# THE SEMANTICS AND APPROXIMATION FOR THE SKYLINE OPERATOR

by

Wen Jin
M.A., Southeast University, 1995
Ph.D Candidate, Fudan University, 1999

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
in the School
of
Computing Science

© Wen Jin 2007
SIMON FRASER UNIVERSITY
Summer 2007

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

## APPROVAL

**Name:** Wen Jin

**Degree:** Doctor of Philosophy

**Title of thesis:** The Semantics and Approximation for the Skyline Operator

**Examining Committee:** Dr. Qianping Gu
Chair

______________________________
Dr. Jiawei Han, Senior Supervisor

______________________________
Dr. Martin Ester, Supervisor

______________________________
Dr. Ke Wang, Supervisor

______________________________
Dr. Binay Bhattacharya, Supervisor

______________________________
Dr. Wo-Shun Luk, SFU Examiner

______________________________
Dr. Joerg Sander, External Examiner,
Professor of Computer Science,
University of Alberta

**Date Approved:** 06/12/2006

SIMON FRASER UNIVERSITY library

# DECLARATION OF PARTIAL COPYRIGHT LICENCE

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection (currently available to the public at the "Institutional Repository" link of the SFU Library website <www.lib.sfu.ca> at: <http://ir.lib.sfu.ca/handle/1892/112>) and, without changing the content, to translate the thesis/project or extended essays, if technically possible, to any medium or format for the purpose of preservation of the digital work.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author's written permission.

Permission for public performance, or limited permission for private scholarly use, of any multimedia materials forming part of this work, may have been granted by the author. This information may be found on the separately catalogued multimedia material and in the signed Partial Copyright Licence.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

Simon Fraser University Library
Burnaby, BC, Canada

# Abstract

The skyline operator of a $d$-dimensional dataset, which returns the points that are not dominated by any other point on all dimensions, has been well recognized its importance in preference queries and multi-criteria decision making applications. Many existing algorithms have been developed to improve the efficiency in computing the exact skyline objects in the full space. However, few previous work involves the following problems: (1) Why and in which subspaces is (or is not) an object in the skyline? (2) How to approximate the skyline objects in a reasonable way? (3) Can the notion of skyline operator facilitate other database operators? (4) How could skyline be computed efficiently over multiple relational tables?

In this work, we explore the semantics of skyline in subspaces, study the approximate skyline objects in databases, apply the notion of skyline to the efficient processing of other database operations such as ranked queries, and propose solutions to implement skyline operator on multiple relations. We develop a class of novel and efficient methods to fulfill these tasks in large databases. A comprehensive performance study on both synthetic datasets and real datasets demonstrates that our proposed methods are not only efficient but also effective.

# Acknowledgments

First of all, my deep gratitude goes to my supervisor and mentor, Dr. Jiawei Han, for his continuous encouragement, guidance and support during the course of this work. His creative thinking and vision and as well as his persistence and industriousness has been inspiring me during my study and will always give me strength in my future career.

I am very thankful to my supervisor, Dr. Martin Ester, for his insightful comments and support during the discussions about research, for his valuable advices and suggestions about career development as a mentor. I am very fortunate to have his help in introducing me to become a Christian.

I wish to express my thanks to my committee members, Dr. Ke Wang and Dr. Binay Bhattacharya. Their helpful discussions enable me to make improvements to this thesis.

My gratitude and appreciation also goes to Dr. Wo-Shun Luk and Dr. Joerg Sander in University of Alberta for agreeing to serve as examiners and for their remarks that help me to better present the results of my research and improve the quality to the thesis.

Part of this work is done in collaboration with Dr. Jian Pei, Dr. Yufei Tao in the Chinese University of Hong Kong and Dr. Anthony K. H. Tung in National University of Singapore. I thank them for the knowledge and skills they imparted through the collaboration. Working with them is always so enjoyable and fruitful.

I also want to thank Mr. Richard Frank for proofreading my thesis. I would also like to thank many people in our department, support staff and faculty, for always being helpful over the years. A particular acknowledgement goes to Rong Ge, Zengjian Hu, Haiming Huang, Flavia Moses, Benjamin C. M. Fung, Val Galat, Fereydoun Hormozdiari, Junqiang Liu, Yuelong Jiang, Benkoczi Robert, Qiaosheng Shi, Yabo Xu and Senqiang Zhou.

Finally, my warmest thanks go to my parents for their patience, support and love in my whole life.

# Contents


**Approval** ii

**Abstract** iii

**Acknowledgments** iv

**Contents** v

**List of Tables** x

**List of Figures** xi

**1 Introduction** 1

1.1 Motivation of the Thesis . . . . . . . . . . 1

1.2 Thesis Plan . . . . . . . . . . 3

**2 Background and Related Work** 4

2.1 Full Space Skyline Computation Approach . . . . . . . . . . 5

2.1.1 Basic Divide and Conquer Approach (BDC) . . . . . . . . . . 5

2.1.2 Extension to BDC . . . . . . . . . . 6

2.1.3 External Algorithms for BDC . . . . . . . . . . 7

2.1.4 Basic Block Nested Loop (BNL) . . . . . . . . . . 7

2.1.5 Sort-Filter-Skyline Computation (SFS) . . . . . . . . . . 8

2.1.6 Linear Elimination Sort For Skyline (LESS) . . . . . . . . . . 9

2.1.7 B-tree Minimum Value Index Approach (MVI) . . . . . . . . . . 9

2.1.8 R-tree Nearest Neighbor Approach (NN) . . . . . . . . . . 11

2.1.9 R-tree Branch and Bound Approach (BBS) . . . . . . . . . . 13

2.1.10 The Bitmap Index Approach . . . . . . . . . . . . . . . . . . . . . . . . 15
2.2 Subspace Skyline Computation . . . . . . . . . . . . . . . . . . . . . . . . . . 16
2.2.1 Computing Subspace Skylines via Multiple Anchors . . . . . . . . . . 17
2.2.2 Computing SKYCUBE with Sharing Strategies . . . . . . . . . . . . . 18
2.2.3 Updating SKYCUBE with the Compressed Structure . . . . . . . . . 19
2.2.4 Mining Frequent Skylines in Subspaces . . . . . . . . . . . . . . . . . . . 20
2.2.5 Mining Strong Skylines in Subspaces . . . . . . . . . . . . . . . . . . . 21
2.3 Miscellaneous Approach . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
2.3.1 Skyline Algorithms for Distributed Environments . . . . . . . . . . . . 21
2.3.2 Skyline Computation in Mobile Environments . . . . . . . . . . . . . . 22
2.3.3 Skyline Computation in Data Streams . . . . . . . . . . . . . . . . . . 22
2.3.4 Skyline Computation in Time Series Databases . . . . . . . . . . . . . 23
2.3.5 Skyline Computation with Partially-Ordered Domains . . . . . . . . . 24
2.3.6 Cooperative Database Retrieval Using High-Dimensional Skylines . . . 24
2.3.7 K-Dominant Skylines Computation . . . . . . . . . . . . . . . . . . . . 25
2.3.8 Skyline Cardinality Estimation . . . . . . . . . . . . . . . . . . . . . . 26

**3 The Semantics of Skyline 27**
3.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
3.2 Multidimensional Subspace Skyline and a Unique-Value Case . . . . . . . . . 31
3.2.1 Subspace Skyline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
3.2.2 Subspace Skylines in Unique Value Data Sets: A Simple Case . . . . . 32
3.3 Skyline Semantics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
3.3.1 Key Observations and Ideas . . . . . . . . . . . . . . . . . . . . . . . . 34
3.3.2 Skyline Groups and Decisive Subspaces . . . . . . . . . . . . . . . . . 35
3.3.3 Semantics of (Subspace) Skyline Objects . . . . . . . . . . . . . . . . . 38
3.3.4 Answering Skyline Membership Queries . . . . . . . . . . . . . . . . . 39
3.4 Subspace Skyline Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
3.4.1 Intuition . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
3.4.2 Skyline Group Lattice . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
3.4.3 Skyline Groups and Skyline Objects . . . . . . . . . . . . . . . . . . . 42
3.4.4 OLAP Analysis on Skylines . . . . . . . . . . . . . . . . . . . . . . . . 44
3.5 Subspace Skyline Computation . . . . . . . . . . . . . . . . . . . . . . . . . 45

3.5.1 Finding Skyline by Sorting . . . . . . . . . . . . . . . . . . . . . . . . . 45
3.5.2 Top-down Subspace Enumeration Tree . . . . . . . . . . . . . . . . . . 46
3.5.3 Algorithm *Skyey* . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
3.6 Experimental Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
3.6.1 Results on Real Data Set *Great NBA Players' Statistics* . . . . . . . . 50
3.6.2 Results on Synthetic Data Sets . . . . . . . . . . . . . . . . . . . . . . 53
3.7 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56

**4 Approximate Skyline: Thick Skyline Operator 58**
4.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
4.2 The Thick Skyline Operator . . . . . . . . . . . . . . . . . . . . . . . . . . 62
4.2.1 Problem Definitions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
4.2.2 The Task of Mining Thick Skylines . . . . . . . . . . . . . . . . . . . . 63
4.3 A Sampling-and-Pruning Method . . . . . . . . . . . . . . . . . . . . . . . . 64
4.3.1 Sampling Strategies . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
4.3.2 Strong Dominating Relationship and the Algorithm . . . . . . . . . . . 66
4.4 An Indexing-and-Estimating Method . . . . . . . . . . . . . . . . . . . . . . 69
4.4.1 Bounding $\epsilon$-neighbors in Minimum Value Index . . . . . . . . . . . . . 69
4.4.2 Slide Windows and the Algorithm . . . . . . . . . . . . . . . . . . . . 71
4.5 Microcluster-based Method . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
4.5.1 CF-tree and Microclusters . . . . . . . . . . . . . . . . . . . . . . . . . 75
4.5.2 Skylining Microclusters and the Algorithm . . . . . . . . . . . . . . . 77
4.5.3 Thick Skyline Operator and Data Mining Applications . . . . . . . . . 81
4.6 Experimental Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
4.6.1 Efficiency Tests . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
4.6.2 Effectiveness Tests . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
4.7 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86

**5 Skyline and Database Ranked Queries 88**
5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
5.2 Foundations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
5.2.1 Top-$k$ Ranked Queries and Skylines . . . . . . . . . . . . . . . . . . . 91
5.2.2 MBR and Microcluster . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
5.3 Computing and Indexing Layered Skylines . . . . . . . . . . . . . . . . . . . 95

5.3.1 A Naive-based Method . . . . . 95
5.3.2 A Topology Sorting-based Method . . . . . 96
5.4 A KNN-based Sweeping Approach for Top-$k$ Queries . . . . . 97
5.4.1 Sweeping over Blocks . . . . . 97
5.4.2 Sweeping within Layered Blocks . . . . . 101
5.5 A Grid-based Sweeping Approach for Top-$k$ Queries . . . . . 103
5.5.1 Shell Grid Partition of CF-tree . . . . . 103
5.5.2 Ranking Algorithm . . . . . 106
5.5.3 Error Bound in Approximate Solution . . . . . 110
5.6 Experimental Results . . . . . 111
5.6.1 Evaluations of Algorithms BBR and SGR . . . . . 112
5.6.2 Comparing with Algorithms of Onion and PREFER . . . . . 114
5.7 Revisit of Top-$k$ Ranked Queries . . . . . 117
5.8 Summary . . . . . 119

**6 The Multi-Relational Skyline Operator** **121**
6.1 Introduction . . . . . 121
6.2 Preliminaries . . . . . 125
6.3 Algorithms for Skylines over a Single Join . . . . . 128
6.3.1 A Naive Approach for Skyline Join . . . . . 128
6.3.2 Integrating with Sort-Merge Join . . . . . 128
6.3.3 Integrating with Nested-Loop Join . . . . . 132
6.4 Cardinality Estimate of Joined Skylines . . . . . 134
6.4.1 Problem of a Naive Solution . . . . . 134
6.4.2 The Model . . . . . 135
6.4.3 The Size of the Join Table . . . . . 135
6.4.4 The Expected Number of Skyline Objects in the Join Table . . . . . 135
6.4.5 Upper Bound . . . . . 136
6.4.6 Lower Bound . . . . . 138
6.5 Extending to Multiple Joins . . . . . 138
6.6 Experimental Evaluation . . . . . 139
6.7 Conclusion . . . . . 142

**7 Summary and Conclusions** **143**

7.1 Summary of the Thesis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 143

7.2 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 145

**8 Ongoing and Future Work** **146**

8.1 Answering Subspace Skyline Queries by Materializing Signatures . . . . . . . 146

8.2 Mining Subspace Thick Skyline Objects . . . . . . . . . . . . . . . . . . . . . 147

8.3 Mining Interesting Non-Skyline Objects . . . . . . . . . . . . . . . . . . . . . 147

8.4 Mining Microeconomic Dominating Neighbors . . . . . . . . . . . . . . . . . . 148

**Bibliography** **149**

# List of Tables


2.1 Minimum Value Index . . . . . . . . . . 10
2.2 Heap Contents . . . . . . . . . . 15
2.3 The Bitmap Approach . . . . . . . . . . 16

3.1 A Set of Objects as Our Running Example . . . . . . . . . . 35
3.2 Some Skyline Players and the Corresponding Decisive Subspaces . . . . . . . 52
3.3 Number of Skyline Players in Subspaces with Different Dimensionality . . . . 52
3.4 Number of Skyline Objects in Subspaces on Synthetic Data Sets . . . . . . . 55

4.1 The Index Approach . . . . . . . . . . 70
4.2 Thick Skyline Size . . . . . . . . . . 86

5.1 Example of Expanding Nodes in R-tree (1) . . . . . . . . . . 100
5.2 Example of Expanding Nodes in R-tree (2) . . . . . . . . . . 103

6.1 Customer(C) Table . . . . . . . . . . 123
6.2 Order(O) Table . . . . . . . . . . 123
6.3 Joined Table of Customer and Order (1) . . . . . . . . . . 124
6.4 Joined Table of Customer and Order (2) . . . . . . . . . . 124
6.5 Group-by D1 (CNum) in Customer . . . . . . . . . . 126
6.6 Group-by D5 (CNum) in Order . . . . . . . . . . 126
6.7 Customer' . . . . . . . . . . 131
6.8 Order' . . . . . . . . . . 131
6.9 Sort Customer by $D_2$ (Age) . . . . . . . . . . 133
6.10 Sort Order by $D_6$ (Quantity) . . . . . . . . . . 133
6.11 Cardinality of Different Datasets . . . . . . . . . . 140

# List of Figures


2.1 Illustration of BDC . . . . . . 5

2.2 Example Data . . . . . . 10

2.3 Discovery of $i$ . . . . . . 12

2.4 Discovery of $a$ . . . . . . 12

2.5 NN Partitioning for 3-dimensions. . . . . . . 13

2.6 Recursion Tree . . . . . . 14

2.7 R-tree example . . . . . . 14

2.8 An Anchor $A_1$ . . . . . . 17

2.9 More Anchors . . . . . . 17

3.1 An Example Showing the Intuition . . . . . . 28

3.2 The Algorithmic Framework of Searching Subspace Skylines in No-sharing Data Sets . . . . . . 34

3.3 The Subspaces Where Object $q$ in Table 3.1 Belongs to the Skyline. . . . . . . 39

3.4 Skyline Group Lattice for Table 3.1 . . . . . . 44

3.5 A Top-down Subspace Enumeration Tree . . . . . . 46

3.6 The *Skyey* Algorithm . . . . . . 50

3.7 Runtime vs. Cardinality on Independent Data Sets . . . . . . 54

3.8 Runtime vs. Cardinality on Correlated Data Sets . . . . . . 54

3.9 Runtime vs. Cardinality on Anti-Correlated Data Sets . . . . . . 54

3.10 Runtime vs. Dimensions on Independent Data Sets . . . . . . 56

3.11 Runtime vs. Dimensions on Correlated Data Sets . . . . . . 56

3.12 Runtime vs. Dimensions on Anti-Correlated Data Sets . . . . . . 56

4.1 Skyline Query of New York Hotels . . . . . . 59

4.2 Thick Skyline Pattern of New York Hotels . . . . . . . . . . . . . . . . . . . . 60
4.3 Sampling Objects to Pruning. . . . . . . . . . . . . . . . . . . . . . . . . . 66
4.4 Evaluate Neighborhood Scope . . . . . . . . . . . . . . . . . . . . . . . . . 72
4.5 Evaluate Neighborhood Scope in 3-$d$ Case . . . . . . . . . . . . . . . . . . 73
4.6 Microclusters of CF-tree . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
4.7 Time vs. Dimensions on Independent Data Sets . . . . . . . . . . . . . . . . 83
4.8 Time vs. Dimensions on Correlated Data Sets . . . . . . . . . . . . . . . . . 83
4.9 Time vs. Cardinality Dimensions on Independent Data Sets . . . . . . . . . . 83
4.10 Time vs. Cardinality Correlated Data Sets . . . . . . . . . . . . . . . . . . 83
4.11 Time vs. Eps on Independent Data Sets . . . . . . . . . . . . . . . . . . . . 83
4.12 #of Comparisons vs Cardinality on Independent Data Sets . . . . . . . . . . 83
4.13 #of Thick Skyline vs. Cardinality on Independent Data Sets . . . . . . . . . 84
4.14 #of Thick Skyline vs. Eps on Independent Data Sets . . . . . . . . . . . . . 84
4.15 Effects of Microclusters(I) . . . . . . . . . . . . . . . . . . . . . . . . . . 84
4.16 Effects of Microclusters(II) . . . . . . . . . . . . . . . . . . . . . . . . . . 84

5.1 Top-k Objects vs. Skyline . . . . . . . . . . . . . . . . . . . . . . . . . . . 90
5.2 Multilayer Skyline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93
5.3 Contact Points to a CF-tree Microcluster . . . . . . . . . . . . . . . . . . . 98
5.4 Sweeping over R-tree Blocks . . . . . . . . . . . . . . . . . . . . . . . . . . 99
5.5 Layered-Skyline in Block N4 . . . . . . . . . . . . . . . . . . . . . . . . . . 102
5.6 Linked List for Layered Skyline in a Block . . . . . . . . . . . . . . . . . . . 102
5.7 Sweeping Multilayered Skyline in R-tree . . . . . . . . . . . . . . . . . . . . 102
5.8 Shell-Grid Partition of Microclusters . . . . . . . . . . . . . . . . . . . . . . 104
5.9 2D SG-Partition and 3D SG-Partition . . . . . . . . . . . . . . . . . . . . . 105
5.10 Sweeping a Shell Grid . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107
5.11 Error Bound of Grid-based Answer . . . . . . . . . . . . . . . . . . . . . . . 110
5.12 Query Time vs #of Results in Correlated Data Set . . . . . . . . . . . . . . . 111
5.13 Query Time vs #of Results in Independent Data Set . . . . . . . . . . . . . . 111
5.14 Effect of Eps(1) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112
5.15 Effect of Eps(2) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112
5.16 Effect of Eps(3) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113
5.17 Preprocessing Time . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113

5.18 Index Size vs #of Layers on Correlated Data Set . . . . . . . . . . . . . . . . 113
5.19 Index Size vs #of Layers on Independent Data Set . . . . . . . . . . . . . . . 113
5.20 Query Time vs #of Results on Correlated Data Set . . . . . . . . . . . . . . . 114
5.21 Query Time vs #of Results on Independent Data Set . . . . . . . . . . . . . . 114
5.22 #of Tuples Visited vs #of Results on Independent Data Set . . . . . . . . . . 114
5.23 #of Tuples Visited vs #of Results on Correlated Data Set . . . . . . . . . . . 114
5.24 Coverage Rate vs #of Results on Independent Data Set . . . . . . . . . . . . 115
5.25 Coverage Rate vs #of Results on Correlated Data Set . . . . . . . . . . . . . 115

6.1 Comparing $a_i \oplus b_j$ with Skylines Found So Far in Joined Table . . . . . . . . 132
6.2 The Size of Data Sets and #of Skylines . . . . . . . . . . . . . . . . . . . . . 141
6.3 Runtime vs Cardinality . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 141
6.4 Runtime vs Dimensions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 142
6.5 #of Skylines Estimate . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 142

# Chapter 1

# Introduction

## 1.1 Motivation of the Thesis

The skyline operator was introduced to the database systems by applying the problem of finding the maxima of a set of points. Given a dataset with $d$-dimensions, the skyline is defined as a subset which contains exactly all interesting objects. An object $A$ is said to be interesting if there is no other object that is better in at least one dimension and not worse in all remaining dimensions than $A$. If we are traveling on vacation, for instance, it is very useful to find a hotel which is cheap and yet in close proximity to a beach. The hotel, by definition, would be considered part of the skyline as long as it is not worse than any other hotel with regard to price and distance. Skyline operator is very important in many applications such as multi-criteria decision-making and preference queries [13]. There are many algorithms which have been developed to improve the efficiency in computing the exact skyline objects in a full space. However, there are still many interesting problems related to the skyline which have not been carefully solved. In this work, we aim to explore the semantics of skyline in subspaces, study the approximate skyline objects in databases, apply the notion of skyline to the efficient processing of other database queries such as ranked queries, and extend the single table skyline operator to the multi-relational skyline operator.

In the first major part of this thesis, we focus on the the fundamental problem on the *semantics* of skylines: Why and in which subspaces is (or is not) an object in the skyline? Practically, users may also be interested in skylines in any subspaces. Then, what is the relationship between the skylines in the subspaces and those in the super-spaces? How

can we effectively analyze the subspace skylines? Can we efficiently compute skylines in various subspaces? We investigate the semantics of skylines, propose the subspace skyline analysis, and extend the full-space skyline computation to subspace skyline computation. We introduce a novel notion of *skyline group* which essentially is a group of objects that are coincidentally in the skylines of some subspaces. We identify the *decisive subspaces* that qualify skyline groups in the subspace skylines. The new notions concisely capture the semantics and the structures of skylines in various subspaces. Multidimensional roll-up and drill-down analysis is introduced. We also develop an efficient algorithm to compute the set of skyline groups and, for each subspace, the set of objects that are in the subspace skyline. A performance study is reported to evaluate our approach.

In the second major part of this thesis, we focus on the computation of the approximation of skyline objects. As the typical skyline query only gives users *thin skylines*, i.e., objects satisfying skyline evaluation conditions. This may not be desirable in many real applications because some non-skyline objects are almost as good as skyline objects and still attract users. We propose a novel concept, called *thick skyline*, which recommends not only skyline objects but also their nearby neighbors within $\varepsilon$-distance. Efficient computation methods are developed including *Sampling-and-Pruning* method, *Indexing-and-Estimating* method and *Microcluster-based* method. The first two methods take advantage of the statistics or indexes in large databases for computing thick skylines. The *Microcluster-based method* uses data summarization to save computation cost and also provides a concise representation of the thick skyline in the case of high cardinalities. Our experimental performance study shows that the proposed methods are both efficient and effective.

In the third major part of this thesis, we focus on the efficient processing of ranked queries based on skylines. Given a linear monotone score function $s$, the top-$k$ ranked query retrieves the best $k$ objects according to the values of $s$. Existing methods for processing such queries employ the techniques of sorting, updating thresholds, materializing views or pre-computing convex structures etc. Motivated by the interesting relationship between the top-$k$ tuples and the skyline objects, we propose the novel idea of sweeping the line/hyperplane of the score function towards the layered skyline, and quickly locate the answer points during the sweeping. We develop efficient algorithms for the exact top-$k$ ranked query and the approximate top-$k$ ranked query, and illustrate these methods can easily be plugged into typical multi-dimensional database indexes. We experimentally demonstrate that the proposed algorithms outperform the existing ones from different evaluating aspects.

In the fourth major part of this thesis, we focus on efficient computation of skyline over multiple relational tables. End users may be interested in getting skyline objects from data involved in several relations, but the cost on computing skylines on the joined table can be increased dramatically due to its potentially increasing cardinality and dimensionality. How to develop efficient methods to share the join processing with skyline computation is central to the skyline query optimization on multiple relations. We systematically study the skyline operator on multi-relational databases, and propose solutions aiming at seamlessly integrating state-of-the-art join methods into skyline computation. To further extend the query optimizer's cost model to accommodate skyline operator over joined tables, we also theoretically estimate the size of joined skylines. Our experiments not only demonstrate that the proposed methods are efficient, but also show the promising applicability of extending skyline operator to other typical database operators such as join and aggregates.

## 1.2 Thesis Plan

In the next chapter, we present background materials and previous works that relate to the central topic of this thesis. These include an introduction to the skyline operator and previous works in computing skyline objects and its variants of this problem in large databases. In Chapter 3, we study the semantics of skylines, propose the subspace skyline analysis, and extend the full-space skyline computation to subspace skyline computation. In Chapter 4, we present a new thick skyline operator which not only returns skyline objects but also some non-skyline objects in their $\varepsilon$-neighborhoods. Furthermore, we develop efficient methods for mining thick skylines. In Chapter 5, we study the relationship between the ranked queries and skyline objects, and propose efficient algorithms based on the novel idea of sweeping the line/hyperplane of the score function towards the layered skyline, and quickly locating the answer points during the sweeping. In Chapter 6, we propose new methods in computing skyline over joined relational tables. In Chapter 7, we summarize the contribution of the thesis. In the last chapter some possible future works based on this thesis will be presented.

# Chapter 2

# Background and Related Work

The skyline computation [13] originates from the *maximal vector problem* in computational geometry, which was proposed by Kung et al. [57]. Basically, let $V$ be a set of $n$ $d$-dimensional vectors and for any vector $v \in V$, let $x_i(v)$ denote the $i$th component of $v$. A dominating relationship $\preccurlyeq$ is defined on $V$ in a natural way, that is, for $v, u \in V$, $v \preccurlyeq u$ if and only if $x_i(v) \leq x_i(u)$ for all $i = 1, \ldots, d$. For $v \in V$, $v$ is defined to be a *maximal element* [1] of $V$ if there does not exist $u \in V$ such that $v \preccurlyeq u$ and $u \neq v$ [57]. Earlier work [9, 10, 65, 84] on computing the skyline was algorithmic in nature where all the data was assumed to be available in memory, and no attention was paid to making the algorithms external. Borzsonyi et al. first introduced the *maximal vector* as a notion of the "skyline operator" to the databases community [13], and studied the efficient ways of computing skyline in the context of large datasets. It attracted much attention by researchers and many algorithms were developed on this topic.

This chapter surveys existing approaches for computing skylines and the variants of this problem in large databases, namely: (1) full space skyline computation approach; (2) subspace skyline computation approach and (3) miscellaneous approach.

[1] In the original definition, all criteria in dimensions are to be maximized, while many algorithms are designed for the criteria to be minimized. The minimum criteria and maximal criteria are actually equivalent to some extent by a reversal of axis.

## 2.1 Full Space Skyline Computation Approach

### 2.1.1 Basic Divide and Conquer Approach (BDC)

Kung et al. present a basic divide-and-conquer algorithm (BDC) for maximal vector [57], which works as follows: (1) Sort dataset $V$ in each of $k$ dimensions and that the sorted dataset $V = \{v_1, \ldots, v_n\}$ is arranged as a sequence $v_1 \leq v_2 \leq \ldots \leq v_n$ along one of the dimensions (say $d_i$). Choose the median $m_i$ of $V$ in $d_i$, and partition $V$ into two parts $P_1 = \{v_1, \ldots, v_{n/2}\}$ and $P_2 = \{v_{n/2+1}, \ldots, v_n\}$ , with respect to the the sorted order over $d_i$ (here $m_i$ is $v_{n/2}$). (2) Find skyline $S_1$ of $P_1$ and $S_2$ of $P_2$ recursively partitioning $P_1$ and $P_2$ until one partition contains only one tuple. (3) Find the overall skyline as the result of *merging* $S_1$ and $S_2$ by eliminating those tuples of $S_2$ which are dominated by a tuple in $S_1$. Note that none of the tuples in $S_1$ can be dominated by a tuple in $S_2$ because a tuple in $S_1$ is better in dimension $d_p$ than every tuple of $S_2$.

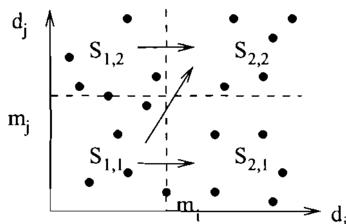



Figure 2.1: Illustration of BDC

As shown in Figure 2.1, the trick is to partition both $S_1$ and $S_2$ according to the median $m_j$ for some other dimension $d_j$, and here four partitions $S_{1,1}$, $S_{1,2}$, $S_{2,1}$ and $S_{2,2}$ are obtained. Now, it needs to merge $S_{1,1}$ and $S_{2,1}$, $S_{1,1}$ and $S_{2,2}$, $S_{1,2}$ and $S_{2,2}$. Note that it actually need not merge $S_{1,2}$ and $S_{2,1}$ since the tuples of these partitions are incomparable. Merging $S_{1,1}$ and $S_{2,1}$ (and the other pairs) is done recursively. That is, $S_{1,1}$ and $S_{2,1}$ are again partitioned. The recursion of the *merging* function terminates if all dimensions have been considered, or if one of the partitions is empty, or contains only one tuple.

BDC method is theoretically the best known algorithm for the worst case, in the order of $O(n(log(n))^{k-2})$ for $k \geq 4$ and $O(nlog(n))$ for $k = 2, 3$. Unfortunately, this is also the complexity of the algorithm in the best case.

### 2.1.2 Extension to BDC

Kung et al. proposed an algorithm [10, 76] which achieved expected running time linear in $n$ for the dataset under the CI assumption[2]. To find the maxima of a set $V$ of $n$ vectors, the algorithm partitions $V$ into two sets $A$ and $B$, each having $n/2$ vectors. It recursively finds the maxima of $A$ and $B$, call $M_A$ and $M_B$ respectively. The set of maxima vectors of $V$ is the set of maxima of $M_A \cup M_B$. To find the set of maxima of $M_A \cup M_B$, BDC method is applied to $M_A \cup M_B$. Unlike the sorting process performed for partitioning the dataset in BDC, this method need not sort the data. It stores the vectors in an $n \times d$ array of scalar values, and each vector is initially represented as a pair of integers which define the top and bottom endpoints of a segment in the array. Partition into further sub-partitions can be accomplished by taking the arithmetic mean of the endpoints as defining two new segments [10].

Bentley et al. applied virtual point technique to BDC approach and developed a fast linear expected time (FLET) [9]. Under the UI assumption[3], a virtual point $x$ not necessarily an actual point in the set is determined so that the probability that no point from the set dominates it is less than 1/n. The set of points is then scanned, and any point that is dominated by $x$ is eliminated. It is shown that the number of points $x$ will dominate, on average, converges on n in the limit, and the number it does not is $o(n)$. It is also tracked while scanning the set whether any point is found that dominates $x$. If some point did dominate $x$, it does not matter that the points that $x$ dominates were thrown away. Those eliminated points are dominated by a real point from the set anyway. BDC is then applied to the $o(n)$ remaining points, for a $O(kn)$ average-case running time. This happens at least $(n-1)/n$ fraction of trials. In the case no point was seen to dominate $x$, which should occur less than $1/n$ fraction of trials, BDC method is applied to the whole set. However, BDC's $O(nlog^{k-2}(n))$ running time in this case is amortized by $1/n$, and so contributes $O(log^{k-2}(n))$, which is $o(n)$. Thus, the amortized, average-case running time of FLET is $O(kn)$. FLET is no worse asymptotically than BDC in worst case.

[2]Consider the following properties of a set of points: (a) (independence) the values of the points over a single dimension are statistically independent of the values of the points along any other dimension; (b) (distinct values) points (mostly) have distinct values along any dimension; and (c) (uniformly) the values of the points along any one dimension are uniformly distributed. The properties of independence and distinct values are called component independence (CI).

[3]The properties of uniformly distributed, independence and distinct values are called uniform independence (UI)

### 2.1.3 External Algorithms for BDC

The BDC method is efficient only for small datasets. If the entire dataset can not fit in memory then the algorithm performs terribly since it needs to read and write the dataset multiple times simply for the partitioning process until a partition fits into main memory. An easy way to improve this is to divide the whole dataset into $m$ partitions so that each partition can fit into main memory [13]. For each partition, BDC algorithm can be applied to compute the skyline for that partition. Pairwise merging has to be done for each partition to compute the final result in a bottom-up way. During the merging for each two partitions, multi-way partitioning has to be done again to make sure that the two sub-partitions can be merged in main memory. Although the I/O behavior of the BDC algorithm can be improved to some extent, it is still not ideal as an external algorithm. Furthermore, this method is not suitable for on-line processing since it cannot produce any skyline until the partitioning phase completes. Another simple extension to the BDC algorithm is as follows: (1) load as many tuples as fit into the available main-memory buffers; (2) apply the BDC algorithm to this block of tuples in order to immediately eliminate tuples which are dominated by others. Here this is referred to as an "Early Skyline". Clearly, applying an "Early Skyline" incurs additional CPU cost, but it also saves I/O because less tuples need to be written and reread in the partitioning steps. In general an Early Skyline is attractive if the result of the whole skyline objects is small.

### 2.1.4 Basic Block Nested Loop (BNL)

A straightforward way to compute skyline is to compare each data object with all the other objects. The naive blocked nested loop-based (BNL) approach is developed to achieve this goal. In the context of large scale database where all the data cannot be loaded to the memory at the same time, every tuple may have to be processed with multiple rounds. An improved method is to keep a *window* of incomparable tuples during the comparison in main memory as a list of skyline candidates [13]. When the algorithm starts, the first data point is added to the list, while for each subsequent point $p$, three cases can occur: (1) if $p$ is dominated by any tuple in the list, it is eliminated as it is not possible to be a skyline object; (2) if $p$ dominates one or more points in the list, it is inserted, and all the dominated points by $p$ are removed; and (3) if $p$ is neither dominated by, nor dominates, any point in the list, it is inserted into the window if the memory capacity permits, otherwise, $p$ is

written to a temporary file on disk. If the temporary file is not empty after an iteration is finished, a new iteration has to be started to process the tuples in the temporary file. The process has to be repeated until there are no tuples being written to the temporary file in that iteration. In regards to when to output the tuples in the window as the final skyline objects after each iteration finishes, only the tuples inserted into the window before the first tuple is written to the temporary file can be output since those tuples have been compared to all the tuples in the dataset. The remaining tuples in the window have to be compared with the data in the temporary file. A technical detail during the computation is: each time if a point dominates other points being added, it is moved to the top of the list. This utilizes the heuristics that the new tuple being added could dominated more tuples than the existent ones so that less number of comparisons need to be conducted.

### 2.1.5 Sort-Filter-Skyline Computation (SFS)

Depending on the size of main memory, BNL may require a number of passes to finish the processing and the I/O cost could be very high. Furthermore, it requires the scanning of the whole dataset to output the first skyline object. Chomicki et al. suggested an improvement by proposing an algorithm called Sort-Filter-Skyline(SFS) [27], which makes a topological sorting of the data according to a monotonic score function. The advocated sorting is by volume descending $\prod_{i=1}^{k} t[d_i]$ or entropy descending $\sum_{i=1}^{k} \ln t[d_i]$ [27, 28] where $t[d_i]$ refers to the value of tuple $t$ in dimension $i$. Similar to BNL, SFS maintains a window and adopts multi-pass processing. The rational of topological sorting is that tuples with a lower score have higher possibility to dominate a larger number of tuples. So if these lower score tuples are inserted to the candidates list first, the number of comparisons can be largely reduced that makes the pruning process much quicker. In addition, when a tuple is added to the candidate list of the window, it is for sure a skyline point since this tuple cannot be dominated by any tuple that has not yet been presented due to the sorting. Such a presorting approach also enables the progressive output of skylines while BNL has to wait till all tuples are being compared. In regards to the window operation, SFS is also better since there is no need to replace the tuples already in the window.

### 2.1.6 Linear Elimination Sort For Skyline (LESS)

The LESS (linear elimination sort for skyline) method [39] combines features of SFS, BNL and FLET. LESS is similar to SFS in that it filters the records via a filter window. It sorts the tuples by their entropy scores. The major change from SFS is that LESS combines some computation into the external sorting process. In the first pass of the external sorting, it uses an *elimination-filter* (EF) window to eliminate tuples quickly. During the sorting, tuples with the best entropy scores seen so far are kept in the EF window. Each time when a block of tuples are read in for sorting, they are compared with the tuples in the EF window first. If being dominated by any tuple in it, they are dropped. At the same time, tuples currently in the EF window are dropped if they are worse than the best of the surviving tuples in the current block. Another combination with the external sorting is that during the last pass of the sorting, skyline filtering starts to be conducted as well. This will always save a pass for computing skylines. In case of an overflow of the SF window happens, LESS requires multiple passes to be done just like SFS.

Due to the filtering in the first pass of the sorting, LESS produces less tuples for skyline computation. At the same time, it is possible that one more pass is needed for the sorting than the normal procedure since some memory is allocated to the EF window. If the data is uniformly distributed along any dimension and also the values of the tuples over a single dimension are statistically independent of the values along any other dimension, LESS has an average-case runtime of $o((kn))$.

All these nested loop based methods have to scan the entire data file to return a complete skyline, since a skyline point could have a very large score due to the large values in all dimensions except one dimension which has a lowest value and thus appear at the end of the sorted list. Also in regard of output order the skyline objects, all these methods have the same fixed order which is decided by the sort order.

### 2.1.7 B-tree Minimum Value Index Approach (MVI)

The approaches introduced in previous subsections perform the computation of skylines "on-the-fly", and they must basically read the whole database at least once. Now in the remaining of this section, we summarize those index-based methods which need to visit only a fraction of the dataset.

The Minimum-Value-Index (MVI) approach [85, 30] organizes a set of $d$-dimensional

points in the form of $p = (p_1, p_2, \ldots, p_d)$ into $d$ lists along each dimension. A point is assigned to the $i$th list ($1 \le i \le d$), if and only if its coordinate $p_i$ on the $i$th axis is the minimum among all dimensions. So if point $p$ is in the $i$th list, $p_i \le p_j$ for any $j \ne i$. Table 2.1 shows the index for an example dataset (chosen from [72]) in Figure 2.2. Each list is sorted in ascending order of the points' minimum coordinate ($minC$, for short) and indexed by a B-tree along that dimension. Points that have the same $i$th coordinate (i.e., $minC$) in the $i$th list are called a batch. In Table 2.1, it is easy to see that points in list 2 are divided into five batches $\{k\}$, $\{i, m\}$, $\{h, n\}$, $\{l\}$, and $\{f\}$, while each point of list 1 constitutes an individual batch because all $x$ coordinates are different.

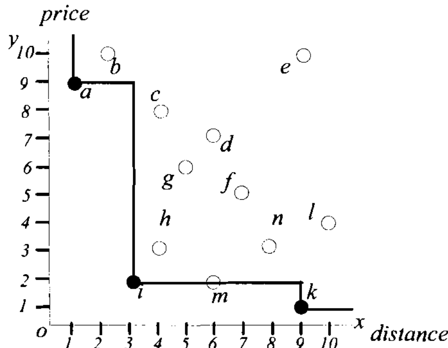



Figure 2.2: Example Data

Table 2.1: Minimum Value Index

| List1 | | List2 | |
|---|---|---|---|
| $a(1,9)$ | $minC = 1$ | $k(9,1)$ | $minC = 1$ |
| $b(2,10)$ | $minC = 2$ | $i(3,2), m(6,2)$ | $minC = 2$ |
| $c(4,8)$ | $minC = 4$ | $h(4,3), n(8,3)$ | $minC = 3$ |
| $g(5,6)$ | $minC = 5$ | $l(10,4)$ | $minC = 4$ |
| $d(6,7)$ | $minC = 6$ | $f(7,5)$ | $minC = 5$ |
| $e(9,10)$ | $minC = 9$ | | |

To compute the skyline, the first batch of each list is loaded by the algorithm, and the one with the minimum $minC$ is processed first since that is the one that has most potential to be a skyline. The processing of a batch involves two steps: (1) the skyline points inside a batch are computed first; (2) the skyline points are compared with the current final skyline points list, and if they are not dominated by any of the existent skyline, they are added to the final skyline list. After a batch in a list is processed, it moves to the next batch in the list. The scanning of the batches keeps moving on until a batch is reached which produces a skyline having all the coordinates smaller than the next batch (after the current batch being processed for that list) of all the lists. To illustrate the procedure with the example index in Table 2.1, the first batches being processed are $\{a\}$, $\{k\}$ which have identical $minC = 1$, in which case the algorithm handles the batch from list 1. Point a is added to the skyline list since the current list is empty. The algorithm then loads the next batch in list 1 $\{b\}$ which has $minC = 2$. Since batch $\{k\}$ from list 2 has a smaller $minC$ than $b$ and $k$ is not

dominated by a, it is inserted in the skyline. Similarly, the next batch handled is $\{b\}$ which is discarded since it is dominated by $\{a\}$. When the algorithm proceeds with batch $\{i, m\}$ and adds $i$ to the skyline, it does not need to proceed further, because both coordinates of $i$ are smaller than or equal to the $minC$ of the next batches (i.e., $\{c\}$, $\{h, n\}$) of lists 1 and 2, hence all the remaining points (in both lists) are dominated by $i$. This technique can return skyline points quickly at the top of the lists, and the order in which the skyline points are returned is fixed. Furthermore, as indicated in Kossmann et al. [56], the index built for $d$ dimensions cannot be used to retrieve the skyline on any subset of the dimensions because the list that an element belongs to may change according the subset of selected dimensions. An exponential number of lists must be precomputed to support the queries on arbitrary dimensions,in general.

### 2.1.8 R-tree Nearest Neighbor Approach (NN)

Nearest Neighbor(NN) processes the regions partitioned by the nearest-neighbor search recursively. A monotonic distance function $f$ of the points to the origin of the axis (e.g. Euclidean distance in L1 form, which equals to the sum of the coordinates of $p$) is fed to the algorithm as a criteria to select the nearest neighbor. The first nearest neighbor is a skyline object since if it is dominated by another object then the other object should be the first nearest neighbor of the origin. R-tree [41] and its variants R+-tree [81], R*-tree [7] are used as index structures since they are suitable for calculating nearest neighbors. As an example to show the algorithm of NN, let us take a look at the application of the algorithm to the 2-$d$ example dataset (chosen from [72]) in Figure 2.2 indexed by a R-tree. NN performs a nearest-neighbor query using function $f$ from the beginning of the axes (point $o$) on the R-tree. This can be done with any of the existing algorithms [78, 43]. Point $i$ is the nearest neighbor to the origin with the minimum distance (*mindist*) of 5 and it is part of the skyline. With this skyline object generated, all the points in the dominance region of $i$ (shaded area in Figure 2.3) can be pruned from further consideration. The remaining space is split into two partitions based on the coordinates $(i_x, i_y)$ of point $i$: (i) $[0, i_x)$ $[0, \infty)$ and (ii) $[0, \infty)$ $[0, i_y)$. In Figure 2.3, the first partition contains subdivisions 1 and 3, while the second one contains subdivisions 1 and 2. These resulting partitions are inserted into a to-do list since they are the only regions that need to be processed to find further skyline objects. While the to-do list is not empty, NN removes one of the partitions from the list and repeats the same process recursively. For instance, point a is the nearest neighbor in partition $[0, i_x)$

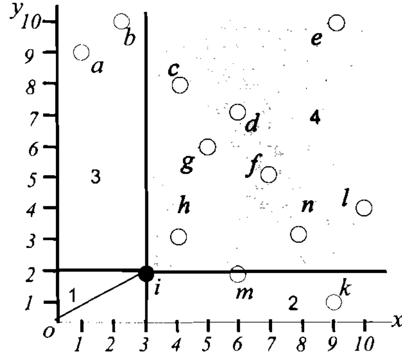



Figure 2.3: Discovery of $i$

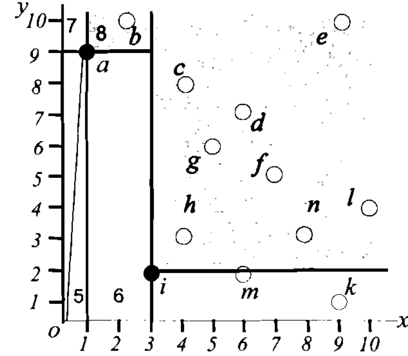
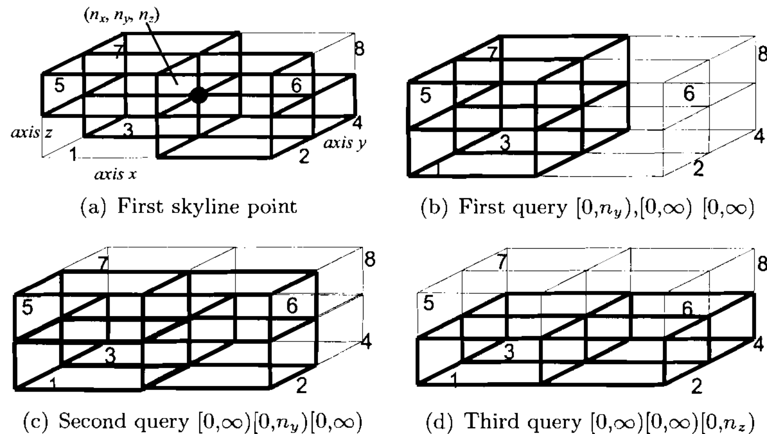



Figure 2.4: Discovery of $a$

$[0,\infty)$, which results in the partitions $[0, a_x)$ $[0,\infty)$ (subdivisions 5 and 7 in Figure 2.4) and $[0, i_x)$ $[0, a_y)$ (subdivisions 5 and 6 in Figure 2.4) being inserted into the to-do list. If a partition is empty, it can be simply discarded.

High dimension creates more complications for this method. Figure 2.5(a) shows a three-dimensional ($3D$) example (chosen from [56, 72]), where point n with coordinates ($n_x$, $n_y$, $n_z$) is the first nearest neighbor (i.e., skyline point). The NN algorithm will be recursively called for the partitions (i) $[0, n_x)$ $[0,\infty)$ $[0,\infty)$ (Figure 2.5(b)), (ii) $[0,\infty)$ $[0, n_y)$ $[0,\infty)$ (Figure 2.5(c)) and (iii) $[0,\infty)$ $[0,\infty)$ $[0, n_z)$ (Figure 2.5(d)). In general, for a $d$-dimensional data-space, a new skyline point causes d recursive applications of NN. In particular, each coordinate of the discovered point splits the corresponding axis, introducing a new search region towards the origin of the axis. In the case of 3 dimensions, one subdivision (the eighth in figure 2.5) among eight will not be searched by any query since it is dominated by the newly discovered skyline. However, each of the remaining subdivisions will be searched by two queries, for example, a skyline point in subdivision 2 will be discovered by both the second and third queries. To eliminate the duplication, Kossmann et al. [56] proposed the following elimination methods: (1) Laisser-faire: All the found skyline points so far are stored in a hash table in main memory. When a new skyline point p is discovered, it is probed to check whether it already exists in the hash table, and it is then either discarded or inserted into the hash table. Though this technique removes duplicates with little CPU overhead, it incurs very high I/O cost since large portion of the space will be accessed by multiple queries. (2)Propagate: To prevent duplicates from occurring, when a skyline point $p$ is found, the to-do list is scanned so that all the partitions containing $p$ are removed

and repartitioned according to $p$. The new partitions are inserted into the to-do list. This method has high CPU cost since the to-do list is scanned each time a skyline point is discovered. (3) Merge: The main idea is to merge partitions in to-do list to reduce the number of queries that have to be performed. Partitions contained in other ones can be eliminated in the process. This method also incurs high CPU cost since it can be expensive to find good candidates for merging. (4) Fine-grained partitioning: Instead of generating $d$ partitions after a skyline point is found, an alternative approach is to generate $2d$ non-overlapping subdivisions (excluding number 1 and 8 in Figure 2.5). But this generates a more complex problem of false hits, that is, it is possible that points in one subdivision (e.g., subdivision 4) are dominated by points in another (e.g., subdivision 2) which makes the algorithm impractical to produce the right results. According to the experimental evaluation of Kossmann et al. [56], propagate method was significantly more efficient than others (with fine-grained partitioning not implemented), but the best results were achieved by a hybrid method combining propagate and laisser-faire.




(a) First skyline point

(b) First query $[0,n_y),[0,\infty)\ [0,\infty)$

(c) Second query $[0,\infty)[0,n_y)[0,\infty)$

(d) Third query $[0,\infty)[0,\infty)[0,n_z)$

Figure 2.5: NN Partitioning for 3-dimensions.

### 2.1.9 R-tree Branch and Bound Approach (BBS)

Branch and Bound Skyline (BBS) algorithm [72, 73] which has been proved to be I/O optimal, takes a similar branch-and-bound paradigm as NN using an $R$-tree to index data. To cope with the disadvantages of NN, BBS takes a different way to create the to-do list.

The to-do list is a heap containing a list of MBRs/points. The heap is sorted according to the minimum distance (in L1 form) of the MBRs/points to the origin of the axis. At the beginning, BBS accesses the root node of the R-Tree and inserts all the entries in the root node to the heap first, then it begins the expanding and pruning process. This is illustrated in an example (chosen from [72]) with the data depicted in Figure 2.2 indexed by an $R$-tree shown in Figure 2.7. Firstly, entries $e_6$, $e_7$ are inserted into the heap. Then entry $e_7$ is removed from the heap and expanded. $e_7$'s children $e_3$, $e_4$, $e_5$ are then inserted into the heap. When $e_3$, which has the minimum distance in the heap, is expanded, the first skyline point $i$ is discovered and it is inserted into the skyline list. The algorithm proceeds with the next entry $e_6$. Among all the children of $e_6$, only the ones that are not dominated by any of the points in the current skyline list are inserted which results in $e_2$ being pruned. The next entry being processed is point $h$ which is also pruned due to the dominance by $i$. The algorithm continues in this way until the heap is empty. Figure 2.2 shows the processing steps in a table.

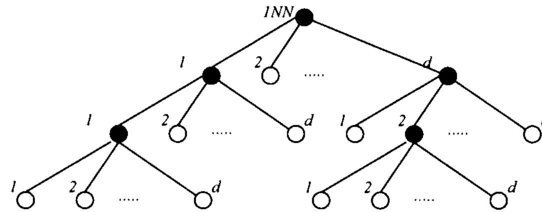



Figure 2.6: Recursion Tree

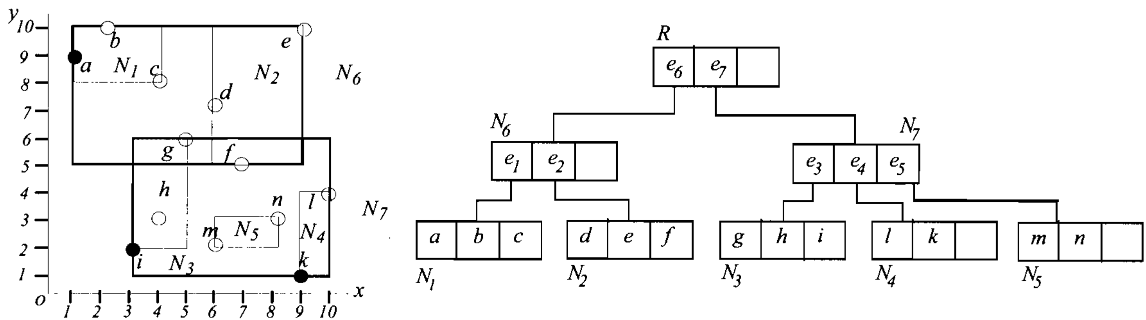



Figure 2.7: R-tree example

Table 2.2: Heap Contents

| Action | Heap contents | Skyline points |
|---|---|---|
| Access root | $< e_7, 4 >< e_6, 6 >$ | $\emptyset$ |
| Expand $e_7$ | $< e_3, 5 >< e_6, 6 >< e_5, 8 >< e_4, 10 >$ | $\emptyset$ |
| Expand $e_3$ | $< \mathbf{i}, 5 >< e_6, 6 >< h, 7 >< e_5, 8 >< e_4, 10 >< g, 11 >$ | $\{i\}$ |
| Expand $e_6$ | $< h, 7 >< e_5, 8 >< e_1, 9 >< e_4, 10 >< g, 11 >$ | $\{i\}$ |
| Expand $e_1$ | $< \mathbf{a}, 10 >< e_4, 10 >< g, 11 >< b, 12 >< c, 12 >$ | $\{i, a\}$ |
| Expand $e_4$ | $< \mathbf{k}, 10 >< g, 11 >< b, 12 >< c, 12 >< l, 14 >$ | $\{i, a, k\}$ |

### 2.1.10 The Bitmap Index Approach

Bitmap technique [85] relies on the fact that bitmap operations are fast, and it encodes a data point $p$ ($p = (p_1, p_2, \ldots, p_d)$, where $d$ is the number of dimensions) in bitmaps so that whether this point is a skyline point can be simply decided from the bitmaps. Information encoded in the bitmap contains the order of a point at each dimension while the real value is not important any more. Each bitmap representing a point is an $m$-bit vector, where $m$ is the total number of distinct values over all dimensions. If there are $m_i$ distinct values at dimension $i$, then $m = \Sigma(m_i)$. As shown in the example of 2-$d$ dataset (chosen from [72]) in Figure 2.2, there are 10 distinct values on each dimension and $m = 20$. Each value on the $i$th dimension is represented by $m_i$ bits. If $p_i$ is the $j_i$th smallest number on the $i$th axis, then its rightmost $(j_i - 1)$ bits are 0, and all the remaining ones 1. Table 2.3 contains the bitmap for points in Figure 2.2, where point $a$ has the smallest value (1) on the $x$ axis, so all bits for $x$ value of a are 1. Similarly, since $y$ value of $a$ is the ninth smallest on the $y$ axis, the rightmost 8 bits are 0. To decide whether a point is a skyline point, a new bit string needs to be created for that point. For example, point $c$ has bitmap representation (1111111000, 1110000000) and note that the rightmost fourth and the eighth bits equal to 1 on dimensions x and y respectively. By juxtaposing the fourth and eighth bit of every point, the algorithm creates two bit-strings, $cX = 1110000110000$ and $cY = 0011011111111$. Each of these bit-strings has 13 bits with one bit from each object in the dataset (with the order from top object a to bottom object n) and each bit is highlighted in bold. The 1s in the result of $cX \& cY = 0010000110000$ represents those points that dominate $c$, which are $c$, $h$, and $i$ in this case. Apparently, if the number is greater than 1, the considered point is not a skyline point. To compute the whole set of skyline points, this process has to be done for every point in the dataset.

Table 2.3: The Bitmap Approach

| id | Coordinate | Bitmap Representation |
|---|---|---|
| a | (1,9) | (1111111111,1100000000) |
| b | (2,10) | (1111111110,1000000000) |
| c | (4,8) | (1111111000,1110000000) |
| d | (6,7) | (1111100000,1111000000) |
| e | (9,10) | (1100000000,1000000000) |
| f | (7,5) | (1111000000,1111110000) |
| g | (5,6) | (1111110000,1111100000) |
| h | (4,3) | (1111111000,1111111100) |
| i | (3,2) | (1111111100,1111111110) |
| k | (9,1) | (1100000000,1111111111) |
| l | (10,4) | (1000000000,1111111000) |
| m | (6,2) | (11111000001,111111110) |
| n | (8,3) | (1110000000,1111111100) |

The efficiency of bitmap relies on the speed of bit-wise operations. However, if the number of distinct values is very large, representing the points in bitmaps will consume huge space. In addition, if the dataset is large, each of the bit-strings constituted for deciding the skyline-ness of a point will contain a large number of bits which could make the bit-wise operation slow. Also a cost to build these bit-strings requires the scanning of the whole dataset. In terms of skyline points output order, this approach depends on the data point insertion order just like BNL and SFS. In the case of dynamic datasets where insertions or deletions happen, this technique can not be applied since the rankings of attribute values could be changed during the data updates.

## 2.2 Subspace Skyline Computation

In a multi-dimensional dataset, skyline queries provide the best points with consideration to all dimensions, while in reality, various user groups may issue queries on only a subset of dimensions that they are interested in. So how to efficiently compute skyline objects in subspaces becomes an interesting topic.

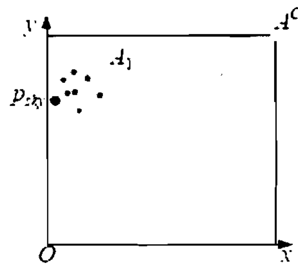


Figure 2.8: An Anchor $A_1$

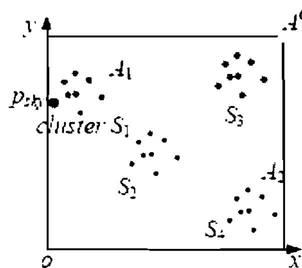


Figure 2.9: More Anchors

### 2.2.1 Computing Subspace Skylines via Multiple Anchors

SUBSKY is a technique which focuses on the problem of answering the query in an arbitrary subspace with a low dimensionality [87]. The core of SUBSKY is a transformation that converts multi-dimensional data to one dimension values so that effective pruning heuristics can be applied. Assuming a unit $d$-dimensional full space where each dimension has domain [0,1], and use the term maximal corner (*anchor*) for the corner $A^c$ of the data space having coordinate 1 on all dimensions. Each point $p$ is converted to a $1D$ value $f(p)$ which equals to the $L_\infty$ distance between $p$: $f(p) = \max_{i=1}^{d}(1 - p[i])$

Given a full space skyline point $p_{sky}$ in subspace $SUB$, if $f(p) < \min_{i \in SUB}(1 - p_{sky}[i])$, then $p$ can not belong to the skyline of $SUB$ since the minimum value among all dimensions of $p$ is still larger than the maximum value among all dimensions of the skyline point. To compute skyline objects of a subspace $SUB$, each point $p$ is accessed in descending order of $f(p)$, and meanwhile (i) the current set $S_{sky}$ of skyline points among the data already examined, and (ii) a value $U$ corresponding to the largest $\min_{i \in SUB}(1 - p_{sky}[i])$ for the points $p_{sky} \in S_{sky}$. The algorithms terminates when $U$ is larger than the $f(p)$ of the next $p$ to be processed.

In the case of uniform data distribution, $f(p)$ is always computed with respect to one *anchors* for the dataset (chosen from [87]) as shown in Figure 2.8. However, in practice where data are usually clustered, the $f(p)$ of various $p$ should be calculated with respect to different anchors to achieve greater pruning power as shown in Figure 2.9. For example, with skyline point $p_{sky}$, using maximal corner as the anchor could not prune any point in the upper left cluster $S_1$ though it could prune all points in $S_3$, while choosing point $A_1$ as the anchor for points in $S_1$ is a much better choice since the square area produced by

$\min(A_1[i] - p_{sky}[i])$ covers most points in $S_1$. So multiple anchors need to be created and points should be assigned to the appropriate anchors.

To find multiple anchors, SUBSKY projects all the data points onto the major perpendicular plane (the $d$-dimensional plane passing the maximal corner and perpendicular to the major diagonal of the data space) and partitions the points by clustering the projected points into $m$ (the predefined number of anchors) clusters. Finding an anchor for a cluster is to decide a point so that the (hyper-)square composed by the bottom-left point of the cluster(in the original space and that anchor point covers every point in the cluster.

Overall, SUBSKY first obtains $m$ anchors as stated in the previous paragraph on a random subset of the database. $f(p)$ (with the assigned anchor) of each point is managed with a B-tree that separates the points assigned to different anchors. Then the algorithm identifies the maximum $f(p)$ among all anchors. The algorithm then scans all the points assigned to each anchor in descending order of $f(p)$ values and updates the skyline list. The pruning capability of SUBSKY decreases very fast in the query of a high sub-dimensionality. However, there are two problems in SUBSKY: (1) the anchor points are never modified after the initial computation. Since the pruning power is largely decided by the choice of the anchor points, SUBSKY is not suitable for dynamic data where the data distribution may change over time. (2) The pruning ability of SUBSKY deteriorates fast with the increase of query dimensionality, which makes it inappropriate for computing the skylines of all subspaces.

### 2.2.2 Computing SKYCUBE with Sharing Strategies

Yuan et al. [95] proposes to compute a SKYCUBE, which consists of skylines of all possible non-empty subsets of a given set of dimensions for answering multiple subspace skyline queries, in order to achieve minimal response time. Naively computing the $2^d$-1 skyline results independently can be extremely expensive, so [95] focuses on the efficient construction of the SKYCUBE based on the sharing strategies in identifying the computation dependencies among multiple related skyline queries. In the SKYCUBE, the set of skyline points in each subspace is termed as a cuboid. Two approaches were proposed: the *BUS(Bottom-Up Skycube)* and the *TDS(Top-Down Skyline)*. *BUS* takes two sharing strategies by sharing the result and the sorting. Under the distinct value condition where values at each dimension of the dataset are different, it extends SFS by sharing $d$ sorted lists of objects during the

computation and each time the sorted dimension is picked for the subspace as the dimension that has largest domain value among all dimensions in that subspace. It computes each cuboid level by level from bottom to the top. Sharing the result means that the lower level cuboids (the set of skyline points in that subspace) are merged to form part of the upper level parent cuboids since the union of child cuboids belongs to the parent cuboids. If the distinct condition does not hold, it needs to check whether a subspace skyline object is really a superspace skyline object.

TDS relies on an extended DC algorithm and computes multiple related skyline queries by sharing the partitions and merging along a path of cuboids. This derives from the fact that in the unique value dataset, computing the skyline for a superspace includes the computing of skyline for the subspaces. While the merging process in the superspace includes the merging of its subspace. This is done by splitting the skyline of superspace into a skyline set in one subspace plus the remaining skyline set. For example, skylines for superspace $ABC$ can be divided into skylines for subspace $AB$ and the skyline tuple set not in $AB$. In particular, a structure called *skylist* is utilized in TDS. A *skylist* represents the skylines of a path without duplications in any subspace. The list contains a number of elements each storing the skyline points uniquely for the corresponding cuboid. For example, for a path $c$ along the lattice structure $c = \langle A, AB, ABC \rangle$, if the *skylist* is $\{p_3, p_2, p_5, p_1\}$, then it means subspace $AB$ has a set of skyline points $\{p_2, p_3, p_5\}$ while subspace $ABC$ has a set of $\{p_1, p_2, p_3, p_5\}$. Operations of *split*, *union* and *filter* are defined for the skyline list. To compute the skycube, TDS first constructs $\binom{d}{\lceil d/2 \rceil}$ skylists/paths that can cover every node in the lattice, and then each cuboid can be computed along the path in the shared BDC as described above. To handle the case of duplicate values in the datasets, it needs to compare each tuple with the superspace skylines to find in which dimensions the tuples to have the same values with the superspace skylines.

### 2.2.3 Updating SKYCUBE with the Compressed Structure

Compared to the context of static data used in [95], Xia et al. [94] investigate the important issue of updating the SKYCUBE in a dynamic environment, which focus on supporting concurrent and unpredictable subspace skyline queries in frequent updated databases. Obviously, the naive method that SKYCUBE is recomputed upon each update, is extremely inefficient and in turn affects the query performance severely.

To improve the storage of the SKYCUBE to support efficient update, they propose a new structure called the *Compressed SKYCUBE*, based on the concept of the minimum subspaces. The compressed SKYCUBE concisely represents the complete SKYCUBE and preserves the essential information of subspace skylines. Each skyline object is stored only in the cuboids which correspond to its minimum subspaces, and the compressed skycube contains only non-empty cuboids. Compared to the original SKYCUBE [95], the compressed SKYCUBE has much less duplicates among cuboids, and does not need to contain all cuboids. Since recomputing everything upon updates is obviously unacceptable because of the expensive costs of cuboid computations and disk accesses in retrieving objects. To minimize such costs during the updates of objects, they propose the object-aware update scheme for the compressed SKYCUBE. More specifically, they differentiate various cases such as when an update needs to retrieve new objects from the disk, when existing objects in the compressed SKYCUBE are affected, and etc.

### 2.2.4 Mining Frequent Skylines in Subspaces

In a high dimensional space, skyline points no longer provide any meaningful insights to the user as there are too many of them. A new metric called *skyline frequency* is introduced to rank the interesting skyline points among all produced. *Skyline frequency* is counted as how many times the skyline point is returned as a skyline in subspaces. On the other side, *dominating frequency* is the number of times a point being dominated in subspaces. Top-$k$ frequent skyline query is to find the best $k$ such skylines (with highest frequencies). The algorithm framework is quite straightforward: for every point $p$ in $D$, count the subspaces it has been dominated and maintain the top-$k$ points after every point is tested [19].

Let $DS(p,q) = (U,V)$ denote the set of all subspaces for which point $p$ dominates $q$ (*dominating subspaces* of $p$ over $q$), where $U$ are the dimensions $p$ is better than $q$, and $V$ are the dimensions $p$ is equal to $q$. Given two collections of subspaces $S_1, S_2 \subseteq S$, $S_1$ covers $S_2$ if $S_1 \supseteq S_2$. A $DS$ of $(U', V')$ is covered by $(U, V)$ if and only if (i) the union of $U$ and $V$ can cover the union of $U'$ and $V'$. (ii) $U$ can cover $U'$. $DS(p,q)$ is defined as a Maximal Dominating Subspace Set (MDSS) for point $q$ if there does not exist a point $p'$ that $DS(p',q)$ can cover $DS(p,q)$. So counting the dominating subspaces is to count the subspaces covered by those maximal dominating subspace sets. The algorithm maintains a threshold $\theta$ to keep track of the $k^{th}$ smallest dominating frequency among all the process points and it is initialized to $2^d - 1$. A top-$k$ frequent skyline set $R$ is also maintained

for storing the result. For each point $p$ in the dataset, the algorithm first compute the set $M$ of all the maximal dominating subspace sets of $p$ which is done by comparing all the other points with $p$ on all dimensions. Then the dominating frequency of $p$ is computed by counting the total number of subspaces that are covered by the maximal dominating subspaces in $M$. Exact counting can be time-consuming due to the large size of $M$, so an approximate counting method is also given by providing solution similar to DNF counting problem. The counting result is then compared with $\theta$ and if it is less than $\theta$ or the size of set $R$ is less than $k$, the new point is inserted or updated in $R$.

### 2.2.5 Mining Strong Skylines in Subspaces

With the aim at mining a set of interesting skyline points in high dimensional space, Zhang et al. [97] proposed *strong skyline points* as follows: given a space $S$, a subspace is said to be $\delta$-subspace if its skyline contains less than $\delta$ points. The union of the skyline points in all $\delta$-subspace of $S$ are called strong skyline points. Intuitively, such skyline points are strong in two reasons: (1) they represent points which are most difficult to dominate even when the number of dimensions being considered is reduced. While weaker skyline points are being removed due to the reduction of dimensions, only the strongest skyline points survive. (2) Since all other points are dominated by these small set of points, they are also considered strong in term of the ability to dominate points. In general, the number of strong skyline points is much less than the original skyline points.

Based on the property of $\delta$-subspace which is very similar to Apriori property of frequent itemset [1], two subspace search algorithms with breadth first strategy and depth first strategy respectively, are developed to find those $\delta$-subspaces. The existing BNL algorithm [13] is also improved to efficiently determine whether a given subspace is $\delta$-subspace by exploiting the properties of strong skyline points.

## 2.3 Miscellaneous Approach

### 2.3.1 Skyline Algorithms for Distributed Environments

In [5] and [4], the authors present an approach for computing skylines in distributed web information systems and show how it can be optimized for categorical data. Data is said to be categorical if the values for some attributes are chosen from a rather limited number

of alternatives (=categories). Consider, for example, hotel rooms: The "smoke" attribute, which can only set to either smoking or non-smoking, or a rating measured by one to five stars are examples for categorical attributes. The algorithm is based on a central mediator system. Attributes of the objects that should be queried (for example some ratings in Zagat [zag], the distance to a local point determined by MapQuest [map] ) are fetched from different sources sorted according to respective score functions. Based on that sorted access, the central skyline engine computes the skyline and incrementally presents its results to the user.

### 2.3.2 Skyline Computation in Mobile Environments

With more and more applications in mobile environments, the skyline computation faces more challenges. Authors in [46] study the skyline queries under this circumstance by assuming (1) each mobile device only holds a portion of the entire dataset; (2) devices communicate through MANETs; (3) and mobile users posing skyline queries are only interested in data pertaining to a limited geographical area, although the queries involve data stored on many mobile devices due to the storage limitations of the devices.

To improve the efficiency of skyline queries, two most important costs are considered: the cost of the communication among the mobile devices and the cost of query execution on the mobile devices. For reducing the communication cost, a distributed query processing strategy is proposed that takes advantage of the skyline dominance relationship to eliminate non-qualifying intermediate tuples, thus reducing the amount of data transmitted. For reducing the latter, on each mobile device involved, the local query processing is optimized using a hybrid storage model for the tuples, which have both spatial and non-spatial attributes.

### 2.3.3 Skyline Computation in Data Streams

Lin et al. focus on computing the skyline against the most recent $n$ of $N$ elements in a data stream [60]. They developed an effective pruning technique to minimize the number of elements to be kept. It can be shown that on average storing only $O(log^d N)$ elements from the most recent $N$ elements is sufficient to support the precise computation of all "$n$ of $N$" skyline queries in the $d$-dimensional space if the data distribution on each dimension is independent. Then, an encoding scheme is proposed, together with efficient

update techniques, for the stored elements, so that computing an "$n$ of $N$" skyline query in the $d$-dimensional takes $O(logN + s)$ time that is reduced to $O(dloglogN + s)$ if the data distribution is independent, where $s$ is the number of skyline points.

Tao et al. [86] study skyline computation in stream systems that consider only the tuples that arrived in a sliding window covering the $W$ most recent timestamps, where $W$ is a system parameter called the *window length.* The stream is append only, meaning that a tuple is not replaced before its expiry. Their objective is to maintain the skyline over the live data, and continuously output the skyline changes. Two general frameworks for maintaining stream skylines are proposed: (1) Lazy method handles two situations associated with the skyline changes (i) a new tuple arrives, or (ii) some skyline point expires. (2) Eager method aims at (i) minimizing the memory consumption by keeping only those tuples that are or may become part of the skyline in the future, and (ii) reducing the cost of the maintenance module, which is responsible for expunging the obsolete (i.e., dead) data from the database, and outputting the skyline stream.

### 2.3.4 Skyline Computation in Time Series Databases

Morse et al. consider the *continuous time-interval skyline* [23] operation involves data points that are continually being added or removed. Each data point has an arrival time and an expiration time associated with it that defines a time interval for which the point is valid. The task for the DBMS is to continuously compute a skyline for the data points that are valid at any given time. Basically, the skyline in the continuous case may change based on one of two events: (i) some existing data point $i$ in the skyline expires, or (ii) a new data point $j$ is introduced into the dataset. An algorithm, called *LookOut* is developed, based on the idea that: in the case of an expiration, the dataset must be checked for new skyline points that previously may have been dominated by $i$. These points must then be added to the skyline if they are domianted by some other existing skyline points. In the case of the insertion, the skyline must be checked to see if $j$ is dominated by a pint already in the skyline. If not, $j$ must be added to the skyline and existing skyline points checked to see if they are dominated by $j$. If so, they must be removed.

### 2.3.5 Skyline Computation with Partially-Ordered Domains

Chan et al. [16, 17] study the evaluation of skyline queries with partially-ordered attributes which include interval data (e.g., temporal data), type/class hierarchies, and set-valued domains. Because such attributes lack a total ordering, traditional index-based evaluation algorithms (e.g., NN and BBS) that are designed for totally-ordered attributes can no longer prune the space as effectively. Their solution is to transform each partially-ordered attribute into a two-integer domain that allows to exploit index-based algorithms to compute skyline queries on the transformed space. Based on this framework, three algorithms are proposed: $BBS^+$ is a straightforward adaptation of BBS, and SDC (Stratification by Dominance Classification) and $SDC^+$ are optimized to handle false positives and support progressive evaluation. Both SDC and $SDC^+$ exploit a dominance relationship to organize the data into strata. While SDC generates its strata at run time, $SDC^+$ partitions the data into strata offline. Two dominance classification strategies (MinPC and MaxPC) are also designed to further optimize the performance of SDC and $SDC^+$. The experimental results show that the proposed techniques outperform existing approaches by a wide margin.

### 2.3.6 Cooperative Database Retrieval Using High-Dimensional Skylines

Balke et al. [6] propose to combine the advantages of intuitive skyline queries and manageable top $k$ answer sets for cooperative retrieval systems by introducing an interactive feedback step presenting a representative sample of the (high-dimensional) skyline to users and evaluating their feedback to derive adequate weightings for subsequent focused top $k$ retrieval. Hence, each users information needs are conveniently and intuitively obtained, and only a limited set of best matching objects is retrieved. However, the huge size of skyline sets and the necessary time for their calculation remains an obstacle to efficient sampling for getting user feedback. Therefore they propose a sampling scheme to give users a first impression of the optimal objects in the database that is representative of the skyline set, manageable in size and efficient to compute, without computing the actual skyline. They also prove these characteristics and show how to subsequently estimate a users compensation functions by evaluating feedback on objects in the sample. Such a approach paves the way to overcome the drawbacks of todays cooperative retrieval systems by utilizing the positive aspects of skyline queries in databases.

### 2.3.7 K-Dominant Skylines Computation

Considering the original definition of skyline, a huge number of skyline objects will be produced in the situation of high dimension as it is getting harder for a point to be dominated by some other points in all dimensions. $k$-dominance is proposed to narrow down the number of skylines so as to make the result skylines more meaningful. It defines $k$-dominant skylines [18] as those points that are not $k$-dominated by any other object. A point $p$ is said to $k$-dominate $q$ if there are $k$ sub-dimensions in which $p$ is better than or equal to $q$ and better in at least one of those $k$ dimensions. Due to the feature that a non-($k$-dominant skyline) point is not necessarily be $k$-dominated by any $k$-dominant skyline point (which is different from a normal non-skyline point), the usual pruning procedure (such as BNL) can not be directly applied. To find the set of all $k$-dominant skyline points, the authors [18] proposed three algorithms: One-Scan algorithm, Two-Scan algorithm and Sorted-Retrieval algorithm.

The One-Scan algorithm compares a point with full space skylines (free skylines) to determine whether the point is $k$-dominant. This is due to the following property: if a point is not a $k$-dominant skyline, then there must exist a free skyline point that $k$-dominates that point. One-Scan algorithm maintains two lists ($R$ and $T$) during the scan to store intermediate $k$-dominant skylines and non-$k$-dominant skylines respectively (these two sets constitute the whole set of free skylines). It takes a similar way as SFS by sorting the points first and starting to find full space skyline progressively. During the process, each point is compared with the so-far found free skylines in $T$ first, if it is dominated, then there is no chance for it to become a $k$-dominant skyline. Otherwise, it is checked whether this point $k$-dominates any point in $R$ (which contains temporary $k$-dominant skylines so far), or being $k$-dominated by any point in $R$. Depending on the result, the point is inserted into $R$ or $T$. Those points being swapped out from $R$ are inserted back to $T$ since they are free skylines.

One-Scan algorithm needs to maintain free skyline points for the comparison, and the sets of free skyline points could consume large amount of space since the number is very big. Two-Scan algorithm computes the temporary $k$-dominant candidates (stored in a set $R$) in the first scan just like a normal SFS does but replacing "dominate" operation with "$k$-dominate" operation. But due to the unusual feature (from a full space skyline computation) as stated above, the result can have false positive points. So in the second scan, to determine whether a point $p$ in $R$ is actually $k$-dominant, it is sufficient to compare $p$ with each point $p'$ which is not in $R$ and $p'$ occurs earlier than $p$ in the original sorted list since all the later

ones have already been compared with $p$ in the first scan.

The third algorithm is "Sorted Retrieval" by first sorting all the points in each dimension which produces $d$ sorted arrays (where $d$ is the number of dimensions). Each sorted array has a cursor pointing to the current batch being processed (a batch is a set of points with the same value on the sorted dimension). Set $T$ is initialized with the whole dataset as candidates list for $k$-dominant skylines. The real ones are progressively moved to set $R$ while non-$k$-dominant skyline points in $T$ are progressively removed. The algorithm chooses one sorted list with the smallest batch value (just like the Minimum-Value-Index method [85] introduced before) to process next until $T$ is empty. Two steps are taken to process the batch. In the first step, if a currently processed point $p$ in the batch $k$-dominates any point in $T$, then that point is removed from $T$. Since each point could be processed $\|d\|$ times, a counter is associated with each point. A point being processed from the current list increases its counter with one. In the second step, if we find the point $p$ is in $T$ and also the value of the counter is $(d - k + 1)$, then $p$ is for sure to be a $k$-dominant skyline and it is moved to $R$. This is based on the observation that if there is a point $p'$ such that $k$-dominates $p$, then $p$ can be processed in an earlier batch of points than $p'$ in at most $(d - k)$ iterations. (i.e. $p$ is better than $p'$ in $(d - k)$ dimensions maximally).

### 2.3.8 Skyline Cardinality Estimation

Bentley et al. [10] established that the average number of skyline tuples is $O((\ln n)^{d-1})$ where $n$ is the number of tuples and $d$ is the number of skyline attributes, under the UI assumption. Godfrey [38] proved that expected skyline cardinality is $\Theta((\ln n)^{d-1}/(d-1)!)$ under the assumptions of sparseness over attributes' domains (namely that there are virtually no duplicate values) and statistical independence across attributes. Chaudhuri et al. [23] aimed to relax the strong assumptions of previous work on skyline cardinality estimate, such as assuming attribute value independence, all attributes are unique and completely ordered etc. They also derive cardinality estimates for categorical attributes.

# Chapter 3

# The Semantics of Skyline

The skyline operator is important for multi-criteria decision making applications. Although many recent studies developed efficient methods to compute skyline objects in a specific space (usually the full space), the fundamental problem on the *semantics* of skylines remains open: Why and in which subspace is (or is not) an object in the skyline? What is the relationship between the skylines in the subspaces and those in the super-spaces? How can we effectively analyze the subspace skylines? Practically, users may also be interested in the skylines in any subspaces depending on the specific application requirements. Then can we efficiently compute skylines in various subspaces?

We explore the above questions in this chapter, investigate the semantics of skylines, propose the subspace skyline analysis, and extend the full-space skyline computation to subspace skyline computation. To the best of our knowledge, this is the first study on the semantics of skylines and the subspace skyline analysis [75]. We introduce a novel notion of *skyline group* which essentially is a group of objects that are in the skylines of some subspaces coincidentally. We identify the *decisive subspaces* that qualify skyline groups in the subspace skylines. The new notions concisely capture the semantics and the structures of skylines in various subspaces. Multidimensional roll-up and drill-down analysis is introduced in exploring the semantics. We also develop an efficient algorithm, *Skyey*, to compute the set of skyline groups and, for each subspace, the set of objects that are in the subspace skyline. A performance study is included to evaluate our approach.

## 3.1 Motivation

The *skyline operator* is important for multi-criteria decision making applications and this has been well recognized by the community of data analysis. A very classic illustrative example of skyline query is to search for hotels in Nassau (Bahamas) which are cheap and close to the beach [13]. Suppose each hotel has two attributes: the price and the distance to the beach. In the context of this example query, hotel $H1$ *dominates* hotel $H2$ (or, $H1$ is a better choice than $H2$ ) if $H1.price \leq H2.price$, $H1.distance \leq H2.distance$ and at least one inequality holds. The skyline hotels are formed by those that are not dominated by others in terms of price and distance to the beach. In other words, the skyline hotels provide candidates for all the possible trade-offs between price and distance to the beach that are superior to other hotels.

There are many recent studies on efficient methods for skyline computation ( please refer to Chapter 2 for a brief review). However, the fundamental questions about the semantics of skyline remain open.

**Example 1 (Intuition)** Let us take a look at the example in Figure 3.1 with the dataset containing 5 objects in 2-d dataspace $(X, Y)$. It is easy to verify that objects $p$, $q$ and $w$ are in the skyline in space $(X, Y)$ since none of them is dominated by any other objects.

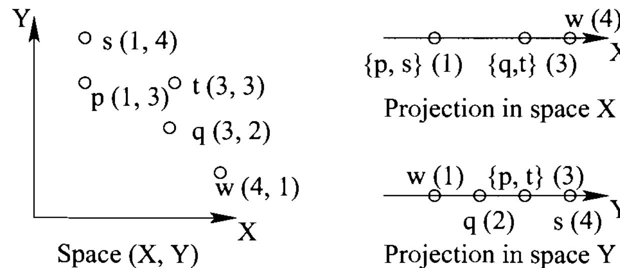



Figure 3.1: An Example Showing the Intuition

In the same figure, we also plot the projections of the objects on dimensions $X$ and $Y$, respectively. The projections of $p$ and $s$ collapse in subspace $X$, and both of them are in the subspace skyline of $X$. In subspace $Y$, the projection of $w$ is in the subspace skyline. As the comparison under trivial subspace $\emptyset$ is usually meaningless to the user, hereafter, we use the term "subspace" to refer to only non-empty ones except when specifically mentioned.

Although $p$, $q$ and $w$ are all skyline objects in the full space $(X, Y)$, there are differences

in terms of how they become part of the skyline and this is worth to study carefully in order to discover other important features. Both $p$ and $w$ have projections that are part of the subspace skylines (i.e., in subspaces $X$ and $Y$, respectively), but for any possible subspaces, no projection of $q$ is in subspace skyline. A closer look reveals that $p$ is already sufficient to qualify as a skyline object by taking a value of 1 on dimension $X$. Similarly, the value 1 of $w$ on dimension $Y$ is a determining factor for the skyline membership of $w$. On the other hand, $q$ is a skyline object only if both dimensions $X$ and $Y$ are considered – it needs two dimensions to qualify.

Although both $s$ and $t$ are not in the skyline in space $(X, Y)$, they are still subtly different if we look at the subspaces. The projection of $s$ is in the skyline in subspace $X$ but $t$ has no projection belonging to a subspace skyline. $s$ is dominated by $p$, nevertheless, the dominance is "partial" – $s$ takes the same value as $p$ in dimension $X$ and thus has the chance to be in the skyline in subspace $X$. ■

With the involvement of only a two dimensional space and a few objects, the skyline in the example of Figure 3.1, is simple and easy to be perceived, while the general situation may be much more complicated when dimensionality is high and dataset is large. Nevertheless, the observations in the above example reveal one important intuition: whether an object is in the skylines of the full space or of some subspaces is determined by the values of the object in some *decisive subspaces.* The decisive subspaces and the values in those subspaces vary from object to object in the skyline. For a particular object, the values in its decisive subspaces justify *why and in which subspaces the object is in the skyline* – the *semantics* of the object with respect to skyline.

*Why should we care about the semantics of skylines?* Semantics is important to understand the data. For example, Section 3.6.1 analyzes a real data set which contains 17,226 records of Great NBA Players' seasonal performance from 1960 to 2001. Wilt Chamberlain's performance in 1960 is in the skyline of the full space, which can be identified by the conventional skyline computation methods. However, one may wonder which merits really make Wilt that outstanding. The semantics analysis in Section 3.6.1 shows that Wilt was outstanding in total rebounds in the season of 1960 by achieving the record of 2149 in the NBA history. The attribute of total rebounds is the decisive subspace that establishes his superior status. In fact, he was not exceptional in any other factors such as total assists. As another example, Michael Jordan does not hold any record high in any single attribute.

However, his performance in 1988 is in the skyline of subspaces of (total points, total rebounds, total assists) and (games played, total points, total assists), and also in the skyline of the full space. Those two subspaces are decisive, and explain why Michael Jordan is an outstanding player. Clearly, such information cannot be captured by traditional skyline computation and analysis in the full space, hence it is hard to get people understand what is/are the key factor(s) that makes an object outstanding.

The concepts of skyline groups and decisive subspaces can also be used immediately for efficient query answering. For example, given an object or a group of objects, the *skyline membership queries* is to determine the subspaces where the object(s) are in the subspace skylines. On the other hand, given a subspace, the *subspace skyline query* finds the set of objects whose projections are in the subspace skyline.

The investigation of skylines in subspaces naturally introduces the problem of *subspace skyline analysis and computation*: for a set of subspaces, find the objects and their projections that are in the skylines of these subspaces, and analyze their relationship. This type of query is interesting and useful in practice since, more often than not, a user may want to interactively examine the skylines with respect to different combinations of attributes.

Motivated by the above observations, in this chapter, we study the problem of multidimensional subspace skyline computation and analysis. We make the following contributions.

- *We develop a theoretical framework to answer the question about semantics of skyline: Why and in which subspaces is an object in the skyline?* The semantics of skyline objects is concisely captured by the novel notions of *skyline groups* and the corresponding *decisive subspaces*. The subspaces, where an object (or a set of objects) is in the skyline, can be effectively determined by the skyline groups that the object belongs to and their decisive subspaces.

- *We investigate the problem of subspace skyline analysis.* Skylines in subspaces can be concisely summarized by skyline groups. Moreover, skyline objects in the full space can be selected as the representatives in skyline groups. They catch the "contour" (i.e., technically, the projections) of the skylines. The multidimensional roll-up and drill-down analysis is useful to support the online analytic processing of skylines.

- *We present efficient algorithms for subspace skyline computation.* We develop an algorithm to compute both the set of skyline groups and, for each subspace, the

set of objects that are in the subspace skyline. The algorithm makes good use of the findings in the semantics research and recursively reduces the set of objects to be searched. Moreover, a local sorting technique is developed so that computing skylines in subspaces can be substantially faster than a naïve method running a skyline computation algorithm on every subspace from scratch.

- *We develop effective methods to index skyline groups and their decisive subspaces, and use them in query answering.* Particularly, we address two types of queries: skyline membership queries and subspace skyline queries. We show that the queries can be answered efficaciously by the proper materialization of the skyline groups and their decisive subspaces.

- *A performance study using both synthetic and real data sets is conducted to evaluate our approach.* We showcase some interesting findings in the skyline semantic analysis using the real data set about technical statistics of NBA players, and justify why they are meaningful in practice. Moreover, we use benchmark synthetic data sets to test the efficiency and the scalability of our algorithm.

The rest of this chapter is organized as follows. In Section 3.2, we extend the concept of skyline to multidimensional subspaces, and examine the subspace skylines in unique value data sets, a simple case where objects do not share values on any dimensions. In Section 3.3, we introduce the notion of skyline groups and decisive subspaces, and justify how the new notions capture the semantics of objects with respect to skyline. In Section 3.4, we tackle the problem of subspace skyline analysis. In Section 3.5, we present an algorithm for subspace skyline computation. An extensive performance study is reported in Section 3.6. Section 3.7 summarizes the whole chapter.

## 3.2 Multidimensional Subspace Skyline and a Unique-Value Case

In this section, we first extend the concept of skyline to multidimensional subspaces, and then examine a simple case where on any dimension, no two objects share the same value on that dimension.

### 3.2.1 Subspace Skyline

Hereafter in this chapter, we consider a set of objects $S$ in an $d$-dimensional space $\mathcal{D} = (D_1, \ldots, D_d)$ by default, where dimensions $D_1, \ldots, D_d$ are in the domain of numbers and usually they are continuous integers as $1, \ldots, d$. For the easiness and clarity in description, sometimes we also use enumeration of letters such as $A, B, C \ldots$. In the rest of the chapter, we often do not explicitly mention $S$ and $\mathcal{D}$ when they are clear in the context for the sake of brevity.

For objects $p, q \in S$, $p$ is said to *dominate* $q$ if $p.D_i \leq q.D_i$ for $1 \leq i \leq d$ and there exists at least one dimension $D_j$ such that $p.D_j < q.D_j$. Object $p$ is a *skyline object* if $p$ is not dominated by any other objects in $S$.

The notion of skyline can be naturally extended to subspaces.

**Definition 3.2.1 (Subspace skyline)** A subset of dimensions $\mathcal{B} \subseteq \mathcal{D}$ ($\mathcal{B} \neq \emptyset$) forms a (non-trivial) $|\mathcal{B}|$-*dimensional subspace* of $\mathcal{D}$. For an object $p$ in space $\mathcal{D}$, the *projection* of $p$ in subspace $\mathcal{B}$, denoted by $p_{\mathcal{B}}$, is a $|\mathcal{B}|$-tuple $(p.D_{i_1}, \ldots, p.D_{i_{|\mathcal{B}|}})$, where $D_{i_1}, \ldots, D_{i_{|\mathcal{B}|}} \in \mathcal{B}$ and $i_1 < \cdots < i_{|\mathcal{B}|}$.

The projection of an object $p$ ($p \in S$) in subspace $\mathcal{B} \subseteq \mathcal{D}$ is in the *subspace skyline* (of $\mathcal{B}$) if $p_{\mathcal{B}}$ is not dominated by any $q_{\mathcal{B}}$ in $\mathcal{B}$ for any other object $q \in S$. $p$ is also called a *subspace skyline object (of $\mathcal{B}$).* ■

For example, in Figure 3.1, the projections of both $p$ and $s$ are in the subspace skyline in subspace $X$, and the projection of $w$ is in the subspace skyline in subspace $Y$.

### 3.2.2 Subspace Skylines in Unique Value Data Sets: A Simple Case

In this subsection, we consider the *unique value* data sets where for any objects $p, q$ and any dimension $D_i$, $p.D_i \neq q.D_i$. Clearly, in a unique value data set, there does not exist two objects that share the same projection in any subspace.

Interestingly in this case, the subspace skyline objects have the monotonicity property as follows.

**Theorem 3.2.2 (Monotonicity of skyline membership in unique value data sets)** *In a unique value data set, if an object $p$ is in the skyline of subspace $\mathcal{B}$, then for any superspace $\mathcal{B}' \supset \mathcal{B}$, $p$ is also in the subspace skyline of $\mathcal{B}'$.*

**Proof.** We prove by contradiction. Suppose there exists a superspace $\mathcal{B}' \supset \mathcal{B}$ such that $p$ is not in the subspace skyline of $\mathcal{B}'$. There must be another object $q$ such that $q$ dominates $p$ in $\mathcal{B}'$. In other words, for any dimension $D_i \in \mathcal{B} \subset \mathcal{B}'$, $q.D_i \leq p.D_i$. In a unique data set, according to the definition, $q.D_i \neq p.D_i$. Thus, $q.D_i < p.D_i$. That means $q$ dominates $p$ in $\mathcal{B}$, which leads to a contradiction to the the assumption that $p$ is in the subspace skyline of $\mathcal{B}$. ■

Theorem 3.2.2 discloses the structure of subspace skylines in a unique value data set: every subspace skyline object must be a projection of some global skyline object.

**Proposition 3.2.3 (Subspace skyline objects in unique value data sets)** *For any object $p$ and subspace $\mathcal{B}$, $p_{\mathcal{B}}$ is in the subspace skyline of $\mathcal{B}$ only if $p$ is a skyline object in the full space.*

**Proof.** Trivially, $\mathcal{B} \subseteq \mathcal{D}$ where $\mathcal{D}$ is the full space. According to Theorem 3.2.2, $u$ must be in the skyline of $\mathcal{D}$. The proposition holds immediately. ■

The monotonicity in the unique value data sets suggests a straightforward framework to compute subspace (true subspaces excluding full space) skylines: we take the set of skyline objects in the full space as seeds and project them into subspaces in the order of descending dimensionality. The recursive projection of a super space skyline object stops once it is not in the subspace skyline of some subspace $\mathcal{B}$. In other words, we do not need to search any subspaces of $\mathcal{B}$. The framework is shown in Figure 3.2. We assume that the size of dataset is greater than 0 (which is always true in the practical cases), and this guarantees that the set of skyline objects in full space is not empty.

The algorithmic framework in Figure 3.2 can be implemented efficiently. For example, we can enumerate the subspaces systematically using the *subspace enumeration tree* which will be discussed in Section 3.5.2, such that each subspace should be checked only once. The correctness and the completeness of the algorithmic framework immediately follows Theorem 3.2.2 and the proposition.

## 3.3 Skyline Semantics

Starting from this section, we consider general data sets which may not have the property of *unique value*. In other words, two objects may share the same values on some dimensions.

**Input:** A unique value data set of objects $S$ in space $\mathcal{D}$;
**Output:** for each subspace, the set of objects in the subspace skyline;
**Method:**

```
1:  find L_D, the set of skyline objects in the full space;
2:  IF |D| > 1 THEN
3:     FOR EACH (|D| - 1)-dimensional subspace B, call recursive-search(B, L_D);

Function recursive-search(B, L)
4:  find L_B, the set of skyline objects in subspace B and in set L;
5:  output L_B as the skyline in subspace B if L_B ≠ ∅;
6:  IF |B| = 1 OR L_B = ∅ THEN RETURN;
7:  FOR EACH (|B| - 1)-dimensional subspace B'
8:     IF B' has not been searched before THEN call recursive-search(B', L_B);
9:  RETURN;
```

Figure 3.2: The Algorithmic Framework of Searching Subspace Skylines in No-sharing Data Sets

As will be shown soon, the situations are much more complicated than the simple case discussed in Section 3.2.2.

In this section, we first show some observations that inspires the ideas. Then, we introduce the new notions. Last, we elaborate on how the new notions capture the semantics of skyline objects and answer skyline membership queries.

### 3.3.1 Key Observations and Ideas

To find the relationship of skyline objects among different subspaces, first let's look at one interesting question: if an object $p$ is in the skylines of subspaces $\mathcal{C}_1$ and $\mathcal{C}_2$ such that $\mathcal{C}_1 \subset \mathcal{C}_2$, can we make an extrapolation that $p$ is also in the skyline of any subspace $\mathcal{C}$ in between, i.e., $\mathcal{C}_1 \subset \mathcal{C} \subset \mathcal{C}_2$? As shown in Section 3.2.2, this property is appealing since it may extremely simplify the determination of skyline membership in subspaces. Unfortunately, the general situation is far from being so simple.

**Example 2 (Key Observations)** Consider the objects in Table 3.1 as our running example. We obtain the following two observations.

First, object $q$ is in the skylines of full space $(A, B, C, D)$ and of subspace $A$. However, it is not in the skyline of subspace $(A, B)$, since its projection, $(2, 5, *, *)$, is dominated by

Table 3.1: A Set of Objects as Our Running Example

| Objects | $A$ | $B$ | $C$ | $D$ |
|---|---|---|---|---|
| $p$ | 2 | 5 | 7 | 8 |
| $q$ | 2 | 5 | 6 | 8 |
| $s$ | 2 | 4 | 7 | 8 |
| $t$ | 3 | 4 | 6 | 9 |

$(2, 4, *, *)$, the projection of $s$. This demonstrates that, in general, for subspaces $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}$ such that $\mathcal{C}_1 \subset \mathcal{C} \subset \mathcal{C}_2$, *even though an object is in the subspace skylines of* $\mathcal{C}_1$ *and* $\mathcal{C}_2$*, it may not be in the subspace skyline of* $\mathcal{C}$.

Second, objects $p$, $q$ and $s$ collapse in subspace $(A, D)$. The projection $(2, *, *, 8)$ is in the subspace skyline of $(A, D)$. Thus, any values on the dimensions in subspaces $(A)$, $(D)$ or $(A, D)$ that qualifies $p$ as a subspace skyline object in those subspaces also results in the same qualification for $q$ and $s$, and vice versa. In other words, *if a group of objects collapse in a subspace* $\mathcal{B}$ *and the shared projection is in the subspace skyline of* $\mathcal{B}$*, then the objects in the same group share the skyline membership in all subspaces of* $\mathcal{B}$. ■

The observations in Example 2 lead to the following ideas.

- Generally, the skyline membership is not monotonic – being in the skyline of subspace $\mathcal{B}$ does not automatically makes that object in the skyline of super-spaces of $\mathcal{B}$. The object may be dominated in the super-spaces of $\mathcal{B}$ by some other objects which have the same values in $\mathcal{B}$ as the current object.

- Objects coincide and form groups in subspaces. The skyline memberships in subspaces are shared by all objects in the same group. The convergence and divergence of groups from subspace to subspace play critical roles in forming skylines of various subspaces. Therefore, it is critical to capture groups of objects of which the shared projections are in the skylines of the projected subspaces.

### 3.3.2 Skyline Groups and Decisive Subspaces

Following the ideas stated in the previous section, let us consider objects that collapse in subspaces. They form groups that are critical in our multidimensional subspace skyline analysis.

**Definition 3.3.1 (C-group)** Let $G \subseteq S$ be a subset of objects and $\mathcal{B} \subseteq \mathcal{D}$ be a subspace. $(G, \mathcal{B})$ is a *coincident group* (or *c-group* for short) if for each dimension in $\mathcal{B}$, all objects in $G$ share a same value. The *projection* of the group in $\mathcal{B}$, denoted by $G_{\mathcal{B}}$, is $u_{\mathcal{B}}$ where $u \in G$.

A c-group $(G, \mathcal{B})$ is *maximal* if there do not exist any other objects $v \in (S - G)$ that share the same values as those in $G$ on dimensions in $\mathcal{B}$, and objects in $G$ do not share the same value on any other dimension $D \in (\mathcal{D} - \mathcal{B})$. $\mathcal{B}$ is called the *signature subspace* of $G$. ∎

**Example 3 (C-group)** Consider objects $p$, $q$ and $s$ in Table 3.1. They share the same value on dimension $A$. Thus, $p$, $q$ and $s$ form a coincident group (or c-group for short) on $A$.

We can not add new objects into the group $(\{p, q, s\}, A)$ since no other objects have the same value of 2 on dimension $A$, but it can be expanded by including more dimensions according to the definition of *maximal C-Group*. $p$, $q$ and $s$ share the same values on dimension $A$ and $D$. Thus, we can maximize the group to include dimension $D$. The maximal c-group is $(\{p, q, s\}, (A, D))$. ∎

Given a subset of objects $G$, we define $\mathcal{I}(G)$ as the maximal set of dimensions that all objects in $G$ share the same values. That is,

$$\mathcal{I}(G) = \{D | D \in \mathcal{D}, \forall u, v \in G : u.D = v.D\}.$$

Moreover, for a subspace $\mathcal{B}$ and a set of objects $G$, we define $\mathcal{O}(G, \mathcal{B})$ as the maximal set of objects that share the same values on dimensions in $\mathcal{B}$ as objects in $G$. That is,

$$\mathcal{O}(G, \mathcal{B}) = \{v | v \in S, \forall D \in \mathcal{B} \forall u \in G : v.D = u.D\}.$$

Using the two operators defined above, maximal c-groups can be derived for any given subset of objects, or a subset of objects and a subspace that forms a c-group. The following lemma gives the derivation, and it holds immediately based on the related definitions.

**Lemma 3.3.2 (C-group)** *For a given subset of objects $G$, $(\mathcal{O}(G, \mathcal{I}(G)), \mathcal{I}(G))$ is a maximal c-group. For a given c-group $(H, \mathcal{B})$, $(\mathcal{O}(H, \mathcal{B}), \mathcal{I}(\mathcal{O}(H, \mathcal{B})))$ is a maximal c-group.* ∎

To understand why the cases for unique value data set are simple, we observe the property below which derives directly from the definitions of unique value data sets and c-groups.

**Lemma 3.3.3 (C-group in unique value data sets)** *In a unique value data set, for any subset of objects $G \neq \emptyset$,*

$$\mathcal{I}(G) = \begin{cases} \mathcal{D} & \text{if } |G| = 1 \\ \emptyset & \text{if } |G| > 1 \end{cases}$$

*For any subset of objects $H \neq \emptyset$ and subspace $\mathcal{B} \neq \emptyset$,*

$$\mathcal{O}(H, \mathcal{B}) = \begin{cases} H & \text{if } |H| = 1 \\ \emptyset & \text{if } |H| > 1 \end{cases}$$

*For any maximal c-group $(G, \mathcal{B})$, $|G| = 1$ and $\mathcal{B} = \mathcal{D}$ if $\mathcal{B} \neq \emptyset$.* ■

We are particularly interested in maximal c-groups whose projections are in the skyline of some subspaces. Intuitively, we want to capture the subsets of values in their projections that are decisive to their skyline memberships.

**Definition 3.3.4 (Skyline group and decisive subspace)** Maximal c-group $(G, \mathcal{B})$ is called a *skyline group* if $G_{\mathcal{B}}$ is in the subspace skyline of $\mathcal{B}$.

For skyline group $(G, \mathcal{B})$, a subspace $\mathcal{C} \subseteq \mathcal{B}$ is called *decisive* if (1) $G_{\mathcal{C}}$ is in the subspace skyline of $\mathcal{C}$; (2) $\mathcal{O}(G, \mathcal{C}) = G$; and (3) there exists no proper subspace $\mathcal{C}' \subset \mathcal{C}$ such that conditions (1) and (2) also hold for $\mathcal{C}'$.

The *signature* of skyline group $(G, \mathcal{B})$ is written as $Sig(G, \mathcal{B}) = \langle G_{\mathcal{B}}, \mathcal{C}_1, \ldots, \mathcal{C}_k \rangle$, where $\mathcal{C}_1, \ldots, \mathcal{C}_k$ are all decisive subspaces of the skyline group. ■

Conditions (1) and (3) are straightforward. Condition (2) requires that the decisive subspaces are exclusive to the group $G$. This reflects our intension to catch the decisive factors for a group of objects that are in the (subspace) skylines. We will revisit this point soon when we discuss the semantics.

**Example 4 (Skyline group)** Consider the objects in Table 3.1 again. In the full space, $q$, $s$ and $t$ are skyline objects, therefore, each of them forms a maximal c-group in space $(A, B, C, D)$. Each group contains only one object.

For group $(q, ABCD)$, where $q$ and $ABCD$ are shorthands for set $\{q\}$ and subspace $(A, B, C, D)$, respectively, subspace $CD$ is decisive. Please note that $AD$ is not a decisive subspace for the group, since $q$ collapses with $p$ and $s$ in $AD$ and the maximal c-group in $AD$

contains three objects, i.e., $\mathcal{O}(q, AD) = pqs$. In other words, condition (2) in the definition is violated. Another decisive subspace for this group is $AC$. Thus,

$$Sig(q, ABCD) = \langle (2,5,6,8), AC, CD \rangle.$$

As another example, for group $(pqs, AD)$, its projection is in the subspace skyline of $AD$. The group has two decisive subspaces, namely $A$ and $D$. Thus,

$$Sig(pqs, AD) = \langle (2,*,*,8), A, D \rangle.$$

Similarly, we have $Sig(qt, C) = \langle (*,*,6,*), C \rangle$. ∎

### 3.3.3 Semantics of (Subspace) Skyline Objects

Now we study the question about the semantics: *For a given object or a group of objects, can we determine the subspaces where the projections of the object(s) are in the subspace skyline?*

**Theorem 3.3.5 (Decisive subspace)** *For skyline group $(G, \mathcal{B})$, if $\mathcal{C}$ is a decisive subspace, then for any subspace $\mathcal{C}'$ such that $\mathcal{C} \subseteq \mathcal{C}' \subseteq \mathcal{B}$, $G_{\mathcal{C}'}$ is in the subspace skyline.*

**Proof.** We prove by contradiction. Suppose $G_{\mathcal{C}'}$ is not in the subspace skyline, and is dominated by an object $w_{\mathcal{C}'}$ in subspace $\mathcal{C}'$. Then, $w \notin G$. For each dimension $D \in \mathcal{C}'$, $w.D \leq G_{\mathcal{B}}.D$ and the inequality holds on at least one dimension. On the other hand, since $\mathcal{C}$ is decisive, $G_{\mathcal{C}}$ is not dominated by the projections of any other objects. Thus, $G_{\mathcal{C}} = w_{\mathcal{C}}$. That means, $\mathcal{O}(G, \mathcal{C}) \supset G$, which violates condition (2) in Definition 3.3.4. ∎

Theorem 3.3.5 indicates how decisive subspaces capture the semantics of skyline objects: *The skyline membership of an object or a group of objects is established by its decisive subspaces.* Comparing with what we have found in observation 1 in section 3.3.1 , Theorem 3.3.5 shows a nice feature for deciding subspace skylines.

**Example 5 (Semantics)** As shown in Example 4, $Sig(q, ABCD) = \langle (2,5,6,8), AC, CD \rangle$. Thus, $q$ is in the skyline of subspaces inclusively bordered by $ABCD$, $AC$ and $CD$, as shown in Figure 3.3(a). This also explains why we opt for the representation of signature. ∎

The signature of skyline group $(q, ABCD)$ explains why and in which subspaces $q$ is in the skyline without any accompanying coincident objects. $q$ coincides with $p, s$ and $t$ in some subspaces and thus may jointly be in some subspace skylines. This is captured by the corresponding skyline groups.

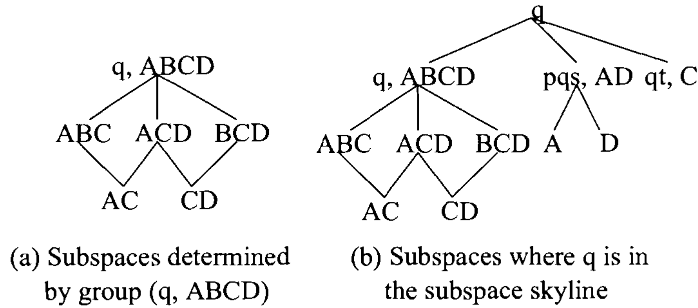



Figure 3.3: The Subspaces Where Object $q$ in Table 3.1 Belongs to the Skyline.

**Theorem 3.3.6 (Semantics)** *An object $u$ is in the skyline of subspace $\mathcal{C}$ if and only if there exists a skyline group $(G, \mathcal{B})$ and its decisive subspace $\mathcal{C}'$ such that $u \in G$ and $\mathcal{C}' \subseteq \mathcal{C} \subseteq \mathcal{B}$.*

**Proof.** (Direction if). Following Theorem 3.3.5, $G_{\mathcal{C}}$ is in the subspace skyline. Since $u \in G$, $u_{\mathcal{C}}$ is also in the subspace skyline.

(Direction only-if). Consider the group of objects $\mathcal{O}(u, \mathcal{C})$. All objects in the group are in the subspace skyline of $\mathcal{C}$ since they share the same values as $u$ on dimensions in $\mathcal{C}$. Following Lemma 3.3.2, $(\mathcal{O}(u, \mathcal{C}), \mathcal{I}(\mathcal{O}(u, \mathcal{C})))$ is a maximal c-group. Furthermore, it is easy to see that the group must be in the skyline of subspace $\mathcal{I}(\mathcal{O}(u, \mathcal{C}))$. Thus, the group is a skyline group. We notice that subspace $\mathcal{C}$ satisfies conditions (1) and (2) of Definition 3.3.4. Thus, if $\mathcal{C}$ is minimal, then $\mathcal{C}$ itself is decisive, i.e., $\mathcal{C}' = \mathcal{C}$. Otherwise, there must exists a $\mathcal{C}' \subset \mathcal{C}$ that $\mathcal{C}'$ is decisive. ∎

### 3.3.4 Answering Skyline Membership Queries

To answer *skyline membership query* which is: given an object or a group of objects, determine the subspaces where the object(s) are in the subspace skylines, Theorem 3.3.6 provides a generic framework with the use of skyline groups and their signatures.

The framework is simple. Suppose the set of skyline groups and their signatures are materialized. (The algorithm for computing skyline groups and their signatures will be given in Section 3.5.) Then, *instead of searching all possible subspaces, we only need to check the skyline groups in which the object is a member.* This is effective since only the signatures of the skyline groups are needed. Moreover, the skyline groups can be indexed by their signatures to speed up the search. To illustrate, we give an intuitive example here.

**Example 6 (Semantics – continued)** Continued from Example 5, $q$ is a member of skyline group $(pqs, AD)$, which has decisive subspaces $A$ and $D$. Thus, $q$ is also in the subspace skylines of $A$, $D$ and $AD$. Similarly, as a member of group $(qt, C)$, $q$ is in the subspace skyline of $C$. The complete set of subspaces where $q$ is in the skyline is shown in Figure 3.3(b).

■

## 3.4 Subspace Skyline Analysis

The notion of skyline groups naturally leads us to explore skylines in subspaces. When skylines in all subspaces are considered, it is imperative to ask: *How are the subspace skylines formed and what is the relationship among them?*

### 3.4.1 Intuition

We try to decipher some elegant structures embedded in the subspace skylines.

Skylines in subspaces consist of projections of objects. For a projection that is in the skyline of a subspace, the set of objects that share the same projection form a c-group. By the c-group containment relationship, a concise lattice structure is formed for the projections in subspace skylines. The lattice is called the *skyline projection lattice* (Theorem 3.4.1 in Section 3.4.2) .

The projection lattice may contain redundant information. The critical point here is that some projections in skylines of different subspaces may be made by the same maximal group of objects. Conceptually, a skyline group is a maximal group of objects that coincide in some subspaces and whose projections are also in the subspace skyline. Therefore, we can use skyline groups to derive a concise representation. The lattice of skyline groups is called the *skyline group lattice* and is a quotient lattice of the skyline projection lattice (Theorem 3.4.2 in Section 3.4.2).

Manipulating groups of objects all the time is still inconvenient. Ideally, we would like to select some representatives for the skyline groups. Fortunately, this is achievable since each skyline group must contain at least one object that is in the skyline of the full space. This indicates that the full space skyline casts the contours of skylines in subspaces.

### 3.4.2 Skyline Group Lattice

For $u \in S$, a projection $u_{\mathcal{B}}$ is called a *skyline projection* if it is in the skyline of $\mathcal{B}$. We can define a relation $\sqsubseteq$ on the set $\mathcal{P}$ of all skyline projections: for $r, h \in \mathcal{P}$ that are in the subspace skylines of $\mathcal{B}_1$ and $\mathcal{B}_2$, respectively, $r \sqsubseteq h$ if $\mathcal{B}_1 \supseteq \mathcal{B}_2$ and $r_{\mathcal{B}_2} = h$.

**Theorem 3.4.1 (Skyline projection lattice)** *Let $\mathcal{P}$ be the set of all skyline projections with respect to a set of objects $S$. $(\mathcal{P}, \sqsubseteq)$ is a complete lattice if $(*, *, \ldots, *)$ and $\emptyset$ are treated as the two trivial skyline projections for the unit element and the zero element, respectively.*
**Proof.** Obviously, $\sqsubseteq$ is a partial order on $\mathcal{P}$. We also notice that $\emptyset$ is the projection of any objects in subspace $\emptyset$ (the trivial subspace), and $(*, *, \ldots, *)$ is the projection of an empty set of objects on all dimensions. They are trivial and just technically make up the lattice. The completeness of the lattice follows from the fact that the number of skyline projections is limited. ∎

To characterize that multiple skyline projections may be made by one maximal group of objects, we define an equivalence among skyline projections as follows. For any projection $r$ in subspace $\mathcal{B}$, define the *pre-image* of $r$ as the set of objects that have $r$ as the projection in $\mathcal{B}$, denoted by $pre(r) = \{u | u \in S, u_{\mathcal{B}} = p\}$. For two skyline projections $r$ and $h$ in subspaces $\mathcal{B}_1$ and $\mathcal{B}_2$, respectively, they are equivalent (in terms of being generated by the same group of objects), denoted by $r \sim h$, provided $pre(r) = pre(h)$.

**Theorem 3.4.2 (Skyline group lattice)** *Let $\mathcal{SG}$ be the set of all skyline groups. $(\mathcal{SG}, \sqsubseteq)$ forms a complete lattice where $\sqsubseteq$ is on the projections in the groups. Moreover, $(\mathcal{SG}, \sqsubseteq) = (\mathcal{P}, \sqsubseteq)/\sim$.*
**Proof.** The claim follows from the fact that a skyline group also expands to include all possible dimensions where the objects share the same projections. For any skyline projections $r$ in subspace $\mathcal{B}_1$ and $h$ in subspace $\mathcal{B}_2$ such that $r \sim h$, let $G = pre(r) = pre(h)$. We show $r$ and $h$ are in fact in the same skyline group.

According to Lemma 3.3.2, $(\mathcal{O}(G, \mathcal{I}(G)), \mathcal{I}(G))$ is a maximal c-group. From the definition of pre-image, we know $\mathcal{O}(G, \mathcal{I}(G)) = G$, $\mathcal{B}_1 \subseteq \mathcal{I}(G)$ and $\mathcal{B}_2 \subseteq \mathcal{I}(G)$. Now, we show that $(G, \mathcal{I}(G))$ is a skyline group by contradiction.

Suppose $(G, \mathcal{I}(G))$ is not a skyline group, i.e., $G_{\mathcal{I}(G)}$ is not in the skyline of subspace $\mathcal{I}(G)$. Then, there must exists an object $u \neq G$ such that $u_{\mathcal{I}(G)}$ dominates $G_{\mathcal{I}(G)}$, i.e., for

each dimension $D \in \mathcal{I}(G)$, $u.D \leq v.D$ where $v \in G$. However, since $G_{\mathcal{B}_1}$ is in the skyline of subspace $\mathcal{B}_1$, $u_{\mathcal{B}_1} = v_{\mathcal{B}_1}$. In other words, $u \in pre(p) = G$. That leads to a contradiction.

Thus, $r$ and $h$ are in fact the projection of skyline group $(\mathcal{O}(G, \mathcal{I}(G)), \mathcal{I}(G))$ on $\mathcal{B}_1$ and $\mathcal{B}_2$, respectively. That means $r$ and $h$ belongs to the same skyline group. ■

Theorem 3.4.2 shows that skyline groups capture skyline projections in subspace skylines effectively, and the signatures of skyline groups serve as the summarization. Immediately, we know that the number of skyline groups is at most the number of skyline projections.

*Practically, is the summarization using skyline groups meaningful?* In practice, data is more or less correlated. Thus, objects may share values in some dimensions and form groups. In addition to capturing the semantics of skyline objects, skyline groups also summarize data records collapsing in some subspaces and appearing in some subspace skylines.

### 3.4.3 Skyline Groups and Skyline Objects

Although skyline groups provide a succinct summarization of the skylines in various subspaces, it can still be inconvenient and costly to manage all group members if a data set contains a large number of objects. *Can we select some representative objects from the skyline groups?*

Encouragingly, we observe that each skyline group contains at least one skyline object in the full space.

**Theorem 3.4.3 (Skyline object)** *For any skyline group $(G, \mathcal{B})$, there exists at least one object $u \in G$ such that $u$ is in the skyline of full space $\mathcal{D}$.*

**Proof.** Let $u$ be an object in $G$ such that, in the full space $\mathcal{D}$, $u$ is not dominated by any other objects in $G$. Such an object exists provided $G \neq \emptyset$. We show that $u$ is a skyline object in $\mathcal{D}$ with respect to the set of all objects $S$.

Suppose $u$ is dominated by $v \notin G$ in the full space $\mathcal{D}$, then two cases may arise. First, $u_{\mathcal{B}} = v_{\mathcal{B}}$, then $v \in G$ and it contradicts the assumption that $u$ is not dominated by any other objects in $G$. Second, if there exists a dimension $D \in \mathcal{B}$ that $u.D > v.D$, then given $u$ is dominated by $v$ in the full space, $u_{\mathcal{B}}$ is dominated by $v_{\mathcal{B}}$. That leads to a contradiction to the definition of skyline group that $u_{\mathcal{B}}$ is in the subspace skyline. ■

Theorem 3.4.3 indicates that the skyline objects in the full space play critical roles in the construction of subspace skylines – their projections are sufficient to represent the "*contour*"

of the skyline, i.e., the dimension values of the projections in the subspace skyline. In other words, *an object that is not in the skyline of full space can be in the skyline of some subspace only if it collapses to some full space skyline object(s) in those dimensions.*

Please note the pre-condition of Theorem 3.4.3 that group $(G, \mathcal{B})$ is a skyline group (not just a maximal group) is important. Generally, a maximal c-group that is not a skyline group may still have a skyline object in the full space as a member. For example, in Figure 3.1, the group $(pt, Y)$ is a maximal c-group and $p$ is a skyline object in the full space $(X, Y)$, but the group itself is not a skyline group.

For a data set $S$, we can obtain the set $SK$ of skyline objects in the full space. An object $u$ is in the skyline of subspace $\mathcal{B}$ in $S$ if and only if there exists an object $v$ that is in the skyline of the full space such that $u_{\mathcal{B}} = v_{\mathcal{B}}$ and $v$ is also in the skyline of the same subspace $\mathcal{B}$ in $SK$.

Moreover, for a data set $S$, let $SK$ be the set of skyline objects in the full space and let $\mathcal{SG}_S$ and $\mathcal{SG}_{SK}$ be the skyline group lattices on data sets $S$ and $SK$, respectively. On the relationship between subspace skyline groups in $S$ and the subspace skyline groups in its set of full space skyline objects $SK$, we have the result that $\mathcal{SG}_{SK}$ is a quotient lattice of $\mathcal{SG}_S$.

This can be verified from two directions. First, according to Theorem 3.4.2, both $\mathcal{SG}_S$ and $\mathcal{SG}_{SK}$ are complete lattices. For any objects $u, v \in SK$, if $u$ and $v$ are in a skyline group $(G, \mathcal{B})$ in $\mathcal{SG}_S$, then we have $u_{\mathcal{B}} = v_{\mathcal{B}}$. Therefore, $u$ and $v$ must also be in the skyline on set $SK$. On set $SK$, let $G' = \mathcal{O}(\{u, v\}, \mathcal{B})$ and $\mathcal{B}' = \mathcal{I}(G')$. $(G', \mathcal{B}')$ is a maximal c-group. Since $u$ and $v$ are in the skyline of $\mathcal{B}$, $(G', \mathcal{B}')$ is also a skyline group on $SK$. According to Theorem 3.4.3, every skyline group on $S$ must contain at least one skyline object in the full space. Thus, any skyline group $(G, \mathcal{B})$ on $S$ can be mapped to a skyline group $(\mathcal{O}(\{u, v\}, \mathcal{B}), \mathcal{I}(G'))$ on $SK$.

On the other hand, for any skyline group $(G'', \mathcal{B}'')$ on $SK$, skyline group $(\mathcal{O}(G'', \mathcal{B}''), \mathcal{B}'')$ on $S$ is mapped to the group on $SK$. Thus, the mapping is a surjection from $\mathcal{SG}_S$ to $\mathcal{SG}_{SK}$. The claim that $\mathcal{SG}_{SK}$ is a quotient lattice of $\mathcal{SG}_S$ follows.

Together with Theorems 3.4.3, the above result immediately have two practically useful applications. First, efficient algorithms can be derived for subspace skyline computation, which will be discussed in Section 3.5. Second, they can also lead to a novel OLAP style analysis of subspace skylines, which will be showcased in Section 3.4.4.

If we are only concerned with the projections in the subspaces skyline, only the skyline

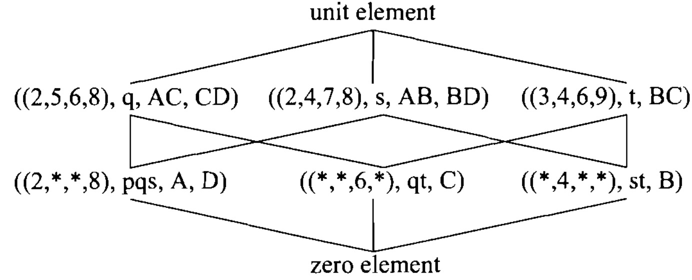



Figure 3.4: Skyline Group Lattice for Table 3.1

objects in the full space are needed for the analysis. In such a case, we do not need to manipulate all objects. This potentially leads to a significant reduction in the computational cost.

### 3.4.4 OLAP Analysis on Skylines

Since the skyline groups form a complete lattice, it is natural to introduce the multidimensional roll-up and drill-down analysis on skyline groups.

**Example 7 (OLAP analysis)** Figure 3.4 shows the skyline group lattice in our running example (Table 3.1). For each node in the lattice, the projection, the skyline objects, and the decisive subspaces are shown.

By browsing Figure 3.4, the following structural information about the subspace skylines can be presented.

- Subspace skylines. The information is recorded in the signatures.
- Relationships between skylines in subspaces. For example, from the figure, we know that an object is in the subspace skyline of $C$ if it has value 6 on $C$. There are two ways to further qualify the object as a skyline object in the full space: either having value 4 on $B$ (i.e., object $t$), or having value 2 on $A$ or 8 on $D$ (i.e., object $q$). The latter two values ($A = 2$ and $D = 8$) always come together.
- Closure information. From the figure, we can learn that it is impossible to have an object in the subspace skyline of $BCD$, but not in the subspace skyline of $ABCD$. Although a naïve method to derive this information has to check all objects in the data set, we derive this information from only the skyline groups. ∎

*In practice, why are such roll-up and drill-down operations useful?* Suppose all objects in our running examples are stock data and the user is not only interested whether a stock is good in general, but also in which attributes it is specially good. For example, when a user examines subspace $C$, she finds that both $q$ and $t$ are subspace skyline objects. This is interesting to her, but she would like to find out further in what other subspaces $q$ is also good and is better than $t$. Then, she finds $AC$ and $CD$ and their super-spaces through a roll-up.

Clearly, the online roll-up and drill-down analysis is not available in the traditional skyline analysis.

## 3.5 Subspace Skyline Computation

Given a data set, the problem of *subspace skyline computation* is to compute, for each non-empty subspace, the set of objects that are in the skyline of the subspace. At the same time, we also want to compute the complete set of skyline groups and their signatures as the summarization of the skylines.

### 3.5.1 Finding Skyline by Sorting

A lexicographic order can be defined on the set of objects $S$. For any objects $u, v \in S$. $u \prec v$ if there exists an $i_0$ $(1 \leq i_0 \leq n)$ such that $u.D_i = v.D_i$ for $(1 \leq i < i_0)$ and $u.D_{i_0} < v.D_{i_0}$. $u \preceq v$ if $u \prec v$ or $u = v$. Apparently, the lexicographic order $\preceq$ is a total order.

As shown in [27], skyline objects in the full space $\mathcal{D}$ can be found in two steps, as illustrated in the following example.

**Example 8 (Skyline computation by sorting)** Let us compute skyline objects in space $(X, Y)$ for the objects in Figure 3.1. In the first step, we sort all objects in the lexicographic order. The sorted list is $p(1,3)$, $s(1,4)$, $q(3,2)$, $t(3,3)$, $w(4,1)$.

The second step is as follows. We initiate the set of skyline objects as empty. Then, we scan the sorted list once. For each object $u$ in the list, we compare $u$ against the current set of skyline objects. If $u$ is not dominated, then $u$ is a skyline object and is inserted into the set.

For example, since $p(1,3)$ is the first one in the sorted list, it is not dominated and is inserted into the set. The next object, $s(1,4)$, is dominated by $p(1,3)$ in the current set of

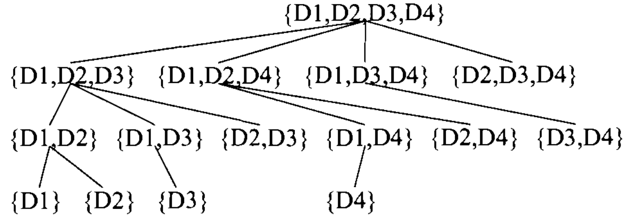



Figure 3.5: A Top-down Subspace Enumeration Tree

skyline objects, and thus is discarded. The third object, $q(3, 2)$, is compared with $p(1, 3)$, and is not dominated. Thus, $q$ is also inserted into the set as a new skyline object. $t(3, 3)$ is discarded since it is dominated by $p(1, 3)$. Last, $w(4, 1)$ is inserted into the set since it is not dominated by either $p$ or $q$. The set of skyline objects $\{p, q, w\}$ is returned. ∎

### 3.5.2 Top-down Subspace Enumeration Tree

In order to search for skylines of all subspaces thoroughly, we search subspaces in a depth-first manner. The complete set of subspaces can be enumerated systematically using a (top-down) subspace enumeration tree. For example, Figure 3.5 shows a tree enumerating subspaces of space $(D_1, D_2, D_3, D_4)$.

A top-down subspace enumeration tree is a variation from a conventional set enumeration tree [79] in the way of enumeration. In a conventional set enumeration tree, search starts from the empty set, and each child adds a new element to the parent set. This is bottom-up. In the top-down subspace enumeration tree here, we start from the full space, and each child explores a proper subspace with one less dimension. In other words, a top-down subspace enumeration tree can be regarded as a bottom-up compliment set enumeration tree. The reason for this arrangement is that the search from super-spaces to subspaces enables us to recursively use Theorem 3.4.3 to prune the set of objects under consideration. This point will become clear in Section 3.5.3.

### 3.5.3 Algorithm *Skyey*

Given a set of objects $S$ in space $\mathcal{D}$ as input, Algorithm *Skyey* returns the set of skyline groups (in the form of signatures) and, for each non-empty subspace, a list of objects that are in the corresponding subspace skyline. We first describe the algorithm. An example (Example 10) will follow.

### Finding Skyline Objects in Full Space

As a first step, *Skyey* sorts all objects in the lexicographic order and finds the set of skyline objects in the full space, as illustrated in Example 8. The list of skyline objects in the full space can be output. Every distinct full space skyline object forms a skyline group. In other words, if two objects $u$ and $v$ have exactly the same value in all dimensions of $\mathcal{D}$, and both of them are in the full space skyline, then they share the same skyline group.

After the set of skyline objects in the full space is found, *Skyey* makes one scan of all the other objects (i.e. objects that are not in the skyline of full space). Each non-skyline object is compared with the skyline objects. For each non-skyline object $u$ and a skyline object $v$, if the maximal set of dimensions on which $u$ and $v$ share common values (i.e., the set of dimensions $\mathcal{I}(\{u, v\})$) is non-empty, then a tag of $(u, \mathcal{I}(\{u, v\}))$ is attached to $v$. The tag means $u$ and $v$ coincide in subspace $\mathcal{I}(\{u, v\})$.

After this step, all non-skyline objects can be discarded. We do not need to access them anymore in the rest of the algorithm. The rest of *Skyey* searches the subspaces by a depth-first traversal of the subspace enumeration tree (Section 3.5.2).

### Efficient Local Sorting

During the traversal of the subspace enumeration tree, a node is going to be visited. Suppose the current node corresponds to a subspace $\mathcal{B}$. To identify the objects in the subspace skyline of $\mathcal{B}$, a naïve method is to sort the skyline objects in the parent node. However, repeatedly sorting objects for different subspaces can be expensive if there are many objects and many subspaces. Interestingly, we can reuse the sorted list in the parent node, which can reduce the sorting cost substantially.

**Example 9 (Local sorting)** Suppose the skyline objects in the full space $(D_1, D_2, D_3, D_4)$ is evaluated by sorting. In order to find the objects in the subspace skyline of $(D_1, D_2, D_3)$ (i.e., the first child of the root node in Figure 3.5), we do not need to sort the objects – they are already sorted since a sorted list by $(D_1, D_2, D_3, D_4)$ is also a sorted list by $(D_1, D_2, D_3)$. In fact, we do not need to do any extra sorting when we search the subspaces in the leftmost branch of the subspace enumeration tree.

Now, let us consider the second leftmost leaf node in Figure 3.5, $D_2$. The objects are sorted by $(D_1, D_2)$ in the parent node. Instead of a complete sorting, merge sort can serve the same purpose and save the computation cost. The idea is as follows.

The sorted list by $(D_1, D_2)$ can be regarded as divided into groups according to $D_1$. Within each group, objects are sorted by $D_2$. We only need to merge the groups according to $D_2$. ■

### Finding Objects in Subspace Skylines

Once the objects are sorted in the subspace, the objects in the subspace skyline can be identified easily. Please note that we only sort the list of skyline objects in the parent node. To make the list of objects in the subspace skyline of the current node complete and the decisive subspace accurate, we need to find the objects that are not in the skyline of the parent node, but are in the skyline of the current subspace. This is because not all the subspace skyline objects are in the skyline objects set of its super space. The finding can be achieved by examining the information recorded in the tags using the following two rules.

- *Identifying objects that are not in the skyline of the parent node, but are in the skyline of the current subspace.* For an object $v$ (a skyline object from the parent node) that is also in the skyline of the current subspace $\mathcal{B}$, we check the tags attached to $v$. For any tag $(u, \mathcal{C})$ such that $\mathcal{B} \subseteq \mathcal{C}$, $u$ should also be output as an object in the skyline of the current subspace, and is put in the same skyline group where $v$ is in. The reason is that $u$ shares the same values as $v$ on all dimensions in $\mathcal{B}$.

- *Identifying decisive subspaces.* For each skyline group $G$, we check whether the number of objects in the skylines of the current subspace is the same as the number of objects in the parent node subspace. If the number of objects changes, that means the group changes, i.e., $\mathcal{O}(G, \mathcal{B}) \supset \mathcal{O}(G, \mathcal{B}')$. Then, we add the parent subspace as a temporary decisive subspace to group $G$, and create a new group for $G' = \mathcal{O}(G, \mathcal{B})$, and $G_{\mathcal{B}}$ is recorded in the signature of the new group. If the group already has a temporary decisive subspace that is a super-space of the newly inserted one, then the super-space should be removed.

The depth-first search proceeds recursively. According to Theorem 3.4.3, only objects that are in the skyline of the current subspace should be passed to the children. Tags should be created for those objects that are in the skyline of the parent subspace but not in the skyline of the current subspace.

**Example 10 (Algorithm *Skyey*)** We demonstrate the algorithm *Skyey* using our running example (Table 3.1).

As the first step, we sort all four objects in the lexicographic order and identify three objects q, s, t in the skyline of the full space $(A, B, C, D)$ and tags are created. We create a skyline group for each object and each group has ABCD as their temporary decisive space.

Then, we go to subspace $(A, B, C)$. We do not need to sort the objects since the sorted list in the root node can be reused. Again, in this subspace, all the three full space skyline objects are in the subspace skyline, and the groups do not change.

We further go to subspace $(A, B)$. Again, we reuse the sorted list. In this subspace, $q$ and $t$ are dominated by $s$. Thus, $ABC$ should be added to the groups of $q$ and $t$, respectively, as temporary decisive subspace. Two tags, $(p, AD)$ and $(t, B)$, should be created and attached to $s$. Later, temporary decisive subspace $ABC$ for group $q$ will be removed when another decisive subspace $AC$ is inserted into the group.

We turn to subspace $A$, where $s$ is still in the subspace skyline. However, the group needs to be expanded, since the tag $(pq, A)$ indicates $p$ and $q$ are also in the subspace skyline. Thus, the parent subspace, $AB$ should be added to group $s$ as a temporary decisive subspace. A new group, $pqs$ is created. Since $p$, $q$ and $s$ share values on dimensions $A$ and $D$, the signature of group $pqs$ should include dimension $D$ as well, i.e., $(pqs, (2, *, *, 8))$.

The other subspaces are searched recursively. ∎

The algorithm is summarized in Figure 3.6. The correctness and the completeness of the algorithm follows from the above discussion.

Comparing to a brute-force search, algorithm *Skyey* has two major advantages. First, by Theorem 3.4.3, *Skyey* recursively reduces the set of skyline objects that need to be searched – only those objects in the skyline of the current node will be passed to the children. Often, only a small subset of objects in a set is in the skyline. The recursive reduction is effective in practice. Second, the adaptive local sorting can reuse the sorted list of the parent node to avoid sorting at all in some nodes, and also use merge sort which is practically more efficient than sorting from scratch.

Since *Skyey* needs to output the skyline of each non-empty subspace, it has to search every subspace once. In fact, if a user is interested in only the skyline groups but not the detailed lists of objects in the skyline of every subspace, the search can be further sped up.

**Input:** A set of objects $S$ in space $\mathcal{D}$;
**Output:** (1) the set of skyline groups, and (2) for each subspace, the set of objects in the subspace skyline;
**Method:**
Call *Skyey*($S$, $\mathcal{D}$);

**Function** *Skyey*($S'$, $\mathcal{D}'$)
1: Sort $S'$ in lexicographic order, try to reuse the existing sorted list as shown in Section 3.5.3;
2: identify objects in the subspace skyline of $\mathcal{D}'$;
3: for each skyline group in the parent node, check whether a temporary decisive subspace should be added; if necessary, create new groups and tags, remove minimal temporary decisive subspaces (Section 3.5.3);
4: let $S''$ be the set of subspace skyline objects in the current subspace;
5: for each $(|\mathcal{D}'| - 1)$-d subspace $\mathcal{D}''$ call *Skyey*($S''$, $\mathcal{D}''$)
6: return;

Figure 3.6: The *Skyey* Algorithm

## 3.6 Experimental Results

We conducted an empirical study of our method using both a real data set and the benchmark synthetic data sets. We evaluated the meaningfulness of skyline groups and their decisive subspaces, as well as the efficiency and the scalability of our approach to subspace skyline computation by comparing with the approach in in different aspects.

All algorithms were implemented using Microsoft Visual C++ V6.0. Experiments were conducted on a PC with an Intel Pentium 4 1.6 GHz CPU, 512 M main memory and a 40GB hard disk, running the Microsoft Windows XP Professional Edition operating system.

### 3.6.1 Results on Real Data Set *Great NBA Players' Statistics*

We downloaded from the NBA official website (www.nba.com) the Great NBA Players' technical statistics from 1960 to 2001. In total there are 17,265 records. Each record is the statistics of a player in a season. We selected six attributes: the number of games played (GP), total points (PTS), total rebounds (REB), total assists (AST), total blocks (BLK), total turnovers (TURNOVER). In this data set, the larger the attribute values, the better. That is, player $A$ dominates player $B$ if $A$'s attribute values are not less than $B$'s, and $A$ has at least one attribute better than $B$.

Finding the skyline in this players' statistics data set makes excellent sense in practice. People are often interested in finding the skyline players – players who have some outstanding merits that are not dominated by some other players. Moreover, finding the semantics of skyline in this application is of great interest – we not only want to know who are the great players, but also want to know exactly on which combinations of factors a player is dominating the other players.

The knowledge of subspace skylines has immediate applications. For example, if a coach wants to find a player with the best total points and rebounds, he should look at the skyline players in the subspace $(PTS, REB)$, instead of all skyline players.

In this data set, we found 166 skyline records in the full 6-$d$ space. The total number of corresponding decisive subspaces is 333, and the average dimensionality of the decisive subspaces is 2. We list some skyline players and their decisive subspaces in Table 3.2.

The first six records are in the skyline since each of them takes the maximum value in one dimension. Interestingly, Wilt Chamberlain's performance in 1961 was also outstanding in some combinations of attributes. Michael Jordan's performance in 1988 was not exceptional in terms of any single attribute. However, it is in the skyline once attribute combinations are considered. Actually he is a skyline object in so many different attribute combinations such as (PTS, REB, AST), (GP, PTS, AST) etc. which show he is an outstanding player. Similar case applies to Gary Payton's performance in 2001 as he is in the skyline when multiple attributes are considered. For Clyde Drexler, his performance in 1990 is in the skyline only if all the attributes are considered.

Clearly the decisive subspaces provide more insightful information than just the list of skyline players in the full space.

We found skyline records in all non-empty subspaces. Some of them may not be skyline records in the full space. The numbers of subspace skyline objects and the skyline groups in subspaces with the same number of dimensions are listed in Table 3.3. These numbers can be explained by the different number of subspaces associated with the given dimensionality and by the fact that the number of skyline records increases with increasing dimensionality.

We also counted the total number of subspaces where a record is in the skyline. This is an interesting measure, which can show the different importance of each subspace. Intuitively, if a player is in the skylines of more subspaces, he has a better overall capability in terms of combinations of attributes. We found that, in addition to dominating all others in total points (PTS), Wilt Chamberlain's performance in 1961 has the highest number, 44,

Table 3.2: Some Skyline Players and the Corresponding Decisive Subspaces

| Player | GP | PTS | REB | AST | BLK | TURNOVER | Decisive subspaces |
|---|---|---|---|---|---|---|---|
| Wilt Chamberlain (1960) | 79 | 3033 | 2149 | 148 | 0 | 0 | (REB) |
| Wilt Chamberlain (1961) | 80 | 4029 | 2052 | 192 | 0 | 0 | (PTS), (GP, REB), (REB, AST) |
| Chuck Williams (1973) | 90 | 1113 | 250 | 557 | 11 | 256 | (GP) |
| John Stockton (1990) | 82 | 1413 | 237 | 1164 | 16 | 298 | (AST) |
| Mark Eaton (1984) | 82 | 794 | 927 | 124 | 456 | 206 | (BLK) |
| George Mcginnis (1974) | 79 | 2353 | 1126 | 495 | 56 | 422 | (TURNOVER), (PTS,REB,AST), (REB, AST, BLK) |
| Michael Jordan (1988) | 81 | 2633 | 652 | 650 | 65 | 290 | (BLK, AST), (AST, PTS, GP), (TURNOVER, PTS, GP), (AST, REB, PTS), (TURNOVER, REB, PTS), (TURNOVER, AST, PTS), (TURNOVER, BLK, PTS), (TURNOVER, AST, REB, GP) |
| Gary Payton (2001) | 82 | 1815 | 396 | 737 | 26 | 209 | (GP, PTS, REB, AST), (BLK, AST, PTS, GP) |
| Clyde Drexler (1990) | 82 | 1767 | 546 | 493 | 60 | 232 | (GP, PTS, REB, AST, BLK, TURNOVER) |

Table 3.3: Number of Skyline Players in Subspaces with Different Dimensionality

| Dimensionality | # of players | # of skyline groups |
|---|---|---|
| 1 | 6 | 0 |
| 2 | 107 | 0 |
| 3 | 465 | 0 |
| 4 | 799 | 0 |
| 5 | 598 | 0 |
| 6 | 166 | 166 |

of subspaces where it is in the skyline. On the other hand, although John Stockton's performance in 1990 dominates all others in total assists (AST), which is decisive, it is only in the skyline of 32 subspaces.

From the preliminary analysis on the real data set, we obtained the interesting and meaningful observations that cannot be derived from the traditional skyline computation. This demonstrates the meaningfulness of the proposed subspace skyline analysis.

### 3.6.2 Results on Synthetic Data Sets

Using the data generator provided by the authors of [13], we generated three types of data sets as described in [13]:

- *Independent data sets* where the attribute values of the generated records are uniformly distributed;
- *Correlated data sets* where if a record is good in one dimension, it is likely that it is also good in other dimensions; and
- *Anti-correlated data sets* where if a record is good in one dimension, it is unlikely to be good in other dimensions.

For the details of the data generator, please refer to [13]. For each type of data distribution, we generated data sets with different sizes (from $100,000$ to $1,000,000$ tuples) and with dimensionality varying from 2 to 10. As the data generated by the tool are in the range of [0,1] with the precision of 6 digits in the fraction part, we discretize the data to allow for duplicates (with a degree of 10, i.e. in dimension, the value of a tuple could be the same with the other 9 tuples), which is more conforming to the real world dataset.

We investigate different aspects of our approach mainly in terms of the features of skyline group and signature for different types of dataset, the efficient representation of subspace skylines using skyline groups, the runtime and the space needed for the processing.

We compare the processing time of skyline group computation as well as the computation for all subspace skyline objects. As there is no previous work on skyline group and signature computation, for the purpose of comparison, we implemented a naïve bottom-up algorithm as a baseline method to compare with the top-down algorithm *Skyey*. The basic idea is to compute skyline objects from the 1-d subspaces to the full space. A depth-first search using

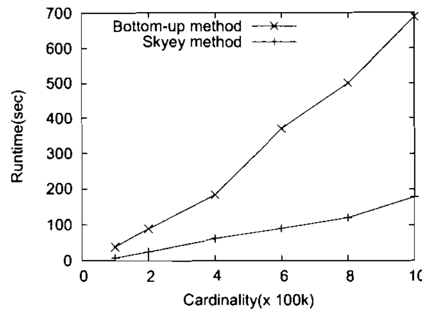



Figure 3.7: Runtime vs. Cardinality on Independent Data Sets

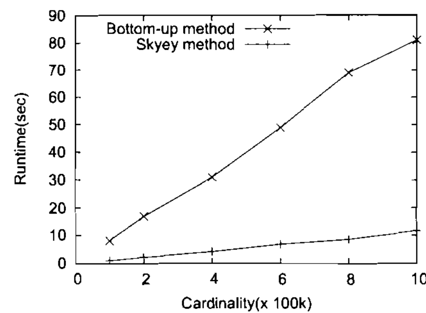



Figure 3.8: Runtime vs. Cardinality on Correlated Data Sets

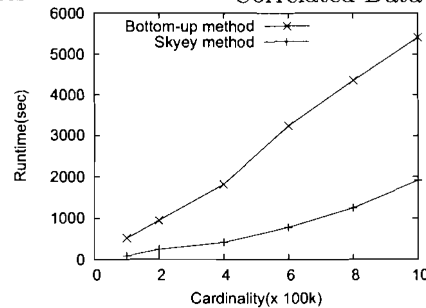



Figure 3.9: Runtime vs. Cardinality on Anti-Correlated Data Sets

the traditional set enumeration tree [79] is used. For example, given a dataset of attributes ($D_1$, $D_2$, $D_3$, $D_4$), we first compute the skyline in a single dimension subspace $D_1$ by sorting all objects by dimension $D_1$. After the sorting, with one scan we can immediately determine the skyline objects in the subspace $D_1$. To compute the skyline objects in subspace $(D_1, D_2)$, we need to sort the objects by dimensions ($D_1$, $D_2$). We apply the local sorting technique to reduce the cost of sorting. We also capture the skyline groups by monitoring objects splitting into different groups as new dimensions are added in. Since the search is bottom-up, once a new group is formed, the decisive subspace is caught.

We evaluated the scalability of algorithm *Skyey* and the bottom-up method with respect to the number of tuples in the data sets. The dimensionality was fixed to 6. The results on the three types of data sets are shown in Figures 3.7, 3.8 and 3.9, respectively. Clearly, both methods are scalable with respect to the size of data sets, but *Skyey* is far more efficient

Table 3.4: Number of Skyline Objects in Subspaces on Synthetic Data Sets

| Dimen-sionality | # of skyline objects (groups) | | |
|---|---|---|---|
| | Independent | Correlated | Anti-correlated |
| 1 | 7 (1) | 611 (6) | 304 (6) |
| 2 | 94 (0) | 45 (1) | 134 (3) |
| 3 | 269 (0) | 703 (1) | 2706 (1) |
| 4 | 306 (0) | 3007 (0) | 16062 (0) |
| 5 | 164 (0) | 4559 (0) | 36087 (0) |
| 6 | 34 (34) | 2239 (2239) | 21694 (21694) |

than the bottom-up method. Among the three data sets, the computation on the correlated data sets is the fastest, and the computation on the anti-correlated data sets is the slowest.

To further understand the effect of different distributions on the number of skyline objects and skyline groups, in Table 3.4, we list the number of skyline objects in subspaces found on synthetic data sets with 6 dimensions and $100,000$ tuples. For this example, the data value is discretized with a duplicated degree of 100 in each dimension.

As can be seen, on the correlated data sets, the number of skyline objects in subspaces is always the smallest among the three types of data sets, while the number of skyline objects on the anti-correlated data sets is always substantially larger than the other two types. This also clearly explains the difference in runtime.

We also tested the scalability of *Skyey* and the bottom-up method with respect to the dimensionality. We show in Figure 3.10, 3.11 and 3.12 the results on the data sets with different data distribution. The size of the data sets was fixed to $100,000$ records.

We observed that *Skyey* is much more scalable than the bottom-up method. In our experiments, *Skyey* is $4-6$ times faster than the bottom-up method when the dimensionality is 6. The speed-up factor is increasing with increasing dimensionality.

While *Skyey* is quite efficient when the dimensionality is not very high, we note that neither method is linearly scalable with respect to dimensionality. This observation is consistent with the results in previous studies (e.g., [72]), which showed that the dimensionality curse is still a grand challenge for skyline computation. The same applies to subspace skyline computation. How to conduct efficient subspace skyline computation and analysis on high dimensional data (e.g., with dimensionality over 100) is still an open problem.

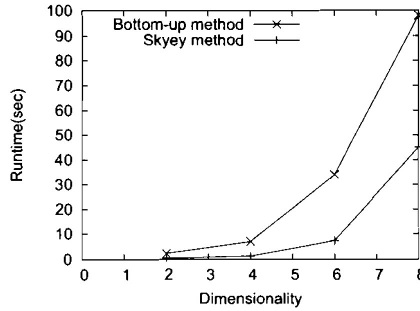



Figure 3.10: Runtime vs. Dimensions on Independent Data Sets

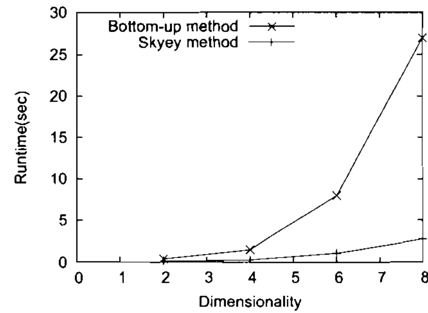



Figure 3.11: Runtime vs. Dimensions on Correlated Data Sets

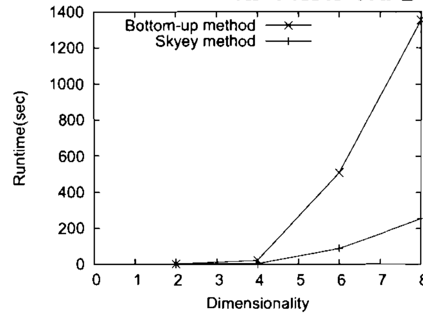



Figure 3.12: Runtime vs. Dimensions on Anti-Correlated Data Sets

## 3.7 Summary

In this chapter, we answered the questions about semantics of skyline objects by introducing the novel notions of skyline groups and decisive subspaces. To the best of our knowledge, this is the first study on the semantics of skylines and the subspace skyline analysis [75]. We proposed the problem of subspace skyline analysis and computation. On the subspace skyline analysis side, a novel roll-up and drill-down analysis of skylines in various subspaces was introduced. On the subspace skyline computation side, an efficient algorithm *Skyey* was developed. A performance study using both real and synthetic data sets was conducted to verify the meaningfulness and the efficiency of our approach. The experimental results suggest that the semantics of skyline objects and subspace skyline analysis are highly meaningful in practice, and algorithm *Skyey* is efficient and scalable.

This study also suggests quite a few interesting research problems. First, optimizing the algorithms for subspace skyline computation, subspace skyline query answering and subspace skyline membership query answering is an important issue. Proper index structures should be developed as well. For example, it would be interesting to explore how to integrate the algorithmic contributions in [95] and this *Skyey* to develop more efficient methods for skyline group computation. Second, the concept of skyline groups can be generalized to fuzzy or approximate skyline groups. This will introduce some interesting opportunities for further summarization and compression of skylines, as well as stronger query answering capability. Last but not least, it is interesting to construct an online analytic processing (OLAP) system for subspace skyline analysis. The real applications of such a system will provide tremendous insights into the skyline analysis and computation problem.

# Chapter 4

# Approximate Skyline: Thick Skyline Operator

The skyline operator returns the objects that are not dominated by any other objects with regard to certain measures in a multi-dimensional space. An example is to find the hotels with no others having better price and shorter distance to the beach. Recent work on the skyline operator [13, 85, 56, 72, 5] as we introduced in the related work of Chapter 2 focuses on improving the computation efficiency of skyline in large relational databases. However, the result from these computation gives users only *thin skylines*, i.e., single objects w.r.t. certain measurement criteria of dimension combination. This may not be desirable in many real applications since some non-skyline objects are almost as good as skyline objects and they are still very attractive to users. In this chapter, we propose a novel concept, called *thick skyline*, which recommends not only skyline objects but also their nearby neighbors within a $\varepsilon$ distance. Three efficient methods with different technical strengths are developed including (1) *Sampling-and-Pruning* method and *Indexing-and-Estimating* method to find such thick skyline with the help of sampling technique or index structure in large databases, and (2) *Microcluster-based algorithm* for mining thick skyline. The *Microcluster-based method* not only provides efficient computation but also provides a concise representation of the thick skyline in the case of high cardinalities. Empirical evaluations demonstrate the efficiency and effectiveness of our proposed methods.

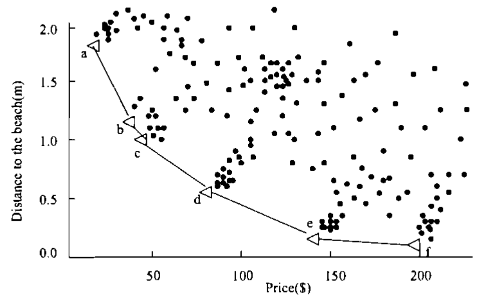



Figure 4.1: Skyline Query of New York Hotels

## 4.1 Motivation

In many situations, a database user is not interested in a list (possibly very long) of all answers that are stored in the database. Instead, they are interested only in a small number of highly-ranked answers which can provide them additional important information for making a decision. The paradigm of rank-aware query processing has recently received a lot of attention in the database system community. The *skyline operator* [13] is a very typical one that has been proposed recently, which aims to compute those objects that are not dominated by any other objects.

A typical example is a dataset about hotels which we mentioned in the previous chapter. Figure 4.1 shows the skyline of hotels among a set of hotels with dimensions of the Distance (to the beach) and the Price. The hotels $(a, b, c, d, e, f)$ are the skyline of hotels for which there is no other hotel that is better on both Price and Distance than any of them. Usually these skyline hotels are ranked as the *best* or *most satisfying* hotels.

The skyline operator can be represented by an (extended) SQL statement in SQL: *SELECT * FROM Hotels WHERE city='New York' SKYLINE OF Price* min, *Distance* min where *min* indicates that the Price and the Distance attributes should be as small as possible. For simplicity, we assume that skylines are computed with respect to *min* conditions on all the dimensions. That is, the "good", or "better" standard is evaluated by minimum condition. However, in general, it can be combined with other conditions, such as *max* [13] in the query.

The skyline operator are very useful in many areas, including database rank query computation, distributed query optimization, data visualization [13], convex hull computation, with application examples such as customer information service, stock analysis, resource

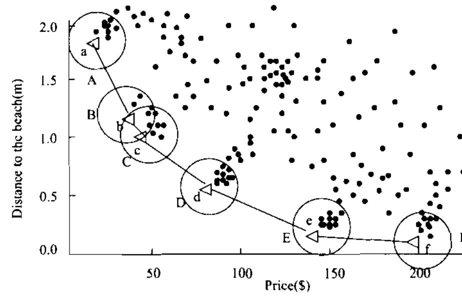



Figure 4.2: Thick Skyline Pattern of New York Hotels

management decision making etc.. Most existing work on skyline queries has been focused on efficient computation of skyline objects in large databases [13, 85, 56, 72, 5].

Although skyline queries provide useful information, the results obtained by the skyline operator may not always contain satisfiable information for users. Let's look at an example.

**Example 11 (Intuition)** *Given the hotel example in Figure 4.1, a conference organizer needs to decide the conference location. He/She usually will be particularly interested in the following questions:*

1. *Can we find a bunch of good choices of hotels (in terms of price and closeness to the beach) which are nearby so as to provide preferable choices to conference attendees?*
2. *If a good (skyline) hotel has no room available, is there any nearby hotel which, though not ranked as high as a skyline hotel, can still be a good candidate?*

■

Since the skyline operator can **only** return a set of skyline objects as the query result, such as the skyline hotels $(a, b, c, d, e, f)$, there is little choice for users's requests such as in question 1 or 2 above. For example, if hotel $f$ is full, while the user has higher preference in the distance attribute (i.e., close to the beach as much as possible), other skyline hotels actually provide too few choices for his preference. In this case, those non-skyline hotels which are close to $f$ are more likely to meet the user's requirement. Another problem in most of the existing studies is that they are based on the assumption of small skyline cardinality [72, 13, 56]. However, there could exist a large amount of skyline objects in a dataset, making it inconvenient for users to browse and manually select interesting ones.

To address the problem of either "two few" or "too many" skyline objects, it seems to be natural to consider a compact and meaningful structure to represent the skyline and its neighborhood.

In this chapter, we propose an extended definition of *skyline*, develop a novel data mining technique to skyline computation, and study the interesting patterns related to the skyline.

The concept of skyline is extended to *thick skyline* by pushing a user-specific constraint into skyline search space. For simplicity, the user-specific constraint is defined as the $\varepsilon$-neighbor of any skyline object. The thick skyline objects are classified into three categories: (1) a single skyline object, called *outlying skyline object*, (2) neighboring skyline objects, called *dense skyline objects*, and (3) neighboring skyline and non-skyline objects, called *hybrid skyline objects*. In particular, we are more interested in thick skyline objects which are composed of all the dense skyline and hybrid skyline objects.

Based on this notion, the questions in Example 11 can be answered as the follows. If a customer is given the knowledge of hotel sets $A$, $B$, $C$, $D$, $E$ and $F$ in Figure 4.2, such that each set consists of one skyline hotel as well as its $\varepsilon$ (say 20, where the value of distance and price can be normalized for the computation) neighbors, he can easily move from one hotel to a neighboring one in the same set. Suppose that he desires to live in a skyline hotel $b$ ($\$40, 1km$) in $B$, or the skyline hotel $e$ ($\$135, 0.15km$) in $E$, but both of them are full. He can then choose any other skyline hotel in $B$, such as hotel $c$ ($\$45, 0.7km$), which is as good as the occupied skyline hotel in $B$ (since $\$40 < \$45$, but $1km > 0.7km$); or choosing any non-skyline hotel in $E$, say hotel ($\$136, 0.155km$), which is actually "almost" as good as the occupied skyline hotel $e$.

Mining the thick skyline is computationally expensive since it has to handle skyline detection and nearest neighbors search, which both require multiple database scans and heavy computation. Can we design algorithms that remove the computational redundancy in skyline detection and nearest neighbors search as much as possible? Furthermore, different configurations may be required for different applications in their database system. For example, some may only allow the dataset to be stored in a single file, others may have additional support of standard index, such as B-tree, while still others may require a special index structure, such as R-tree or CF-tree. To obtain good performance in these situations, how can we exploit the related technique and develop efficient approaches that are most suitable to the different configurations respectively?

Our contributions in this topic are as the follows:

- A novel model of *thick skyline* is proposed that extends the existing skyline operator based on the distance constraint of skyline objects and their nearest neighbors.
- Three efficient algorithms, *Sampling-and-Pruning*, *Indexing-and-Estimating* and *Microcluster-based*, are developed under three typical scenarios, for mining the thick skyline in large databases. Especially, the Microcluster-based method not only leads to efficient computation but also provides a concise representation of thick skyline in the case of high cardinalities.
- Our experimental performance study shows that the proposed methods are both efficient and effective.

The remaining of the chapter is organized as the follows. Section 4.2 proposes the problem definition of a thick skyline. Section 4.3, Section 4.4 and Section 4.5 proposes three algorithms, Sampling-and-Pruning, Indexing-and-Estimating, and Microcluster-based respectively, for mining thick skyline over large databases. The results of our performance study are analyzed in Section 4.6. Section 4.7 concludes the chapter with a summary and discussion of future research directions on this topic.

## 4.2 The Thick Skyline Operator

In this section, the concept of the thick skyline and its properties are introduced. To be complete, we first briefly revisit the related notions of skyline, which is built on the dominating relationship between pairs of data objects. Then we give the definition of $\varepsilon$-neighbor of an object.

### 4.2.1 Problem Definitions

**Definition 4.2.1 (Dominating Relationship)** *Given a d-dimensional database $S$, an object $p \in S$* dominates *another object $q \in S$, denoted as $p \succ q$, if $p$ is as good or better in all dimensions and better in at least one dimension. On the other hand, $q$ is a dominated object.* ∎

**Definition 4.2.2 (Skyline)** *Given a d-dimensional database $S$ containing $n$ objects, the skyline is the set of objects $\{s_1, s_2, \ldots, s_m\}$ in $S$ such that they are not dominated by any object in $S$.* ∎

**Definition 4.2.3 ($\varepsilon$-neighbor of an object)** *The $\varepsilon$-neighborhood of an object $p \in S$, denoted by $N_\varepsilon(p)$, is defined by $N_\varepsilon(p) = \{q \in S \,|dist(p,q) \leq \varepsilon\}$.* ■

Here, the "good", or "better" standard in the dominating relationship is again evaluated by minimum condition, and the distance function $dist(p,q)$ simply uses $L_2$ distance, i.e., $dist(p,q) = \sqrt{\sum_{i=1}^{d} |p_i - q_i|^2}$.

As illustrated in Example 1, some non-skyline hotels are actually almost as good as skyline hotels in terms of the Price and the Distance, and they can be recommended as reasonable replacements for their nearby skyline counterparts. Thus, the criterion of an object being a *skyline* can be extended to a more general notion as follows.

**Definition 4.2.4 (Thick Skyline)** *Given a d-dimensional database $S$ and the skyline = $\{s_1, s_2, \ldots, s_m\}$, the thick skyline is the set of all the following objects:*

- *the skyline objects, and*
- *the non-skyline objects which are $\varepsilon$-neighbors of a skyline object.*

■

Note that a thick skyline object must belong to one of the following three categories: (a) an *outlying skyline object* which does not have any $\varepsilon$-neighbors, (b) a *dense skyline object*, which is in a set of nearby skyline objects, and (c) a *hybrid skyline object*, which is in a set of objects consisting of a mixture of nearby skyline objects and non-skyline objects.

From the data mining point of view, we are particularly interested in identifying the patterns of skyline information represented by clusters of types (b) and (c) as object of type (a) is already presented in the result of a normal skyline query.

Note that a skyline object must be a thick skyline object, but a non-skyline object can still be a thick skyline object.

### 4.2.2 The Task of Mining Thick Skylines

As the task of mining thick skylines is to find the set of skyline objects and their $\varepsilon$-neighbors, intuitively we can apply any existing algorithm such as [13, 56, 72, 85] to identify every skyline object $p$, and executing a range search to find neighboring objects $N_\varepsilon(p)$ (within the distance of $\varepsilon$) to each skyline object $p$. If there are no objects in $N_\varepsilon(p)$, the object $p$ would

be an *outlying skyline object.* If every neighbor in $N_\varepsilon(p)$ is a skyline object, then each of $N_\varepsilon(p)$ is a *dense skyline object.* Finally, if the neighbors of $N_\varepsilon(p)$ are a mixture of skyline objects and nearby non-skyline objects, each of $N_\varepsilon(p)$ would be of a *hybrid skyline object.*

The cost of mining thick skyline depends on the cost of (1) finding the skyline, and (2) finding the corresponding $\varepsilon$-neighbors. As the major cost of part (1) is evaluating the "skylineness" of each object which is usually done in a progressive manner, reducing the number of unnecessary comparisons between the potential skyline objects found so far and the remaining objects, is important. The cost of part (2) can be reduced by combining the phase of identifying $\varepsilon$-neighbors into part (1) as much as possible.

In the remaining sections, we will explore different approaches to mining the thick skyline under three typical situations. The first approach applies sampling and pruning technique for the pure relational files, and exploits the statistics in the relational database system. The second approach estimates and identifies the range of the thick skyline based on general index structures in the relational databases, which is not only suitable for the thick skyline recommendation, but also combinable with other relational operators. The third approach exploits the special summarization structure of microclusters which is widely used in data mining applications, and finds the thick skyline by employing bounding and pruning techniques.

## 4.3 A Sampling-and-Pruning Method

### 4.3.1 Sampling Strategies

Sampling-and-Pruning algorithm runs with the support of the database system where statistics, such as order statistics and quantile in each dimension, can be obtained from the system catalog.

For any single file, the identification of thick skyline objects relies on the comparisons between objects. Clearly, the access order of objects in the comparisons crucially determines the number of comparisons. For example, the access order of objects (1, 1), (2,4), (4,2) and (3, 3) needs 3 comparisons to find skyline (1, 1), while the access order of objects (2, 4), (4,2), (3,3) and (1, 1) needs 6 comparisons. Obviously, we wish to pick some objects with a high dominating capacity at the beginning to prune many dominated objects, thus reducing the overall number of object comparisons and accesses. Motivated by this, a sampling method is developed to achieve a good pruning performance with a rather limited cost.

The basic idea of the Sampling-and-Pruning method is as follows. We first randomly populate a set $X$ with sample of $k$ objects with high dominating capacity as initial "seeds" and compare them with the remaining objects. For example, in Figure 4.3, $p_1$, $p_2$ and $p_3$ are three sampled objects. Several criteria are required during the sampling step: (1) it prefers to choose objects with smaller values in dimension domains which appear to be more dominating, and (2) the $k$ objects are not dominated by each other (it is equivalent to first randomly sampling a subset of objects $M$ from the dataset, then finding $k$ skyline objects from $M$).

Each object of $X$ (usually $k \ll |S|$, where $|S|$ is the size of the database) can be taken temporarily as a "skyline" object. Afterwards, we compare the objects of $S$ with each object of $X$ to update those temporary "skyline" objects , and determine the final thick skyline objects with less number of comparisons by a useful early-pruning technique.

If the values in each dimension are distributed independently, an alternative but more aggressive sampling method [9] can be also applied to construct $k$ sampling objects by choosing $d$ (smaller) values in each dimension (i.e., such $k$ objects may not necessarily exist in the dataset). The sampling step is very efficient since the $k$ objects are built immediately after picking smaller order statistics in each dimension. The pruning capacity of this sampling can be analyzed probabilistically. Figure 4.3 shows a 2-dimensional hotel dataset partitioned into regions 1, 2, 3 and 4 by a randomly sampling point $p_1$ in this way. The objects in region 4 are dominated by the objects in region 1. Assuming there are $n$ objects and the largest values in Price axis and Distance axis are $s$ and $t$ respectively (in Figure 4.3, $s = 230$ and $t = 2.5$), that is, $s$ and $t$ are the $n$-th largest value in the Price and Distance axis respectively. Obviously, if $p_1$ is chosen properly, region 1 should not be empty, which will lead to the pruning of region 4. Otherwise, it means that $p_1$ is a poor sampling object. Suppose the constructed coordinate of $p_1$ is $(u, v)$. Then the probability of $p_1$ in region 2, 3 or 4, i.e., the probability of region 1 being empty, is $(\frac{s \cdot t - u \cdot v}{s \cdot t})^n = (1 - \frac{u \cdot v}{s \cdot t})^n = (1 - \frac{u}{s} \cdot \frac{v}{t})^n$, if $u$ and $v$ are chosen according to the first criterion above. For example, if $u$ is chosen as the $\lceil \sqrt{n \ln n} \rceil$th value in Price, and $v$ is chosen as the $\lceil \sqrt{n \ln n} \rceil$-th value in Distance, which means $u$ and $v$ are relatively small values in the corresponding axis, then the probability is $(1 - \frac{\sqrt{n \ln n}}{n} \cdot \frac{\sqrt{n \ln n}}{n})^n = (1 - \sqrt{\frac{\ln n}{n}} \cdot \sqrt{\frac{\ln n}{n}})^n = (1 - \frac{\ln n}{n})^n \leq e^{-\ln n}$[67] $= \frac{1}{n}$. That is, the chance of $p_1$ not being a good sampling object is very small. In addition, we sample $k$ points instead of 1 which decreases the possibility that none of the sampling objects are good.

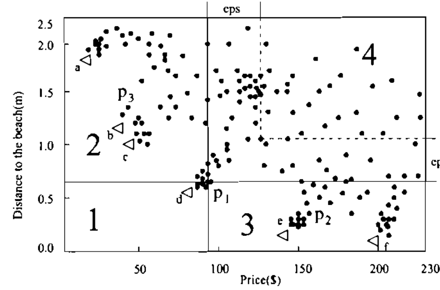



Figure 4.3: Sampling Objects to Pruning.

### 4.3.2 Strong Dominating Relationship and the Algorithm

As the thick skyline objects are not exactly the skyline objects, those non-skyline objects need to be investigated during the comparison step. For instance, if object $q$ is dominated by object $p$, and $q$ is not an $\varepsilon$-neighbor of any skyline objects found so far, $q$ could still be a thick skyline object since $q$ may be an $\varepsilon$-neighbor of some other skyline objects which are to be identified in the remaining comparisons. In other words, $q$ has to be kept and compared with every newly identified skyline objects(even temporarily). Basically, the dominating relationship and distance comparisons cannot determine the complete $\varepsilon$-neighbors of skyline unless all the skyline objects have been confirmed first. Although none of the skyline objects can be missed during the comparisons, the dataset to be used in the comparisons can be pruned to a large extent by removing those objects which definitely cannot become the $\varepsilon$-neighbors of any skyline objects. So in order to avoid unnecessary comparisons, we introduce a strong dominating relationship which is used to prune many non $\varepsilon$-neighbors of any skyline objects in the course of comparisons.

**Definition 4.3.1 (Strongly Dominating Relationship)** *Given a d-dimensional database $S$, an object $p \in S$ strong dominates another object $q \in S$, noted as $p \rhd q$, if $p + \varepsilon$ is as good or better in all dimensions and better in at least one dimension than $q$, i.e. $\forall i$, $1 \leq i \leq d$, $p_i + \varepsilon \leq q_i$, and $p_i + \varepsilon < q_i$ in at least one dimension. On the other hand, $q$ is a strong dominated object.* ■

The strong dominating relationship is illustrated by Figure 4.3, where those objects strong dominated by $p_1$ are in the dashed-lines rectangle which is a sub-region of region 4. Based on the definition of strong dominating relationship, we have the following lemmas.

**Lemma 4.3.2** *Given a dataset $S$, objects $p, q, r \in S$, the following properties are satisfied:*

1. *if $p \rhd q$, $q \rhd r$, then $p \rhd r$;*
2. *if $p \succ q$, $q \rhd r$, then $p \rhd r$.*

■

Lemma 4.3.2 illustrates the transitive property of strong dominating relationship between the objects.

**Lemma 4.3.3** *Given a dataset $S$, objects $p, q \in S$, if $p \rhd q$, then $q$ cannot be a thick skyline object.*

**Proof.** (1) If $p$ is a skyline object, by the definition of the strong dominating relationship, $q$ apparently can not be an $\varepsilon$-neighbor of $p$. Assuming $q$ is a $\varepsilon$-neighbor of another skyline object $p'$, based on the definition, $\sqrt{\sum_{i=1}^{d} |q_i - p'_i|^2} \leq \varepsilon$. We show this does not hold by contradiction. Since $p'$ and $p$ are skylines, so they can not dominate each other. Without loss of generality, we assume $p'_i < p_i$ and $p'_j > p_j$ $(i \neq j)$. Since $p \rhd q$, so $p_i + \varepsilon < q_i$, hence $p'_i + \varepsilon < p_i + \varepsilon < q_i$, then we have $q_i - p'_i > \varepsilon$ which leads to $\sqrt{\sum_{i=1}^{d} |q_i - p'_i|^2} \geq \varepsilon$. This contradicts to our assumption. Thus no $p'$ exists such that $q$ is a $\varepsilon$-neighbor of $p'$. (2) If $p$ is not a skyline object, $p$ must be dominated by some skyline object $p''$, $p'' \succ p$, since $p \rhd q$, based on Lemma 4.3.2, we have $p'' \rhd q$, which means $q$ can not be an $\varepsilon$-neighbor of $p''$. The remaining proof is similar to (1). ■

Lemma 4.3.3 shows the property that if any object $q$ is strong dominated by another object $p$, which could even be a non-skyline object, $q$ will never be a thick skyline object. Thus $q$ can be pruned immediately whenever it is identified as being strong dominated by any other object, and the unnecessary comparisons for $\varepsilon$-neighbors will be largely reduced.

The pseudo-code for the Sampling-and-Pruning algorithm is as follows.

**Algorithm 4.3.1** A Sampling-and-Pruning Method.

**Input:** A dataset $S$ of $n$ objects, distance threshold $\varepsilon$
**Output:** The thick skyline in $T$.
**Method:**

```
1.  Sampling k objects X which do not dominate each other;
2.  Label 'skyline' for X; //assuming objects in X are skylines temporarily
3.  T =X;
4.  FOR each object s_i ∈ S DO
5.      IF any 'skyline' object x(∈ X) ≻ s_i THEN
6.          IF x ⊳ s_i THEN;
7.              Continue;
8.          ELSE
9.                  T = T ∪ {s_i};
10.     ELSE IF s_i ≻ x THEN
11.             X = X − x;
12.             X = X ∪ {s_i}; T = T ∪ {s_i};
13.             IF s_i ⊳ x THEN;
14.                 T = T − x;
15.     ELSE //x and s_i cannot dominate each other
16.             X = X ∪ {s_i};
17.             T = T ∪ {s_i};
18. Update X with the skyline objects computed from list X;
19. Prune objects in T that are not ε-neighbor of any objects in X;
20. Output T;
```

In line 1 of the algorithm, we collect sampling data $X$ from the catalog statistics in the database, in particular, the order statistics and quantile statistics in picking/composing up objects that have stronger dominating capability. As depicted above, such sampling can provide a reasonable guarantee for constructing a good pruning space. $X$ is temporarily added into the thick skyline list in line 2. If an object $s_i$ is strong dominated by an object $x$ in $X$ (lines 5-7), it can be removed. Otherwise, it is either a $\varepsilon$-neighbor of $s_i$ or it is an $\varepsilon$-neighbor of some other object, or not be the $\varepsilon$-neighbor of any other object at this stage. In either case, this object should be added to the candidate thick skyline list $T$ (lines 8-9). In the case that $x$ does not dominate $s_i$, but $s_i$ dominates $x$, then $x$ should be removed from the candidate skyline list $X$ and $s_i$ should be added into the list (lines 10-12). Note that in this case, we can not remove $x$ from $T$ since $x$ could be a thick skyline although it is dominated by another object (in the simplest case, $x$ could be an $\varepsilon$-neighbor of the newly

added skyline object candidate if this object turns out to be a final skyline object). Of course, if $s_i$ strong dominates $x$, $x$ is removed from $T$ (line 14). If $s_i$ cannot be dominated by any $x$, update both $X$ and $T$ with $s_i$ (lines 15-17). After the pruning process, the final skyline object set can be identified from a small candidate list $X$ (line 18). The thick skyline object candidate set $T$ is updated to contain the final right objects (line 19). Note that it need not perform any $\varepsilon$-neighborhood search to find all the neighbors of the $x \in X$, but only need check whether any $t \in T$ locates within $\varepsilon$-distance of $x$. Thick skyline objects are output at line 20.

## 4.4 An Indexing-and-Estimating Method

Most commercial database systems support index structures, such as B-tree or Hash table. Based on these index techniques, by combining range estimate of the batches on the "Minimum Value Index" [85] with an elaborate search technique, we can find the skyline and the corresponding $\varepsilon$-neighbors within at most one scan of the database elegantly.

### 4.4.1 Bounding $\epsilon$-neighbors in Minimum Value Index

Assuming that the objects of $S$ are partitioned into a set of $d$ lists such that an object $p = (p_1, p_2, \ldots, p_d)$ is assigned into the $i$-th list ($1 \leq i \leq d$) if and only if the value of $p_i$ on the $i$-th coordinate is the minimum among all dimensions. Table 4.1 shows the lists for a simple dataset $\{a(2,10)$, $b(2,13)$, $c(4,11)$, $d(5,6)$, $e(6,8)$, $f(7,7)$, $g(8,15)$, $h(10,11)$, $i(11,13)$, $j(13,2)$, $k(9,3)$, $l(13,4)$, $m(8,5)$, $n(10,5)$, $o(12,9)\}$. Objects in each list are sorted in ascending order of their minimum coordinate ($minC$, for short) and indexed by a B-tree. A *batch* in the $i$-th list consists of objects that have the same $i$-th coordinate $minC$. In Table 4.1, $\{a, b\}$ of list 1 forms a batch while every other object constitutes an individual batch since all their first coordinates are different; while objects in list 2 are divided into 5 batches $\{j\}$, $\{k\}$, $\{m, n\}$, $\{l\}$ and $\{o\}$.

Let us have a look about how skyline objects (or "thin" skylines to discern the thick skylines we are studying) are produced first. Basically the algorithm scans the batches in each list in an interleaving way. Processing a *batch* involves (1) computing the skyline inside the batch, and (2) among the objects computed from the first step, adding those not dominated by any of the already-found skyline objects into the skyline list.

During the identification of skyline, the position of their $\varepsilon$-neighbors can be located by

Table 4.1: The Index Approach

| List1 | | List2 | |
|---|---|---|---|
| $a(2,10)$, $b(2,13)$ | $minC = 2$ | $j(13,2)$ | $minC = 2$ |
| $c(4,11)$ | $minC = 4$ | $k(9,3)$ | $minC = 3$ |
| $d(5,6)$ | $minC = 5$ | $l(13,4)$ | $minC = 4$ |
| $e(6,8)$ | $minC = 6$ | $m(8,5)$, $n(10,5)$ | $minC = 5$ |
| $f(7,7)$ | $minC = 7$ | $o(12,10)$ | $minC = 10$ |
| $g(9,15)$ | $minC = 9$ | | |
| $h(10,11)$ | $minC = 10$ | | |
| $i(11,13)$ | $minC = 11$ | | |

taking advantage of the sorted feature of each list. The following lemma gives the range of $\varepsilon$-neighbors for each skyline object in the $d$ lists.

**Lemma 4.4.1** *Given a dataset $X$ indexed by $d$ lists, and a skyline object $p = (p_1, p_2, \ldots, p_d)$ is in the batch $minC = p_i$ of the $i$th list, then:*

*(a) the $\varepsilon$-neighbors of $p$ can only possibly exist in the batch range $[p_i - \varepsilon, p_i + \varepsilon]$ of the $i$-th list; and the batch range $\left[p_j - \varepsilon, p_j + \frac{\varepsilon}{\sqrt{2}}\right]$ of the $j$-th list ($1 \le j \le d$ and $i \ne j$);*

*(b) $p$ does not have any $\varepsilon$-neighbor in any $j$th lists ($1 \le j \le d$ and $i \ne j$) if $(p_j - p_i) > \sqrt{2} \cdot \varepsilon$.*

**Proof.** (a) Suppose any $\varepsilon$-neighbor of $p$ in the $j$-th list is $p' = (p'_1, p'_2, \ldots, p'_d)$. The proof of bounds in the $i$-th list and the proof of the lower bound in the $j$-th list can be straightforwardly proved. For the upper bound in the $j$-th list, assuming the $\varepsilon$-neighbor $p'$ can exist in some batch with larger value than $p_j + \frac{\varepsilon}{\sqrt{2}}$ in the $j$-th list, that is, the batch of the $j$-th list with $minC = p'_j > p_j + \frac{\varepsilon}{\sqrt{2}}$. Since $p_i$ and $p'_j$ are the minimum values of $p$ and $p'$ in the $i$-th list and $j$-th list respectively, $p'_i > p'_j > p_j + \frac{\varepsilon}{\sqrt{2}} > p_i + \frac{\varepsilon}{\sqrt{2}}$, so we have $|p'_i - p_i| > |p'_i - p_j| > \frac{\varepsilon}{\sqrt{2}}$. On the other hand, $\left|p'_j - p_j\right| > \frac{\varepsilon}{\sqrt{2}}$, so $\sqrt{\sum_{i=1}^{d} |p'_i - p_i|^2} > \varepsilon$. Thus, $p'_j$ cannot go beyond the batch $p_j + \frac{\varepsilon}{\sqrt{2}}$ in the $j$th list, i.e., $p'_j < p_j + \frac{\varepsilon}{\sqrt{2}}$. (b) As shown in Figure 4.4 (*Eps* in the figure represents $\varepsilon$), all the objects in the $j$-th list can only appear in area I, while those in the $i$-th list are in area II, and the diagonal equally partitions I and II. Also, $q$ is the intersection point of the diagonal line and the line starting

from $p$ paralleled to $j$-axis, $dist(p,q) = p_j - p_i$, and the distance between $p$ and its possible nearest neighbor $o$ in I, $dist(p,o)$, is the shortest distance from $p$ to the diagonal (plane). If $dist(p,o) > \varepsilon$, which means $dist(p,q) > \sqrt{2} \cdot \varepsilon$, then it is impossible for any object in I (such as object $s$) to be the $\varepsilon$-neighbor of $p$. ∎

Though it looks like that Figure 4.4 only shows a two dimensional case, it is actually a representation of higher dimensional situation in terms of measuring minimum distance between a point in list $i$ and a point in list $j$. The reason is that when calculating the possible minimum distance, the optimal case is that the values of all the other dimensions (except $i$ and $j$ which we are studying at) of the point in list $j$ have the same value as the point being studied in list $i$. Any difference in value in those dimensions will relax the condition. Figure 4.5 illustrates the same situation in three dimension.

### 4.4.2 Slide Windows and the Algorithm

Based on Lemma 4.4.1, when a skyline point is found in $i$-th list, the $\varepsilon$-neighbor search of this point in the $j$-th list ($1 \leq j \leq d$, and $i \neq j$) will go toward two directions "around" batch $minC = p_j$ if condition (b) in Lemma 4.4.1 is not satisfied. The search toward upper bound direction is in $\left[p_j, p_j + \frac{\varepsilon}{\sqrt{2}}\right]$, and the search toward lower bound direction is in $[p_j - \varepsilon, p_j]$. Suppose the batch currently processed in the $j$-th list is $minC_j$. For every skyline object $p$ detected in the $i$-th list, batch $minC = p_j$ ($p_j \geq p_i$) could be located at any position of the $j$-th list. So it could scan many objects in the $j$-th list, from the currently processed batch $minC_j$ to the batch around $minC = p_j$, in order to find $\varepsilon$-neighbors. As such, for the $\varepsilon$-neighbors search of all the skyline objects, there could be many scans of the lists. To avoid this situation, a length of $\varepsilon$ sliding window around the current batch $minC_j$ (denoted as $SW_{minC_j}$) is maintained for each list during the sequential scan of the lists. In other words, the batch number $minC$ within the slide window $SW_{minC_j}$ satisfies $minC_j - \varepsilon < minC < minC_j$. Since $\varepsilon$ is usually a relatively small value, the memory space $O(d \cdot \varepsilon)$ used by slide windows is small too. Clearly, we can leave the upper bound search to the remaining sequential scan of the lists without incurring any extra cost, but only focus on the search of lower bound part $[p_j - \varepsilon, p_j]$ since it could lead to the repeated backwards scans of the list and incur more overhead.

We demonstrate by the following lemma that the $\varepsilon$-neighbor search in the $j$-th list toward the lower bound direction only needs to be done within the slide window of $minC_j$.

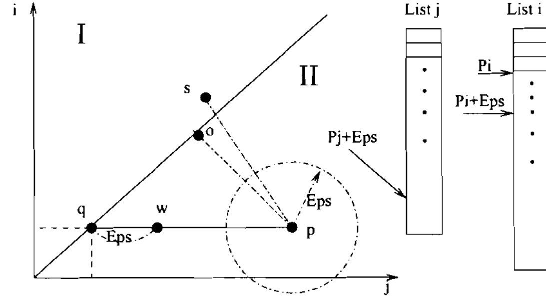



Figure 4.4: Evaluate Neighborhood Scope

**Lemma 4.4.2** *Given a dataset $X$ indexed by $d$ lists stated above, and a skyline object $p = (p_1, p_2, \ldots, p_d)$ is in the current processing batch $minC = p_i$ of the $i$-th list. Suppose the last processed batch in the $j$-th list is $minC_j$ $(1 \leq j \leq d$ and $i \neq j)$. The $\varepsilon$-neighbors of $p$ in range $[p_j - \varepsilon, p_j]$ of $j$-th list can only be found in the slide window $SW_{minC_j}$.*

**Proof.** Since batch $minC_i$ in the $i$-th list is the one the algorithm is currently processing, so $minC_i \geq minC_j$ (because the algorithm requires to process the batch in the ascending order of batch number). While $p$ is a skyline object in the $i$-th list, we have $p_j > p_i$ and $p_i = minC_i$. If $p_j - \varepsilon > minC_j$, it means that the search will involve the batches after current batch $minC_j$ in the $j$-th list, so we can leave it as well to the remaining sequential scan of the $j$-th list. Otherwise, $p_j - \varepsilon \leq minC_j$. In the meantime, we have $p_j - \varepsilon > p_i - \varepsilon = minC_i - \varepsilon \geq minC_j - \varepsilon$, so $minC_j - \varepsilon < p_j - \varepsilon < minC_j$. It shows that the $\varepsilon$-neighbors of $p$ in range $[p_j - \varepsilon, p_j]$ are covered by the slide window $SW_{minC_j}$. ∎

Lemma 4.4.2 also ensures an important property: whenever an object, if it cannot be identified as a skyline object or an $\varepsilon$-neighbor during its "life span" in the current slide window, it will never become an $\varepsilon$-neighbor for any skyline object in the remaining lists. Motivated by this property, we can find thick skylines within one scan of lists: initializing $minC_1$, $minC_2$, ..., $minC_d$ and building $d$ slide windows of $minC_i$. In the ascending order of batch number among $d$ lists, comparing every object $p$ in the minimum $minC_i$ with the thin skyline objects found so far. If $p$ is a skyline object, the range of its $\varepsilon$ is determined. The different batch upper bounds of $\varepsilon$-neighbors for skyline objects can be recorded (and have a maximum batch upper bound). Later on, the search of $p'$s $\varepsilon$-neighbors towards the lower bound can be done immediately in the slide window of each list; while the search of $p'$s $\varepsilon$-neighbors toward upper bound can be carried out during the remaining sequential scan

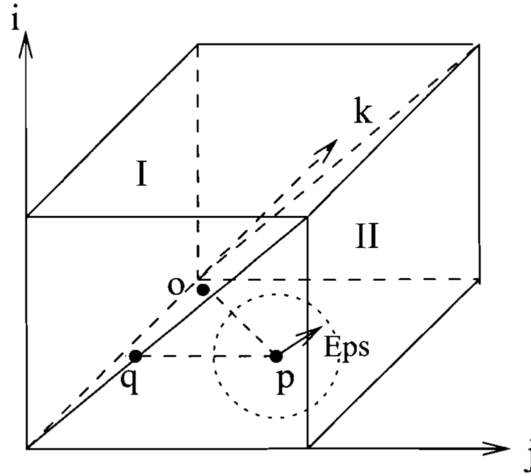



Figure 4.5: Evaluate Neighborhood Scope in 3-$d$ Case

of the lists. If $p$ is not a skyline object, check whether $p$ is the $\varepsilon$-neighbor of any skyline object if it is within the recorded upper bound $minC$ limit to be checked. If it is, $p$ is a thick skyline object; otherwise scanning forward to find any new skyline object until $p$ moves out of the current slide window. With the increasing $minC_i$ in each list, the slide window will move to the next batch until the batch number in each list reaches the maximum batch upper bound.

Let us take a look at the example dataset shown in Table 4.1. We aim to find thick skyline objects with $\varepsilon$ equal to 4. Initially, the first batches of all the lists are accessed and the one with the minimum $minC$ is processed. Let the minimum current batches in both lists are $minC_1$ and $minC_2$ respectively. Since batch $\{a, b\}$ and $\{j\}$ have identical $minC$, i.e., $minC_1 = minC_2 = 2$, the algorithm picks $\{a, b\}$ and computes the skyline objects in the batch, it then add the skyline object $a$ to the skyline list. Since 10-2=8 is greater than $4 \cdot \sqrt{2}$, we are sure $a$ does not have $\varepsilon$ neighbor in list 2 (Lemma 4.4.1). But object $b$ in the current batch is a $\varepsilon$ neighbor of $a$, so $b$ is added to the thick skyline list. As the next batch $\{c\}$ in list 1 has $minC_1 = 4$, $\{j\}$ in list 2 with $minC_2 = 2$ is processed and inserted into the skyline list since it is not dominated by $a$. Again $j$ does not have $\varepsilon$ neighbor in list 1 due to the same reason as $a$. Then, the next batch handled is $\{k\}$ in list 2, and it is added to the skyline list without $\varepsilon$ neighbor search in list 1. It searches the $\varepsilon$ window of local list 2 and $j$ is not its $\varepsilon$ neighbor. The next batch being processed is $l$ (it has the same $minC$

as the current batch in list 1). $l$ is dominated by a $j$, but $l$ is an $\varepsilon$-neighbor of $j$, so it is added to the thick skyline list. The algorithm processing now goes back to batch $c$ of list 1, $c$ is dominated by $a$, but an $\varepsilon$ neighbor of $a$ as well, so it is part of the thick skyline. The next batch to process is $d$ (since the next batch in list 2 has same $minC$ as $d$), and it is added to the skyline list. There is no $\varepsilon$ neighbor for $d$ in the $\varepsilon$ window of list 1 and neither does that exist in $\varepsilon$ window (from the current batch $l$) of list 2. The algorithm moves to next batch $m, n$ of list 2 and discovers that $m$ is a skyline object and $n$ is an $\varepsilon$ neighbor of $m$ (once discovered being an $\varepsilon$ neighbor of any skyline object such as $m$, $n$ does not to be checked with other skyline objects although it is also a neighbor of $k$). Moving back to list 1, it turns out that $e$ and $f$ are $\varepsilon$ neighbors of $d$. Up to this point, no further batches need to be processed, because both coordinates of $d$ are smaller than the $minC_1$ and $minC_2$ of next batches of list 1 and list 2, which means that the remaining objects are dominated by $d$. In addition, there is no thick skyline objects which could exist further in either list since $minC$ of both $o$ and $g$ have passed (or reached) the $\varepsilon$ upper bound window of the last possible skyline objects ($m$ and $d$ respectively) that could have them as $\varepsilon$ neighbors.

The pseudo-code of the Indexing-and-Estimating algorithm is as follows.

**Algorithm 4.4.1** An Indexing-and-Estimating Method.
**Input:** B-tree of $d$ lists index and distance threshold $\varepsilon$.
**Output:** The thick skyline $T$.
**Method:**

```
1.  S = ∅; T = ∅;
2.  FOR i = 1 to d DO;
3.      SW_i = ∅; minC_i = min list_i; upper_i = minC_i + ε;
4.  WHILE (minC_1 < upper_1) ∨ ... ∨ (minC_d < upper_d) DO;
5.      Choose the batch minC_i (with minC_i = min minC_1, ..., minC_d);
6.      Check each object p in batch minC_i;
7.      IF p is a skyline object THEN
8.          S = S ∪ {p};
9.          Check SW_j for list j such that ((p_j − p_i) ≤ √2 · ε);
10.         IF any q is a ε neighbor THEN
11.             T = T ∪ {q};
12.             Update upper_i = max{minC_i + ε, upper_i};
```

13. Update $upper_j = \max\{p_j + \frac{\varepsilon}{\sqrt{2}}, upper_j\}$;
14. ELSE IF $p$ is an $\varepsilon$-neighbor THEN
15. $T = T \cup \{p\}$;
16. Update $minC_1, \ldots, minC_d$ and $SW_1, \ldots, SW_d$;
17. $T = T \cup S$;
18. Output thick skyline $T$;

The algorithm creates a skyline objects list and $\varepsilon$-neighbors list (line 1), current batches and slide windows in each list (line 2-3). Each object $p$ in the minimum $minC_i$ is compared with the skyline list (line 6). If $p$ is a skyline object (line 7-8), the slide window of each related list is checked for finding its $\varepsilon$-neighbors, and the range of the upper bound window of each list is updated (line 12-13). Part of $p$'s $\varepsilon$-neighbors will be left to the remaining access of the lists(line 14). If $p$ is not a skyline object, it is checked with the skyline list and add it to $T$ if it is an $\varepsilon$-neighbor of a skyline object(line 14-15). The slide windows will move to the next batch (line 16) and keep moving until the batch number in every list reaches the maximum batch upper bound (line 4). Finally, it will output both skyline objects and $\varepsilon$-neighbors (line 17-18).

## 4.5 Microcluster-based Method

In order to scale-up data mining methods to large databases, a general strategy is to apply data compression or summarization. A typical approach is to summarize the database into microclusters based on CF-tree [96, 52] in which data is organized in a balanced tree with branching factor $B$ and a threshold $T$. Each internal node of the tree has at most $B$ entries and the diameter of all entries in a leaf node is at most $T$. An illustration is shown in Figure 4.6). These microclusters can be built from a hierarchical cluster feature tree (CF-tree) which is one of the most frequently used index structure in data mining tasks. The detailed process of building microclusters can be referred to [96, 52].

### 4.5.1 CF-tree and Microclusters

The formal definition of microcluster is given below. For an object $X_i$, $X_i^p$ represents the value on its $p$th dimension.

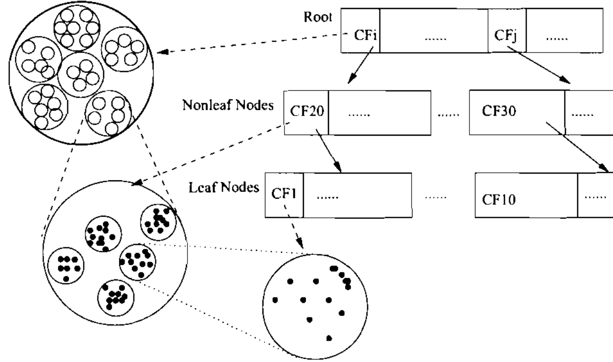



Figure 4.6: Microclusters of CF-tree

**Definition 4.5.1 (Microcluster)** *A microcluster for a set of d-dimensional objects* $X = X_1 \ldots X_n$ *is defined as a tuple* $(\overline{CF1^X}, \overline{CF2^X}, \overline{CF3^X}, n)$, *where* $\overline{CF1^X}$, $\overline{CF2^X}$, *and* $\overline{CF3^X}$ *each corresponds to a vector of d entries. The definition of each of these entries is as the follows:*

- *For each dimension, the sum of the data values is maintained in* $\overline{CF1^X}$. *Thus,* $\overline{CF1^X}$ *contains d values. The p-th entry of* $\overline{CF1^X}$ *is equal to* $\sum_{j=1}^{n} X_j^p$.
- *For each dimension, the sum of the squares of the data values is maintained in* $\overline{CF2^X}$. *Thus,* $\overline{CF2^X}$ *contains d values. The p-th entry of* $\overline{CF2^X}$ *is equal to* $\sum_{j=1}^{n} (X_j^p)^2$.
- *For each dimension, the minimum of data values is maintained in* $\overline{CF3^X}$. *Thus,* $\overline{CF3^X}$ *contains d values. The p-th entry of* $\overline{CF3^X}$ *is equal to* $\min_{j=1}^{n}(X_j^p)$. *The* $\overline{CF3^X}$ *can work as a best possible vector in the microcluster in terms of dominating relationship.*
- *The number of data points are maintained in n.*

∎

The centroid $X_a$ of a microcluster $mc_a$ can be represented by $CF1^X$ as: $X_a = \frac{\overline{CF1^X}}{n}$, and the radius $r_a$ of a microcluster $mc_a$ can be represented by $X_a$ and $\overline{CF1^X}$ and $\overline{CF2^X}$ vectors as

$$r_a = \sqrt{\frac{\sum_{j=1}^{n} (X_j - X_a)^2}{n}} = \sqrt{\frac{\overline{CF2^X} + n \cdot (X_a)^2 - 2 \cdot X_a \cdot \overline{CF1^X}}{n}}.$$

The minimum distance $mdist_o$ between each microcluster and the origin can be easily determined by $\overline{CF3^X}$. With the data organized by microclusters, we can limit the search space of the complete $\varepsilon$-neighbors of an object by first examining the neighborhood relationship between microclusters. Suppose object $p$ is in a microcluster $mc_a$, the distance between $mc_a$ and any microcluster $mc_b$ is represented as: $dist_m(mc_a, mc_b) = dist(X_a, X_b) - r_a - r_b$. If $dist_m(mc_a, mc_b) < \varepsilon$, then the objects in $mc_b$ and $mc_a$ are candidates of $\varepsilon$-neighbors of $p$.

The CF-vector values in a microcluster also satisfy additive property: if $CF_1(X) = (\overline{CF1_1^X}, \overline{CF2_1^X}, \overline{CF3_1^X}, n_1)$

and

$CF_2(X) = (\overline{CF1_2^X}, \overline{CF2_2^X}, \overline{CF3_2^X}, n_2)$

are the CFs for sets of objects $X$ and $X'$ respectively then

$CF_1 + CF_2 = (\overline{CF1_1^X} + \overline{CF1_2^X}, \overline{CF2_1^X} + \overline{CF2_2^X}, \min(\overline{CF3_1^X}, \overline{CF3_2^X}), n_1 + n_2)$

is the microcluster feature vector for the sets $X$ and $X'$.

In order to facilitate the task of mining thick skyline, the database can be partitioned into a set of microclusters with radius $r_i$ (for example, it could be $r_i \leq \varepsilon$) in the leaf nodes of CF-tree, where each non-leaf node represents a larger microcluster consisting of all the sub-microclusters represented by its entries, and the first level microclusters are represented by root node entries shown in Figure 4.6. Notice that there may exist overlapping between some microclusters and methods such as those in [42], [52] can be used to remedy this problem.

Some spatial index such as $R$-tree [72] can be still applied in data partitioning. However, such partitioning lacks the inherent statistics information and the additive property as the microcluster does. Specifically, sphere-like structure of a microcluster is more suitable in handling the problem of thick skyline with the searching of $\varepsilon$-neighbors. The microcluster structure in CF-tree can provide a hierarchical, compact representation of the thick skyline distribution for future query refinement. Moreover, the cost of building a CF-tree is very cheap and only need a linear scan of database.

### 4.5.2 Skylining Microclusters and the Algorithm

Based on the property that each microcluster can be represented by the statistics in its CF-vector, the basic idea of mining thick skyline is by taking two steps. Firstly, instead of accessing every object in the dataset, we only need to identify the microclusters that could contain skyline objects (called *skylining microclusters*), then find which microclusters are

their $\varepsilon$-neighbors. In the second step, the thick skyline objects can finally be determined from those microclusters. Using these skylining microclusters to represent a group of skyline objects and nearby objects is an appropriate summarization of thick skyline in the case of large number of skylines. It is also a good structure to maintain the skyline in dynamic dataset, where the skyline can vary from time to time.

With the data represented in microclusters, the dominating relationship can be be applied to the microclusters, that is, for any two microclusters $mc_a$ and $mc_b$, we compare their centroid vector and $\overline{CF3^x}$ respectively (since centroid vector represents the average case of all the objects in the microcluster while $\overline{CF3^x}$ represents the best possible point similar to the bottom left corner of MBR in an R-tree). If $x_a \succ \overline{CF3_b^x}$, then $mc_a \succ mc_b$, that is, the objects in $mc_b$ must be dominated by some objects in $mc_a$. As the number of microclusters is much less than the total number of objects in the dataset, the computation cost is very low.

Now we apply the nearest neighbor search to find thick skylines based on the statistics in microclusters. The algorithm starts at the root node of the CF-tree and searches for the microclusters in the ascending order of distance $mdist_o$. All the visited microclusters are added to a heap $h$ sorted by the distance $mdist_o$. Initially, the microcluster with the minimum distance to the origin is selected from the first level microclusters. Since the CF-tree is a hierarchical structure, the corresponding entry in the root node can be quickly located, and the search then goes further to its sub-microclusters and recursively goes on until the expected microcluster $mc_i$ is located in a leaf node. $mc_i$ is a skylining microcluster and is added into a heap $h_1$ sorted by the distance $mdist_o$.

Next, the algorithm continues to expand the microclusters in heap $h$ and selects the next one with the minimum distance to the origin. If the $\overline{CF3^x}$ vector of the next selected microcluster is dominated by the centroid vector of any microcluster in $h_1$, it means that the objects of the selected microcluster are dominated by some objects in a microcluster in heap $h_1$ and the selected microcluster cannot contain skyline objects. This microcluster can be certainly skipped for skyline objects consideration. Furthermore, if it is strong dominated by any microcluster in $h_1$ which means its containing objects are not possible to become any $\varepsilon$-neighbors of any skyline object, it can be pruned without any further consideration. This is in a similar situation as we show in Lemma 4.3.3 in the first method. Otherwise, the selected microcluster should be added into $h_1$ as the thick skyline object candidates. The algorithm continues selecting the remaining microclusters until all of them are visited or

prunned. Here only the statistics instead of every object in the microcluster being accessed for evaluating whether it is a skylining microcluster, thus the cost is low.

Afterwards, the algorithm visits the heap $h_1$ which only now contains a small number of microclusters, and again expands them according to the points/microclusters' $mdist_o$. We process the expanding in a batch way. The general idea is that all the objects in a expanded microcluster have to be identified as either a skyline object or not before a $\varepsilon$-neighbors search is launched for that/those microclusters. If there is any object that is still in an undefined status (i.e. it is not sure whether it is a skyline object or not), the next best microcluster is expanded.

At the beginning, all the objects in the microcluster with the smallest $mdist_o$ are expanded. Object $p$ at the top of heap $h_1$ (if the top element is a microcluster, then that microcluster is expanded until the top element is a single object) which is nearest to the origin is the skyline object, and the remaining objects are examined in the order of $mdist_o$ and see whether they are skyline objects (this property guarantees that whenever an object is not dominated by skyline objects found so far, it is a skyline object [56]). Notice that the skyline objects in the microclusters expanded so far are maintained in a skyline list. To save the on-the-fly search of $\varepsilon$-neighbors for each skyline object, a group $\varepsilon$-neighbors search of all the skyline objects in the expanded microcluster(s) is launched whenever it is discovered that all the objects in an expanded microcluster $mc'_i$ have been inspected. If there exists skyline objects in microcluster $mc'_i$, the neighboring microclusters of $mc'_i$ are also examined. As we know previously, for any microcluster $mc_a$, if $dist_m(mc_a, mc'_i) = dist(x_a, x'_i) - r_a - r'_i \leq \varepsilon$, it may contain the $\varepsilon$-neighbors of $mc'_i$. The hierarchical structure of CF-tree provides an efficient way to the search of the neighboring microclusters. We simply use a top-down traversing approach by checking whether $mc'_i$ intersects with some first level microclusters in the root node, then with the non-leaf nodes, and finally locate the desired microclusters in the leaf nodes. The search complexity which is bounded by the tree height and the intersected number of microclusters in the tree, is practically small. The objects in these neighboring microclusters are examined whether they are $\varepsilon$-neighbors of skyline objects in $mc'_i$. All the object in microcluster $mc'_i$ are then removed from $h_1$, and $h_1$ is updated. The algorithm repeats the expanding process and any of the expanded objects/microclusters are pruned if they are strong dominated by any object in the current skyline list. The algorithm terminates when $h_1$ is empty.

The pseudo-code for the Microcluster-based algorithm is as follows.

**Algorithm 4.5.1** A Microcluster-based Method.

**Input:** CF-tree with $m$ leaf microclusters, and the distance threshold $\varepsilon$.

**Output:** The thick skyline.

**Method:**

1. $S = \emptyset$; $T = \emptyset$; $heap_1 = \emptyset$;
2. $heap$ = first level nodes of the CF-tree;
3. WHILE $heap$ is not empty DO
4.     WHILE top node $mc_k$ is not a leaf node
5.         IF $(mc_j \in heap_1 \succ mc_k)$ THEN
6.             Remove $mc_k$;
7.         ELSE Expand top node in $heap$;
8.     Extract top node $mc_i$;
9.     IF $\neg(mc_j \in heap_1 \succ mc_i)$ THEN
10.         Add $mc_i$ to $heap_1$;
11. WHILE $heap_1$ is not empty DO
12.     WHILE top element $mc_k$ of $heap_1$ is not an object
13.         IF $(p \in S \succ mc_k)$ THEN;
14.             Remove $mc_k$;
15.         ELSE Expand $mc_k$; $lastExpandedMc = mc_k$;
16.     $lastExpandedMcSet \bigcup = mc_k$;
17.     WHILE there exist microclusters before the last object in $lastExpandedMc$ DO
18.         Expand those microclusters and add fully expanded MCs to $lastExpandedMcSet$;
19.     WHILE top elements of $heap_1$ is an object DO
20.         Extract object $p$ at the top of $heap_1$;
21.         IF $\neg(p \in S \succ p)$ THEN;
22.             Add $p$ to $S$;
23.     Find neighboring microclusters of each $mc'_i$ in $lastExpandedMcSet$;
24.     Add $\varepsilon$-neighbors of skyline in $mc'_i$ to $T$;
25. Output thick skyline $S \cup T$;

Lines 2-10 identifies *skylining microclusters* and thick skyline candidates microclusters. Starting from line 11, thick skyline objects are being identified. Lines 12-15 find the first microcluster that contains the first skyline object which is the one has the closest distance

to the origin and this microcluster is remembered in the variable *lastExpandedMc*. Lines 16-18 continues the expanding until all the objects in *lastExpandedMc* are listed in the front end of the heap than any microclusters. All the fully expanded microclusters during this expanding are recorded in the set of *lastExpandedMcSet*. Then skyline objects are evaluated for all the fully expanded microclusters so far based on $NN$ property (lines 19-22). The group search is performed to locate the neighboring microclusters, and the $\varepsilon$-neighbors of skyline are determined (lines 23-24).

### 4.5.3 Thick Skyline Operator and Data Mining Applications

Interestingly, the task of mining the thick skyline can be regarded as a by-product of some typical data mining tasks by naturally pushing the skyline constraint into the mining process. Recent studies in [53, 22] encourage a unified framework of data mining in that not only the input of one data mining operation can be the output of another, but also multiple mining operations can be properly integrated. Here we discuss the usability of this approach. Many data mining tasks, tend to find *large* patterns which means every object is a member of the patterns. For instance, DBSCAN is a popular density-based clustering method [31] to find arbitrarily shape clusters in databases. The thick skyline can be regarded as a *small* pattern, which consists of groups of *special* objects. So it is cost-saving to extract a *small* pattern of thick skylines along with the process of mining *large* patterns of clustering.

DBSCAN chooses one object $p$ as a seed and starts a $\varepsilon$-neighbors range search to find neighboring objects $N_\varepsilon(p)$. If the number of objects in $N_\varepsilon(p)$ satisfies $MinPts$ threshold, $N_\varepsilon(p)$ and $p$ will be merged into a *local cluster*, and the cluster expands with further range search and continues to go on. Basically, every object in the database will be involved in $\varepsilon$-neighbors search. So we can neither enforce any extra database access, nor violate any clustering target, only need to push the constraint of (strong) dominating relationship into the $\varepsilon$-neighbor search to check whether the retrieved object is dominated by the query object. Thus the skyline and its neighbors can be identified after the whole clusters are identified. Obviously, if taking advantage of the microcluster technique, it would be easier to find both clusters and the thick skyline.

## 4.6 Experimental Results

In this section, we report the results of our experimental evaluation in terms of efficiency and effectiveness. Section 4.6.1 shows the runtime performance and the number of comparisons needed for computing the thick skyline. Section 4.6.2 evaluates several factors that affect the size of thick skylines including the choice of the value of $\varepsilon$ and the using of microclusters. Here, we focus on the cost of mining thick skylines in the computing stage instead of preprocessing stage such as the index building or the CF-tree creation.

Following similar data generation methods in [13], we employ two types of datasets: independent databases where the attribute values of the tuples are generated using uniform distributions and anti-correlated datasets which contain tuples whose attribute values are good in one dimension but are bad in one or all of the other dimensions. The dimensionality of datasets $d$ is in between 2 and 5, the value of each dimension is in the integer range of [1, 1000] and the number of data objects(cardinality) $N$ is between 100k (100,000) and 2000k (2000,000). We have implemented the Sampling-and-Pruning, Indexing-and-Estimating and Microcluster-based methods in C++. All the experiments were conducted on an Intel 1.6GHZ processor with 512M RAM, 40G hard disk, running on Windows 2000.

### 4.6.1 Efficiency Tests

**Runtime Performance.** We first investigate the runtime issue in the case of different dimensionality. For the independent data distribution, we use a dataset with the cardinality N of 1000,000, while for the anti-correlated dataset, the cardinality is much smaller with the number of 100,000 since it generally takes longer time to compute the skyline in anti-correlated distribution and more skyline objects are generated, but this does not affect the relative comparison of the algorithms. $\varepsilon$ in both cases equal to 5. Figures 4.7 and 4.8 depict the results of the runtime w.r.t various dimensionality in the independent and anti-correlated distribution respectively. In both cases, the Indexing-and-Estimating method achieves as good performance as that for the Microcluster-based method in case of small dimensionality($d = 2$, 3), due to its sorted list structure is most suitable to the relative small dimension so that the $\varepsilon$ neighbors do not scatter in too many lists that needs to search. Microcluster-based method is getting better towards larger dimensionality ($d > 3$) and when the skyline size becomes larger. The Sampling-and-Pruning method ranks the third due to its lack of index structure and all the computation needs to be done online.

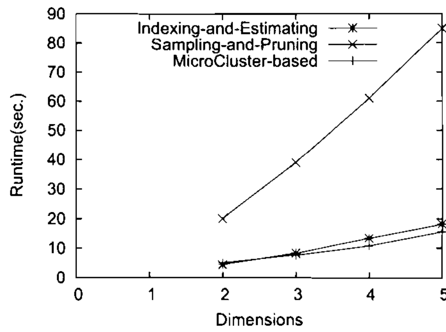



Figure 4.7: Time vs. Dimensions on Independent Data Sets

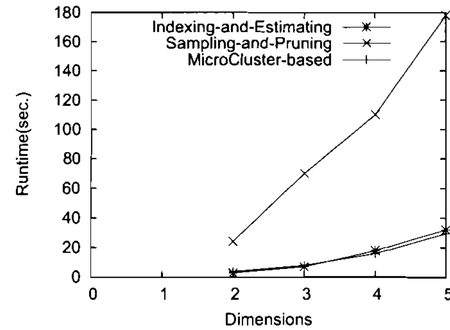



Figure 4.8: Time vs. Dimensions on Correlated Data Sets

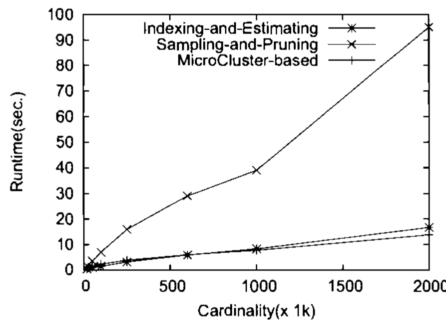



Figure 4.9: Time vs. Cardinality Dimensions on Independent Data Sets

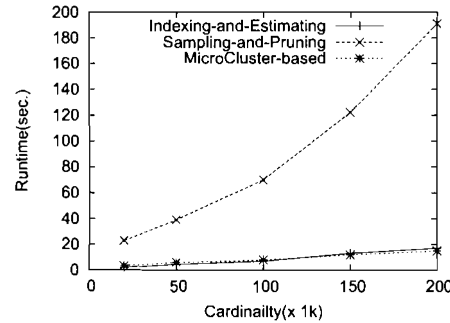



Figure 4.10: Time vs. Cardinality Correlated Data Sets

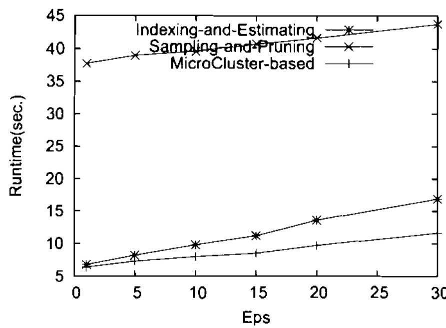



Figure 4.11: Time vs. Eps on Independent Data Sets

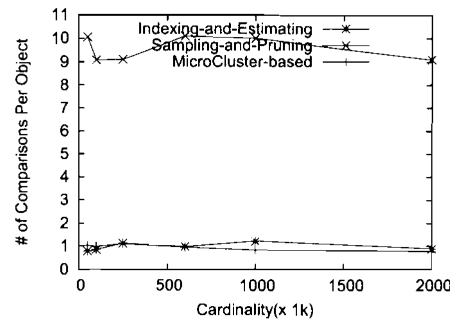



Figure 4.12: #of Comparisons vs Cardinality on Independent Data Sets

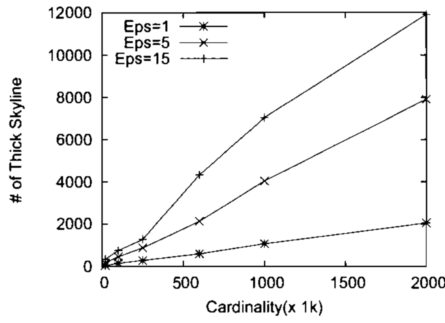



Figure 4.13: #of Thick Skyline vs. Cardinality on Independent Data Sets

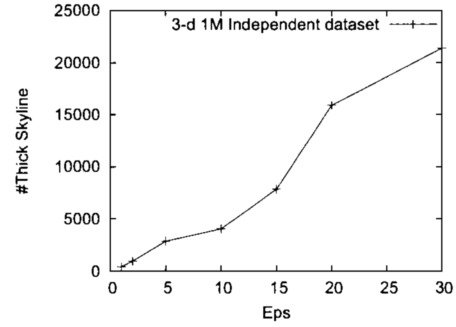



Figure 4.14: #of Thick Skyline vs. Eps on Independent Data Sets

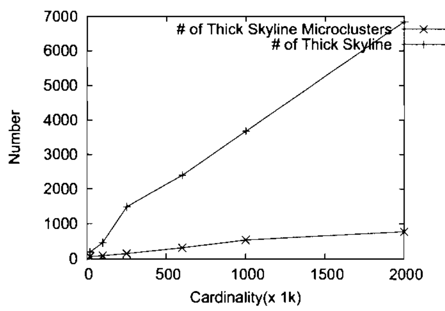



Figure 4.15: Effects of Microclusters(I)

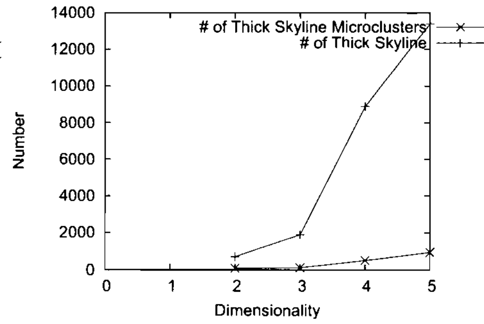



Figure 4.16: Effects of Microclusters(II)

Figures 4.9 and 4.10 show the runtime w.r.t. various number of tuples in independent and anti-correlated distributed 3-$d$ datasets respectively($\varepsilon = 5$). Again, due to the same reason as stated in previous paragraph, we tested independent dataset with the scale to 2000,000, while for anti-correlated dataset, we test the dataset with maximum size of 200,000. In both cases, we can see that the Indexing-and-Estimating method is better than Microcluster-based method in run time when the cardinality is relatively small ($600K$ for the independent dataset and around $100K$ for the anti-correlated dataset). After that, Microcluster-based method starts to run faster than the Indexing-and-Estimating method due to its region pruning capability of microclusters and good scalability of the hierarchical structure. Since there is no traditional or specialized index to facilitate computation, Sampling-and-Indexing still ranks the third in the comparisons, but the run time is not bad even when the cardinality is very large.

**Number of Comparisons.** In order to examine the average number of comparisons per object(equivalent to the total number of comparisons divided by the number of objects), we test the three methods on different sizes of 3-$d$ independent distributed datasets (with $\varepsilon = 5$). Figure 4.12 shows the result and it indicates that both the Indexing-and-Estimating and the Microcluster-based methods have the similar number of comparisons per object(about one time), since many objects are pruned earlier or later. Sampling-and-Pruning method (with the sample size of 30) has at most 10 comparisons for each object.

### 4.6.2 Effectiveness Tests

**The Effect of $\varepsilon$.** Obviously, the choice of $\varepsilon$ values will affect the size of thick skylines, and this is related to the specific applications. Some applications would want to output more choices for the user, hence a relatively bigger $\varepsilon$ is chosen, while other applications may want to have a more refined set of choices thus a smaller value of $\varepsilon$ is selected. But in general, $\varepsilon$ is practically a small value w.r.t. the domain values, which reflects the "local neighborhood". The choice of value can also be recommended by the system as an initial parameter for the future user interaction. When we increase $\varepsilon$ value from 1 to 30 in 1000,000 independently distributed 3-$d$ dataset, Figure 4.14 (here denote the number of thick skyline as "TSkyline") shows that the number of thick skyline objects increases, and Figure 4.11 shows the run time of the three algorithms also increase. In particular, Microcluster-based method is always the best and keeps good scalability. Indexing-and-Estimating method

is still better than Sampling-and-Pruning method in runtime w.r.t $\varepsilon$, but we can see that the Indexing-and-Estimating method increases faster than the sampling method with the increase of $\varepsilon$ due to the fact that it needs to scan a bigger window in each list.

**The Effect of Dimensionality and Cardinality.** The change of dimensionality will affect the size of thick skylines, this can be illustrated in Table 4.2. For example, if $\varepsilon$ is chosen as the square root of sum of 0.3% of the maximum value in each dimension, Table 4.2 shows the size of thick skylines (cardinality of 100K tuples) in different dimensions, where the number in brackets is the number of skyline objects, which can be small in some case. We notice that if increasing dimensionality, both independent and anti-correlated distributed dataset will increase the number of the thick skyline objects, and the latter will increase more. The affect of cardinality is shown in Figure 4.13 (here denote the number of thick skyline as "TSkyline") that the number of thick skyline increases w.r.t the large size of datasets.

Table 4.2: Thick Skyline Size

| Dimension | Independent | Anti-Corelated |
|---|---|---|
| 2 | 65(10) | 187(41) |
| 3 | 161(19) | 1075(353) |
| 4 | 892(134) | 7436(2084) |
| 5 | 1435(328) | 26486(9320) |

**The Effect of Microclusters.** Finally, we examine the compact results of using thick skylining microclusters which contain the skyline objects and their $\varepsilon$-neighbors to represent all thick skyline objects. Figure 4.15 shows that in a 3-$d$ independent distributed dataset, when $\varepsilon = 3$, the number of microclusters maintained is always far less than that of the thick skyline objects. This effect is especially evident when data size tends to be larger. We have the similar compact effect w.r.t. the change of dimensionality in a $N = 1M$, independent distributed dataset($\varepsilon = 5$) shown in Figure 4.16.

## 4.7 Summary

The paradigm of rank-aware query processing has recently received a lot of attention in the database systems community. In particular, the new *skyline operator* has been proposed. In

this chapter, we proposed a novel notion of the *thick skyline* based on the distance constraint of a skyline object from its nearest neighbors [51]. The task of mining thick skylines is to recommend skyline objects as well as their nearest neighbors within $\varepsilon$-distance. We also develop three algorithms, Sampling-and-Pruning, Indexing-and-Estimating, and Microcluster-based, to find such thick skylines in large databases. Our experimental evaluation demonstrates the efficiency and effectiveness of our algorithms. We believe the notion of thick skyline and mining methods not only extends the skyline operator in database query, but also provides interesting patterns for data mining tasks. This studies also suggests several interesting topics by pushing the data mining operation into the skyline concept. For example, we can investigate the task of (1) mining subspace thick skylines; (2) mining the most interesting subspaces for the thick skyline objects or (3) mining the exceptional thick skyline objects in the full space or subspaces and so on.

# Chapter 5

# Skyline and Database Ranked Queries

Given a linear monotone score function $f$, the top-$k$ ranked query retrieves the best $k$ objects according to the values of $f$. Existing methods for processing such queries employ the techniques of sorting, updating thresholds, materializing views and convex hull. In this chapter, motivated by the interesting relationship between the top-$k$ tuples and the skyline objects, we propose two novel index-based techniques for top-$k$ ranked query: (1) indexing the layered skyline, and (2) indexing microclusters of objects into a grid structure. We develop efficient algorithms for ranked queries by locating the answer points during the sweeping of the line/hyperplane of the score function over the indexed objects. Both methods can be easily plugged into typical multi-dimensional database indexes. The experimental results from different evaluating aspects are also reported.

## 5.1 Introduction

Rank-aware query processing has become more and more important for database users. The answer to the top-$k$ ranked query is a set of $k$ resulting tuples ordered according to a specific score (preference) function that combines the value of each input (attribute). The combined score function is usually linear and monotone with regard to the individual input. For example, given a hotel database with attributes of *distance* to the beach ($x$ axis), *price* ($y$ axis) and the score function $f(x, y) = 0.3x + 0.2y$, the top-3 hotel records are the best

three hotels that minimize $f$.

A straightforward method to answer top-$k$ ranked queries is to first calculate the score of each tuple during the scan of the dataset, then output the top-$k$ tuples from all the calculated ones by comparison or sorting. This fully ranked approach obviously is undesirable for querying a relatively small number $k$ out of a large number of objects. Thus, several methods towards improving the efficiency of top-$k$ ranked queries over databases have been developed. Fagin et al. a set of dynamic threshold-based scanning algorithms are introduced in [33, 48, 47], and Natsev et al. [68] provides a solution for combining the results of multiple top-$k$ queries based on user-specified join predicates. Recent work includes materializing techniques based on pre-computing ranked views and convex hulls etc [44, 21, 89]. But these methods are either specific to joined relations of two dimensions, or incompatible with other database indexing techniques, or computationally expensive.

In this chapter, we propose novel solutions to the ranked query problem motivated by the relationship between the top-$k$ ranked objects and the skyline objects. Let us first look at the following example.

**Example 12 (Top-$k$ ranked queries)** Consider a set of hotels with attributes of *distance* ($x$ axis) and *price* ($y$ axis) as shown in Figure 5.1. The points marked as triangles $(a, b, c, d, e, f)$ are skyline hotels as there is no other hotel that is better on both *price* and *distance* than any of them. Suppose a score function $f(x, y) = a_1x + a_2y$ where $0 \leq a_1, a_2 \leq 1$, is used to rank top-$k$ hotels. From geometry point of view, ranking top-$k$ hotels can be regarded as the process of sweeping the line $a_1x + a_2y$ in the plane as shown in Figure 5.1 (illustrated with the lines sweeping from $F_0 \rightarrow F_1 \rightarrow F_2 \rightarrow F_3 \rightarrow \ldots$), and intersecting the hotel points during the sweeping. The order that the first $k$ points are intersected is the order that top-$k$ objects are ranked. ■

Note that in practice, users might be interested in different score functions, so that $a_1$ and $a_2$ could be arbitrarily changed and the angle of the sweeping line to the axis is changed correspondingly. Furthermore, the same process can be applied to high dimensional cases as well. When $d$ ($d > 2$) attributes are involved in the score function of $a_1 \cdot x_1 + \ldots + a_d \cdot x_d$ where $0 \leq a_i \leq 1$, querying the top-$k$ objects is equivalent to sweeping a $d$-dimensional hyperplane representing the score function $f(x_1, \cdot, x_d) = \sum_{i=1}^{d} a_i \cdot x_i$ and finding the first $k$ objects intersecting with it.

Obviously, the sweeping process naturally captures the order of ranking top-$k$ objects

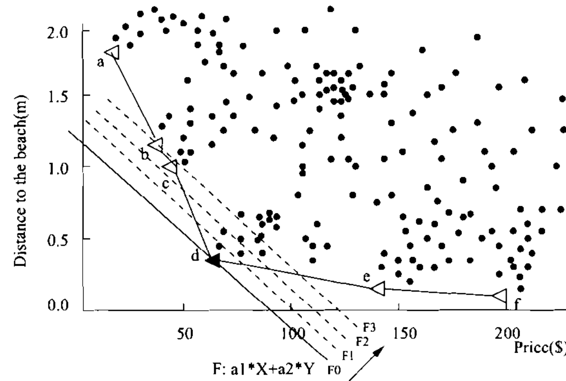



Figure 5.1: Top-k Objects vs. Skyline

and it only retrieves the least number of objects needed from the database in the ideal case. Furthermore, we have the general observations as follows.

1. The dominating relationship actually decides the ranking order between a pair of objects. That is, if an object $p$ dominates object $q$, then $p$ ranks higher than $q$ w.r.t. any above score functions.

2. The top-1 object is always a skyline object no matter how the function changes; for example, in Figure 5.1 the best hotel is the skyline hotel $d$. The $i$th best object is either another skyline object or an object dominated by at least one of the top-($i$-1) objects.

3. If multi-layer skylines [70, 29] are constructed where each object in some layer is dominated by at least one object in its previous layer, the top-$k$ ranked objects must be contained in the first $k$ skyline layers.

We can see that although the skyline objects (and their different layers) are independent of any chosen score functions, the higher dominating capacity of skyline objects leads to the higher ranking of skyline objects over non-skyline objects.

Motivated us by these observations, we propose two novel index-based approaches for top-$k$ ranked queries. The first approach is indexing the layered skylines, and the second approach is indexing microclusters of objects into a grid structure. We also develop efficient algorithms for ranked queries by locating the answer points during the sweeping of the line/hyperplane of the score function over the indexed objects. Both approaches can be

easily plugged into typical multi-dimensional database indexes. For example, the layered skyline can be maintained in a multi-dimensional database index structure with node blocks described by (1) MBR(Minimum Bounded Rectangle) such as R-tree[41], R*-tree[7] and X-tree[12], or (2) spherical Microcluster such as CF-tree[96], SS-tree[91] and SR-tree[54].

Our contributions on top-$k$ ranked queries can be summarized as the follows.

1. We propose a framework to process top-$k$ ranked queries by indexing skyline layers or indexing microclusters;

2. We propose efficient sweeping algorithms for the exact top-$k$ ranked queries and approximate top-$k$ ranked queries, and illustrate the instantiations applied for different index structures;

3. The comprehensive experiments not only demonstrate that our algorithms provide better performance than state-of-the-art methods, but also illustrate that the application of data mining technique (microclustering) is a useful and effective solution for database query processing.

The rest of the chapter is organized as follows. Section 5.2 presents the problem definitions. Several algorithms on computing multi-layer skylines are given in Section 5.3. Section 5.4 and Section 5.5 give the KNN-based and Grid-based sweeping algorithms for the exact and approximate ranked queries respectively, and illustrate their plug-in adaptations for R-trees and CF-trees respectively. In Section 5.6, we perform a comprehensive experimental evaluation, and Section 5.7 briefly reviews the work of top-$k$ ranked queries. We conclude the chapter in Section 5.8.

## 5.2 Foundations

In this section, we introduce the important definitions and properties that will be used through this chapter.

### 5.2.1 Top-$k$ Ranked Queries and Skylines

Similar to the previous chapters, we denote any $d$-dimensional dataset as $X$, and also denote any linear monotone score function as $f$. To represent users' preferences on different

attributes in a dataset, different weights at each attribute are set for the ranked queries. Typically, a weighted score function is specified to combine all the preferences as the follows.

**Definition 5.2.1 (Score Function)** *Given any d-dimensional dataset $X$, and the linear monotone score function $f$ on $d$ attributes is: $f(x) = \sum_{i=1}^{d} a_i \cdot x_i$ where $x \in X$, $a_i$ is the weight on the attribute value $x_i$ where $0 \leq a_i \leq 1$.* ∎

Without loss of generality, we assume the lower the score value, the higher rank the object has. The concept of the above score function can be used to define the top-$k$ ranked query as follows:

**Definition 5.2.2 (Top-$k$ Ranked Query)** *Given a dataset $X$ and the linear monotone score function $f$ on $d$ attributes, a top-$k$ ranked query returns a collection $T \subset X$ of $k$ objects ordered by $f$ ascendingly, such that for any $t \in T$, there does not exist $x \in X$ and $x \notin T$, such that $f(t_1, \ldots, t_d) \leq f(x_1, \ldots, x_d)$.* ∎

Since the dominating relationship($\succ$), which is defined in Definition 4.2.1, describes a min/max preference on all the attributes between objects. That is, the smaller the value, the better ranking for the object. Therefore, it can be used to identify the ranking order between any pair of objects as the follows.

**Lemma 5.2.3** *Given a dataset $X$ and the linear monotone score function $f$, for any objects $p \in X, q \in X$, if $p \succ q$ then $f(p_1, \ldots, p_d) < f(q_1, \ldots, q_d)$.*

**Proof.** The proof is easy, since $p \succ q$, we have $p_i \leq q_i (1 \leq i \leq d)$ and at least for one dimension say $j$, $p_j < q_j$, thus $f(p_1, \ldots, p_d) < f(q_1, \ldots, q_d)$ due to the fact that $f$ is a monotone function. ∎

The skyline operator [13] is based on the dominating relationship among the dataset. Interestingly, the following notion of multi-layer skyline [70, 29] is built on top of the skyline operator and can be regarded as a stratification of the dominating relationship in the dataset.

**Definition 5.2.4 (Multi-layer Skyline)** *Given a dataset $X$, the multi-layer skyline is organized as the follows:*

1. *The first layer skyline $L_1$ is the set of regular skyline objects of $X$.*

2. *For $i > 1$, the ith layer skyline $L_i$ is the set of skyline objects in $X - \bigcup_{j=1}^{i-1} L_j$.*

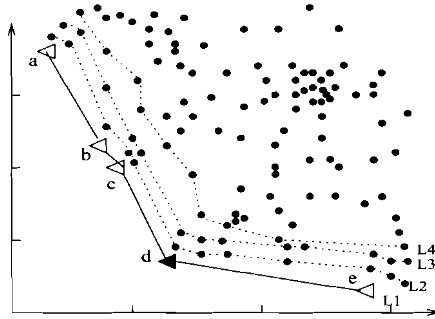



Figure 5.2: Multilayer Skyline

■

As shown in Figure 5.2, layer $L_1, L_2, L_3, L_4$ (in dashed lines) are the four layers of skyline objects. We can observe that no points in this figure the can dominate points in $L_1$, and only points in $L_1$ can dominate points in $L_2$. Any point in $L_3$ is dominated by at least one point in $L_2$ (and of course $L_1$), while points in $L_4$ can only dominate points in layers after $L_4$.

We can generalize this dominating containment property of the multi-layer skyline as follows:

**Lemma 5.2.5** *Given a dataset $X$ and score function $f$, for any $i < j$, if object $q \in L_j$, there exists at least one object $p \in L_i$ s.t. $f(p_1, \ldots, p_d) < f(q_1, \ldots, q_d)$.*

**Proof.** Suppose there is no object in $L_i$ that can dominate $q$, then $q$ must belong to the layer of $L_i$, $\therefore$ the assumption does not hold. Thus there exists at least one object in $L_i$ say $p \succ q$. Based on Lemma 5.2.3, $f(p_1, \ldots, p_d) < f(q_1, \ldots, q_d)$. ■

Clearly, $q$ in some layer is always ranked after some other object $p$ in its previous layer. On the other hand, $q$ may not be dominated by some other objects in its previous layer and may be ranked higher than them.

Hence, the multi-layer skyline provides relevant references for evaluating the ranking orders between objects in different layers. It partitions the whole dataset into different parts with different dominating capacities, which can measure not only the dominating relationship between skyline objects and non-skyline objects, but also the dominating relationship between non-skyline objects.

Lemma 5.2.5 also implies a useful property that the results of the top-$k$ ranked query locate in the first $k$ layer skylines. Actually, the set of first $k$ layer skylines is a worst-case guarantee to obtain the correct answer. In many cases, the top-$k$ ranked tuples can be found in less than $k$ skyline layers.

Assuming that we know $K$, the maximum value of the regular user-preferred $k$ in ranked queries (as an example $K = 100$ in [89]), all top-$k$ ranked queries in large databases can be processed using the first $K$ skyline layers.

**Theorem 5.2.6** *Given K layers of skyline $L_1, \ldots, L_K$, any top-k tuples w.r.t. f must be contained in the objects set $\bigcup_{j=1}^{K} L_j$, i.e. in the first K skyline layers.*

**Proof.** If $i < j$, according to Lemma 5.2.5, if the rank query number $k$ does not exceed K which we assume is the upper bound of the normal rank queries, each object $q$ in $L_j$ ranks lower than at least some object $p$ in $L_i$. In the worst case, $q$ only ranks lower than exact one object, say $h$ in $L_i$, and if this is the same case for $h$ and all the upstream dominating layers, then each of the top-$k$ tuples just locates at each of the $k$ layers of the skyline. While in the normal cases, it may take much less than $k$ layers to finish the top-$k$ query. ∎

### 5.2.2 MBR and Microcluster

As mentioned in the introduction section, the layered skyline can be organized into rectangle-like and sphere-like *blocks*, hence supported by both types of multi-dimensional database indexes. We choose R-tree and CF-tree as the typical representatives of these two types of indexes. The basic storage of a *block* in R-tree is a Minimum Bounding Rectangle(MBR) in a leaf node. On the other hand, the basic storage of a *block* in a CF-tree is a sphere-like microcluster in the leaf nodes.

The microcluster $C$ used in this chapter is represented as $(n, \overline{CF1(C)}, \overline{CF2(C)}, r)$, which is a simplification version of Definition 4.5.1, where $\overline{CF3(C)}$ is no more required.

Since the typical top-$k$ query applications usually involve up to ten dimensions [21], [44], under such circumstance, both R-tree and CF-tree can provide good performance during the data access and processing. Without any specification, in this chapter *block* is used as a general term for either a MBR in the R-tree or a sphere Microcluster in the CF-tree.

## 5.3 Computing and Indexing Layered Skylines

In the preprocessing step for the top-$k$ ranked queries, we compute the $K(k \leq K)$ layers of skyline objects out of the dataset.

Note that the skyline computation has much less worst-case complexity $(O(n(log(n))^{d-2}))$ $(d > 3)$ [57] than the convex points computation $(O(n^{d/2}))$ $(d > 4)$ [26] for a typical size of databases (say $n \leq 10^6, d \leq 10$). If the number of layered skyline objects is much smaller than the size of the database, for example, in a five dimensional dataset with 100,000 tuples where the distribution in each dimension is correlated, the total number of skyline objects is around 840 [85] while there are about 100 layers of skylines in this dataset. Hence, we just store these layered skyline objects as *blocks* in the leaf nodes of index structures. If the number of layered skyline happens to be large, considering the fact that databases usually has already been maintained by multi-dimensional indexes to support its query operations, we only need to make some minor modifications to the original index structure for the sweeping processing when answering top-$k$ queries.

### 5.3.1 A Naive-based Method

Intuitively, we can apply any existing algorithm such as [13], [85], [56] or [72] to identify and output skyline $L_1$ in $X$, then removing $L_1$ from $X$ to iteratively find skyline $L_2$ in $X - L_1$ and so on. The top-$k$ layer skylines can thus be obtained. The pseudo-code of naive-based algorithm is given as follows:

**Algorithm 5.3.1** A naive-based method for multi-layer skyline.
**Input:** a dataset $X$, layer threshold $k$.
**Output:** top-$k$ layer skyline $T$.

1. $T = \emptyset$;
2. FOR $i = 1$ to $k$ DO;
3. Find $L_i$ in X;
4. $T = T \cup L_i$;
5. $X = X - L_i$;
6. Output top-$k$ layer skyline $T$;

Obviously, this naive-based method needs multiple scans of the database, and incurs more I/O overhead.

### 5.3.2 A Topology Sorting-based Method

In many database systems, queries and applications are supported by indexes (B+tree index or Hash index). Without loss of generality, supposed the dataset $X$ is stored in the leaf nodes of B+tree in the order of topology sorting, that is, for any tuple $u, v \in X$, $u = (u_1, u_2, ..., u_d)$, $v = (v_1, v_2, ..., v_d)$, we say $(u_1, u_2, ..., u_d) \lessdot (v_1 ,v_2 ,...,v_d)$ if $u_1 \leq v_1$ and $(u_2, ..., u_d) \lessdot (v_2 ,..., v_d)$. Obviously, this order is defined recursively.

We can take advantage of the information in the topology sorting index and propose the corresponding algorithm to identify multi-layer skylines. To facilitate the search of the skyline layer for each object, we use $k$ linked lists : $i$-List ($1 \leq i \leq k$) to represent top-$k$ layer skyline objects, at the beginning, these lists are empty and are updated during the scan of $X = \{x_1, \ldots, x_N\}$ in a topology sorting order. The basic idea of this algorithm is: we start from the first object in $X$, obviously $x_1$ is a skyline object and inserted into 1-List. We then compare $x_2$ with $x_1$, if $x_2$ is dominated, it is inserted into 2-List, otherwise it is inserted into 1-List and so on. Suppose $x_i$ is accessed, and there are $j$ non-empty lists at this moment. We first compare $x_i$ with the objects in 1-List, if it is not dominated by all the objects in 1-List, insert it into 1-List. Otherwise, whenever it is dominated by some object in 1-List, stop comparing with remaining objects in this (layer)list, and start the comparisons with with objects in 2-List. This process may continue until it finishes comparisons with objects in $j$-List. Like SFS [27] where the object can be determined whether it is a skyline or not by only comparing with the objects appearing before this object in the sorting order, here the skyline layer at which an object locates only relies on the comparisons with those objects appearing before this object in the topology sorting order. The pseudo-code of the topology sorting-based method is in the following:

**Algorithm 5.3.2** A topology sorting based method for multi-layer skyline.
**Input:** a dataset $X$ in the topology sorting order, layer threshold $k$.
**Output:** top-$k$ layer skyline $1-$List, ..., $k-$List.

1. FOR $i = 1$ to $k$ DO
2. $\quad i-$List$= \emptyset$;
3. FOR $i = 1$ to $N$ DO

```
4.      j=1; LocateLayer = True;
5.      WHILE NOT LocateLayer DO
6.          IF every object p ∈ j−List cannot dominate x_i
7.              THEN j−List=j−List ∪ {x_i};
8.                      LocateLayer = False;
9.              ELSE j = j + 1;
10. Output top-k layer skyline 1−List, ..., k−List;
```

## 5.4 A KNN-based Sweeping Approach for Top-$k$ Queries

In this section, we apply the strategy of $k$ nearest neighbor (KNN) search to examine those nodes in index trees with potentially best scores, prunes those nodes which cannot contain top-$k$ objects and retrieves the objects layer-by-layer.

The proposed methods include two levels of sweeping: (1) sweeping over blocks such that the I/O cost of accessing blocks is aimed to be minimized; and (2) sweeping within a block such that the CPU cost of retrieving objects within a block is aimed to be minimized.

### 5.4.1 Sweeping over Blocks

During the sweeping process, the hyperplane which represents the weighted score function always contacts the best point first, the next best point second and so on. Based on the indexed $K$ skyline layers, we develop an efficient branch-and-bound algorithm to answer the top-$k$ rank query by using the optimal $KNN$ search paradigm [12, 43, 78, 89]. That is, in each step those data blocks with potentially the best scores are examined first, meanwhile the blocks which cannot contain any top-$k$ object will be pruned. Since the lowest point of a data block represents the point which has the smallest score in that block and the highest point of a block is the one that has the highest score. Thus, if the lowest point of a data block $M$ is dominated by the current $k$th ranked object, then none of the objects in $M$ can be a top-$k$ answer. During the examining process, a list is maintained for all the current/temporary top $k$ points.

While for a sphere Microcluster in CF-tree, the lowest and highest points in the CF-tree case can be accurately captured by using the two contact points of the sweeping hyperplane that contacts the sphere of the microcluster. This is shown in Figure 5.3 with points $p$ and

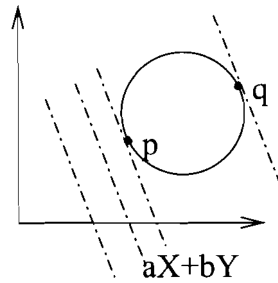



Figure 5.3: Contact Points to a CF-tree Microcluster

$q$. Details of how to compute the coordinates of contact points is given at the end of this subsection.

In the algorithm, a sorted queue $Q$ is used to maintain the processed points and blocks in the ascending order according to their score values. For an object, the score is the value of the score function with the coordinates of the object as the input, while for a data block, the score is the function value of its lowest point. The algorithm starts from the root node and inserts all its contained blocks to the queue. The first block entry in the queue will then be expanded, if the entry is at the leaf node, we will access the data points in it with some strategy. In the expanding process, we also keep track of how many data points are already presented, and if an object or a block is dominated by enough ($> k$) objects (single objects and objects in some blocks) lining in the queue before it, then it can be pruned. The expanding process stops when the queue has $k$ continuous data objects in the front. The pseudo-code of the algorithm is described below. In the algorithm, we assume there are at least $k$ tuples in the dataset.

**Algorithm 5.4.1** Branch-and-Bound Ranking(BBR) Method.
**Input:** A multi-dimensional index tree, $k$

**Output:** Top-k answers in $Q$
**Method:**

1. $Q$ :=Root Block;
2. WHILE top-$k$ tuples not found DO;
3. F:=the first non-object element from $Q$;
4. $S$ := SweepIntoBlock($F$); /$S$ is a set of blocks and/or objects
5. FOR each block/object $s$ in $S$ Do

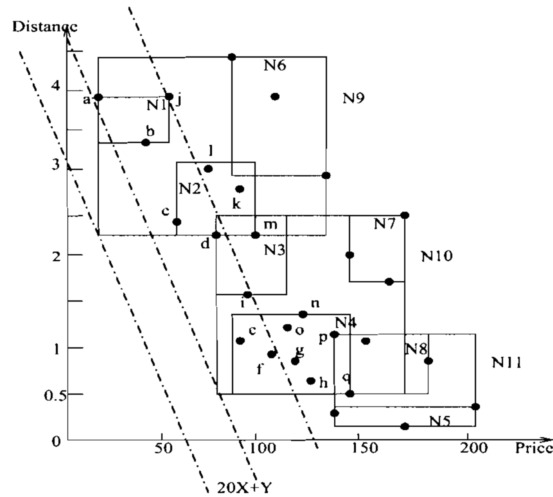



Figure 5.4: Sweeping over R-tree Blocks

6. IF more than $k$ objects in $Q$ having smaller scores than $s$
7. Discard $s$;
8. ELSE
9. Insert $s$ to $Q$;
10. Output $k$ objects from $Q$;

In line 4 of the algorithm description above, $S$ is a set of blocks and/or points. Here, the $k$ "better" objects in line 6 refer to single data points or points in a block whose highest point has a lower score than the current entry $s$. The insertion of $s$ to $Q$ is according to the score of its lowest point if $s$ is a block in line 9.

For the sweeping process **SweepIntoBlock**, Algorithm 5.4.1 can have different implementations when we sweep into leaf blocks. The straightforward approach is to simply expand the block and access all the objects contained (noted as method **BBR1**). It incurs unnecessary computational cost as each time it has to expand and access all the points in the leaf block, even if many of these points are actually impossible to be part of the top-$k$ answer. In order to avoid processing these unnecessary objects in each block, we make use of the property of the layered skylines since they give a contour of the data distribution, and develop an efficient sweeping within-layered-blocks method (noted as **BBR2**), as a procedure **SweepIntoBlock** in Algorithm 5.4.1. Details will be introduced in the next

subsection.

Lets look at an example of applying the Algorithm 5.4.1 to the dataset using the R-tree index when the implementation of *SweepIntoBlock* is to expand all the points in a leaf MBR each time. Assume we want to find top 7 tuples according to the score function $20X + Y(a = 20, b = 1)$ as illustrated in Figure 5.4. The algorithm starts from the root node and inserts all its entries $N_9$, $N_{10}$, $N_{11}$ into the queue sorted w.r.t. their scores. We then start to expand node $N_9$. The lowest point of a *block* in this case is the lower-left corner point of the MBR. The whole expanding steps in finding the top-7 tuples are shown below in Table 5.1. The first seven objects in the priority queue are the results.

Table 5.1: Example of Expanding Nodes in R-tree (1)

| Contents of priority queue | Expanded MBR |
|---|---|
| $N_9$, $N_{10}$, $N_{11}$ | Root |
| $N_1$, $N_{10}$, $N_2$, $N_6$,$N_{11}$ | $N_9$ |
| $a$,$N_{10}$,$N_2$,$b$, $N_6$, $j$, $N_{11}$ | $N_1$ |
| $a$,$N_4$,$N_2$,$b$,$N_3$, $N_6$,$j$, $N_{11}$ | $N_{10}$($N_7$ pruned) |
| $a$, $N_2$, $b$, $N_3$, $e$, $f$, $g$, $o$, $N_6$, $h$, $n$, $q$, $p$, $N_{11}$ | $N_4$ |
| $a$, $b$, $c$, $N_3$, $e$, $d$, $f$, $l$, $k$, $g$ | $N_2$($N_6$,$h$,$N_{11}$ pruned) |
| $a$, $b$, $c$, $d$, $e$, $f$, $i$, $l$ | $N_3$ |

The algorithm can also be easily applied to the dataset using the index structure of CF-tree. The crucial point is how to find the accurate lowest and highest points in a sphere microcluster for the score function. Given a sweeping hyperplane $y$ and a microcluster $F$ with radius $R$ centered at the origin as below:

$$y = a_1 \cdot x_1 + \ldots + a_d \cdot x_d$$

$$F(x_1, ..., x_d) = {x_1}^2 + {x_2}^2 + ... + {x_d}^2 - R^2 = 0 \quad (1)$$

To find the contact points, we only need to solve the following equation together with (1).

$$\nabla F(x'_1, ..., x'_d) = c \cdot \vec{A} \quad (2)$$

In (2), vector $\vec{A} = a_1, \ldots, a_d$, and $c$ is a free variable. $\nabla F(x_1, ..., x_d)$ is the gradient of $F$ at point $X(x'_1, ..., x'_d)$ representing the directional derivative and it is a $d$-vector ([3]):

$$\nabla F(x'_1, ..., x'_d) = (\frac{\partial F}{\partial x_1}(x'_1, \ldots, x'_d), \ldots, \frac{\partial F}{\partial x_d}(x'_1, \ldots, x'_d))$$

There are totally $(d+1)$ equations and $(d+1)$ variables to solve. Within only a constant computation time $O(d)$, the two contact points representing the lowest point and highest point are obtained.

### 5.4.2 Sweeping within Layered Blocks

In this subsection, we propose a layered accessing technique for the SweepIntoBlock procedure in Algorithm 5.4.1.

For each *block* in a leaf node with any index built in subsection 5.3, we create some additional data structures. Suppose the block contains $m$ layers of skylines, we maintain the lowest and highest object as well as the total number objects for that layer. The lowest and highest points in each layer correspond to the lower-left and higher-right points of the minimum bounding hyper-rectangle of the skyline objects in that layer. As shown in Figure 5.5 which is an enlargement of the block $N_4$ in Figure 5.4, objects $e$, $f$, $g$ and $h$ are layer 1 skyline objects in node $N_4$, and all these skyline objects in layer 1 are minimally bounded by a hyper-rectangle (see the red dashed box, called a pseudo node) denoted as $L_1_N_4$ (as used in the algorithm description example below). The same way applies to layer 2 skyline objects $o$, $q$ bounded by a pseudo node denoted as $L_2_N_4$, and similarly, layer 3 skyline objects $n$, $p$ are bounded by a pseudo node denoted as $L_3_N_4$. The linked list storage structure for the layered skylines is shown in Figure 5.6, where the header is the summarization information of the pseudo node which links to its bounding skyline objects. Now if a leaf *block* is chosen from the queue, we only expand the pseudo nodes in it that has the best lowest point according to the score function instead of all the objects.

Let us see an example of how BBR2 is applied in R-tree. The same dataset in Figure 5.4 now has been organized into pseudo nodes shown in Figure 5.7. The whole expanding steps in finding top-7 tuples are shown in Table 5.2, and it is clear to see that less objects are accessed compared to the objects in Table 5.1.

The case of applying BBR2 to CF-tree is similar to the processing in subsection 5.5.2, and we omit the details here.

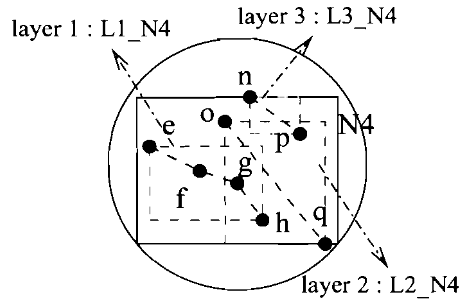



Figure 5.5: Layered-Skyline in Block N4

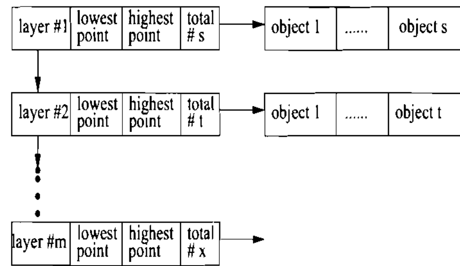



Figure 5.6: Linked List for Layered Skyline in a Block

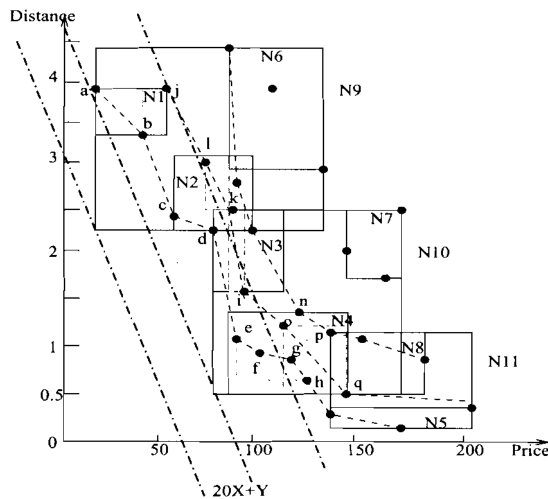



Figure 5.7: Sweeping Multilayered Skyline in R-tree

Table 5.2: Example of Expanding Nodes in R-tree (2)

| Contents of priority queue | Expanded MBR |
|---|---|
| (1)$N_9$, $N_{10}$, $N_{11}$ | Root |
| (2)$N_1$, $N_{10}$, $N_2$, $N_6$,$N_{11}$ | $N_9$ |
| (3)$a$, $N_{10}$, $N_2$, $b$, $L_2$_$N_1$, $N_6$, $N_{11}$ | $N_1$ |
| (4)$a$, $N_4$, $N_2$, $b$, $N_3$, $L_2$_$N_1$, $N_6$, $N_{11}$ | $N_{10}$($N_7$ prunned as $N_2$ and $N_3$ dominate it with enough objects) |
| (5)$a$, $N_2$, $b$, $N_3$, $e$, $f$, $L_2$_$N_4$, $L_2$_$N_1$, $g$, $N_6$, $h$, $N_{11}$ | $N_4$ |
| (6)$a$, $b$, $c$, $N_3$, $d$, $e$, $L_2$_$N_2$, $f$, $L_2$_$N_4$, $L_2$_$N_1$, $g$ | $N_2$ ($N_6$, $h$, $N_{11}$ pruned already enough tuples dominate them) |
| (7)$a$, $b$, $c$, $d$, $e$, $L_2$_$N_2$, $f$, $i$ | $N_3$($L_2$_$N_4$, $L_2$_$N_1$, $k$, $g$, $L_3$_$N_3$ pruned) |
| (8)$a$, $b$, $c$, $d$, $e$, $f$, $i$, $l$ | $L_2$_$N_2$ |

## 5.5 A Grid-based Sweeping Approach for Top-$k$ Queries

Although the KNN-based approach can efficiently obtain the top-$k$ objects, it may still visit and compare all the objects in a block even when the layering technique is applied. In this section, we present an alternative grid-based method for more efficiently organizing the objects. Since the user-specified weights of a score function will often have a fuzzy rather than a crisp semantic, approximate query processing seems to be acceptable if this allows significantly improved response time.

The basic idea is to build a grid-like partition of the *blocks* and access the objects within a block along the grid. For CF-tree, a shell-grid partition is made over microclusters. The microclusters are then assigned to the grid cells, and the sweeping algorithm is applied. This approach reduces the number of comparisons, but it may lead to a non-exact result if $k$ answer objects are found before a further grid cell with better objects is accessed. Since any MBR can be bounded in a sphere, we only illustrate the case of microclusters.

### 5.5.1 Shell Grid Partition of CF-tree

We first have an overview of a CF-tree with the extension of shell grid partition shown in Figure 5.8, where Figure 5.8($b$) depicts the anatomy of the intermediate microcluster node

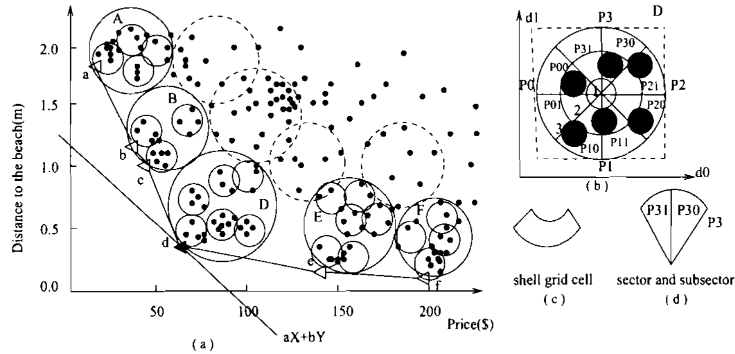



Figure 5.8: Shell-Grid Partition of Microclusters

$D$ in Figure 5.8($a$). The shape of a single **shell grid cell** (or **cell**) is shown in Figure 5.8($c$). The microclusters can be obtained during the construction of the CF-tree[96], where one threshold parameter to input is the diameter $T$ for the entries in a leaf node. We can enforce the this parameter to be smaller than a value of $\varepsilon$.

In order to obtain the shell-grid partitions described as above, we develop a novel partitioning method motivated by the technique of the Pyramid indexing [11]. The partition made is in partial shells (i.e. from a high dimensional point of view, the base of the partition is a part of the ($d$-1)-dimensional spherical surface) and the center of the sphere/circle is the pre-computed center using the statistics of CF-vector. We partition the space in a more granular fashion as described below. Assuming the sphere center $o$ of a node has coordinates ($o_0$, $o_1$,..., $o_{d-1}$), the spherical data space will be split into $2^{d-1} \cdot 2d$ fan-like partitions, with each partition having the sphere center as the top, and $1/(2^{d-1} \cdot 2d)$ part of the ($d$-1)-dimensional spherical surface as the base. The detailed partitioning process is given as the follows.

First, we split the sphere into $2d$ **sectors** as $P_0$,...,$P_{2d-1}$ according to the square cube with $2d$ surfaces (dashed square in Figure 5.8($b$) that encloses the sphere in the two dimensional case), as $P_3$ in Figure 5.8($d$)). Then we use the hyperplane perpendicular to each axis and passing through the sphere center to split the whole space. As a result, each sector $P_i$ is divided into $2^{d-1}$ **subsectors** as $P_{i0}$, $P_{i1}$, ..., $P_{i(2^{d-1}-1)}$. Then the whole space is divided with parallel complete spherical **shells** starting from the center. For example, in Figure 5.8($b$) the shell numbering/level 3 (in thick lines) refers to the outmost shell, and the one with 2 refers to the middle layer shell while the one with 1 refers to the innermost shell.

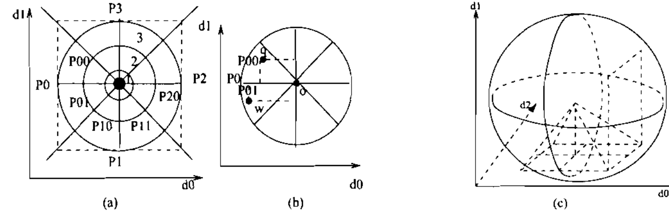


Figure 5.9: 2D SG-Partition and 3D SG-Partition

Figure 5.8($b$) or Figure 5.9($a$) shows a 2-dimensional case, and Figure 5.9($c$) illustrates the scenario of 3-dimensional partition. Finally, the intersections of sectors and shells form into **cells** as shown in Figure 5.8($c$).

We observe that any object $x$ with coordinates ($x_0$, ..., $x_{d-1}$) in sector $P_i$ has the following property:

**Lemma 5.5.1** *For any object $x$ in sector $P_i$ where $i < d$, it satisfies: for any dimension $j$, $0 \leq j < d$, $i \neq j$, $|o_i - x_i| \geq |o_j - x_j|$; for sector $P_i$ where $i \geq d$, it satisfies: for any dimension $j$, $0 \leq j < d$, $j \neq (i - d)$, $|o_{i-d} - x_{i-d}| \geq |o_j - x_j|$.*

**Proof.** Based on the construction of shell grid, it is clear to derive that for any point in $P_i$ sector, the distance of its $i$th coordinate (if $i < d$) or $i$-$d$th (if $i > d$) from the coordinate of the center of the sphere is greater than the distance of all the other coordinates. ∎

For example, for $q, w$ in Figure 5.9($b$), it satisfies $|o_0 - q_0| \geq |o_1 - q_1|$. This is the same for $w$. With this property, when we want to check which sector $P_i$ an object belongs to, it only needs to calculate the dimension number $i$ for this object whose coordinate has the largest distance to that of the center. If its coordinate value in $i$ is less than that of the center, the sector number is $i$, otherwise it is $i + d$.

The data objects in every subsector in $P_i$ certainly follow the Lemma 5.5.1. We further number these subsectors $j$ (for subsector $P_{ij}$) from 0 to $2^{d-1} - 1$ in ($d$-1) bits of binary format $s_0$, ..., $s_{d-1}$. If $i < d$, then the bit $s_i$ does not exist in the binary string, and if $i \geq d$, then the bit $s_{i-d}$ is excluded. For each subsector number with the above numbered scheme, if the bit $s_j = 1$, then points in that subsector have the property of $x_j > x_i$, and it is opposite for $s_j = 0$. As an example, subsectors in sector $P_0$ are numbered with a single bit $s_1$ since bit $s_0$ is excluded from the 2-bit string $s_0$, $s_1$. If $s_1$ is equal to one (i.e. the subsector is $P_01$), then any point $x$ in this subsector has the property $x_1 > x_0$.

Note that each subsector has $2^{d-1}-1$ direct neighbors in the same sector, and $(d-1)\cdot 2^{d-2}$ ones in the neighboring sector. For example, in the 2-$d$ case shown in Figure 5.8($b$), $P_{01}$ has one neighbor subsector $P_{00}$ in the same sector $P_0$, and another neighbor $P_{10}$ in its neighboring sector $P_1$. From now on, all the partitions we mentioned will be implicitly subsectors. To facilitate the grid index building, we also define the height of a microcluster in the shell partition.

**Definition 5.5.2 (Height of a Microcluster)** *Given a microcluster mc with its statistical information and the center $O(x_0, ..., x_{d-1})$ of the node where mc resides, the height of the microcluster $H_{mc}$ in the grid index is defined as: $H_{mc} = dist(center(mc), O)$, where $center(mc)$ denotes the center of mc.* ∎

For the leaf microcluster nodes, we set the grid shell width as $\varepsilon$ so that no microclusters can stay across more than two shells as shown in Figure 5.8, and the center point can only reside in a single shell. Each microcluster has a high and low shell number between which it resides (a microcluster could reside right within one shell layer or run across two layers where high and low shells are the two layers it runs across). From the definition of the height of a microcluster, we can easily compute which shell(s) a microcluster resides in.

Based on all the above definitions and Lemma 5.5.1, with CF-tree as an input, we can start the hierarchical internal node grid index building process. The the microcluster entries in a node are scanned and their subsector number and shell number are computed (according to the center of the node and the microcluster). Then in each partition, microclusters are stored in a linked list so that they can be accessed in a shell ascending/descending way during the sweeping process. The largest shell number of microclusters in each node are also recorded.

### 5.5.2 Ranking Algorithm

We now present the sweeping algorithm for the top-$k$ ranked query, and illustrate that the error bound is within $\varepsilon$. The main idea of our algorithm is to utilize the shell grid partition and provide direction on how to expand necessary cells. To start the sweeping, we calculate which sector the sweeping hyperplane contacts first, and the corresponding sector number applies to any level of internal nodes. As described in subsection , within only a constant computation time $O(d)$, the coordinates of the two contact points $L$ and $H$ representing the lower point and higher point are obtained.

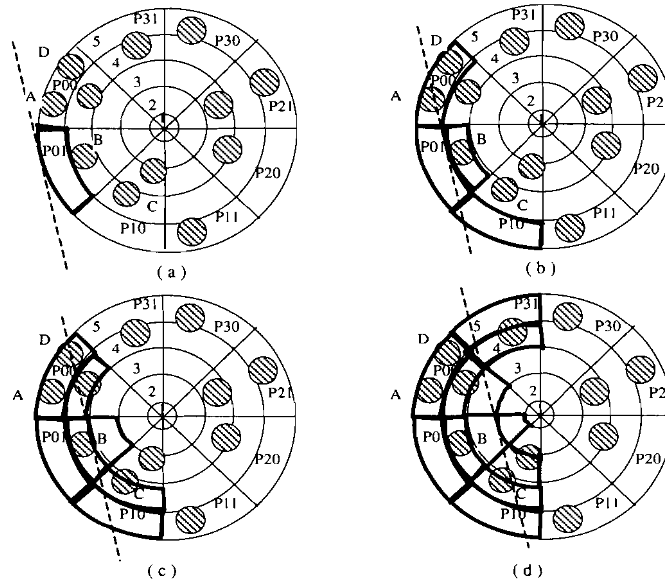



Figure 5.10: Sweeping a Shell Grid

Afterwards, we can easily find which sector the lower point belongs to according to the coordinates of the contact point. In Figure 5.10($a$), the sweeping line will contact sector $P_{01}$ first.

One useful property in the sweeping algorithm is as follows:

**Property 5.5.3** *The sweeping process first explores the outmost shell grid cell of the partition which the sweeping hyperplane tangent contacts, then goes to its directed neighboring cells in the same level of shell. If there is no data in those neighboring cells, sweeping should go to the inner cell in the same partition directly.* ■

This property is illustrated in an example shown in Figure 5.10. Microcluster $A$ has the shell number 5 which is the largest of this node. During the sweeping process, the hyperplane touches sector $P_{01}$ first shown in Figure 5.10($a$) and there is no data in this cell with shell number 5 in partition $P_{01}$, then we should access $P_{00}$ next (i.e. access microcluster $A$ and $D$) shown in Figure 5.10($b$). Assuming $A$ and $D$ are not there, then we should access microcluster $B$ in the inner cell of the current partition $P_{01}$ next (since $P_{10}$ contains no data) instead of going further along the same shell to partitions $P_{11}$ and $P_{31}$ shown in Figure 5.10($b$). According to the score function represented by the sweeping hyperplane,

cells with shell number 5 (cells $P_{10}$ and $P_{00}$) could be as good or better than the cell in $P_{01}$ with shell number 4. Here the "good or better" means the score of the lowest point of the microcluster in the cell is equal or smaller. In the example, the sweeping process shown in Figure 5.10($c$) and Figure 5.10($d$) continues as more neighboring cells are accessed until top-$k$ objects are found.

Now let us describe the sweeping algorithm. Assuming we have already calculated out the two standard contacting points and the partition number $P_{mn}$ that the sweeping plane contacts first, the sweeping process works in a hierarchical way. A sorted queue is used to store the expanding entities including the intermediate microclusters, leaf microclusters, a pseudo node (as introduced below) and any shell of a node. These entities are put in the sorted queue according to the score of their standard contacting point.

We start from the outmost spherical shell of the root node, then expand this entity based on Property 5.5.3. To illustrate this, we use Figure 5.10($a$) again as an example. The symbol $P_{ij_l}$ denotes the cell in shell level $l$ of subsector $P_{ij}$ and $PS_l$ denotes the whole spherical shell of $l$. First we expand the microclusters in the cell of subsector $P_{mn}$ ($P_{01}$ in this case, or if no data exists there, then we go to its directly neighboring cells ($P_{00_5}$ and $P_{10_5}$). At the same time, the inner spherical shell ($PS_4$) is added to the queue too. Besides expanding on the subsector $P_{mn}$ ($P_{01}$ in the example) and its direct neighboring cells ($P_{00_5}$ and $P_{10_5}$ in the example), the sweeping process also needs to decide when to expand the unvisited cells of other subsectors (such as those in $P_{31}$ and $P_{11}$ ). To handle this, in the sweeping process we add to the queue a special pseudo entity which has the subsector number and shell number recorded in it. A typical pseudo entity is to use a special copy of the worst microcluster entry in the expanding cell, so that when that entry is being scanned, we know that it has potential to explore cells in its neighboring sector. An example in Figure 5.10 is that when $P_{00_5}$ is being expanded, microcluster $A$ and $D$ are inserted into the queue, but in addition, a copy entry of $D$ (which is the worst in this cell) $D_e$ is also inserted to indicate that the cell in its neighboring subsector $P_{31}$ should be explored before expanding some worse microclusters in $P_{01_3}$ or the shell $PS_2$. If there is no data in the current expanding cell, we can relax the Property 5.5.3 a bit in the implementation by finding its neighboring cells that contain data.

When using the standard contacting points of a shell in the sweeping process, a little detail needs to be taken care of. Before the sweeping passes the half sphere (shell number decreases to 1), the lowest contacting point is used to compute the score, while after passing

the half sphere (shell number increases again from 1) , the highest contact point is used instead. Actually when $k$ is not a big value, the sweeping hardly needs to pass over the half sphere. Moreover, after passing the half sphere, we access microclusters in a cell in the order of from a lower shell to a higher shell which is easily done through the doubly linked list. In the sweeping process, when a leaf entry microcluster appears in the front of the queue, we remove it off and put in an output list. When the total data objects number contained in the list begins to exceed $k$, the sweeping process stops. The following pseudo-code describes the processing of the approximate top-$k$ query. MC is used as an abbreviation of microcluster.

**Algorithm 5.5.1** A Shell-Grid Ranking(SGR) Method.
**Input:** CF-tree with Grid Shell Partitions, $k$.
**Output:** Top-$k$ answers in list T.
**Method:**

```
1.  Calculate standard contacting points and the contained
    subsector number P_mn ;
2.  Q = ∅; T = ∅;
3.  Insert into Q the outmost cell of root node of CF;
4.  WHILE the first k tuples are not found DO;
5.      Remove the first entity E in Q;
6.      IF E is a cell
7.          Insert its microclusters in subsector P_mn and its direct
            neighbors as well as the pseudo-entities;
8.      ELSE IF E is an intermediate node;
9.          Insert into Q the outmost cell of E;
10.     ELSE IF E is a pseudo-entity;
11.         Insert into Q microclusters in its neighbor subsectors
            of the same cell and the pseudo entities;
12.     ELSE IF E is a leaf node
13.         Add entries in E to Q;
14.     ELSE IF E is a leaf entry
15.         Add E to T;
16. Output k points from T;
```

### 5.5.3 Error Bound in Approximate Solution

To analyze the corresponding error bound which is measured in the score difference of the resulting objects and the actual objects which should be in the result set in the exact result case.

We observe that in the sweeping process shown in Figure 5.11, some objects in other microclusters(i.e. MC2) are better than the objects in the current selected microcluster (i.e. MC1). In the extreme case, point $q$ is computed as part of the answer instead of point $w$ whose score is just a little bit larger than that of $p$. The following lemma gives the error bound of the approximate solution in SGR.

**Lemma 5.5.4** *The maximum error in the approximate top-k objects is $O(\varepsilon)$.*

**Proof.** Assuming the coordinates of point $p$ is $(x_1, ..., x_d)$ while those of point $q$ is $(x'_1, ..., x'_d)$, then the score difference of $q$ and $w$ is: $f(q) - f(w) < f(q) - f(p) = a_1 \cdot (x'_1 - x_1) + ... + a_d \cdot (x'_d - x_d) < \varepsilon \cdot \sum a_i = \varepsilon$. ∎

So the score difference between the worst point in the selected microcluster and the best point in the unexplored one is less than $\varepsilon$. Apparently this is the extreme case, while in many circumstances of the sweeping process, the error is much smaller or no error at all. On the other hand, there is a tradeoff between the choice of accuracy and the index cost. A large radius will affect the computation accuracy, but can save index space and facilitate fast processing, while a small radius can bring higher level of accuracy in the cost of high indexing cost.

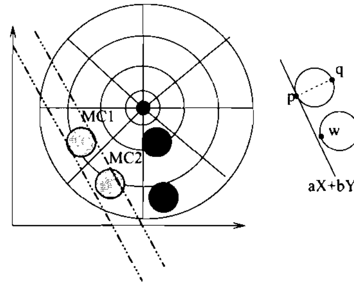



Figure 5.11: Error Bound of Grid-based Answer

## 5.6 Experimental Results

In this section, we report the results of experimental evaluation for our algorithms and other related algorithms. The evaluation investigates different aspects including preprocessing time, query time, error bound and the number of tuples retrieved.

We implement the proposed methods and Onion in C++, and obtain PREFER in *www.db.ucsd.edu/prefer*. The experiments were run on a Intel 883MHZ machine with 512M RAM running Windows 2000.

For a fair comparison of the algorithms, we use the data generator in [13] to generate two types of synthetic relational datasets of 100,000 records with 5 attributes. As the same as that described in the Onion technique [21] and PREFER system [44], the attributes of the first type of dataset are independently distributed, and the attributes of the second type of dataset are correlated. For comparison purpose, all the attribute values are scaled to positive integer values (with the value domain of each dimension between 1 and 1000) as the PREFER system application only accepts this type of data.

The experiment consists of two parts: (1) the comparisons of our algorithms BBR and SGR. More specifically, "BBR1" refers to the Branch-and-Bound Ranking with sweeping of all objects in a MBR, "BBR2" refers to the Branch-and-Bound Ranking with sweeping of objects layer-by-layer in a MBR and "SGR" refers to the Shell Grid Ranking algorithm with microclusters indexed in a CF-tree. We also show how the query time and the size choice of the microclusters affect the results. (2) The comparisons of our algorithms with the most related rank query work on the algorithms of the Onion technique and the PREFER system.

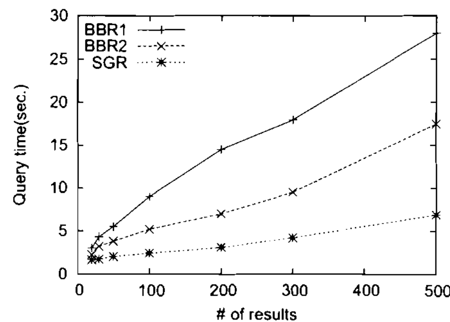



Figure 5.12: Query Time vs #of Results in Correlated Data Set

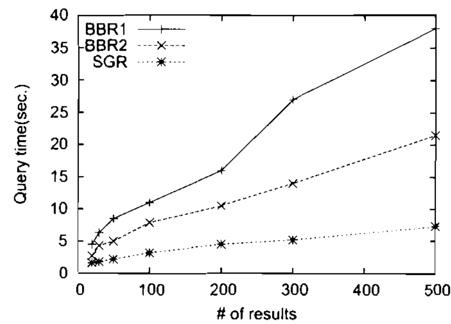



Figure 5.13: Query Time vs #of Results in Independent Data Set

Similar to our method, Onion and PREFER use the pre-computing approach to facilitate ranked query, where Onion materializes the layers of convex hull while PREFER materializes the views of answer sets of the ranked queries. A more detailed description is given in the section of the related work. The evaluation is based on the issues of preprocessing time, top-$k$ rank query time, the number of objects retrieved for answering the query and the approximation quality.

### 5.6.1 Evaluations of Algorithms BBR and SGR

**Runtime Comparison.** We show the results of the runtime for the three algorithms w.r.t various number of resulting objects ($k$). Figure 5.12 and 5.13 show the results for the correlated and independent dataset respectively. The number of attributes used in this experiment is 4. In general, all the algorithms for correlated data have relative better performance than in independent distribution data since there are less number of MBRs/Microclusters to be accessed in correlated distribution. BBR2 method is much faster than the BBR1 method because of its visiting less objects (in the experiment we only pre-computed layered skylines for $K$=200 and we found that is enough for answering the top-$k$ query for $k$ increasing to as big as 500). While the SGR method (here the $\varepsilon$ used in this experiment is 10) is faster than BBR2 due to the fact that it always checks for the most relevant data.

**The Effect of $\varepsilon$ on SGR Method.** Next we investigate how the query time and error rate change with different choices of the value of $\varepsilon$. We set $\varepsilon$ as the value of 5, 10 and 15 respectively and perform tests on the 4-$d$ 100,000 records correlated dataset. The error rate is computed as the ratio of the average resulting score difference from that of the true

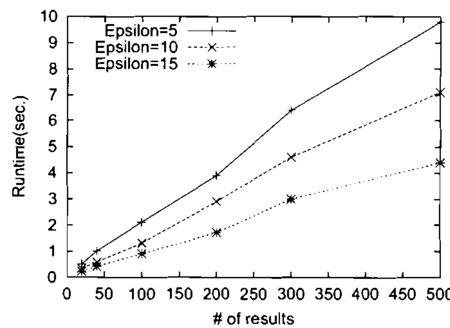



Figure 5.14: Effect of Eps(1)

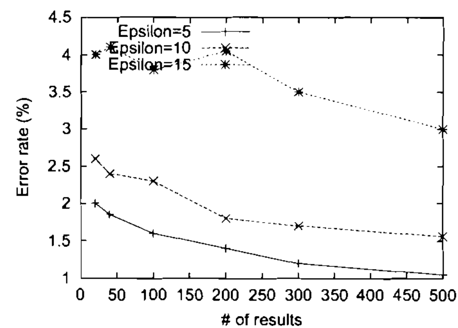



Figure 5.15: Effect of Eps(2)

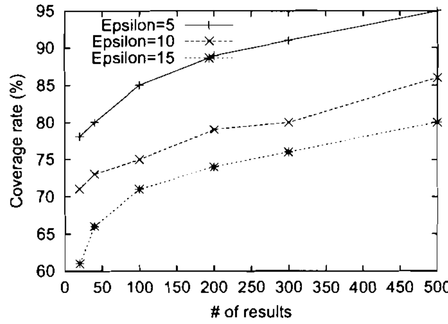



Figure 5.16: Effect of Eps(3)

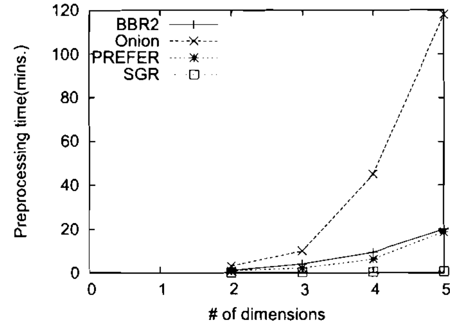



Figure 5.17: Preprocessing Time

answer set to the average score of the true answer set.

We also measure the coverage rate which is defined as the percentage of the true top-$k$ points covered by the approximate answer tuples. The independent dataset is used in the comparisons. The result of Figure 5.14, Figure 5.15 and Figure 5.16 show that when the value of $\varepsilon$ becomes larger, the runtime becomes shorter due to the decreasing number of microclusters that need to be accessed, and the error rate as well as the coverage rate are getting relatively higher due to the increasing size of microclusters. But for the same $\varepsilon$, with the number of requested objects increasing, the error rate decreases and coverage rate increases since more truely top-$k$ ranked objects are found in the resulting microclusters (i.e. most of the early returned microclusters contains fully the actual top ranked objects). So the approximate method is in favor of a large $k$.

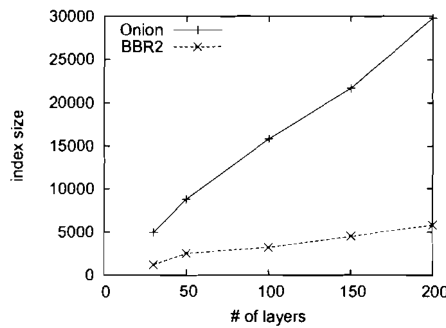



Figure 5.18: Index Size vs #of Layers on Correlated Data Set

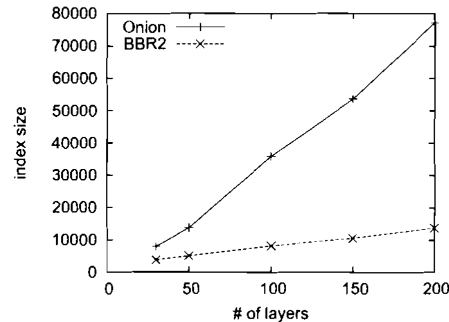



Figure 5.19: Index Size vs #of Layers on Independent Data Set

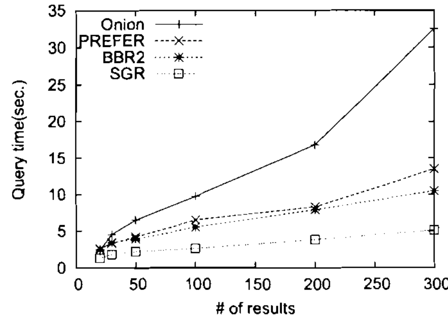



Figure 5.20: Query Time vs #of Results on Correlated Data Set

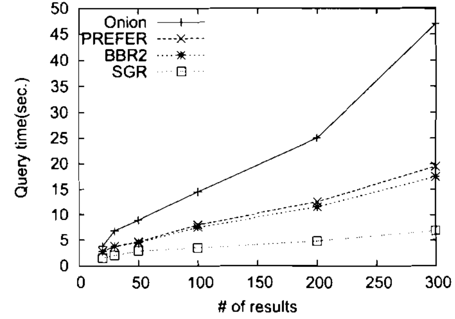



Figure 5.21: Query Time vs #of Results on Independent Data Set

### 5.6.2 Comparing with Algorithms of Onion and PREFER

We compare our algorithms with the Onion technique and the PREFER system from different aspects. The Onion technique does not need any specific parameters, while the PREFER needs the number of views to be pre-materialized and the depth value of tables where the small values for both input can easily lead to the large size of storage. There are two factors to be considered as the criterion of the evaluations. First, we need to examine the preprocessing time which refers to the index building time for each method as all these methods has the pre-computing process. Second, we need to consider the query answering time and how many records each method retrieves to answer the top-$k$ ranked query.

**Preprocessing Time Comparison.** We compare the preprocessing time of the BBR2 method, SGR method, the Onion technique and the PREFER method. For the PREFER,

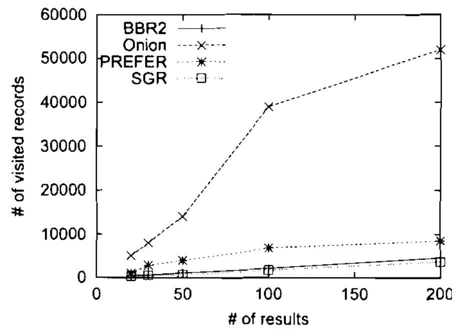



Figure 5.22: #of Tuples Visited vs #of Results on Independent Data Set

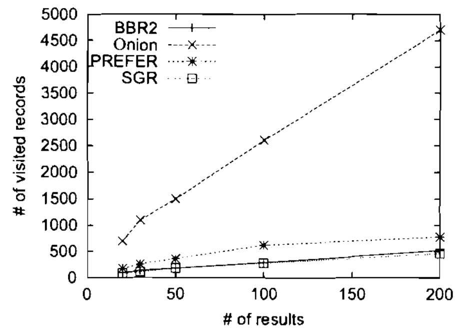



Figure 5.23: #of Tuples Visited vs #of Results on Correlated Data Set

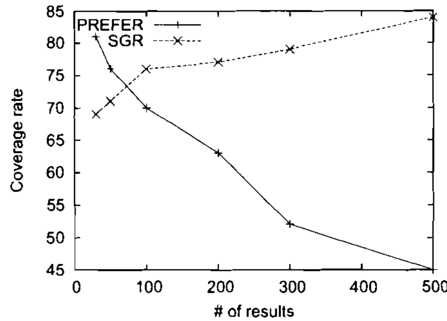



Figure 5.24: Coverage Rate vs #of Results on Independent Data Set

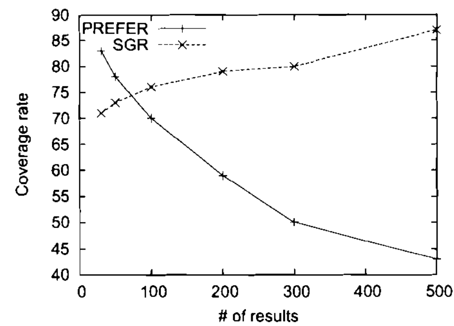



Figure 5.25: Coverage Rate vs #of Results on Correlated Data Set

as there is no estimated number of tuples needed to store in each view for answering the top-$k$ query, we have to choose the view depth as the same as the dataset size which results in very long building time for each view. Also the views building time depends on how many views are to be built, and the number of views needed will increase dramatically with the dimension increasing in order to achieve good performance since the pre-built views are tending to cover all the possible queries. Our SGR does not have this problem because it always starts from the right point/partition no matter how the coefficients of the score function change. In addition, a large number of views each with a high depth cost huge amount of disk space, which is not acceptable for very large database. We compare the time to construct $K$ layers of skylines for BBR method and $K$ layers of convex hulls for the Onion method w.r.t. $K = 200$ as well as the shell index building for the SGR method. As index building takes a very long time to finish for the Onion technique especially in high dimension, so we restrict the number of dimension within 5. We experiment with the correlated dataset with size of 100,000 records and dimensions of 2,3,4 and 5 respectively. For each dimension, the number of views constructed for the PREFER are 3, 10, 30, 80 respectively.

Figure 5.17 shows the preprocessing time in minutes for these four techniques. It is clear that the SGR method has the least preprocessing time among all the methods since it only naturally does one scan of the data when building the CF-tree with extended shell grid index. The preprocessing time of BBR2 is much faster than the Onion technique, especially when the dimension getting higher. When more dimensions involve, the preprocessing time for the PREFER starts to increase more than BBR2 in this correlated dataset.

**Size of Materialization Comparison.** The relationship between the index size and the maximum number of requested objects (the number of layers is equal to this number) is shown in Figure 5.18 and Figure 5.19 which depict the comparison results of BBR2 and Onion in the case of correlated and independent dataset respectively. The dataset size is 100,000 and the dimension is 5. The PREFER system is not listed for comparison, because it has no notion of layer as an input and basically the materialized size is the number of views times the whole dataset size. From the Figure 5.18 and Figure 5.19, it verifies that the correlated dataset always contains smaller number of skylines, so it requires smaller materialized space than the independent dataset, and in each case BBR2 builds smaller size of index than the Onion does.

**Query Time Comparison.** The comparison of the query time of the BBR2, the Onion technique, the PREFER system and the SGR w.r.t. various number of requested objects $k$ in independent dataset and correlated dataset are present in Figure 5.20 and Figure 5.21 respectively. The dataset size is 100,000 and the dimension of the dataset is 5. The computation of the BBR2 method and the Onion technique are based on the index building from the previous paragraph. Similarly, we can see that the results of the correlated dataset outperforms that of the independent dataset. From the results, we can also see that our SGR method has apparent advantage over the other three methods, BBR2 ranks second and PREFER is better than Onion in query time.

**Comparison of the Number of Tuples Retrieved.** Figure 5.22 and Figure 5.23 show the number of retrieved tuples visited under the independent dataset and correlated dataset respectively, to answer the top-$k$ query for the BRR2, the Onion, the PREFER and the SGR. The experiment setting is the same as that for the query time comparison. We observe that the Onion method retrieve much more tuples than the other two, and both the SGR method and the BBR2 method show a better performance over the PREFER and the Onion. Also, the independent dataset does not work as good as correlated dataset.

**Quality of the Approximate Answer.** As the PREFER system uses materialized views to speed up the ranked query, the size of the materialized views determines the accuracy of the results. In case of limited space is given for storing the pre-materialized views, the PREFER can only give partial correct answers which we take as approximate answers. So based on the same size for the materialized objects or views, we compare the quality of approximate answers of the SGR and the PREFER. Figure 5.24 and Figure 5.25 show the cases for the independent dataset and the correlated dataset respectively. When $k$ increases,

the quality of the top-$k$ answers evaluated by the coverage rate for the SGR becomes much more higher than that of the PREFER. In fact we can see that the coverage rate increases for the SGR while that for the PREFER degrades quickly. Also from the Figure 5.24 and Figure 5.25 we know, the distributions in two dataset do not make much difference to the result of the approximate ranked queries.

From the above experiments, it shows that our sweeping algorithms outperform the Onion and the PREFER in a variant of tests, and demonstrates the efficiency and the effectiveness of our algorithms.

## 5.7 Revisit of Top-$k$ Ranked Queries

In database systems, the nearest-neighbors query, rank query and skyline query are very common and related query operations. The nearest-neighbors query corresponds to a query object $q$ and returns $k$ objects which are closest to $q$. Implicitly, each attribute has the same weight in calculating the overall similarity of each object to $q$. The skyline query returns a set of objects which are not dominated by any other object. If the dimensionality is not high, say less than 5, the cardinality of skyline is usually much smaller than the size of the dataset. Compare the top-$k$ query and nearest-neighbor query, no weights are assigned for skyline query.

Agrawal et. al. in their pioneering work [2] put the notion on preferences into perspective and introduce a framework for their expression and combination. The top-$k$ ranked query problem was formally raised by Fagin in multimedia database systems in [32]. Methods can be basically categorized into the following types:

- **Sorted Accessing and Ranking:** this approach mainly applies some strategies to sequentially searching the sorted list of each attributes until the top-$k$ tuples has been retrieved.

  The authors in [32] assume independent sub-systems handle each atomic top-$k$ query, and a middleware is responsible for combining these results for a final result. Wimmers et. al. [92] describe the implementation of Fagin's algorithm for merging ordered streams of ranked results in the Garlic multimedia middleware system. The MARS system [71] uses variations of Fagin's algorithm and views queries as binary trees. Guntzer et. al [36] proposed an incremental method to compute and output top-$k$

query. Compared to Fagin's algorithm, they develop an improved termination condition in tuned combination with heuristic control flow adopting itself narrowly to the particular score distribution.

In [33], Fagin develops a "threshold algorithm (TA)" , which scans all query-relevant index lists in an interleaved manner, and maintains the worst score among the current top-$k$ results and the best possible score for all other candidates and items not yet encountered. Nepal and Ramakrishna [69] define an algorithm that is equivalent to TA, but the notion of optimality is weaker. The paper [88] focuses on providing approximate TA variants based on probabilistic arguments. The main purpose is to prune candidate items and to reduce the index scans with high probability, and the authors provide solutions in different distributions such as uniform, Poisson etc.

- **Random Accessing and Ranking:** this approach supports mainly random access over the dataset until the top-$k$ tuples has been retrieved.

  The work in [82] proposes an approximate method of top-$k$ ranked query in terms of a score vector and a weight vector. Whether a tuple matches the query is expressed in a weighted cosine similarity between the query and tuple score vector. The accuracy criteria is based on the sum score ratio of the results to the accurate ones and the overlapping tuples in the two sets. The paper [40] uses foot-rule distance to measure the two rankings and model the rank problem as as the minimum cost perfect matching (mcpm) problem. A Hungarian Algorithm (HA) and a Successive Shortest Paths (SSP) algorithm are given to solve the top-$k$ query.

  Chang and Hwang [20] developed $MPro$, a method to optimize the execution of expensive predicates for top-$k$ queries. While [25] proposes a way to translate the top-$k$ query into a range query of the relational database. A score $s_q$ is determined heuristically by database statistics. Then a range query, composed of the provided scoring function, $s_q$ and the query object, retrieves all the tuples with score greater than $s_q$ and gives the final result. [63] presents an algorithm for web-based top-$k$ query, which interleaves both random access and sorted access.

- **Materialization and Rank Indices:** this approach organizes the tuples in a special way or pre-computes answers for a set of ranked queries as views, then applies similarity match for the answer of any new ranked query.

An index technique for any linear optimization queries is introduced in [21] with the geometry property that the optimal value is achieved at some vertices in convex hull. The index is constructed by building layered convex hulls for all the data points. Queries are evaluated from the most outward convex to inner side until the top $k$ are found. Building such index for very large databases costs expensive due to the convex hull finding complexity.

To overcome the drawbacks of [21], authors in [44, 45] proposes a method based on the pre-materialized views to answer top-$k$ query. When the query's preference function is close to that of a view's, a small number of the tuples in the view is necessary for the top tuples. Then the query result is produced in a pipelined fashion by excluding the top tuples found previously from the view and repeating the process until all answers are identified. As there is no guarantee how many tuples should each view stores to answer top-$k$ query, it usually ends up with storing the whole data set in each view in a sorted manner, which costs huge space, but is unavoidable when the number of requested tuples is large. Also the number of views will increase dramatically with the dimension increases in order to cover the whole space.

Tsaparas et.al.[89] proposes a ranked index to support top-$k$ query ($k \leq K$) where the point with coordinates value of the two weights forms a vector together with the origin coordinate. When this vector sweeps the positive quadrant of the plane, it partitions the plane into regions where two adjacent regions have different set of top-$K$ tuples and all these top-$K$ tuples are materialized as index for later query search. The drawback is that it only applies to 2 dimensions practically and a huge number of materialized partitions could exist.

## 5.8 Summary

Rank-aware query processing, where the system returns only the top-ranked answers w.r.t. user-defined weights of the different attributes, has recently emerged as an important paradigm in database systems. Only few existing methods for ranked queries exploit pre-materialization and index structures. In this chapter, we introduce a novel approach based on the observation that for any score function (weights) the top-$k$ ranked answers must be contained in the first $k$ skyline layers [49]. These skyline layers can be efficiently determined and are stored in existing multi-dimensional index structures that are enhanced by special data structures

representing the different skyline layers. As a consequence, we propose indexing layered skylines and indexing shell-grid microclusters for top-$k$ ranked queries, and present methods for sweeping the hyperplane of the score function over the indexed objects. Our methods can be easily adapted to the existing multi-dimensional index structures. The experimental results demonstrate the strength of our methods and the usefulness of the microclustering technique in top-$k$ query processing. This study suggests the following interesting topics for future research. The choice of an appropriate maximum number $K$ of layers to be materialized, e.g. based on query statistics, deserves further attention. The proposed methods for ranked queries are generic and independent from the particular index structure. They should be implemented and experimentally evaluated also in the context of other index structures. Finally, it would be interesting to investigate the impact of high-dimensional data, where only a small number of attributes would be assigned non-zero weights, on ranked queries and the performance of the methods presented.

# Chapter 6

# The Multi-Relational Skyline Operator

Most of the existing work on skyline query has been extensively used in decision support, recommending systems etc, and mainly focuses on the efficiency issue for a single table. However the data retrieved by users for the targeting skylines may often be stored in multiple tables, and requires to perform join operations among tables. As a result, the cost on computing skylines on the joined table will be increased dramatically due to its potentially increasing cardinality and dimensionality. How to develop efficient methods to share the join processing with skyline computation is central to the skyline query optimization on multiple relations. In this chapter, we systematically study the skyline operator on multi-relational databases, and propose solutions aiming at seamlessly integrating state-of-the-art join methods into skyline computation. To further extend the query optimizer's cost model to accommodate skyline operator over joined tables, we also theoretically estimate the size of the joined skylines. Our experiments not only demonstrate that the proposed methods are efficient, but also show the promising applicability of extending skyline operator to other typical database operators such as join and aggregates.

## 6.1 Introduction

The skyline query has been recently proposed and extensively studied as an important query operator for preference queries, decision support and recommending systems in database

communities. However, most of the existing work on skyline queries for databases mainly discusses the computation efficiency in one single relational table. While in many database applications, users are often interested in querying skylines from multiple relational tables which requires join operations. Formally, we call the skyline operator over a multi-relational joined table $A_1 \bowtie A_2 \ldots \bowtie A_k$ as the *multi-relational skyline operator*, or simply as the *skyline join operator*[1], denoted by $A_1 \bowtie_s A_2 \ldots \bowtie_s A_k$ where $A_1, \ldots, A_k$ are relations.

The following example skyline queries aim to find model customers in TPC-D datasets (www.tpc.org) for the company such that the higher the *account balance*, the lower the *age* of a customer (in relation *customer*), and also the higher the *quantity* of parts and the *amount* of price (in relation *order*) this customer orders, the "better" this customer.

**Example 13** *Given customer table* Customer (**<u>CNum</u>**, Age, Account Balance) *(as shown in Table 6.1) and part order table* Order (**<u>ONum</u>**, CNum, PNum, Quantity, Amount) *(as shown in Table 6.2). Consider the following queries:*

Question (1): *Who are those young customers with high account balance and having* **ONE** *order with high quantities of parts and high amount of price?*

Question (2): *Who are those young customers with high account balance and high quantities of parts order with high* **TOTAL** *amount of price (over all orders of this customer)?*

Answer: *The SQL statements for answering the above questions are as follows:*

**SELECT** *
**FROM** *Customer C, Order O*
**WHERE** *C.CNum = O.CNum*
**SKYLINE** *OF C.Age* **Min**, *C.AccountBalance* **Max**, *O.Quantity* **Max**, *O.Amount* **Max**;

**SELECT** *C.CNum*, **SUM**(*O.Quantity*), **SUM**(*O.Amount*)
**FROM** *Customer C, Order O*
**WHERE** *C.CNum = O.CNum*
**SKYLINE** *OF C.Age* **Min**, *C.AccountBalance* **Max**, *O.Quantity* **Max**, *O.Amount* **Max**
**GROUP By** *C.CNum;*

*The join results and the skyline objects in Customer $\bowtie$ Order where (1) attributes CNum of C and CNum of O are* join attributes; *(2) attributes Age, AccountBalance, Quantity and Amount*

[1]In this chapter, terms of "multi-relational skyline operator", "skyline join operator" or "joined skylines" all refer to the skyline objects in the joined table

*are* descriptive attributes *participating in skyline evaluation (under shaded regions), are shown in Table 6.3 and Table 6.4 respectively (***skyline objects are in bold font***)*[2]. *The joined skylines may contain tuples that are not in the skyline of the individual input tables, such as the tuple with* $D1 = 105$. *Notice that the skyline queries w.r.t. (1) and (2) return different answers due to the effect of the aggregate constraints being pushed on descriptive attributes. A customer may be a best buyer according to the quantity and price for one sale, but may not be a best buyer if considering all the orders he/she made.*

Table 6.1: Customer(C) Table

| **CNum** | *Age* | *Account Balance* ($) |
|---|---|---|
| **101** | **35** | **90k** |
| 102 | 40 | 40k |
| 103 | 50 | 78k |
| **104** | **35** | **90k** |
| 105 | 58 | 90k |

Table 6.2: Order(O) Table

| ONum | **CNum** | PNum | *Quantity* | *Amount*($) |
|---|---|---|---|---|
| 1 | 101 | 001 | 1 | 274 |
| 2 | 101 | 001 | 6 | 1644 |
| **3** | **102** | **002** | **10** | **1999.9** |
| 4 | 103 | 003 | 1 | 400 |
| 5 | 104 | 004 | 5 | 900 |
| 6 | 104 | 004 | 6 | 1080 |
| 7 | 105 | 005 | 2 | 1900 |

The join operation in Example 13 is *non-reductive* [15] (this assumption holds in the remaining of the chapter) in terms that the size of joined table is larger or equal to that of any table participating in the join operation. Also the tuples have higher dimensions, so that the cost of applying skyline operator after joins would be much more expensive[3]. For arbitrary join operations, since both the cardinality and dimensionality of the joined table might increase, so the cost of finding skylines in the joined table will be even larger. How to

[2] *For simplicity, we denote C.CNum as variables $D_1$, Age as $D_2$, AccountBalance as $D_3$, ONum as $D_4$, O.CNum as $D_5$, Quantity as $D_6$ and Amount as $D_7$ (for simplicity here we omit PNum in the joined table), and these notations are used in the remaining of this chapter.*

[3] The cost of skyline computation methods increases w.r.t. the increase of dimensionality.

Table 6.3: Joined Table of Customer and Order (1)

| **D1** | *D2* | *D3* | D4 | **D5** | *D6* | *D7* |
|---|---|---|---|---|---|---|
| 101 | 35 | 90k | 1 | 101 | 1 | 274 |
| **101** | **35** | **90k** | **2** | **101** | **6** | **1644** |
| **102** | **40** | **40k** | **3** | **102** | **10** | **1999.9** |
| 103 | 50 | 78k | 4 | 103 | 1 | 400 |
| 104 | 35 | 90k | 5 | 104 | 5 | 900 |
| 104 | 35 | 90k | 6 | 104 | 6 | 1080 |
| **105** | **58** | **90k** | **7** | **105** | **2** | **1900** |

Table 6.4: Joined Table of Customer and Order (2)

| **D1** | *D2* | *D3* | **D5** | *SUM(D6)* | *SUM(D7)* |
|---|---|---|---|---|---|
| 101 | 35 | 90k | 101 | 7 | 1918 |
| **102** | **40** | **40k** | **102** | **10** | **1999.9** |
| 103 | 50 | 78k | 103 | 1 | 400 |
| **104** | **35** | **90k** | **104** | **11** | **1980** |
| 105 | 58 | 90k | 105 | 2 | 1900 |

develop efficient methods to share the join processing with skyline computation is central to the skyline query optimization on multiple relations.

Motivated by the above observations, in this chapter, we systematically study the multi-relational skyline operator and make the following contributions:

- We study the problem of skyline computation with or without aggregate constraints in multiple tables when join operation works on them.
- We propose different approaches which aim to combine state-of-the-art join methods into skyline computation for any single join operation and extend to the case of multiple join operations.
- We present an effective method to estimate the size of skylines over joined relations, which can help to optimize join operations in the case of the skyline operator being involved.
- Our experiments on TPC-D benchmark demonstrate the efficiency and scalability of our proposed methods.

The rest of the chapter is organized as follows. Section 2 introduces the preliminaries, and Section 3 introduces the approaches of answering skylines over single join. Section 4

gives an method for estimating the skyline size in the joined table. Section 5 extends the methods in Section 3 to the case of multiple joins. We present experimental results in section 6 and conclude the chapter in section 7.

## 6.2 Preliminaries

Let denote a relation $X$ in an $n$-dimensional space $D = (D_1, \ldots, D_n)$, where dimensions $D_1, \ldots, D_n$ are in the domain of numbers. For any $p \succ q$, $q$ is called a dominated object, and denote the set of objects dominating $q$ as $Dom(q)$. We also have the following definition:

**Definition 6.2.1 (Join Attributes and Descriptive Attributes)** *Given two relations $A(d_1, \ldots, d_i, d_{i+1}, \ldots, d_{n_1})$ and $B(d'_1, \ldots, d'_j, d'_{j+1}, \ldots, d'_{n_2})$, where attributes $d_1, \ldots, d_i$ in $A$ and $d'_1, \ldots, d'_j$ in $B$ are called* **join attributes** *such that they are only used in the join operation*[4]*. Attributes $d_{i+1}, \ldots, d_{n_1}$ in $A$ and attributes $d'_{j+1}, \ldots, d'_{n_2}$ in $B$ are called* **descriptive attributes** *that participate in the evaluation of skylines.*

Corresponding to the scenarios in Example 13, we generalize two typical skyline problems as follows.

**Problem 1 (Skyline Join Problem)** *Given two relations $A(d_1, \ldots, d_i, d_{i+1}, \ldots, d_{n_1})$ and $B(d'_1, \ldots, d'_j, d'_{j+1}, \ldots, d'_{n_2})$, find skylines over the joined table $A \bowtie B$ excluding the join attributes.*

**Problem 2 (Skyline Join with Aggregate Constraint Problem)** *Given two relations $A(d_1, \ldots, d_i, d_{i+1}, \ldots, d_{n_1})$ and $B(d'_1, \ldots, d'_j, d'_{j+1}, \ldots, d'_{n_2})$, find skylines over the joined table $A \bowtie B$ where aggregate constraints on descriptive attributes in $B$ are applied on corresponding tuples in $A$. Here the aggregates include Max, Min, COUNT, SUM and AVG etc.*

We assume the join operator to be *non-reductive* [15] i.e. the join has the following properties:(1) the predicate is of the form $x = y$, where $x$ is an expression computable from one table and $y$ is an expression involving the other table; (2) it can be inferred that i) $x$ cannot be null, and ii) for each $x$ there must exist at least one $y$ such that $x = y$ holds.

[4] *It is often the case that join attributes refer to the primary key in $A$ and foreign key(s) in $B$. However, in general case a join attribute could be any attribute. For ease of analysis, we assume whenever an attribute participates in the join, it does not participate in skyline evaluation.*

In general, an SQL-like statement for expressing a skyline join query based on the SKYLINE OF clause [13] is given as follows:

**SELECT** $R_i.D_j|AGG(R_i.D_j)$
**FROM** $R_1, R_2, \ldots, R_m$
**WHERE** join_condition $(R_1, R_2, \ldots, R_m)$
**GROUP BY**...
**HAVING**...
**SKYLINE** OF [DISTINCT] $R_1.D'_{n_1}$ **[MIN|MAX|DIFF]**, ..., $R_m.D'_{n_m}$ **[MIN|MAX|DIFF]**
**ORDER BY**...

Here, (1) $R_i.D_j$ refers to any attribute in $R_i$, $R_i.D'_{n_i}$ refers to the descriptive attributes in $R_i$ and (2) $AGG \in \{MIN, MAX, SUM, COUNT, AVG\}$.

Table 6.5: Group-by D1 (CNum) in Customer

| **D1** | *D2* | *D3* | |
|---|---|---|---|
| 101 | 35 | 90k | **LS(S)** |
| 102 | 40 | 40k | **LS(N)** |
| 103 | 50 | 78k | **LS(N)** |
| 104 | 35 | 90k | **LS(S)** |
| 105 | 58 | 90k | **LS(N)** |

Table 6.6: Group-by D5 (CNum) in Order

| D4 | **D5** | *D6* | *D7* | |
|---|---|---|---|---|
| 1 | 101 | 1 | 274 | **LN(N)** |
| 2 | 101 | 6 | 1644 | **LS(N)** |
| 3 | 102 | 10 | 1999.9 | **LS(S)** |
| 4 | 103 | 1 | 400 | **LS(N)** |
| 5 | 104 | 5 | 900 | **LN(N)** |
| 6 | 104 | 6 | 1080 | **LS(N)** |
| 7 | 105 | 2 | 1900 | **LS(N)** |

In each table, we can group tuples according to the values of *join attributes* in ascending order. As shown in Table 6.5 and Table 6.6, relations *Customer* and *Order* are grouped by join attributes $D_1$ and $D_5$ respectively. Every tuple belongs to one of three cases: (1) **LS(S)** means this tuple is a *local skyline* in its group and also a *skyline in the whole table*, such as tuple (101, 35, 90k) in *Customer*. (2) Label **LS(N)** means this tuple is only a *local skyline* in the group but *not a skyline in the whole table*. For example, tuple (2, 101, 6, 1644) in *Order* is a local skyline but it is not a global skyline since it is dominated by another tuple (3, 102, 10, 1999.9). (3) Label **LN(N)** means this tuple is *locally not a skyline* in the group (and of course it is *not a skyline in the whole table*), such as the tuple (1, 101, 1 , 274) in *Order* since it is dominated by the tuple (2, 101, 6, 1644) in the same group. If we use symbol "$\oplus$" to denote the concatenate operator to combine two joinable tuples in $A$ and $B$ into a joined tuple, then each joined tuple in the Table $A \bowtie B$ has the following six cases:

$LS(S) \oplus LS(S)$, $LS(S) \oplus LS(N)$, $LS(S) \oplus LN(N)$, $LS(N) \oplus LN(N)$, $LN(N) \oplus LN(N)$ and $LS(N) \oplus LS(N)$. Note that the join operator is symmetric so that the order of its argument does not matter. We have the following properties:

**Lemma 6.2.2** *The joined tuple $LS(S) \oplus LS(S)$ or $LS(S) \oplus LS(N)$ is also a skyline in $A \bowtie B$. The joined tuple $LS(S) \oplus LN(N)$, $LS(N) \oplus LN(N)$, $LN(N) \oplus LN(N)$ cannot be a skyline in $A \bowtie B$.*

**Proof:** (1) The proof for the case of $LS(S) \oplus LS(S)$ is straightforward, to prove the case of $LS(S) \oplus LS(N)$. Let $a \in A$ be labeled $LS(S)$ in $A$ which means no other tuple can dominate $a$, and $b \in B$ labels $LS(N)$ which means no other tuple can dominate $b$ in the same group but there exists some tuple dominating $b$. If there exists a joined tuple in $A \bowtie B$ which can dominate $a \oplus b$, it must be in the form $a \oplus b'$, since no other tuple in $A$ can dominate $a$. Thus, we have $b' \in B$ and $b' \succ b$. In particular, $b'$ must have the same joined attribute value with $b$ in order to be joinable with $a$. That is, $b$ and $b'$ are in the same group, so $b$ cannot be labeled as $LS(N)$, which contradicts the assumption. (2) For the proof of the second part, let $a \in A$ be labeled as $LN(N)$, which means in the same group, there must exist at least another tuple $a'$ such that $a' \succ a$. We assume that $a \oplus b, b \in B$, is the skyline of $A \bowtie B$. Since $a'$ has the same value on the join attribute as $a$, $a' \oplus b$ is also in $A \bowtie B$. $a \succ a$ implies $a' \oplus b \succ a \oplus b$ which contradicts the assumption. Therefore, $a \oplus b$ is not in the skyline of $A \bowtie B$.

This lemma can help us easily identify the skyline if one tuple participating is a local skyline and the other tuple is a global skyline when they meet the join condition.

**Property 6.2.3** *The joined tuple $LS(N) \oplus LS(N)$ may be or may not be a skyline in $A \bowtie B$.*

In this case, whether the joined tuple $a \oplus b$ in $A \bowtie B$ is a skyline or not depends on the successful "matching" on the join attribute values between their dominators.

In this chapter, we mainly focus on the skyline computation over $A \bowtie B$ where a one-to-many relationship [77] exists between $A$ and $B$ such that the join operation is performed if $A$ has a primary key matching a foreign key in $B$. The case of the join operation over $A$ and $B$ with many-to-many relationship [77] can be solved by introducing a third table $C$ including attributes of $B$ and join attribute of $A$, then perform join between $A$ and $C$, and join between the join results and $B$.

## 6.3 Algorithms for Skylines over a Single Join

We present in this section different approaches of integrating the skyline computation into state-of-the-art join algorithms. Basically, motivated by the properties discussed in Section 2, we can apply a skyline algorithm to $A$ and $B$, then perform a selective join of $A$ and $B$. Without loss of generality, assuming the join operation between $A$ and $B$ corresponds to one-to-many relationship, attribute $d_i$ in $A$ and attribute $d'_j$ in $B$ are join attributes respectively. Denote the Boolean function of evaluating join condition between $p \in A$, $q \in B$ as $\theta(p, q)$.

### 6.3.1 A Naive Approach for Skyline Join

A naive approach for the skyline join operator works as follows: we first compute the join, then apply existing skyline algorithms on the joined relation to find the corresponding skyline objects.

As the join operation is assumed to be *non-reductive* [15], the number of attributes of the joined table is larger than that of each single table participating in join operation, so the cost of running skyline algorithm in the entire joined table is much more expensive than in any single table. If we consider the common case of the increase in both cardinality and dimensionality of the joined table, the cost of the naive approach would be even more expensive.

### 6.3.2 Integrating with Sort-Merge Join

#### Basic Approach

As illustrated in the previous section, the basic idea of this approach is that without computing skyline in the entire joined table, we can process the joined skyline only based on the property of being skyline or dominated for tuple $p \in A$ and $q \in B$, to quickly identify the skyline object for the joined tuple $p \oplus q$ during the join processing. Lemmas 6.2.2 can be exploited to efficiently determine skyline and non-skyline objects based on the labels of the input tables only.

The only difficult case is as described in Property 6.2.3 when two tuples participating in the join are both labeled as "LS(N)". The following lemma identify whether the joined tuple is a skyline object.

**Lemma 6.3.1** *Let $p \in A$ and $q \in B$ be joinable tuples labeled as "LS(N)", if $\theta(p', q') =$*

*$false$ for all $p' \in Dom(p), q' \in Dom(q)$, then $p \oplus q$ is a joined skyline object; otherwise $p \oplus q$ cannot be a joined skyline object.*

**Proof:** *If $\theta(p', q') = false$, then none of $p'$s dominator can match up $q'$s dominator in join attribute, so in the joined table, $p \oplus q$ cannot be dominated by any other joined tuple, thus it is a joined skyline object.*

We now introduce how to find joined skylines by integrating with the sort-merge join algorithm [77]. Essentially, according to the label of "$LS(S)$", "$LS(N)$" or "$LN(N)$" for tuple $p \in A$ and $q \in B$, we decide whether the joined tuple $p \oplus q$ is a skyline object in $A \bowtie B$ depending on whether they satisfy the properties illustrated in Lemma 6.2.2 and Lemma 6.3.1. The pseudo-code of the Sort-Merge based algorithm is shown as follows.

Our skyline computation strategy which consists of (1) pre-processing (lines 1-4) and (2) skyline identification in $A \bowtie B$ (lines 17-30) can be seamlessly embedded into the sort-merge join algorithm (lines 5-16 and lines 32-33). The above method generates complete skyline objects in $A \bowtie B$.

**Algorithm 6.3.1 Input:** *A, B, join attributes of $d_i \in A$ and $d'_j \in B$;*
**Output:** *Skyline in $A \bowtie B$*
**Method:**

1. *FindSkyline(A); //using existing skyline computation algorithms*
2. *Label(A); //label each tuple in A*
3. *FindSkyline(B); //using existing skyline computation algorithms*
4. *Label(B); //label each tuple in B*
5. $T_a$ = *first tuple in A;*
6. $T_b$ = *first tuple in B;*
7. $G_b$ = *first tuple in B;*
8. WHILE $T_a \neq eof$ *AND* $G_b \neq eof$ DO{
9.     WHILE $T_{a_i} < G_{b_j}$ DO;
10.         $T_a$ = *next tuple in A after* $T_a$;
11.     WHILE $T_{a_i} > G_{b_j}$ DO;
12.         $G_b$ = *next tuple in B after* $G_b$;
13.     $T_b = G_b$;
14.     WHILE $T_{a_i} == G_{b_j}$ DO{
15.         $T_b = G_b$;
16.         WHILE $T_{b_j} == T_{a_i}$ DO{
17.             IF $T_a$, $T_b$ *both labeled "LS(S)"* THEN

```
18.             Add skyline Ta ⊕ Tb to result;
19.         IF Ta, Tb each labeled "LS(S)", LS(N) THEN
20.             Add skyline Ta ⊕ Tb to result;
21.         IF one of Ta, Tb labeled "LN(N)" THEN
22.             Ta ⊕ Tb is not a skyline in A ⋈ B;
23.         IF Ta, Tb both labeled "LS(N)" THEN
24.             Flag = FALSE;
25.             IF there are p ∈ Dom(Ta) and q ∈ Dom(Tb)
26.             with θ(p, q) = TRUE THEN
27.                 Flag = TRUE;
28.             IF ¬Flag THEN
29.                 Add skyline Ta ⊕ Tb to result;
30.             ELSE Ta ⊕ Tb is not a skyline in A ⋈ B;
31.         Tb = next tuple in B after Tb;}
32.     Ta = next tuple in A after Ta;}
33. Gb = Tb;}
34. Output result as the skyline in A ⋈ B;
```

The non-trivial cost in this method is the *search and match of dominators*, so the optimization work is to reduce the cost in these two aspects. To quickly output the match of dominators ($Dom(p)$ and $Dom(q)$) (lines 25-30), we build linked list (as a temporary data structure) to maintain their dominators in ascending order of values of the join attribute from the skyline computation process. The cost used for checking the matching of any pair of dominators of $p, q$ labeled as "LS(N)" , is $O(|Dom(p)| + |Dom(q)|)$. For the efficient searching of dominators, different strategies will be introduced in the remaining sections.

The above algorithm SSMJ addresses our Problem 1, for Problem 2 which includes some aggregation operator can be solved with a slightly modified algorithm by first treating the tuples in the same group as a single tuple with the aggregated values, then applying the same procedure to identify skylines.

### Efficient Dominators Match Using R-trees

As discussed in previous section, in order to know whether $p \oplus q$ is a skyline object where $p, q$ are labeled as "LS(N)", we need to obtain $Dom(p)$ and $Dom(q)$. However, computing the dominators $Dom(p)/Dom(q)$ for each tuple $p$ with label "LS(N)" is not trivial when

the dataset is large. To overcome this problem, we propose an R-tree-based approach to facilitate the search of dominators.

Basically, the descriptive attributes of each tuple in each table is indexed in R-tree (as a typical multi-dimensional index structure, R-tree has been widely used). The dominators of any object can be easily obtained by range queries in R-tree, then the dominators are sorted with respect to values of the join attribute, and the match of dominators is tested. To facilitate the match, we can record the range of join attribute in each MBR node, and prune the unnecessary comparisons where join attribute value range does not overlap as much as possible.

### Comparing with Joined Skylines During Join Process

To eliminate the need for dominating test, in this section we propose a method for the computation of joined skylines based on comparing each $LS(N) \oplus LS(N)$ tuple efficiently with the joined skylines found so far in the join process. Here we still follow the strategy of sort-merge join, but adopt two modifications: (1) we discard tuples with label "LN(N)" since they have no contribution to the final skylines in the joined table. (2) we sort each table as follows: (a) the primary sorting for each table is by the order of $LS(S) < LS(N)$, so tuples with label "LS(S)" are placed before tuples with label "LS(N)" in each table, as shown in Figure 6.1; (b) within each group of "LS(S)" and "LS(N)" obtained by (a), the secondary sorting is performed by the join attribute value of each tuple. The reason of these modifications is that we can determine which tuples in the joined table are skylines as early as possible, and also can reduce the sizes of input tables as small as possible.

Table 6.7 and Table 6.8 show the modified tables *Customer'* and *Order'* after modifications (1) and (2) are made to tables *Customer* and *Order* in Table 6.5 and Table 6.6 respectively.

Table 6.7: Customer'

| **D1** | *D2* | D3 | |
|---|---|---|---|
| 101 | 35 | 90k | **LS(S)** |
| 104 | 35 | 90k | **LS(S)** |
| 102 | 40 | 40k | **LS(N)** |
| 103 | 50 | 78k | **LS(N)** |
| 105 | 58 | 90k | **LS(N)** |

Table 6.8: Order'

| D4 | **D5** | *D6* | *D7* | |
|---|---|---|---|---|
| 3 | 102 | 10 | 1999.9 | **LS(S)** |
| 2 | 101 | 6 | 1644 | **LS(N)** |
| 4 | 103 | 1 | 400 | **LS(N)** |
| 6 | 104 | 6 | 1080 | **LS(N)** |
| 7 | 105 | 2 | 1900 | **LS(N)** |

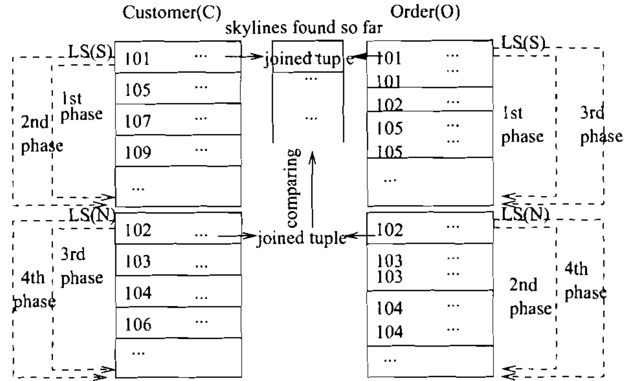



Figure 6.1: Comparing $a_i \oplus b_j$ with Skylines Found So Far in Joined Table

Now the sort-merge join consists of four phases. The first three phases are $LS(S) \oplus LS(S)$, $LS(S) \oplus LS(N)$ and $LS(N) \oplus LS(S)$, are indicated by those dashed lines in Figure 6.1.

The above tuples which can be simply combined if they match up, are skylines $S$ (labeled as "LS(S)") in the joined table, and they are sorted with entropy value in ascending order for quickly identify dominating relationship later. The fourth phase is $LS(N) \oplus LS(N)$ (as shown in Figure 6.1 for the dashed lines with number 4). If any "LS(N)" matches "LS(N)", compare with the joined tuple in $S$, if dominated, it cannot be a joined skyline; otherwise output to the temporary set $S'$. In the future, any new $LS(N) \oplus LS(N)$ will compare with $S$ and $S'$.

### 6.3.3 Integrating with Nested-Loop Join

Motivated by the idea of comparing with joined skylines during the join process in Section 6.3.2, we now propose a method integrating with the nested-loop join [77]. The intuition is that if the joined tuples obtained during the join process keep an "order" in some *descriptive attributes*, then if a new joined tuple is not dominated by joined skylines found so far, it is easily identified as a joined skyline.

Here we sort each join table in one of *descriptive attributes*, and meanwhile maintain the information about different labels as described in the previous section. Note that, those labels are obtained in the Group-By operation with respect to *join attributes* in each table. Table 6.9 and Table 6.10 show the same tuples as Table 6.5 and Table 6.6 but with the

Table 6.9: Sort Customer by $D_2$ (Age)

| **D1** | *D2* | D3 | |
|---|---|---|---|
| 101 | **35** | 90k | **LS(S)** |
| 104 | **35** | 90k | **LS(S)** |
| 102 | **40** | 40k | **LS(N)** |
| 103 | **50** | 78k | **LS(N)** |
| 105 | **58** | 90k | **LS(N)** |

Table 6.10: Sort Order by $D_6$ (Quantity)

| D4 | **D5** | *D6* | D7 | |
|---|---|---|---|---|
| 3 | 102 | **10** | 1999.9 | **LS(S)** |
| 2 | 101 | **6** | 1644 | **LS(N)** |
| 6 | 104 | 6 | 1080 | **LS(N)** |
| 5 | 104 | **5** | 900 | **LN(N)** |
| 7 | 105 | **2** | 1900 | **LS(N)** |
| 4 | 103 | **1** | 400 | **LS(N)** |
| 1 | 101 | **1** | 274 | **LN(N)** |

sorting order in $D2$ (*Age*) and $D6$ (*Quantity*) respectively. Note that Table 6.7 is sorted in ascending order while Table 6.8 is sorted in descending order due to the criterion of "the less the better" in attribute *Age* and "the more the better" in attribute *Quantity* in the query.

**Lemma 6.3.2** *Given relations $A$, $B$ sorted by one of descriptive attributes and each tuple in $A$ and $B$ is labeled as "LS(S)" "LS(N)" and "LN(N)" according to join attributes in each table as described in Section 2. $A$ joins $B$ in the nested loop approach. If tuple $a \in A$, $b \in B$ meet the join condition $\theta(a, b) == TRUE$ and both have a label "LS(N)", then $a \oplus b$ can only be dominated by $c \oplus d$ where $c \in A$, $d \in B$, $\theta(c, d) == TRUE$ and $c, d$ appear before $a, b$ respectively in the sorted tables.*

This lemma illustrates that *if any pair of joinable, "LS(N)" tuples is not dominated by any skyline found in joined table so far, it is a skyline in the joined table and need not compare with any joined tuple in the future since it will never be dominated by a joined tuple appearing after itself.* The pseudo-code of integrating the skyline processing with the nested-loop join algorithm is shown as follows.

**Algorithm 6.3.2**
**Input:** *$A$, $B$, join attributes of $d_i \in A$ and $d'_j \in B$;*
**Output:** *Skyline $A \bowtie B$*
**Method:**

1. *FindSkyline(A); //using existing skyline computation algorithms*
2. *Label(A); //label each tuple in A*

3. *FindSkyline(B); //using existing skyline computation algorithms*
4. *Label(B); //label each tuple in B*
5. FOR *each tuple* $a \in A$ DO
6.     FOR *each tuple* $b \in B$ DO
7.     IF $a_i == b_j$ THEN
8.         IF $T_a$, $T_b$ *both labeled "LS(S)"* THEN
9.             *Add skyline* $T_a \oplus T_b$ *to result;*
10.         IF $T_a$, $T_b$ *each labeled "LS(S)" and* $LS(N)$ THEN
11.             *Add skyline* $T_a \oplus T_b$ *to result;*
12.         IF $T_a$, $T_b$ *both labeled "LS(N)"* THEN
13.             **Compare** $T_a \oplus T_b$ **with each skyline in result**;
14.             IF *NO tuple in result dominates* $T_a \oplus T_b$ THEN
15.                 *Add skyline* $T_a \oplus T_b$ *to result;*
16.             ELSE $T_a \oplus T_b$ *is not a skyline in* $A \bowtie B$*;*
17. *Output result in* $A \bowtie B$*;*

The algorithm consists of (1) pre-processing (lines 1-4) and (2) skyline identification in $A \bowtie B$ (lines 8-18) can be seamlessly embedded into the nested-loop join algorithm (lines 5-7).

## 6.4 Cardinality Estimate of Joined Skylines

In order to provide the query optimizer with more efficient plans for evaluating skyline queries with joins, it is important to estimate the skyline size over any joined table. For example, the query optimizer can adjust the join order among multiple relations so that the tables with smaller estimate size of joined skyline cardinality will join before tables with larger estimate size etc.

### 6.4.1 Problem of a Naive Solution

One might think of estimating the number of skyline objects in a join table by first calculating the size of the join table, and then applying some classic skyline size estimation method, e.g., in [37]. Unfortunately, this naive solution does not work. Recall that the classic skyline size estimation method requires that first, all attributes are independent, second, all values in the same attribute are distinct. Both conditions are violated in the case of the multi-relational skyline operator since the value of the primary key determines the values

of the remaining attributes and one tuple in $A$ in general has several matching tuples in $B$ so that the attribute values of the $A$ tuple are repeated multiple times. Hence, we need to estimate the number of skyline objects directly from the original tables. Authors in [80] study the cardinality estimation of join $T_1 \bowtie T_2$ in a very special case that only attributes in $T_1$ participate in skyline evaluation. However, it is more meaningful to estimate the cardinality of skylines over joined table in general case when attributes in both $T_1$ and $T_2$ participate the skyline evaluation.

### 6.4.2 The Model

Let $A, B$ denote two tables consisting of $d_1$ and $d_2$ descriptive attributes, respectively. $A, B$ each has one join attribute. There are two basic assumptions: First, every data attribute of $A, B$ is drawn randomly from some probability distribution and is distinct. Second, all attributes (including both descriptive and join attributes) of $A, B$ are independent (note that this is only for the input tables but not for the output table).

The join attributes of $A$, $B$ both range on some discrete set $\Omega = \{\omega_1, \ldots \omega_t\}$. Note that $A \bowtie B$ consists of $d_1 + d_2$ descriptive attributes and one join attribute. Our goal is to estimate the number of skyline records in $A \bowtie B$ *solely* on the descriptive attributes. $\forall 1 \leq i \leq t$, let $A_i$ denote the set of records in $A$ whose value on the join attribute is $\omega_i$. We have $A = \bigcup_{i=1}^{t} A_i$ and $\forall 1 \leq i, j \leq t, i \neq j, A_i \bigcap A_j = \phi$. We define $B_i$ similarly. Let $a_i = |A_i|$ and $b_i = |B_i|$. $\forall u \in A$ (or $B$), we write $A_i$ (or $B_i$) $\succ u$, if there is some record in $A_i$ (or $B_i$) except for $u$ itself that dominates $u$.

### 6.4.3 The Size of the Join Table

Since every record in $A$ joins with each record in $B$ with the same value on the join attribute, $A \bowtie B = \bigcup_i A_i \bowtie B_i$ and $|A \bowtie B| = \sum_i a_i \cdot b_i$.

### 6.4.4 The Expected Number of Skyline Objects in the Join Table

To estimate the number of skyline objects, we fix an arbitrary record $r = x \bowtie y \in A_i \bowtie B_i$ where $x = \{x_1, \ldots x_{d_1}\} \in A_i, y = \{y_1, \ldots y_{d_2}\} \in B_i$. Next, we calculate the probability that $r$ is skyline (i.e., none of the records in $A \bowtie B$ except for $r$ dominates $r$).

Note that for $r$ to be a skyline object, none of the records in $A_i$ except for $x$ dominates $x$ *and* none of the records in $B_i$ except for $y$ dominates $y$ (see Lemma 6.3.1). Besides, for

any $j \neq i$, none of the records in $A_j$ dominates $x$, *or* none of the records in $B_j$ dominates $y$.

Integrating over $x$ and $y$, we get Pr[$r$ is skyline] =

$$\int_{-\infty}^{+\infty} \cdots \int_{-\infty}^{+\infty} \prod_{j=1, j \neq i}^{t} \{\Pr[\neg(A_j \succ x) \text{ or } \neg(B_j \succ y)]\} \cdot$$

$$\Pr[\neg(A_i \succ x) \text{ and } \neg(B_i \succ y)] \prod_{k=1}^{d_1} dF(x_k) \prod_{k=1}^{d_2} dF(y_k). \tag{6.1}$$

Unfortunately, it is difficult to solve the above integration directly. We shall prove the upper and lower bounds instead. Let $p(m, d)$ be the probability that a random $d$-dimensional record is skyline among $m$ random records. Note that $p(m, d) \approx \frac{(\log m)^{d-1}}{m(d-1)!}$, according to [14].

### 6.4.5 Upper Bound

To show the upper bound, we need the following lemma which can be proved using the majorization technique in [64].

**Lemma 6.4.1** $\forall 0 < p, q < 1, k \geq 1, (1-p)^k + (1-q)^k - (1-p)^k \cdot (1-q)^k \leq (1-pq)^k$.

By inclusion-exclusion and Lemma 6.4.1, we get $\forall j \neq i$, $\Pr[\neg(A_j \succ x) \text{ or } \neg(B_j \succ y)]$

$$\begin{aligned} &= \left(1 - \prod_{k=1}^{d_1} (1 - F(x_k))\right)^{a_j} + \left(1 - \prod_{k=1}^{d_2} (1 - F(y_k))\right)^{b_j} - \\ &\quad \left(1 - \prod_{k=1}^{d_1} (1 - F(x_k))\right)^{a_j} \cdot \left(1 - \prod_{k=1}^{d_2} (1 - F(y_k))\right)^{b_j} \\ &\leq \left(1 - \prod_{k=1}^{d_1} (1 - F(x_k)) \prod_{k=1}^{d_2} (1 - F(y_k))\right)^{\min\{a_j, b_j\}}, \end{aligned}$$

Let $s = \sum_{j=1}^{t} \min\{a_j, b_j\}$ and $s_{-i} = \sum_{j=1, j\neq i}^{t} \min\{a_j, b_j\}$. By Equation 6.1, Pr[$r$ is skyline]

$$
\begin{aligned}
&\leq \int_{-\infty}^{+\infty} \cdots \int_{-\infty}^{+\infty} \prod_{j=1, j\neq i}^{t} \left(1 - \prod_{k=1}^{d_1} (1 - F(x_k)) \prod_{k=1}^{d_2} (1 - F(y_k))\right)^{\min\{a_j, b_j\}} \\
&\quad \left(1 - \prod_{k=1}^{d_1} (1 - F(x_k))\right)^{a_i - 1} \left(1 - \prod_{k=1}^{d_2} (1 - F(y_k))\right)^{b_i - 1} \\
&\quad \prod_{k=1}^{d_1} dF(x_k) \prod_{k=1}^{d_2} dF(y_k) \\
&\leq \int_0^1 \cdots \int_0^1 \prod_{j=1, j\neq i}^{t} \left(1 - \prod_{k=1}^{d_1} x_k \prod_{k=1}^{d_2} y_k\right)^{\min\{a_j, b_j\}} \\
&\quad \left(1 - \prod_{k=1}^{d_1} x_k\right)^{a_i - 1} \left(1 - \prod_{k=1}^{d_2} y_k\right)^{b_i - 1} \prod_{k=1}^{d_1} dx_k \prod_{k=1}^{d_2} dy_k \\
&= \int_0^1 \cdots \int_0^1 \left(1 - \prod_{k=1}^{d_1} x_k \prod_{k=1}^{d_2} y_k\right)^{s_{-i}} \left(1 - \prod_{k=1}^{d_1} x_k\right)^{a_i - 1} \\
&\quad \left(1 - \prod_{k=1}^{d_2} y_k\right)^{b_i - 1} \prod_{k=1}^{d_1} dx_k \prod_{k=1}^{d_2} dy_k \\
&\leq \int_0^1 \cdots \int_0^1 \left(1 - \prod_{k=1}^{d_1} x_k \prod_{k=1}^{d_2} y_k\right)^{s-1} \prod_{k=1}^{d_1} dx_k \prod_{k=1}^{d_2} dy_k \\
&= p(s - 1, d_1 + d_2).
\end{aligned}
$$

Finally, since there are $|A \bowtie B|$ records in the join table, the expected number of skyline objects in $A \bowtie B$ is upper bounded by $|A \bowtie B| \cdot p(s - 1, d_1 + d_2)$.

### 6.4.6 Lower Bound

By Equation 6.1, Pr[$r$ is skyline]

$$
\begin{aligned}
&> \int_{-\infty}^{+\infty} \dots \int_{-\infty}^{+\infty} \Pr[\neg(B_i \succ y)] \cdot \Pr[\neg(A_i \succ x)] \cdot \\
&\quad \prod_{j=1, j \neq i}^{t} \{\Pr[\neg(A_j \succ x)\} \prod_{k=1}^{d_1} dF(x_k) \prod_{k=1}^{d_2} dF(y_k) \\
&= \int_{-\infty}^{+\infty} \dots \int_{-\infty}^{+\infty} \Pr[\neg(B_i \succ y)] \cdot \prod_{j=1}^{t} \{\Pr[\neg(A_j \succ x)\} \\
&\quad \prod_{k=1}^{d_1} dF(x_k) \prod_{k=1}^{d_2} dF(y_k) \\
&= \left( \int_{-\infty}^{+\infty} \dots \int_{-\infty}^{+\infty} \prod_{j=1}^{t} \{\Pr[\neg(A_j \succ x)\} \prod_{k=1}^{d_1} dF(x_k) \right) \cdot \\
&\quad \left( \int_{-\infty}^{+\infty} \dots \int_{-\infty}^{+\infty} \Pr[\neg(B_i \succ y)] \prod_{k=1}^{d_2} dF(y_k) \right) \\
&= p(|A|, d_1) p(b_i, d_2).
\end{aligned}
$$

Consequently, the number of skyline objects in $A \bowtie B$ is lowerbounded by $p(|A|, d_1) \cdot \sum_{i=1}^{t} a_i b_i p(b_i, d_2)$.

In summary, the number of joined skylines is between $p(|A|, d_1) \cdot \sum_{i=1}^{t} a_i b_i p(b_i, d_2)$ and $|A \bowtie B| \cdot p(s-1, d_1 + d_2)$.

## 6.5 Extending to Multiple Joins

The previous section presents the methods for querying skyline in the joined table where the participating tables for a single join have a one-to-many relationship. We can apply these methods for the many-to-many relationship by multiple joins.

**Example 14** *Given the table* Customer *and table* Order *described in Example 13, we have another table* Part (**PNum**, PQuantity, Price). *Consider the following queries.*

Question (1): *Who are those young customers with high account balance and having* **ONE** *order of part with high quantities in stock and and high amount of price?*

Question (2): *Who are those young customers with high account balance and high quantities of in-stock parts he/she ordered, with high* **TOTAL** *amount of price (over all orders of this customer)?*

Answer: *We can basically answer the above queries by performing similar SQL statements as Example 13 (we omit details due to the space limitation) over* $Customer \bowtie Order \bowtie Part$ *which first finds skyline over* $C = Customer \bowtie Order$*, then over* $C \bowtie Part$.

## 6.6 Experimental Evaluation

In this section, we report the results of our experimental evaluation.

**Experimental Design** All methods proposed in this chapter were implemented using Microsoft Visual C++ V6.0, including (1) the improved sort-merge join based skyline methods by (i) using R-tree MBRs (noted as **SMJS1**), and by (ii) comparing with joined skylines during the join process (noted as **SMJS2**); and (2) the block nested-loop join based skyline method (noted as **NLJS**). For the purpose of comparison, we also implemented the naive-based skyline algorithm (noted as **Naive**) including a sort-merge join process and a skyline computing method LESS in [39] which computes skylines over joined tables. We choose LESS due to its good average performance. Note that like the average complexity analysis in most of skyline computation methods, the analysis of LESS makes the assumption UI [5][39]. However, it is difficult to achieve the same performance in practice since each table has duplicate values, and the joined table have even more duplicate values in each attribute. Experiments were conducted on a PC with an Intel Pentium 4 1.6 GHz CPU, 512 M main memory and a 40 GB hard disk, running the Microsoft Windows XP operating system.

**Testing Datasets** Using the data generator provided by www.tpc.org, we generated several types of TPC-D benchmark tables: *Customer*[6] (each tuple has 44 bytes), *Order*[7] ((each tuple has 84 bytes)) and *Part*[8] (each tuple has 60 bytes). For each type of table, we generated data sets with different sizes (from $10,000$ to $1,000,000$ tuples).

[5]The properties of uniformly distributed, independence and distinct values are called uniform independence (UI)

[6]Customer relation has 3 descriptive attributes: account balance, salary, age)

[7]Order relation has 7 descriptive attributes: quantity, total price, priority, discount, ship cost, tax, delivery duration)

[8]Part relation has 5 descriptive attributes:quantity, price, cost, discount range, weight, size

Table 6.11: Cardinality of Different Datasets

| Dataset1 | C, 20k | O, 100x1k | P, 20x1k |
|---|---|---|---|
| Dataset2 | C, 50k | O, 200x1k | P, 50x1k |
| Dataset3 | C, 100k | O, 500x1k | P, 100x1k |
| Dataset4 | C, 200k | O, 1000x1k | P, 200x1k |

By choosing four different datasets as shown in Table 6.11, the physical sizes of *Customer*, *Order Part*, *Customer*⋈*Order* and *Customer*⋈*Order*⋈*Part* are listed in Figure 6.2(a) with respect to different cardinalities[9]. On the other hand, Figure 6.2(b) illustrates the number of skylines in *C*,*O* and *P*, *C*⋈*O* and *C*⋈*O*⋈*P* illustrated in Figure 6.2(b).

We observe that in both cases, the size differences are evident and become larger when the cardinality increases significantly, and when more relations participate the join operation. The experiments are conducted in different aspects as follows.

**Run Time** The time cost for **Naive** method includes applying the sort-merge join to obtain the joined table and then applying **LESS** to obtain skylines. **SMJS1**, **SMJS2** and **NLJS** also use **LESS** to find skylines in each table and labeling[10]. Additionally **SMJS2** moves "LS(S)" tuples ahead of other tuples. The time used in join process for **SMJS1**, **SMJS2** and **NLJS** shares with the time in computing joined skylines. The computation of joined skylines with aggregate constraints uses less time due to the smaller size of input tables after the aggregation.

The run time comparisons of different methods for computing joined skylines with/without aggregate constraints with respect to different sizes of the joined table *C*⋈*O* (with totally 10 descriptive attributes) are depicted in Figure 6.3(a) and Figure 6.3(b) respectively. Here, we typically test the "SUM" aggregate as mentioned in Example 13. In both cases, our proposed methods run much faster than **Naive** method. In particular, **SMJS2** finishes first, and **SMJS1** takes less time than **NLJS**.

To study the relationship between run time and the dimensionality of the joined table, we compare with the run time of different methods for computing joined skylines with/without aggregate constraints as shown in Figure 6.4(a) and Figure 6.4(b) respectively. The joined table is obtained from the joining of *O* of 500,000 records and *P* of 100,000 records. Each

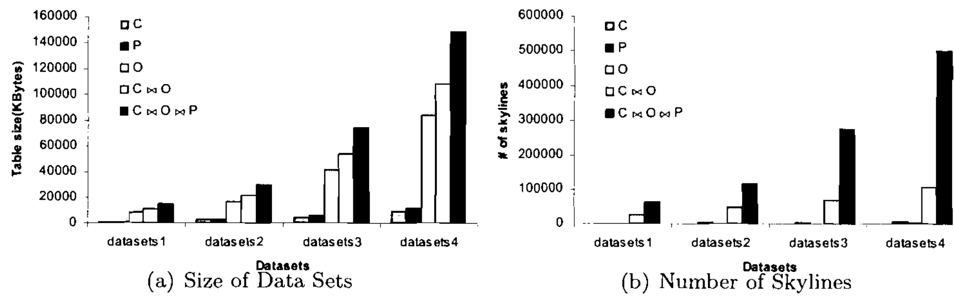
[9] In the legend of figures, *C*-Customer, *O*-Order, *P*-Part, *CjoinO*-C⋈ O, *C* join *O* join *P*-C ⋈ O⋈ P

[10] The **LESS** method can be easily adapted to label "LS(S)" as well as "LS(N)" and "LS(S)" in the order of join attribute value.

time we choose 2, 3, 4 and 5 dimensions per table respectively to participate join operation, thus the joined table has the dimensionality of 4, 6, 8 and 10 respectively. It shows the same ranks of run time as the test of run time w.r.t. different sizes of joined tables.

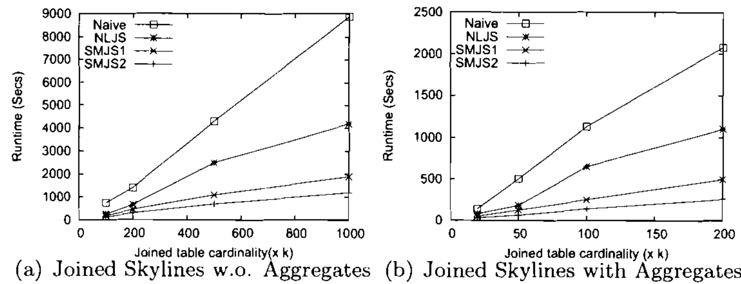
**Test of Cardinality Estimate** It is also interesting to compare the actual size of joined skylines with the expected size of the joined skylines by our proposed estimator (we choose the upper bound estimate). Figure 6.5(a) and Figure 6.5(b) give the results of our method and the classical estimator [37] in different sizes of joined tables (with totally 10 descriptive attributes) and different dimensionality (with 500,000 joined records) respectively, and show that our method can always provide better estimate results.

**Summary** From the experimental evaluations, we conclude that our proposed skyline join algorithms are efficient and scalable to large databases, and also our joined skyline estimator can provide a good estimate of the correct size.



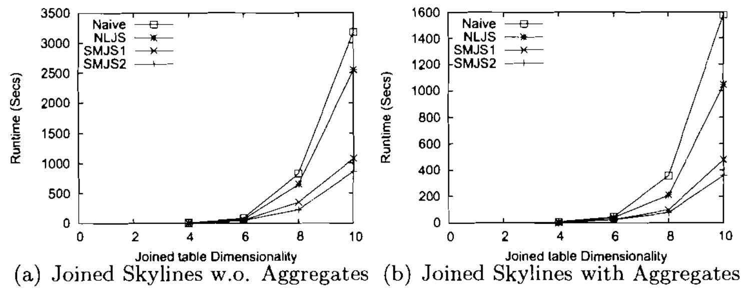

(a) Size of Data Sets (b) Number of Skylines

Figure 6.2: The Size of Data Sets and #of Skylines



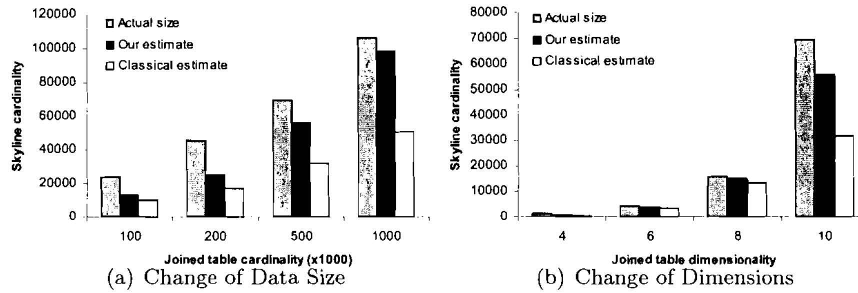

(a) Joined Skylines w.o. Aggregates (b) Joined Skylines with Aggregates

Figure 6.3: Runtime vs Cardinality




(a) Joined Skylines w.o. Aggregates (b) Joined Skylines with Aggregates

Figure 6.4: Runtime vs Dimensions




(a) Change of Data Size (b) Change of Dimensions

Figure 6.5: #of Skylines Estimate

## 6.7 Conclusion

In this chapter, we study the skyline operator over multi-relational tables, and propose solutions by incorporating state-of-the-art join methods into skyline computation [50]. We also showed that the current methods for cardinality estimations are not adequate for the scenario of multiple tables involved with join operations, and provided a method to theoretically estimate the size of joined skylines. The experiments on TPC-D datasets demonstrate the efficiency and scalability of the proposed methods. We believe that this research does not only meaningfully extend the skyline operator to the multi-relational database systems, but also indicate the interesting topics such as joined skylines in the case of updated data and other types of aggregates.

# Chapter 7

# Summary and Conclusions

In this chapter, we summarize our results and present our conclusions. In the following chapter, we suggest areas for future work, including some topics for which we are now working.

## 7.1 Summary of the Thesis

The notion of skyline operator has been demonstrated its importance in many applications such as multi-criteria decision making, data mining and visualization and user-preference queries. Most previous studies focus on the improvement in computational efficiency of finding skyline objects in a full space. However, there still exists the following crucial questions: (1) Why and in which subspaces is (or is not) an object in the skyline? (2) How to reasonably approximate the skyline objects, i.e., mining "approximate skyline objects"? (3) Can the notion of skyline operator facilitate other database operators? (4) How to efficiently compute skyline on multiple relations? In this thesis, we propose a class of methods for answering these questions the and make the following contributions.

- We answered the question about semantics of skyline objects by introducing the novel notions of skyline groups and decisive subspaces. We proposed the problem of subspace skyline analysis and computation. On the subspace skyline analysis side, a novel roll-up and drill-down analysis of skylines in various subspaces was introduced. On the subspace skyline computation side, an efficient algorithm *Skyey* was developed. Our performance study using both real and synthetic data sets was conducted to verify the

meaningfulness and the efficiency of our approach. The experimental results strongly suggest that the semantics of skyline objects and subspace skyline analysis are highly meaningful in practice, and algorithm *Skyey* is efficient and scalable.

- We answered the question of the approximate skyline objects by proposing a novel notion of *thick skyline* based on the distance constraint of a skyline object from its nearest neighbors. The task of mining thick skyline is to recommend skyline objects as well as their nearest neighbors within $\varepsilon$-distance. We also develop Sampling-and-Pruning algorithm, Indexing-and-Estimating algorithm and Microcluster-based algorithm to find such thick skylines in large databases. Our experimental evaluation demonstrates the efficiency and effectiveness of our algorithms. We believe the notion of thick skyline and mining methods not only extends the skyline operator in database query, but also provides interesting patterns for data mining tasks.

- We answered the third question listed above by introducing materializing layered skylines for ranked queries processing. Such a novel approach is based on the observation that for any score function (weights) the top-$k$ ranked answers must be contained in the first $k$ skyline layers. These skyline layers can be efficiently determined and are stored in existing multi-dimensional index structures that are enhanced by special data structures representing the different skyline layers. For this purpose, we propose indexing layered skylines and indexing shell-grid microclusters for top-$k$ ranked query, and present methods for sweeping the hyperplane of the score function over the indexed objects. Our methods can be easily adapted to the existing multi-dimensional index structures. The experimental evaluation demonstrates that our methods clearly outperform state-of-the-art methods of rank-aware query processing in terms of efficiency and, as far as approximate methods are concerned, accuracy.

- We answered the question of how to achieve skyline query optimization on multiple relations by developing efficient methods to share the join processing with skyline computation. We also theoretically estimate the size of the joined skylines to extend the query optimizer's cost model to accommodate skyline operator over joined tables. The experiments not only demonstrate that the methods proposed are efficient, but also show the promising applicability of extending skyline operator to other typical database operators such as join and aggregates.

## 7.2 Conclusion

In conclusion, the analytic and experimental results presented in this thesis have shown the reason of objects being skylines in different subspaces, the degree to which of a skyline object being approximated by a non-skyline object, the usage of skyline objects by speeding up typical database operations such as ranked-aware queries, and the extension to the multi-relational skyline operator.

Hopefully, this thesis has convinced the reader that skyline objects are meaningful semantically, and can provide useful support for data mining and database operations.

# Chapter 8

# Ongoing and Future Work

Although the goal of this thesis has been met, there still remain several aspects that are worthy of consideration and exploration in many related problems, extensions and applications. Some of them are listed here.

## 8.1 Answering Subspace Skyline Queries by Materializing Signatures

In general, the skyline queries in databases can be categorized into two types:

- **Subspace Skyline Query *SS-query*:**
  1. Given a subspace $s$, find all objects that are skyline objects in $s$;
  2. Given a subspace $s$, find all objects that are skyline objects in all the subspaces of $s$;
- **Dominating Subspace Query *DS-query*:**
  1. Given an object $o$, find whether $o$ is a skyline object in any subspace; if so, list all such subspaces;
  2. Given an object $o$, find skyline objects that dominate $o$ in any subspace $s$.

Based on the properties in Chapter 3, the skyline in every subspace can be derived by the signatures of skyline groups, and the size of signatures is usually smaller than size of subspace skyline objects in every subspace. So we can compress the skyline objects of

different subspaces into a set of *signatures of skyline groups* and materialize these signatures for the skyline queries in arbitrarily subspace. Based on the different requirements on the time and space used in pre-processing, we can devise different materializing schema: (1) *indexing signatures of every subspace*, and build inverse index on the decisive subspaces occurred in each signature of the skyline groups; (2) *indexing signatures in the full space*, each node indexed by a decisive subspace stores the corresponding subspace skyline groups in the full space; and (3) *hybrid indexing method* which aims to find a tradeoff between (1) and (2). Accordingly, different search techniques can be developed to search the indexes and retrieves the subspace skyline objects.

## 8.2 Mining Subspace Thick Skyline Objects

A natural extension of the thick skyline operator is applied to the case of any subspace. For example, users would be like to see the thick skyline objects only for the attributes of "price" (*price*) and "distance to the beach" (*distance*) among many attributes of the hotel database. To find thick skyline objects in any subspace, we can integrating the $\varepsilon$ neighborhood search into the *Skyey* algorithm. We can take an alternative approach by indexing signatures in every subspace, then locate the corresponding entry in the index for the input subspace and find its skyline group, finally execute the $\varepsilon$ neighborhood search for output thick skyline objects.

## 8.3 Mining Interesting Non-Skyline Objects

Many existing methods focus on the notion of skyline and its properties, however, it lacks the work in *mining interesting non-skyline objects.* For example, even neither Hyatt hotel nor Hilton hotel is a skyline hotel among all the hotels in New York city with respect to attributes of *price* and *distance.* People often want to know how "good" these non skyline hotels are? Which hotel is better, Haytt or Hilton? Moreover, since the set of skyline objects often occupies a small portion of the database, it is very meaningful to measure the importance (factor) of those non-skyline objects. Typically, each non-skyline object has its dominating degree (to which this object dominates other objects) and dominated degree (to which this object is dominated by other objects). Based on the gradient feature of multi-layered skylines, the dominating degree and the dominated degree can be meaningfully

quantified. Therefore, by combining both dominating degree and dominated degree of each non-skyline object, we can measure each object including non-skyline objects in a reasonable way and mining the most interesting non-skyline objects in large databases.

## 8.4 Mining Microeconomic Dominating Neighbors

Current methods on skyline objects have only considered so-called min/max attributes like price and quality which a user wants to minimize or maximize. However, objects can also have spatial attributes like $x$, $y$ coordinates which can be used to represent relevant constraints on the query results. We introduce novel skyline query types taking into account not only min/max attributes but also spatial attributes and the relationships between these different attribute types [90]. Such kind of queries supports a micro-economic approach to decision making, considering both the quality of query objects and the cost of solutions. We investigate two alternative approaches for efficient query processing, a symmetrical one based on off-the-shelf index structures, and an asymmetrical one based on index structures with special purpose extensions. Our experimental evaluation using a real dataset and various synthetic datasets demonstrates that the new query types are indeed meaningful and the proposed algorithms are efficient and scalable.

# Bibliography


[1] R. Agrawal and R. Srikant. Fast algorithm for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, pages 487–499, 1994.

[2] R. Agrawal and E. L. Wimmers. A framework for expressing and combining preferences. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 297–306, 2000.

[3] Rodin B. *Calculus and Analytic Geometry.* Prentice-Hall, Inc, 1970.

[4] W.-T. Balke and U. Guntzer. Multi-objective query processing for database systems. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB)*, pages 936–947, 2004.

[5] W.-T. Balke, U. Guntzer, and J. X. Zheng. Efficient distributed skylining for web information systems. In *Proceedings of 9th International Conference on Extending Database Technology (EBDT)*, pages 256–273, 2004.

[6] W.-T. Balke, J. X. Zheng, and U. Guntzer. Approaching the efficient frontier: Cooperative database retrieval using high-dimensional skylines. In *Proceedings of the 10th International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 410–421, 2005.

[7] 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 International Conference on Management of Data (SIGMOD)*, pages 322–331, 1990.

[8] 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, 1990.

[9] J. L. Bentley, K. L. Clarkson, and D. B. Levine. Fast linear expected-time alogrithms for computing maxima and convex hulls. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 179–187, 1990.

[10] 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 (JACM)*, 25(4):536–543, October 1978.

[11] S. Berchtold, C. Bhm, and H.P. Kriegel. The pyramid-technique: Towards breaking the curse of dimensionality. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1998.

[12] S. Berchtold, D. A. Keim, and H. P. Kriegel. The x-tree : An index structure for high-dimensional data. In *Proceedings of 22th International Conference on Very Large Data Base (VLDB)*, pages 28–39, 1996.

[13] S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proceedings of the 17th International Conference on Data Engineering (ICDE)*, pages 421–430, 2001.

[14] C. Buchta. On the average number of maxima in a set of vectors. *Inf. Process. Lett.*, 33(2):63–65, 1989.

[15] M. Carey and D.Kossman. On saying "enough already!" in sql. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages "219–230", 1997.

[16] C. Y. Chan, P. K. Eng, and K. L. Tan. Efficient processing of skyline queries with partially-ordered domains. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, pages 190–191, 2005.

[17] 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, 2005.

[18] C. Y. Chan, H. V. Jagadish, K. L. Tan, K. H. Tung, and Z. J. Zhang. Finding k-dominant skylines in high dimensional space. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages "503–514", 2006.

[19] C. Y. Chan, H. V. Jagadish, K. L. Tan, K. H. Tung, and Z. J. Zhang. On high dimensional skylines. In *Proceedings of the 10th International Conference on Extending Database Technology (EDBT)*, pages 478–495, 2006.

[20] K. C. Chang and S. W. Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 346–357, 2002.

[21] Y. C. Chang, L. D. Bergman, V. Castelli, C. S. Li, M. L. Lo, and J. R. Smith. The onion technique: Indexing for linear optimization queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 391–402, 2000.

[22] S. Chaudhuri. Data mining and database systems: Where is the intersection. *IEEE Data Enginerring Bulletin*, 21(1):4–8, 1998.

[23] S. Chaudhuri, N. Dalvi, and K. Raghav. Robust cardinality and cost estimation for skyline operator. In *Proceedings of the 22nd IEEE International Conference on Data Engineering (ICDE)*, 2006.

[24] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic ranking of database query results. In *Proceedings of 30th International Conference on Very Large Data Bases (VLDB)*, pages 888–899, 2004.

[25] S. Chaudhuri and L. Gravano. Evaluating top-k selection queries. In *Proceedings of 25th International Conference on Very Large Data Bases (VLDB)*, pages 397–410, 1999.

[26] B. Chazelle. An optimal convex hull algorithm in any fixed dimension. *Discrete Computational Geometry*, 10.

[27] J. Chomicki, P. Godfrey, J. Gryz, and D. M. Liang. Skyline with presorting. In *Proceedings of the 19th International Conference on Data Engineering (ICDE)*, pages 717–816, 2003.

[28] J. Chomicki, P. Godfrey, J. Gryz, and D. M. Liang. Skyline with presorting: Theory and optimizations. In *Intelligent Information Processing and Web Mining, Proceedings of the International Intelligent Information Systems:IIPWM'05*, pages 595–604, 2005.

[29] T. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, second edition.* The MIT Press, 55 Hayward Street Cambridge, MA 02142-1315, September 2001.

[30] P. K. Eng, B. C. Ooi, and K. L. Tan. Indexing for progressive skyline computation. *Data Knowl. Eng.*, 46(2):169–201, 2003.

[31] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 226–231, 1996.

[32] R. Fagin. Fuzzy queries in multimedia database systems. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS)*, pages 1–10, 1998.

[33] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS)*, pages 102–113, 2001.

[34] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations.* The MIT Press, Springer, 1996.

[35] F. Geerts, H. Mannila, and E.Terzi. Relational link-based ranking. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB)*, pages 552–563, 2004.

[36] U. Gntzer, W.-T.Balke, and W. Kieling. Optimizing multi-feature queries for image databases. In *Proceedings of 26th International Conference on Very Large Data Bases (VLDB)*, pages 419–428, 2000.

[37] P. Godfrey. Cardinality estimation of skyline queries: Harmonics in data. Technique Report CS-2002-03, York University, Computer Science Department, October 2002.

[38] P. Godfrey. Skyline cardinality for relational processing. In *Proceedings of Foundations of Information and Knowledge Systems, Third International Symposium (FoIKS)*, pages 78–97, 2004.

[39] P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, pages 229–240, 2005.

[40] S. Guha, N. Koudas, A. Marathe, and D. Srivastava. Merging the results of approximate match operations. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, pages 636–647, 2004.

[41] 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, 1984.

[42] A. Hinneburg and D. A. Keim. Optimal grid-clustering: Towards breaking the curse of dimensionality in high-dimensional clustering. In *Proceedings of 25th International Conference on Very Large Data Bases (VLDB)*, pages 506–517, 1999.

[43] G. R. Hjaltason and H.Samet. Distance browsing in spatial databases. *ACM Transanction of Database System*, 24(2):265–318, 1999.

[44] V. Hristidis, N. Koudas, and Y.Papakonstantinou. Prefer:a system for the efficient execution of multi-parametric ranked queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 259–270, 2001.

[45] V. Hristidis and Y. Papakonstantinou. Algorithms and applicatins for answering ranked queries using ranked views. *The VLDB Journal*, 13(1):49–70, 2004.

[46] Z. Huang, C.S. Jensen, H. Lu, and B. C. Ooi. Skyline queries against mobile lightweight devices in manets. In *Proceedings of the 22nd IEEE International Conference on Data Engineering (ICDE)*, 2006.

[47] I. F. Ilyas, R. Shah, W.G.Aref, J.S.Vitter, and A.K.Elmagramid. Rank-aware query optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 203–214, 2004.

[48] I. F. Ilyas, W.G.Aref, and A.K.Elmagramid. Supporting top-k join queries in relational databases. In *Proceedings of 29th International Conference on Very Large Data Bases (VLDB)*, pages 754–765, 2003.

[49] W. Jin, M. Ester, and J. W. Han. Efficient processing of ranked queries with sweeping selection. In *Proceedings of the 9th European Conference on Principles and Practice of Knowledge Discovery in Databases(PKDD)*, pages 527–535, 2005.

[50] W. Jin, M. Ester, Z. J. Hu, and J. W. Han. The multi-relational skyline operator. In *Proceedings of the 23rd International Conference on Data Engineering (ICDE)*, 2007.

[51] W. Jin, J. W. Han, and M. Ester. Mining thick skylines over large databases. In *Proceedings of the 8th European Conference on Principles and Practice of Knowledge Discovery in Databases(PKDD)*, pages 255–266, 2004.

[52] W. Jin, Anthony K. H. Tung, and Jiawei Han. Mining top-n local outliers in large databases. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge Discovery and Data Mining (KDD)*, pages 293–298, 2001.

[53] T. Johnson, L. V. S. Lakshmanan, and R. T. Ng. The 3w model and algebra for unified data mining. In *Proceedings of 26th International Conference on Very Large Data Bases (VLDB)*, pages 21–32, 2000.

[54] N. Katayama and S. Satoh. The sr-tree: An index structure for high-dimensional nearest neighbor queries. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 369–380, 1997.

[55] E. M. Knorr and R. T. Ng. Finding intensional knowledge of distance-based outliers. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*, pages 211–222, 1996.

[56] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: an online algorithm for skyline queries. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB)*, pages 275–286, 2002.

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

[58] L. Lakshmanan, J. Pei, and J. W. Han. Quotient cube: How to summarize the semantics of a data cube. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB)*, pages 778–789, 2002.

[59] C. P. Li, B. C. Ooi, K. H. Tung, and S. Wang. Dada: A data cube for dominant relationship analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages "659–670", 2006.

[60] X. M. Lin, Y.D.Yuan, W. Wang, and H.J. Lu. Stabbing the sky:efficient skyline computation over sliding windows. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, pages 502–513, 2005.

[61] H. X. Lu, Y. Luo, and X. M. Lin. An optimal divide-conquer algorithm for 2d skyline queries. In *Proceedings of Advances in Databases and Information Systems, 7th East European Conference (ADBIS)*, pages 46–60, 2003.

[62] Y. Luo, H. X. Lu, and X.M. Lin. A scalable and i/o optimal skyline processing algorithm. In *Proceedings of Advances in Web-Age Information Management: 5th International Conference (WAIM)*, pages 218–228, 2004.

[63] A. Marian, N. Bruno, and L. Gravano. Evaluating top-k queries over web-accessible database. *ACM Transactions on Database Systems*, 29(2):1–44, June 2004.

[64] A. W. Marshall and I. Olkin. *Inequalities: Theory of Majorization and Its Applications.* New York : Academic Press.

[65] J. Matousek. Computing dominances in $e^n$. *Information Processing Letters*, 38(5):277–278, 1991.

[66] M. Morse, J. M. Patel, and W. Grosky. Efficient continuous skyline computation.

[67] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, The Pitt Building, Trumpington Street, Cambridge, United Kingdom, October 1995.

[68] A. Natsev, Y. C. Chang, J. R. Smith, C.S. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. In *Proceedings of 27th International Conference on Very Large Data Bases (VLDB)*, pages 281–290, 2001.

[69] S. Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *Proceedings of the 15th International Conference on Data Engineering (ICDE)*, pages 22–29, 1999.

[70] F. Nielsen. Output-sensitive peeling of convex and maximal layers. *Information Processing Letters*, 59(5):255–259, 1996.

[71] M. Ortega, Y. Rui, K. Chakrabarti, K.Porkaew, S. Mehrotra, and T.S. Huang. Supporting ranked boolean similarity queries in mars. *IEEE Transactions on Knowledge Data Engineering*, 10(6):905–925, 1998.

[72] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algoroithm for skyline queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 467–478, 2003.

[73] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems, 2005.

[74] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *Proceedings of 7th International Conference on Database Theory (ICDT)*, pages 398–416, 1999.

[75] J. Pei, W. Jin, M. Ester, and Y.F. Tao. Catching the best views of skyline: a semantic approach. In *Proceedings of 31st International Conference on Very Large Data Bases (VLDB)*, pages 253–264, 2005.

[76] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, August 1993.

[77] R. Ramakrishnan and J. Gehrke. *Database Management Systems.* 3rd Edition, McGraw-Hill.

[78] N. Roussopoulos, S.Kelley, and F.Vincent. Nearest neighbor queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 71–79, 1995.

[79] R. Rymon. Search through systematic set enumeration. In *Proceedings of the International Conference on Principle of Knowledge Representation and Reasoning (KR)*, pages 539–550, 1992.

[80] N. Dalvi S. Chaudhuri and R. Kaushik. Robust cardinality and cost estimation for skyline operator. In *Proceedings of the 22ndt International Conference on Data Engineering (ICDE)*, pages 64–74, 2006.

[81] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the Thirteenth International Conference on Very Large Data Bases (VLDB)*, pages 507–518, 1987.

[82] P. K. C. Singitham, M. Mahabhashyam, and P. Raghavan. Efficiency-quality tradeoffs for vector score aggregation. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB)*, pages 624–635, 2004.

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

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

[85] K. L. Tan, P. K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *Proceedings of 27th International Conference on Very Large Data Bases (VLDB)*, pages 301–310, 2001.

[86] Y. Tao and D. Papadias. Maintaining sliding window skylines on data streams. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 18(3):377–391, 2006.

[87] Y. Tao, X. K. Xiao, and J. Pei. Subsky: Efficient computation of skylines in subspaces. In *Proceedings of the 22nd IEEE International Conference on Data Engineering (ICDE)*, 2006.

[88] M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB)*, pages 648–659, 2004.

[89] P. Tsaparas, T. Palpanas, Y. Kotidis, N. Koudas, and D. Srivastava. Ranked join indices. In *Proceedings of the 19th International Conference on Data Engineering (ICDE)*, pages 277–290, 2003.

[90] Anthony K. H. Tung, W. Jin, and M. Ester. On dominating your neighborhood profitably. In *Submit to Conference*, 2006.

[91] D. A. White and R. Jain. Similarity indexiug with the ss-tree. In *Proceedings of the 12th International Conference on Data Engineering (ICDE)*, pages 516–523, 1996.

[92] E. L. Wimmers, L. M. Haas, M. T.Roth, and C.Braendli. Using fagin's algorithm for merging ranked results in multimedia middleware. In *Proceedings of the Fourth IFCIS International Conference on Cooperative Information Systems (CoopIS)*, pages 267–278, 1999.

[93] P. Wu, C. Zhang, Y. Feng, B. Y. Zhao, D. Agrawal, and A. E. Abbadi. Parallelizing skyline queries for scalable distribution. In *Proceedings of the 10th International Conference on Extending Database Technology (EDBT)*, pages 112–130, 2006.

[94] T. Xia and D. H. Zhang. Refreshing the sky: The compressed skycube with efficient support for frequent updates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2006.

[95] Y. D. Yuan, X. M. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang. Efficient computation of the skyline cube. In *Proceedings of 31st International Conference on Very Large Data Bases (VLDB)*, pages 241–252, 2005.

[96] T. Zhang, R. Ramakrishnan, and M. Livny. Birch:an efficient data clustering method for very large databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 103–114, 1996.

[97] Z. J. Zhang, X. Guo, H. Lu, K. H. Tung, and N. Wang. Discovering strong skyline points in high dimensional spaces. In *Proceedings of the 2005 ACM CIKM International Conference on Information and Knowledge Management (CIKM)*, pages 247-248, 2005.