim. Skylines (90000 point inserts) (b) 5-dim. Skylines (10000 point inserts)

Figure 23.7: *Memory usage for 10K queries*

23.8.5 Variations

In this subsection we take a look at the sensitivity to grouping parameters, that is, we vary the deviation in Skyline dimensions for the Analyzing approach and the number of learning points for the Learning approach.

In Figure 23.10 we show the running times to insert 1000 points into 10,000 queries. We varied δ_{sky}, the allowed distance in Skyline dimensions and f_{dim} the penalty for differences in Skyline dimensions and noted the running time to insert the points and the number of trees produced by the grouping (by analyzing) algorithm. We used two-dimensional Skyline queries in a five-dimensional correlated data set, uniform predicate distribution with previously mentioned parameters.

The grouping takes place in clusters. Each cluster has a specific number of trees (for example, 369, 481, 1351). The running time is the same for each cluster. Grouping with no difference in Skyline dimensions obviously yields the best running time. But also grouping with differences in Skyline dimensions yields better running times than sequential computation (running time 4.2 secs).

In Figure 23.11 we varied the number of learning points: Instead of 10,000 learning points we only used 1000 learning points.

The interesting result is that for a correlated and correlation groups data set a shorter learning period yields a slightly better running time. The gain, however, is not much, so for previous results we kept the learning period at 10,000 points. Varying the distance function, that is, changing the percentage of Skyline points in the intersection, does not yield better results.

23.9 Speeding up the Delete Operation

Having talked extensively about speeding up the insert operation we should spend some thoughts on speeding up the delete operation. Can the same idea, arranging Skyline queries in a hierarchy of queries, be applied for the delete operation?

Grouping is also possible for the delete operation. Consider two grouped queries Q_1 and Q_2 and root node R in Figure 23.8. There are overall 5 points $(p_1, ..., p_5)$ in Skylines. Two situation can occur: a Skyline point in a leaf-Skyline is deleted, for example, p_3, or a Skyline point in the root Skyline is deleted, for example, p_2. Overall, the computation steps for the grouped delete operation are the same. First, delete the point and compute the candidate points, second, compute new Skyline points from the candidate points. If p_3 is deleted the particular steps are

δ_{sky}	f_{dim}	Running Time (in s)	Trees
0.10	0.025	3.9	369
0.10	0.05	3.8	481
0.10	0.01	1.7	1351
0.20	0.025	3.9	369
0.20	0.05	3.9	369
0.20	0.10	3.8	481
0.20	0.15	1.6	1351
0.20	0.20	1.6	1351
0.40	0.025	3.9	369
0.40	0.05	3.9	369
0.40	0.10	3.9	369
0.40	0.15	3.8	481
0.40	0.20	3.8	481
0.40	0.30	1.6	1351
0.40	0.4	1.6	1351

Table 23.10: *Varying δ_{sky} and f_{dim}*

	Ungrouped	Analysis	Learning (1000)	Learning (10000)
Anti-corr.	44.8	15.5	22.0	26.4
Corr.	46.8	14.8	26.5	23.6
Unif. dist..	39.6	15.4	18.8	27.6
Corr.-groups	45.7	19.6	26.0	22.3

Table 23.11: *Running time (in secs, 1000 inserts) for 100K queries, uniform predicate distribution, different number of learning points*

the following:

- Delete p_3.
- Compute candidate points for Q_1 according to schema in Figure 22.4.
- Compute new Skyline points from candidate points. The new Skyline points reside in leaf-Skyline of Q_1.

If p_2 is deleted the following steps are necessary:

- Delete p_2.
- Compute candidate points for Q_1 and Q_2 according to schema in Figure 22.4 in separate computations.
- Compute new Skyline points for Q_1 and Q_2 from the candidate points separately for each query. These points reside in the leaf-Skylines of Q_1 respectively Q_2
- Compute the intersection of the Skylines from Q_1 and Q_2 to get the new root Skyline.

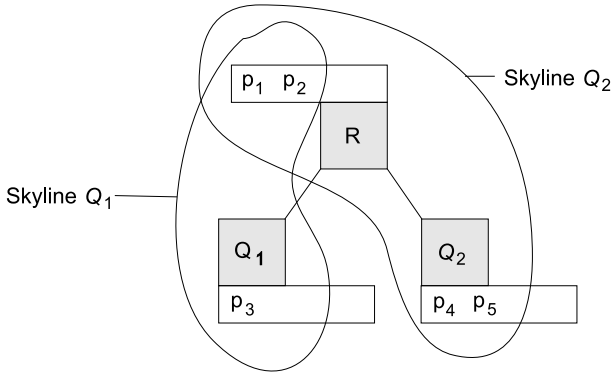


Figure 23.8: *Grouping for delete operation*

Taking the previous example into account, only one step, that is, the actual deletion of the point, can be speed up by grouping. The time consuming step, that is, the re-computation of Skyline points from candidate points, has to be carried out separately for each query. Also the computation of the candidate points has to be done for each query separately. So speeding up the delete operation by grouping the queries like it is done for the insert operation would yield only marginal improvements in performance. For this reason, we did not perpetuate grouped deletion any further.
The performance of the delete operation is given in Chapter 22.2. We consider these results “as good as they get”.

CHAPTER 24

Summary: Continuous Skyline Computation

In Chapters 21 and 23 we took Skyline computation on step further. For all Skyline algorithms presented in Part II and Part III the underlying data set for computing the Skyline was static, that is, during Skyline computation no points were added or deleted from the data set.

Inspired by the characteristics of the Online algorithms, the Nearest Neighbor Algorithm and the Branch-and-Bound Skyline Algorithm, we thought of a scenario where the underlying data set for computing the Skyline is not static, that is, many Skyline points are added to the data set during Skyline computation and a few points are deleted from the data set during Skyline computation. A striking question had to be answered: Do we have to re-compute the complete Skyline computed so far or can we use parts of the Skyline to adjust it reflecting the new situation, that is, having an additional point in the data set or missing one point? The answers was: We can re-use the already computed Skyline to adjust to the new situation saving lots of time.

In our understanding each Skyline query represents a filter for a user who is interested in certain details, for example, certain attributes of a used car. Now there are thousands of users looking for a used car. Another striking question arose: Can we answer similar queries together saving time for computation and memory consumption. And the answers was: We can answer similar queries together. By building up an "index" of queries, they can be answered together in an elegant way. As with any index it must be used with caution. An index used in a wrong way does not help at all, mostly it takes longer to get results. We showed circumstances where the index works very well and we showed circumstances where indexing queries should not be used.

Part V

Conclusion

CHAPTER 25

Conclusion

The title of this thesis, “Skyline Query Processing”, gives the reader a first impression what to expect, a thorough discussion about *Skyline*, that is, coming from practical examples showing some modern day problems with vast amounts of data, going over different ways of determining the Skyline, showing you different aspects of Skyline computation, and finally extending the topic for new applications of the Skyline. This thesis is, therefore, divided into three major parts: The first part deals with Skyline in general, from Skyline examples to mathematical foundations. The second part deals with Skyline algorithms different authors have thought of. Each Skyline algorithm is presented using the same example, so the reader can easily see how each algorithm works. Finally, the third part brings something new, the extension of Skyline computation to dynamic data sets and thousands of Skyline queries to be answered in an efficient way.

This is not the first thesis on Skyline. Some problems have already been discussed in [Ros01]. So this thesis can be seen as a direct sequel of [Ros01], picking up some parts and playing them a little further.

In Chapter 2 we introduced a set of examples which illustrate different application scenarios Skyline processing can be used for. This chapter merely is used as teaser to give you a glance of how the Skyline can be used. All scenarios are take from “daily life” since we did not want to create any artificial - solely scientific - scenario giving the reader the chance to get accustomed with Skyline problems making the Skyline easier to understand. This is also the reason why we put the scenarios at the beginning of this thesis not having talked *Skyline* at all. Please note: Not all scenarios were suitable for all parts of Skyline However, each part refered to the scenario which can be used for it.

Chapter 3 took you to the mathematical basics of the Skyline. The Skyline is based on the maximum vector problem. Trying to compute the maximum of a set of (multi-dimensional) vectors it is, in our consideration, a direct extension of finding the maximum of a set of (one-dimensional) numbers. For finding the maximum (or minimum) of a set of numbers there are a bunch of efficient algorithms that work well for different application scenarios. For finding the maxima (or minima) of a set of vectors there are, now, a bunch of efficient algorithms that work

well for different application scenarios. We also took a look of some Skyline properties which come in handy in later chapters.

Chapter 4 introduced you to the basic wording used in this thesis. It also introduced the Skyline query, the "star" in this thesis. All discussions "orbit" around the Skyline query, like planets around a star. The chapter also served as the first step to a classification of Skyline algorithms according to their computational behavior.

Chapters 6 and 7 gave you the technical knowledge you needed to understand this thesis. It introduced our basic data sets, our basic example to explain the working of the different Skyline algorithms, and a pseudo-code all Skyline algorithms in this thesis are displayed in.

Chapter 8 classified the Skyline algorithms given in Chapters 9, 10, 11, 12, and 13 according to our taxonomy of Skyline algorithms. The algorithms in these chapters all classify as batch or progressive algorithms, that is the Standard Algorithm, the Block-Nested-Loops Algorithm, the Divide-and-Conquer Algorithm, the Bitmap Based Algorithm, and the Partition-Index Algorithm. The basic example from Chapter 7 is used to demonstrate the techniques of each algorithm. Attached to this part is Chapter 14 which demonstrated a way of doing some prefiltering to facilitate later Skyline computation.

Chapters 16 through 18 took the Skyline application one step ahead. In Chapter 16 we introduced an online scenario for Skyline computation giving several properties for a Skyline algorithm to be classified as an online algorithm. In Chapter 17 we presented the first online Skyline algorithm, the Nearest Neighbor Algorithm. It is explained in great detail, demonstrating how it works, its particularities, and its properties to be classified as an online Skyline algorithm. The quality of results plays a central role for the Nearest Neighbor algorithm. What is meant by quality of results and a comparison to other Skyline algorithms concluded this chapter. Chapter 18 showed the second Skyline algorithm to be called an online Skyline algorithm, the Branch-and-Bound Skyline Algorithm. The properties of this Skyline algorithm and its classification to be an online Skyline algorithm is discussed. Both Skyline algorithms are presented using our basic example.

Finally, Chapters 21 to 23 took the Skyline application another step ahead. Instead of dealing with static data sets we now faced the challenge of changing data sets, data was inserted into and deleted from the data set. In Chapter 21 we span the idea. Chapter 22 gave you all necessary operations we needed to cope with changing data sets, finding out that we could reuse parts of the Skyline so far to deal with incoming and outgoing data, making a complete recalculation of the Skyline unnecessary. Chapter 23, again, took us one step ahead. Now dealing with many queries which worked on changing data sets. We developed techniques which allowed us to answer queries not one by one but in groups, saving computation time and memory usage.

CHAPTER 26

Closing Words

Recently the Skyline has reached some considerable attention. From introducing new efficient Skyline algorithms for "classical" Skyline problems to new Skyline application domains there has been considerable research effort. Funnily, the theoretical foundations of the Skyline were proposed more than 30 years ago, the Skyline has begun to attract attention in the database context not before the late 90s.

The Skyline is a neat subject which can help to reduce the ever-enlarging amounts of data. All papers on the Skyline subject have the reduction of data in common.

Reduction of data in any field of information processing has become essential, even vital. The volume of data which decisions are based on increases every day and decisions have to be made in a shorter period of time. This is because the world chances faster today. So concepts for reducing data have to provide reliable results for the decisions to be reasonable.

The fact that the world changes faster today is, of course, somehow a "self inflicted wound", without computers this would have probably never happened. But turning back the time is not possible and undoubtedly computers help us in myriad ways.

So data reduction is the only way we can go. Skyline is casted for a role in this play of data reduction. One among many others. For each act there is a role which plays a leading part, some roles that have a supporting part, and some roles that are not on stage for this particular act. Skyline certainly has acts where it has a leading part. As it is with all plays a director is need who coordinates the performances of each role and assembles them for a scene.

Part VI

Appendix

APPENDIX A

Numbers

Tables A.1 and A.2 show the size of the Skyline for 100,000 points and 1,000,000 points for the anti-correlated, correlated, and uniformly distributed data sets.

Dimensions	Anti-correlated	Correlated	Uniformly distributed
2	49	1	12
3	632	3	69
4	4239	11	267
5	12615	17	1032
6	26843	21	1986
7	41484	43	5560
8	55691	121	9662
9	67101	243	16847
10	75028	378	26047

Table A.1: *Skyline sizes for a 100,000 points data set*

The following data sets were barely used in this thesis. The table just illustrates how the Skyline size grows when the number of points in the data set grows.

Dimensions	Anti-correlated	Correlated	Uniformly distributed
2	54	1	12
3	1012	5	88
4	8545	25	453
5	35947	21	1851

Table A.2: *Skyline sizes for a 1,000,000 points data set*

For the interested reader we note the number of nodes within the R*-tree that is used for the Nearest Neighbor Algorithm in Part III. Tables A.3, A.4, and A.5 depict the number of intermediate and leaf nodes of an R*-Tree built with no specific ordering of the points. The R*-Tree contained 100,000 points.

Dimension	Internal nodes	Data nodes	∑ nodes
2	4	418	422
3	7	556	563
4	10	716	726
5	17	866	883
6	22	1025	1047
7	30	1151	1181
8	36	1298	1334
9	43	1442	1485
10	54	1600	1654

Table A.3: *Number of nodes for a anti-correlated data set, 100,000 points*

Dimension	Internal nodes	Data nodes	∑ nodes
2	4	412	416
3	8	561	569
4	11	714	725
5	15	848	863
6	22	975	997
7	25	1111	1136
8	31	1220	1251
9	39	1358	1397
10	50	1501	1551

Table A.4: *Number of nodes for a correlated data set, 100,000 points*

Dimension	Internal nodes	Data nodes	∑ nodes
2	5	421	427
3	7	571	578
4	10	725	735
5	16	859	875
6	21	999	1020
7	28	1157	1185
8	34	1295	1329
9	44	1431	1475
10	55	1599	1654

Table A.5: *Number of nodes for a uniformly distributed data set, 100,000 points*

List of Figures

List of Tables

Bibliography

[AF00] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 53–64, Cairo, Egypt, September 10-14, 2000.

[AH00] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 261–272, Dallas, Texas, USA, May 16-18, 2000.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts, USA, 1974.

[AMR+98] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. L. Wiener. Incremental maintenance for materialized views over semistructured data. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 38–49, New York City, New York, USA, August 24-27, 1998.

[AT78] S. G. Akl and G. T. Toussaint. A fast convex hull algorithm. *Information Processing Letters*, 7(5):219–222, 1978.

[Aut06] Autoscout24, website, 2006. http://www.autoscout24.de/.

[Bal06] W.-T. Balke. Efficient evaluation of numerical preferences: Top k queries, skylines and multi-objective retrieval. In *Preferences: Specification, Inference, Applications*, Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2006.

[Bay97] R. Bayer. The universal B-tree for multidimensional indexing: General concepts. In *World-Wide Computing and Its Applications '97 (WWCA '97)*, Tsukuba, Japan, March 10-11, 1997.

[BBD+02] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–16, Madison, Wisconsin, USA, June 3-5, 2002.

[BBKK97] S. Berchtold, C. Böhm, D. A. Keim, and H.-P. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 78–86, Tucson, Arizona, USA, May 12-14, 1997.

[BD83] D. Bitton and D. J. DeWitt. Duplicate record elimination in large data files. *ACM Transactions on Database Systems*, 8(2):255–265, 1983.

[BGS01] P. Bonnet, J. E. Gehrke, and P. Seshadri. Towards sensor database systems. In *Proceedings of the International Conference on Mobile Data Management*, Hong Kong, China, January 8-10, 2001.

[BGZ04] W.-T. Balke, U. Güntzer, and J. X. Zheng. Efficient distributed skylining for web information systems. In *Proceedings of Extending Database Technology*, pages 256–273, Heraklion, Crete, Greece, March 14-18, 2004.

[BK01] C. Böhm and H.-P. Kriegel. Determining the convex hull in large multidimensional databases. In *Proceedings of International Conference on Data Warehousing and Knowledge Discovery*, pages 294–306, Munich, Germany, September 5-7, 2001.

[BKS01] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline operator. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 421–430, Heidelberg, Germany, April 2-6, 2001.

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: an efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 322–331, Atlantic City, New Jersey, USA, May 23-25, 1990.

[BKST78] J. L. Bentley, H. T. Kung, M. Schkolnick, and C. D. Thompson. On the average number of maxima in a set of vectors and applications. *Journal of the ACM*, 25(4):536–543, 1978.

[BM72] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1(3):173–189, 1972.

[BO03] B. Babcock and C. Olston. Distributed top-k monitoring. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 28–39, San Diego, California, USA, June 9-12, 2003.

[Boe06] Deutsche Börse, website, 2006. http://deutsche-boerse.com/.

[BW01] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3):109–120, 2001.

[BZG05] W.-T. Balke, J. X. Zheng, and U. Güntzer. Approaching the efficient frontier: Cooperative database retrieval using high-dimensional skylines. In *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 410–421, Beijing, China, April 17-20, 2005.

[Bör99] S. Börzsönyi. Der Skyline-Operator. Technical report, Fakultät für Mathematik und Informatik, Universität Passau, Germany, 1999.

[CDK06] S. Chaudhuri, N. N. Dalvi, and R. Kaushik. Robust cardinality and cost estimation for skyline operator. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, page 64, Atlanta, Georgia, USA, April 3 - 8, 2006.

[CDTW00] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 379–390, Dallas, Texas, USA, May 16-18, 2000.

[CET05a] C. Y. Chan, P.-K. Eng, and K.-L. Tan. Efficient processing of skyline queries with partially-ordered domains. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 190–191, Tokyo, Japan, April 5-8, 2005.

[CET05b] C. Y. Chan, P.-K. Eng, and K.-L. Tan. Stratified computation of skylines with partially-ordered domains. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 203–214, Baltimore, Maryland, USA, June 13-16, 2005.

[CGGL03] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *Proceedings of the International Conference on Data Engineering*, pages 717–816, Bangalore, India, March 5-8, 2003.

[CGGL05] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting: Theory and optimizations. In *Proceedings of the Intelligent Information Systems Conference (IIS)*, pages 595–604, Gdansk, Poland, June, 2005.

[CK97a] M. J. Carey and D. Kossmann. On saying "enough already!" in SQL. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 219–230, Tucson, Arizona, USA, May 13-15, 1997.

[CK97b] M. J. Carey and D. Kossmann. Processing top n and bottom n queries. *IEEE Data Engineering Bulletin*, 20(3):12–19, 1997.

[Com79] D. Comer. The ubiquitous B-tree. *Computing Surveys*, 11(2):121–137, 1979.

[Cou05] COUGAR project, website, 2005. http://www.cs.cornell.edu/database/cougar/.

[CÇC+02] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Monitoring streams - a new class of data management applications. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 215–226, Hong Kong, China, August 20-23, 2002.

[DFZF03] Y. Diao, M. J. Franklin, H. Zhang, and P. M. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Transactions on Database Systems (TODS)*, 28(4):467–516, 2003.

[EOT03] P.-K. Eng, B.C. Ooi, and K.-L. Tan. Indexing for progressive Skyline computation. *Data & Knowledge Engineering*, 46(2):169–201, 2003.

[FJL+01] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 115–126, Santa Barbara, CA, USA, May 21-24, 2001.

[FSAA01] H. Ferhatosmanoglu, I. Stanoi, D. Agrawal, and A. El Abbadi. Constrained nearest neighbor queries. *Lecture Notes in Computer Science*, 2121:257–278, 2001.

[GG98] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[GMS93] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 157–166, Washington, D.C., USA, May 26-28, 1993.

[God04] P. Godfrey. Skyline cardinality for relational processing. In *Foundations of Information and Knowledge Systems, Third International Symposium*, pages 78–97, Wilhelminenburg Castle, Austria, February 17-20, 2004.

[GS79] P. J. Green and B. W. Silverman. Constructing the convex hull of a set of points in the plane. *The Computer Journal*, 22(3):262–266, 1979.

[GS03] A. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Diego, California, USA, June 9-12, 2003.

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 47–57, Boston, Massachusetts, USA, June 18-21, 1984.

[HAC+99] J. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Rotha, and P. Haas. Interactive data analysis: The control project. *IEEE Computer*, 32(8):51–59, 1999.

[HCH+99] E. N. Hanson, Chris Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park, and A. Vernon. Scalable trigger processing. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 266–275, Sydney, Australia, March 23-26, 1999.

[HFC+00] J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. A. Shah. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, 2000.

[HS99] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems*, 24(2):265–318, 1999.

[IS88] M. Miyakawa I. Stojmenovic. An optimal parallel algorithm for solving the maximal elements problem in the plane. *Parallel Computing*, 7(2):249–251, 1988.

[Jes06] Lido di Jesolo, website, 2006. http://www.jesolo.it/.

[Kie02] W. Kießling. Foundations of preferences in database systems. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 311–322, Hong Kong, China, August 20-23, 2002.

[KLP75] H.T. Kung, F. Luccio, and F.P. Preparata. On finding the maxima of a set of vectors. *Journal of the ACM*, 22(4):469–476, 1975.

[KRR02] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for Skyline queries. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 275–286, Hong Kong, China, August 20-23, 2002.

[Lar99] P.-A. Larson. Grouping and duplicate elimination: Benefits of early aggregation. Technical report, Microsoft Corporation , Redmond, Washington, USA, 1999. http://www.research.microsoft.com/ palarson/.

[LPBZ96] L. Liu, C. Pu, R. S. Barga, and T. Zhou. Differential evaluation of continual queries. In *International Conference on Distributed Computing Systems*, pages 458–465, Hong Kong, May 27-30, 1996.

[LPT99] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *Knowledge and Data Engineering*, 11(4):610–628, 1999.

[LYLC04] E. Lo, K. Yip, K.-I. Lin, and D. W. Cheung. Progressive skylining over web-accessible database. unpublished, August, 2004.

[LYWL05] X. Lin, Y. Yuan, W. Wang, and H. Lu. Stabbing the sky: Efficient skyline computation over sliding windows. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 502–513, Tokyo, Japan, April 5-8, 2005.

[Mar99] V. Markl. *MISTRAL: Processing Relational Queries using a Multidimensional Access Technique*. PhD thesis, Fakultät für Informatik, Technische Universität München, Germany, January, 1999.

[Mat91] J. Matoušek. Computing dominances in E^n. *Information Processing Letters*, 38(5):277–278, 1991.

[MF02] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, San Jose, California, USA, February 26 - March 1, 2002.

[MPG06] M. D. Morse, J. M. Patel, and W. I. Grosky. Efficient continuous skyline computation. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, page 108, Atlanta, Georgia, USA, April 3 - 8, 2006.

[MSHR02] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 49–60, Madison, Wisconsin, USA, June 2-5, 2002.

[NDM+00] J. Naughton, D. DeWitt, D. Maier, J. Chen, L. Galanis, K. Tufte, J. Kang, Q. Luo, N. Prakash, F. Tian, J. Shanmugasundaram, C. Zhang R. Ramamurthy, B. Jackson, Y. Wang, and A. Gupta. The niagara internet query system. Technical report, Computer Sciences Department, University of Wisconsin-Madison and Computer Science Department, Oregon Graduate Institute, USA, 2000.

[Nia04] Niagara Internet Query System Project, website, 2004. http://www-db.cs.wisc.edu/niagara/.

[Nyc06] New York City, official website, 2006. http://www.nyc.gov/.

[PJET05] J. Pei, W. Jin, M. Ester, and Y. Tao. Catching the best views of skyline: A semantic approach based on decisive subspaces. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 253–264, Trondheim, Norway, August 30 - September 2, 2005.

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

[PTFS03] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for Skyline queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Diego, California, USA, June 9-12, 2003.

[PTFS05] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM Transactions on Database Systems (TODS)*, 30(1):41–82, 2005.

[RDL95] C. J. Rhee, S. K. Dhall, and S. Lakshmivarahan. The minimum weight dominating set problem for permutation graphs is in nc. *Journal of Parallel and Distributed Computing*, 28(2):109–112, 1995.

[RKV95] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 71–79, San Jose, CA, May 22-25, 1995.

[Ros01] S. Rost. Efficient implementation of the Skyline operator. Master's thesis, Fakultät für Informatik, Technische Universität München, Germany, June, 2001.

[Sel88] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, 1988.

[Set06] SETI Institute, website, 2006. http://www.seti.org/.

[SK98] T. Seidl and H.-P. Kriegel. Optimal multi-step k-nearest neighbor search. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 154–165, Seattle, Washington, USA, June 2-4, 1998.

[Sta04] Stanford Data Stream Management (STREAM) Project, website, 2004. http://www-db.standford.edu/stream/.

[Tel06] Deutsche Telekom, website, 2006. http://www.telekom.de/.

[TEO01] K.-L. Tan, P.-K. Eng, and B.C. Ooi. Efficient progressive Skyline computation. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 301–310, Roma, Italy, September 11-14, 2001.

[TGNO92] D. B. Terry, D. Goldberg, D. Nichols, and B. M. Oki. Continuous queries over append-only databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 321–330, San Diego, California, USA, June 2-5, 1992.

[TP06] Y. Tao and D. Papadias. Maintaining sliding window skylines on data streams. *IEEE Transactions on Knowledge and Data Engineering*, 18(2):377–391, 2006.

[TPS02] Y. Tao, D. Papadias, and Q. Shen. Continuous nearest neighbor search. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, pages 287–298, Hong Kong, China, August 20-23, 2002.

[TXP06] Y. Tao, X. Xiao, and J. Pei. Subsky: Efficient computation of skylines in subspaces. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, page 65, Atlanta, Georgia, USA, April 3 - 8, 2006.

[YG03] Y. Yao and J. E. Gehrke. Query processing in sensor networks. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research*, Asilomar, California, USA, January 5-8, 2003.

[YGM99] T. W. Yan and H. Garcia-Molina. The SIFT information dissemination system. *ACM Transactions on Database Systems (TODS)*, 24(4):529–565, 1999.

[YLL+05] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang. Efficient computation of the skyline cube. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 241–252, Trondheim, Norway, August 30 - September 2, 2005.

[YOTJ01] C. Yu, B. C. Ooi, K.-L. Tan, and H. V. Jagadish. Indexing the distance: An efficient method to knn processing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 421–430, Roma, Italy, September 11-14, 2001.

[ZSC+03] S. Zdonik, M. Stonebraker, M. Cherniack, U. Çetintemel, M. Balazinska, and H. Balakrishnan. The aurora and medusa projects. *IEEE Data Engineering Bulletin*, 26(2):3–10, 2003.

Index