Mining and Managing Neighbor-Based Patterns in Data Streams

by

Di Yang

A Dissertation

Submitted to the Faculty of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Ph.D. in

Computer Science

by

____

January 6th, 2012

APPROVED:

Professor Elke A. Rundensteiner, Advisor ______________

Professor Matthew O. Ward, Co-Advisor ______________

Professor Daniel J. Dougherty, Committee Member ______________

Professor Evimaria Terzi , External Committee Member ______________

Professor Craig Wills, Head of Department ______________

Abstract

The current data-intensive world is continuously producing huge volumes of live streaming data through various kinds of electronic devices, such as sensor networks, smart phones, GPS and RFID systems. To understand these data sources and thus better leverage them to serve human society, the demands for mining complex patterns from these high speed data streams have significantly increased in a broad range of application domains, such as financial analysis, social network analysis, credit fraud detection, and moving object monitoring.

In this dissertation, we present a framework to tackle the mining and management problem for the family of neighbor-based patterns in data streams, which covers a broad range of popular pattern types, including clusters, outliers, k-nearest neighbors and others.

First, we study the problem of efficiently executing single neighbor-based pattern mining queries. We propose a general optimization principle for incremental pattern maintenance in data streams, called "Predicted Views". This general optimization principle exploits the "predictability" of sliding window semantics to eliminate both the computational and storage effort needed for handling the expiration of stream objects, which usually constitutes the most expensive operations for incremental pattern maintenance.

Second, the problem of multiple query optimization for neighbor-based pattern mining queries is analyzed, which aims to efficiently execute a heavy workload of neighbor-based pattern mining queries using shared execution strategies. We present an integrated pattern maintenance strategy to represent and incrementally maintain the patterns identified by queries with different query parameters within a single compact structure. Our solution realizes fully shared execution of multiple queries with arbitrary parameter settings.

Third, the problem of summarization and matching for neighbor-based patterns is examined. To solve this problem, we first propose a summarization format for each pattern type. Then, we present computation strategies, which efficiently summarize the neighbor-based patterns either during or after the online pattern extraction process. Lastly, to compare patterns extracted on different time horizon of the stream, we design an efficient matching mechanism to identify similar patterns in the stream history for any given pattern of interest to an analyst.

Our comprehensive experimental studies, using both synthetic as well as real data from domains of stock trades and moving object monitoring, demonstrate superiority of our proposed strategies over alternate methods in both effectiveness and efficiency.

Publications

Publications Contributing to this Dissertation

Publications Related to Part I

Part I of this dissertation discusses single query optimization for neighbor-based pattern mining queries in streaming environments.

- 1. Di Yang, Elke A. Rundensteiner and Matthew O. Ward, "Neighbor-Based Pattern Detection for Windows Over Streaming Data". EDBT, 2009, pages 517-528.

  Relationship to this dissertation: This work presents the single query execution strategies for density-based cluster mining queries in streams. Chapters 5 and 6 in Part I of this dissertation are based on this work.

- 2. Di Yang, Avani Shastri, Elke A. Rundensteiner and Matthew O. Ward, "An optimal strategy for monitoring top-k queries in streaming windows", EDBT, 2011, pages 57-68.

  Relationship to this dissertation: This work presents the single query execution strategies for kNN queries in streams. Chapter 7 in Part I of this dissertation is based on this work.

- 3. Di Yang, Elke A. Rundensteiner and Matthew O. Ward, "Mining Neighbor-Based Patterns from Streaming Data". WPI Technical Report, 2008.

  Relationship to this dissertation: This work conducts an in-depth performance analysis for alternative neighbor-based pattern mining algorithms, including cost models, cost analysis and more experimental studies. Chapter 8 in Part I of this dissertation is based on this work.

Publications Related to Part II

Part II of this dissertation discusses multiple query optimization strategies for neighbor-based pattern mining query workloads in streaming environments.

- 4. Di Yang, Elke A. Rundensteiner, Matthew O. Ward, "A Shared Execution Strategy for Multiple Pattern Mining Requests over Streaming Data". VLDB, 2009, pages 874-885.

  Relationship to this dissertation: This work presents the multiple query execution strategies for density-based cluster mining queries in streams. The key content in Part II of this dissertation about multiple query execution strategies for density-based cluster mining queries is based on this work.

- 5. Avani Shastri, Di Yang, Elke A. Rundensteiner and Matthew O. Ward, "MTopS: Scalable Processing of Continuous Top-K Multi-Query Workloads", CIKM 2011,1107-1116.

  Relationship to this dissertation: This work presents the multiple query execution strategies for density-based cluster mining queries in streams. The content in Part II of this dissertation about multiple query execution strategies for kNN mining queries shows the key ideas presented in this work.

- 6. Di Yang, Elke A. Rundensteiner and Matthew O. Ward, "Multiple Query Optimization for Neighbor-Based Pattern Mining Requests over Data Streams. TODS 2011. Accepted.

  Relationship to this dissertation: This work concludes several general optimization principles for multiple neighbor-based pattern mining query workloads in streaming environments. Part II of this dissertation has the same structure as this work does.

- 7. Di Yang, Zhenyu Guo, Zaixian Xie, Elke A. Rundensteiner and Matthew O. Ward, "Interactive Visual Exploration of Neighbor-Based Patters in Data Stream", SIGMOD demonstration paper, 2010, pages 1151-1154.

  Relationship to this dissertation: This work demonstrates our prototype system composed of a computational backend integrating the mining techniques proposed in the Part I and Part II of this dissertation, and a visual frontend, allowing human users to interact with the stream mining process.

Publications Related to Part III

Part III of this dissertation discusses summarization and matching of neighbor-based patterns in streaming environments.

- 8. Di Yang, Elke A. Rundensteiner and Matthew O. Ward, "Summarization and Matching of Density-Based Clusters in Steaming Environments", PVLDB vol.5(2) pages 121-132 2011.

  Relationship to this dissertation: This work presents the summarization and matching for density-based clusters in streaming environments. The major component of Part III of this dissertation about summarization and matching for density-based clusters in streaming environments is based on this work.

- 9. Di Yang, Kaiyu Zhao, Hanyuan Lu, Elke A. Rundensteiner and Matthew O. Ward, "ViStream T: An Integrated Platform for Mining Complex Patterns in the Past, the Present and the Future of Streams WPI Technical Report, 2011.

  Relationship to this dissertation: This work demonstrates our prototype system composed by a computational backend integrating all the major mining techniques proposed in this whole dissertation, and a visual frontend, allowing human users to interact with the stream mining process.

Publications Related to Future Work

- 10: Di Yang, Zhenyu Guo, Elke A. Rundensteiner and Matthew O. Ward, "CLUES: A Unified Framework Supporting Interactive Exploration of Density-Based Clusters in Streams", CIKM, pages 815-824, 2011.

  Relationship to this dissertation: This work proposes evolution model and evolution tracking methods for density-based clusters in streaming environments This is an initial effort into our future work of tracking the evolution of more types of neighbor-based patterns in streams.

Other Publications

My other publications were mainly published during my master program at WPI and related to my Master Thesis: "Analysis Guided Visual Exploration of Multivariate Data".

- 11. Di Yang, Elke A. Rundensteiner, Matthew O. Ward, "Nugget Discovery in Visual Exploration Environments by Query Consolidation", CIKM 2007, pages 603-612.
- 12. Di Yang, Elke A. Rundensteiner, Matthew O. Ward, "Analysis Guided Visual Exploration to Multivariate Data", VAST, October 2007, pages 83-90.
- 13. Di Yang, Z. Xie, Elke. A. Rundensteiner and Matthew O. Ward, "Managing Discoveries in The Visual Analytics Process", ACM SIGKDD Explorations special issue on visual analytics, Volume 9, Issue 2, pages 22-30, 2007.
- 14. Elke Rundensteiner, Matthew Ward, Zaixian Xie, Qingguang Cui, Charudatta Wad, Di Yang, Shiping Huang, "$XmdvTool^Q$: Quality-Aware Interactive Data Exploration", SIGMOD 2007, demonstration paper, pages 1109-1112, 2007.

- 15. Matthew O. Ward, Zaixian Xie, Di Yang and Elke A. Rundensteiner. “Quality-Aware Visual Data Analysis” Computational Statistics. Volume 26, Number 4, 567-584.

Acknowledgments

At this moment of reaching the greatest achievement of my life so far, I need to first give my deepest gratitude to my parents, namely my dad Yang Zhiming and my mom Wang Caiyun, and my wife Pu Di. Without their selfless support and continuous trust, finishing this dissertation is simply impossible.

I would like to sincerely thank my advisors Professor Elke A. Rundensteiner and Professor Matthew O. Ward, and my committee members Professor Daniel J. Dougherty and Professor Evimaria Terzi. It is their excellent inspiration and patient guidance, which has guided me through my Ph.D career.

I also would like to thank all my colleagues in XMDV and DSRG labs at WPI, including Zhu Yali, Wei Mingzhu, Ding Luping, Wang Song, Wang Ling, Venkatesh Raghavan, Xie Zaixian, Cui Qingguang, Abhishek Mukherji, Guo Zhenyu, Wang Di, Liu Mo, Zhao Kaiyu, Karen Works, Lin Xika, Lei Chuan, Cao Lei, Lu Hanyuan, Qi Yingmei, Zhang Dazhi and Medhabi Ray, for all the help that they have given me, and all the wonderful time that we have spent together.

Last but not the least, I thank every member of my family besides my parents, including my grandparents Wang Zhizhen, Yu Qinglian, Yang Yulong, Wu Yanmei, my anties Wang Caiwen, Wang Caixia, Wang Cailin, Wan Caihong, Yang Ying, Yang Suqin, and all my cousins Zhang Yujia, Lang Shuang, Ma Lele, Zhang Jing, Yin Jinglei and Peng Kaifeng, for always taking me as their pride.

My Master and Ph.D programs at WPI from 2005 to 2012, including the accomplishment of this dissertation, are supported by NSF, under a series of research grants, including CCF 0811510, IIS 0812027 and IIS 1018443.

Contents

1 Introduction **1**
1.1 Motivation 1
1.1.1 Neighbor-Based Pattern Mining 3
1.1.2 Efficient Pattern Extraction Query Execution 4
1.1.3 Multiple Pattern Extraction Query Optimization . . . 5
1.1.4 Pattern Summarization and Matching 7
1.2 State-Of-the-Art 9
1.2.1 Mining Data Streams 9
1.2.2 Multiple Mining Query Optimization 11
1.2.3 Summarization and Matching for Mined Patterns . . 12
1.3 Proposed Solutions 13
1.3.1 Solutions for Pattern Extraction Query Execution . . 13
1.3.2 Solutions for Multiple Query Optimization 16
1.3.3 Solutions for Pattern Summarization and Matching . 20
1.4 Where our Proposed Techniques May Not Work 23
1.5 Research Impact and Open Research Questions 25
1.6 Dissertation Organization 27

2 Preliminaries **28**
2.1 General Notion of Neighbor-Based Patterns 28
2.2 Definitions for Specific Neighbor-Based Patterns 30
2.2.1 Density-Based Cluster 30
2.2.2 Distance-Based Outlier 31
2.2.3 Top-k Nearest Neighbors (kNN 33
2.3 Sliding Window Semantics 33
2.4 Where Streaming Neighbor-Based Pattern Mining Lies in Data Mining Family 36

I Efficient Processing of Single Neighbor-Based Pattern Extraction Queries 40

3 Naive vs. Incremental Approach 41
3.1 Naive Solution–Recompute from Scratch at Each Window . 41
3.2 Why Incremental Computation 43
3.3 Basic Incremental Computation Strategies 44
3.4 Limitations of Basic Incremental Strategies 47

4 Predicted View Technique 50
4.1 Performance Bottleneck – Handle Expiration 50
4.2 Predicted View 51

5 Proposed Algorithms for Density-Based Clusters 54
5.1 Abstract Neighborship Maintenance – Abstract-C 54
5.2 Exact+abstracted *neighborship* Based Solution (Extra-N) . . . 57
5.2.1 What Abstract-C Suffers From : "Amnesia" 59
5.2.2 Enhanced Abstracted *Neighborship* 59
5.2.3 Challenges for Maintaining Cluster Memberships . . 61
5.2.4 View Prediction for Cluster Membership Maintenance 62
5.2.5 A Stepping-Stone Algorithm: Abstract-M 64
5.2.6 Proposed Solution: Extra-N 66

6 Proposed Algorithms for Distance-Based Outliers 73

7 Proposed Algorithms for kNN 75
7.1 Equivalence to Top-k Problem 75
7.2 Theoretical Foundations 76
7.3 An Initial Approach: PreTopk 80
7.4 Proposed Solution: MinTopk 83
7.4.1 Properties of Predicted Top-k Results 83
7.4.2 Solution: Integrated View Maintenance 84
7.4.3 Optimal Integration Strategy based on Continuous Top-k Career 86
7.4.4 Super-Top-K List Maintenance 88
7.4.5 Final Move Towards Optimality 91

8 Cost Analysis for Proposed Algorithms 96
8.1 Cost Analysis for Density-Based Cluster and Distance-Based Outlier Detection 96
8.1.1 Cost Models 97

8.1.2 Performance Analysis 103
8.2 Cost Analysis for kNN Detection Algorithms 107
8.2.1 Cost Analysis for PreTopk Algorithm 107
8.2.2 Cost Analysis for MinTopk Algorithm 109

9 Experimental Study 115
9.1 Real and Synthetic Streaming Datasets 115
9.1.1 Real Streaming Datasets 115
9.1.2 Synthetic Streaming Datasets 116
9.2 Experimental Studies for Density-Based Cluster Extraction . 117
9.2.1 Experimental Platform 117
9.2.2 Experimental Methodologies 117
9.2.3 Overall Evaluation 118
9.2.4 Abstract-M vs. Extra-N 122
9.2.5 Scalability Analysis 124
9.2.6 Evaluation with Real Data 126
9.3 Experimental Studies for Distance-Based Outlier Extraction 127
9.3.1 Experimental Platform 127
9.3.2 Experimental Methodologies 128
9.3.3 Alternative Methods 128
9.3.4 Performance Evaluation 130
9.4 Experimental Studies for kNN (top-k) Extraction 132
9.4.1 Experimental Platform 132
9.4.2 Alternative Methods 132
9.4.3 Experimental Methodologies 133
9.4.4 Evaluation for Different k Cases 133
9.4.5 Evaluation for Different win Cases 135
9.4.6 Evaluation for Different $slide$ Cases 136
9.4.7 Evaluation for Non-Uniform Arrival Rate Cases . . . 138

10 Related Work for Part I 141

II Multiple Neighbor-Based Pattern Extraction Query Optimization 146

11 Problem Definition 147

12 A Preliminary Sharing Effort: Share Range Query Searches 148

13 Sharing Among Queries with Arbitrary Pattern Parameters 151
13.1 "Containment" among Neighbor-Based Pattern Sets 152
13.1.1 "Growth Property" 152
13.1.2 Hierarchical Pattern Representation 154
13.1.3 Incremental Representation for Other Pattern Types . 156
13.2 Integrated Maintenance for Multiple Density-Based Clustering Queries 157
13.3 Integrated Maintenance for Multiple Distance-Based Outlier Detection Queries 168
13.4 Integrated Maintenance for Multiple kNN Queries 171

14 Sharing Among Queries with Arbitrary Window Parameters 175
14.1 Same win, Arbitrary $slide$ Case. 176
14.2 Same $slide$, Arbitrary win Case 179
14.3 Arbitrary $slide$, Arbitrary win Case 180

15 Putting IT All Together: The General Case 183

16 Experimental Study 192
16.1 Experimental Platform 192
16.2 Real and Synthetic Streaming Datasets 192
16.3 Alternative Algorithms 193
16.4 Experimental Methodologies 194
16.5 Evaluation for One-Arbitrary-Parameter Cases 194
16.5.1 Arbitrary θ^{cnt} case 195
16.5.2 Arbitrary θ^{range} case 196
16.5.3 Arbitrary win case 198
16.5.4 Arbitrary $slide$ case 198
16.6 Evaluation for Two-Arbitrary-Parameter Cases 199
16.6.1 Arbitrary Pattern Parameters 200
16.6.2 Arbitrary Window Parameters 200
16.7 General Case: Four Arbitrary Parameters. 202
16.8 Evaluation for Scalability 204
16.9 Conclusion for Experimental Study of Density-Based Cluster Detection Algorithms 206
16.10 Performance Analysis for Distance-Based Outlier and kNN Detection Algorithms 207

17 Related Work for Part II 211

III Summarization and Matching of Neighbor-Based Patterns 214

18 Supported Queries and System Overview 216
18.1 Continuous Pattern Extraction Queries 216
18.2 Pattern Matching Queries 217
18.3 System Overview 217

19 Pattern Summarization 219
19.1 Summarization for Density-Based Clusters 219
19.1.1 Features of Density-Based Clusters 219
19.1.2 Skeletal Point Summarization 221
19.1.3 Proposed Solution: Skeletal Grid Summarization . . 224
19.2 Summarization for Distance-Based Outliers 230
19.3 Summarization for kNN 231

20 Pattern Extractor 232
20.1 Extracting and Summarizing Density-Based Clusters 232
20.1.1 A Two Stage Strategy and Its Limitations 232
20.1.2 Incremental Computation and Challenges 234
20.1.3 "Lifespan" Analysis 235
20.1.4 Proposed C-SGS Algorithm 239
20.2 Extracting and Summarizing Distance-Based Outliers 244
20.3 Extracting and Summarizing kNN 245

21 Pattern Archiver 246
21.1 Pattern Compression 246
21.1.1 Cluster Summarization in Multi-Resolutions. 247
21.1.2 Distance-Based Outlier Summarization in Multi-Resolutions 248
21.1.3 kNN Summarization in Multi-Resolutions. 249
21.2 Budget- and Accuracy-Aware Resolution Selection. 249
21.3 Selective Pattern Archiving 251

22 Pattern Storage and Match 252
22.1 Storage and Matching for Density-Based Clusters 252
22.1.1 Cluster Organization in Pattern Base 252
22.1.2 Cluster Matching Process 253
22.2 Storage and Matching for Distance-Based Outliers and kNN 256

23 Experimental Study 259
23.1 Experimental Platform 259
23.2 Real and Synthetic Streaming Datasets 259
23.3 Alternative Summarization Formats 259
23.4 Performance of Cluster Extraction and Summarization . . . 260
23.5 Effectiveness for Cluster Matching Queries 268
23.6 Efficiency of Cluster Matching Queries 272
23.7 Evaluation for SGS at Multiple Resolutions 275

24 Related Work for Part III 281

25 Conclusions of This Dissertation 284

26 Future Work 288
26.1 Efficient Pattern Extraction for Other Complex Pattern Types 288
26.2 Pattern Evolution Model and Evolution Tracking 289
26.3 Dynamic Stream Mining Queries 291

List of Figures

2.1 An Example of A Density-Based Cluster 31
2.2 An Example of 3 Distance-Based Oultiers 32
2.3 An Example of $k = 4$ Nearest Neighbors 34
2.4 Template for a Density-Based Cluster Detection Query for Sliding Windows over a Data Steam 35
2.5 Template for a Distance-Based Outlier Detection Query for Sliding Windows over a Data Stream 35
2.6 Template for a kNN Detection Query for Sliding Windows over a Data Stream 35

3.1 Pseudo-Code for Exact-N Algorithm (Part 1) 46
3.2 Pseudo-Code for Exact-N Algorithm (Part 2) 46

4.1 Predicted views of four consecutive windows at W_0 53
4.2 Updated predicted views of four windows at W_1 53

5.1 Pseudo-Code for Abstract-C Part 1 58
5.2 Pseudo-Code for Abstract-C Part 2 58
5.3 "Predicted Views" of 4 Successive Windows at W_0 65
5.4 Updated "Predicted Views" of 4 Successive Windows at W_1 65
5.5 The H-Marks of the Data Points at W_0 68
5.6 Psuedo Code of Extra-N Algorithm 71

7.1 (Predicted) Top-k Results Four Consecutive Windows at time of W_0 (slide size = 4 objects) 77
7.2 Updated Predicted Top-k Results of Four Consecutive Windows at time of W_1 (slide size = 4 objects) 81
7.3 Update Process of Predicted Top-k sets From Time of W_0 to W_1 81
7.4 Independent Top-k Result Sets vs. *Super-top-k* Structure using Complete and Summarized Window Marks 86

7.5 Update Process of *super-top-k list* From W_0 to W_1 90
7.6 Proposed Solution: MinTopk Algorithm 95

9.1 Comparison on CPU Performances of Five Algorithms . . . 119
9.2 Comparison on Memory Performances of Five Algorithms . 120
9.3 Comparison on CPU Time of Abstract-M and MPS in $N^{-}_{(p_i)} = 5\%$ cases 123
9.4 Comparison on Memory Usage of Abstract-M and MPS in $N^{-}_{(p_i)} = 5\%$ cases 123
9.5 Comparison of CPU Scalability on Base (Window) Size . . . 125
9.6 Comparison of Memory Scalability on Base (Window) Size . 125
9.7 Comparison of CPU Scalability on Dimensionality 126
9.8 Comparison on CPU Time with GMTI data 126
9.9 Comparison on Memory Usage with GMTI data 126
9.10 Comparison on CPU Time with STT data 127
9.11 Comparison on Memory Usage with STT data 127
9.12 Comparison on CPU Time for Count-Based Window Scenario 131
9.13 Comparison on Memory Usage for Count-Based Window Scenario 131
9.14 Comparison on CPU Time for Time-Based Window Scenario 131
9.15 Comparison on Memory Usage for Time-Based Window Scenario 131
9.16 CPU time used by three algorithms with different k values . 134
9.17 Memory space used by three algorithms with different k values 134
9.18 CPU times for varying window sizes 136
9.19 Memory space for varying window sizes 136
9.20 CPU times for varying slide sizes 137
9.21 Memory space for varying slide sizes 137
9.22 Response time for processing each window given non-uniform arrival rate. 139

13.1 Cluster Set 1 containing 3 clusters 153
13.2 Cluster Set 2 containing 3 clusters, which is a growth of Cluster Set1 153
13.3 Independent Cluster Membership Storage for Cluster Sets 1 and 2 155
13.4 Hierarchical Cluster Membership Storage for Cluster Sets 1 and 2 155
13.5 Cluster sets identified by three different queries 159

13.6 $IntView\_\theta^{cnt}$: Integrated Representation for density-based clusters identified by three different queries 160
13.7 Distance-Based Outliers Identified by Q_1 and Q_2 172
13.8 Independent vs. Integrated Representation for Distance-Based Outliers Identified by Q_1 and Q_2 172

14.1 Predicted Views Maintained By Three Queries Q1, Q2 and Q3 Independently versus Those Maintained By a Single Meta Query 182

15.1 $IntView$: Integrated Representation for Predicted Views Identified by 3 Queries in 5 Predicted Windows 185
15.2 *Chandi*: Proposed Algorithm for Multiple Density-Based Clustering Queries (Part 1) 186
15.3 *Chandi*: Proposed Algorithm for Multiple Density-Based Clustering Queries (Part 2) 187
15.4 *SDOD*: Proposed Algorithm for Processing Multiple Distance-Based Outlier Detection Queries 188
15.5 *SkNN*: : Proposed Algorithm for Processing Multiple kNN Queries 189

16.1 CPU Time used by Five Competitors in Arbitrary θ^{cnt} Cases 196
16.2 Memory Space used by Five Competitors in Arbitrary θ^{cnt} Cases 196
16.3 CPU Time used by Five Competitors in Arbitrary θ^{range} Cases 197
16.4 Memory Space used by Five Competitors in Arbitrary θ^{range} Cases 197
16.5 CPU Time used by Five Competitors in Arbitrary win Cases 198
16.6 Memory Space used by Five Competitors in Arbitrary win Cases 198
16.7 CPU Time Used by Five Competitors in Arbitrary $slide$ Cases 199
16.8 Memory Space used by Five Competitors in Arbitrary $slide$ Cases 199
16.9 CPU Time used by Five Competitors in Arbitrary Pattern Parameter Cases 201
16.10 Memory Space used by Five Competitors in Arbitrary Pattern Parameter Cases 201
16.11 CPU Time used by Five Competitors in Arbitrary Window Parameter Cases 201

16.12 Memory Space used by Five Competitors Arbitrary Window Parameter Cases . . . 201
16.13 Detailed Comparison on CPU Time Consumption of Five Algorithms . . . 203
16.14 CPU Time used by Five Competitors in Logarithmic Scale . 205
16.15 Memory Space used by Five Competitors in Logarithmic Scale 205

18.1 Continuous Pattern Extraction Query Returning Full *(f)* and Summarized *(s)* Representations of Neighbor-Based Patterns 216
18.2 Pattern Matching Query finding Patterns Similar to the To-Be-Matched Cluster Based on Cluster Summarization 217
18.3 System Overview . . . 218

19.1 Example of full representation, basic SGS and compressed SGS of a 2D cluster . . . 228

20.1 Examples of updating cell status. $\theta^c = 4$, grey circle=*edge point*, black circle=*core point*, number on each object= number of windows the object can survive. . . . 242

23.1 CPU Time and Memory Comparison for Generating Alternative Summarizations on STT Stream . . . 264
23.2 CPU Time and Memory Comparison for Generating Alternative Summarizations on GMTI Stream . . . 264
23.3 CPU Time Comparison Using Time-Based Window (STT Stream) 265
23.4 CPU Time Comparison using Time-Based Window (GMTI Stream) . . . 267
23.5 Similar Rating by Users for Matched Clusters found by Alternative Summarization Methods . . . 272
23.6 CPU Time Comparison for Cluster Matching Queries using Alternative Cluster Summarization Methods . . . 275
23.7 Memory Comparison for Cluster Matching Queries using Alternative Cluster Summarization Methods (in Logarithmic measure) . . . 275
23.8 CPU Time Comparison for Cluster Matching Queries using Alternative Cluster Summarization Methods . . . 278
23.9 Memory Comparison for Cluster Matching Queries using Alternative Cluster Summarization Methods . . . 278
23.10 Similar Ratings by Users For Matched Clusters using Multiple Resolutions . . . 279

List of Tables

8.1 Symbols Used In The Cost Models. 98
8.2 CPU Cost of Individual Operations. 98
8.3 CPU Costs of Alternative Algorithms at Four Stages 100
8.4 Memory Costs of Individual Data Structures. 102
8.5 Memory Costs of Each Algorithm. 104
8.6 Cost Analysis of Each Algorithm (↓=small, ⇓=very small, ↑=large, we use () if impact is minor). 104

Chapter 1

Introduction

1.1 Motivation

As the ability of hardware to collect and transmit large amounts of live data continues to advance [SPP$^+$06, CKW$^+$12, DSJ$^+$12, CLW12], such as the development of RFID systems, smart phones, GPS systems and sensor networks, the need for processing streaming data in information infrastructures has become significant. Similar to the case of static data on which data mining techniques have been applied to find useful information for decades [DHZS02, GRS98, NH94, AGGR98, KN98a], our modern applications now post strong demands on analyzing and thus drawing value from streaming data resources. For example, to make proper investment decisions, financial analysts may want to continuously monitor the trade pattern changes in the stock market by mining the stock transaction streams. As another example, to analyze traffic patterns, such as the congestion areas in the daily traffic and their changes over time, the traffic analysts may need to be not only continuously updated with the patterns mined from the vehicle position streams, but also to have the capabilities of comparing and contrasting the patterns mined at different time points.

Along with the rapidly increasing volume and speed of streaming data, the difficulties for analyzing and understanding them are also correspondingly mounting. Traditional data mining techniques, which focus on analyzing large volumes of static data collections in an offline fashion, are no longer sufficient to handle the needs of mining such highly dynamic streaming data sources. This is because the dynamic characteristics of both streaming data and stream analysis tasks impose many new challenges for data mining systems. The crucial ones: 1) The data sources are no longer statically collected but continuously arriving in high speed. 2) The data sources may be infinite and thus impossible to be all stored in the disk. 3) The mining process to the data sources may no longer be an one-time analysis tasks rather may consist of continuous assessment of patterns across a long time duration. 4) The mining process may need to be highly efficient in order to return mining results in real-time. 5) Stream mining systems may have very limited computational, storage or even power resources, as they may be sensors or other micro-systems that are remotely located. To overcome these challenges, new mining infrastructures and techniques need to be designed to catch up with the high speed of the data streams and also provide long-term analysis support for live streams.

1.1.1 Neighbor-Based Pattern Mining

In my dissertation, I focus on both real-time and long-term analysis of **Neighbor-Based Patterns** in streaming data. The neighbor-based pattern family is an important class of complex patterns covering a broad range of popular pattern types, including density-based cluster detection [EKSX96, GM06, CT07, CEQZ06], distance-based outlier detection [KN98b, AF07], top-k nearest neighbors search (kNN) [MP07, YOTJ01] and reverse top-k nearest neighbor search (R-kNN) [AKK$^+$09, ABK$^+$06]. Neighbor-based pattern mining queries share the important property that their target patterns are defined based on the "neighbor relationships" (links) among objects. Such requirement for identifying neighbors, namely similar or closely related objects, is essential for many pattern mining tasks. This is because the similarity (distance) is an important interrelationship among objects and thus constitutes a key evidence from which analysts can draw conclusions about the data. The class of the neighbor-based pattern mining requests is defined in Chapter 2.

Although the problem of efficiently mining neighbor-based patterns in static environments has been well studied [KN98b, KN99, LS09, EKSX96, AKK$^+$09], the problem of mining and managing them in highly dynamic streaming environments remains unsolved. Due to the new challenges brought by streaming environments as listed earlier, those traditional neighbor-based pattern mining strategies designed for static environments are not sufficient to handle high-speed input streams. To study the problem of neighbor-based pattern mining and management in streaming environments, my dissertation investigates three different aspects of it, namely **efficient neighbor-based pattern extraction**, **multiple query optimization for neighbor-based pattern mining queries** and **summarization and matching for neighbor-based patterns**. I discuss the motivation for each of these three topics below.

1.1.2 Efficient Pattern Extraction Query Execution

Efficient real-time pattern extraction constitutes the foundation for neighbor-based pattern analysis. Clearly, only when a stream processing system is able to extract the patterns within the time requirement of the applications, the subsequent pattern analysis processes can become meaningful.

Many applications providing monitoring services over streaming data require this capability of real-time pattern detection. For example, to monitor main trends as well as the abnormal phenomena arising in the stock market, a financial analyst may want to be kept updated about major clusters as well as the outliers existing in the latest stock transactions. The major clusters formed by similar transactions in the stream may reveal key trading patterns, such as which stocks are intensively being traded in which time periods, currently exist in the market. In contrast, the abnormal transactions, such as individual transactions with unusually high transaction volumes or transaction prices, may indicate some special trading behaviors arising in the market. As another example, to understand the major threats of an enemy's air force, a battlefield commander needs to be continuously aware of the "clusters" formed by enemy warcraft based on the objects' most recent positions reported from satellites or ground stations.

In contrast, the "outliers" in our own solider groups may be especially vulnerable units who need immediate help.

However, the previous research effort for stream data processing mainly focus on on processing traditional SPJ queries [BW01, ABB$^+$03, VNB03] and simple aggregation queries [GKS01, GKMS01]. Little research effort has been made toward efficiently extracting neighbor-based patterns from data streams (See Chapter 1.2.1 for detailed state-of-the-art analysis). Thus, it is the first topic that I studied in my dissertation.

1.1.3 Multiple Pattern Extraction Query Optimization

Parameterized Queries. Complex pattern detection queries are usually parameterized, because pattern detection processes are driven by the domain knowledge of the analysts and the specific analysis tasks. A neighbor-based pattern mining request over sliding windows typically has two sets of input parameters, namely a set of pattern-specific parameters and a set of window-specific parameters. Using the density-based clustering as example, it has two pattern-specific parameters: a range threshold θ^{range} and a count threshold θ^{cnt}, which together define the minimum density that a group objects has to reach to qualify for a cluster. A density-based clustering query over sliding window also has two window-specific parameters: window size win and slide size $slide$, which represent how large is the query window and how often the query window slides respectively.

Why Multiple Queries. Given the prevalence of parameterized pattern mining queries, stream processing systems often need to handle a large number of such queries. This is caused for two major reasons. First, it is well known that in many applications a popular data stream is monitored by a large number of analysts [WRGB06, ZKOS05, HFAE03, LMT$^+$05, AW04]. For example, the stock transaction stream from NYSE is monitored by thousands of financial analysts every day. In my case, due to the specific domain knowledge and analytical tasks of different analysts, the analysts may submit the same types of neighbor-based pattern mining queries but with different parameter settings. For example, while many analysts are monitoring the same pattern type, say outliers, in the NYSE stock transaction stream, they may have their own customized interpretation about the pattern mining parameter settings. In particular, some of them may have very strict definitions of what constitutes an outlier in the stream. They may require the system to report only very abnormal transactions (outliers), while others may be interested in all abnormal transaction behaviors and thus request much more frequent updates to the outlier report.

Second, determining a priori the most appropriate parameter settings is a difficult problem for almost all data mining tasks, especially when faced with an unknown input stream or an unpredictable fluctuating input stream. In static environments, this problem is usually tackled by conducting pre-analysis of the static datasets or repeatedly trying different parameter settings until satisfactory results have been obtained. In streaming environments, the nonrepeatability of streaming data requires analysts to supply the most appropriate input parameters early on. Otherwise, they may permanently lose the opportunity to accurately discover the patterns in the portion of the stream just gone by. Therefore, even a single analyst may submit multiple queries of the same type but with different parameter settings, when she is not sure which parameter setting is the best. An ideal stream processing system should be able to accommodate such multiple query workloads covering many, if not all, major parameter settings of a parameterized query. Note that given the number of parameters required by streaming neighbor-based pattern mining queries, even allowing a very limited number of optional settings on each parameter, say four or five, can easily end up with hundreds of parameter combinations, namely hundreds of different queries.

The general problem of multiple query optimization has gained much research attention in the database community. However, similar to single query processing, the main focus of previous work on shared execution strategies for multiple queries in streaming environments has also been on SPJ and aggregation queries [HFAE03, KFHJ04, WRGB06] (see Chapter 1.2.2 for a detailed state-of-the-art analysis). Thus, the unsolved problem of multi-query optimization for neighbor-based pattern mining queries is the second research topic of my dissertation.

1.1.4 Pattern Summarization and Matching

To extend the horizon of neighbor-based pattern mining in streams from pattern extraction only to now also analyze and manage the extracted patterns, a streaming pattern mining system does not only need to be equipped with highly efficient pattern extraction algorithms, but more importantly, it must also provide effective pattern analysis support, as motivated below:

1) **Pattern feature abstraction.** The key features of detected patterns may be complex and thus may not be easily comprehensible for human

analysts without analytical assistance. For example, in real-time traffic monitoring, a cluster representing a congestion area in the traffic of Beijing may be composed of 10K or even more vehicles and may spread to an area over $10km^2$. By simply looking at the information about individual cluster members (vehicles), such as their positions and moving speed, an analyst may not be able to identify the key features of this cluster in real time, such as where is the key bottleneck causing the congestion.

2) **Pattern compression.** Some patterns need to be kept for long-term analysis, yet keeping the full representation of the complex patterns tends to be impractical in streaming environments. In the previous example, storing the full representation of the detected traffic congestion patterns (arbitrarily shaped clusters), namely the individual cluster member tuples (tens of thousands of tuples for each cluster) would cause not only a huge burden on the storage space but also low efficiency for pattern transmission.

3) **Pattern retrieval (matching).** For stream analysis, the archived patterns may need to be retrieved based on their features. Using the above example, when a new traffic congestion arises, the analysts may ask whether similar congestion patterns have been detected before. If yes, rather than figuring out a new congestion-relief plan from scratch, the previous proven-to-work solution for such congestion patterns could be directly applied.

In short, an effective pattern summarization method is the key for complex pattern analysis and management. It is needed for many different aspects of pattern analysis, including feature abstraction, compression and pattern retrieval (as mentioned above). Also, the pattern summarization can be used for approximate pattern representation. For example, one can design pattern visualization or full representation re-generation techniques based on pattern summarizations.

Although the general problems of information summarization and matching have been studied by many previous researcher [CMR05, HYJS06, LLYZ03, MVW98], the problems of summarizing and matching neighbor-based patterns that are defined based on the neighbor relationships among stream objects are not solved yet (see Chapter 1.2.3 for a detailed state-of-the-art analysis). Thus, the third topic of my dissertation is effective **summarization** and **matching** techniques for neighbor-based patterns in streaming environments.

1.2 State-Of-the-Art

1.2.1 Mining Data Streams

The majority of the previous working studying streaming data processing focus on processing traditional SPJ (Selection, Projection and Join) queries [BW01, ABB$^+$03, VNB03] or aggregation queries [GKS01, GKMS01]. Recently, research effort toward mining more complex patterns in streams, such as clusters [GMMO00, AHWY03, YRW09], outliers [Agg05, SPP$^+$06, AF07] and association rules [Pei09], emerges. However, two major differences distinguish my work from these existing works.

First, I focus on the general notion of neighbor-based patterns, which is a family of important pattern types defined by the topological relationships among individual tuples, including density-based clusters [EKSX96, YRW09], distance-based outliers [KN98a, AF07] and k nearest neighbors

[MP07, YOTJ01] and reverse k nearest neighbors [AKK$^+$09, ABK$^+$06]. There is no existing work discussing the general principles of efficiently mining this type of pattern in streaming environments. In general, the previous works studying the problem of mining data streams usually deal with patterns represented using statistical summarizations. For example, [AHWY03] and its variations [DHYC06, BH06] use a summarization method, called Cluster Feature Vectors (CFV), to represent the clusters in the streams. The neighbor-based patterns that are tackled in my dissertation, however, need to be computed at the individual tuple level by considering topological relationships among the tuples. Such pattern mining at the individual tuple level is critical for applications where every individual tuple is of great importance, such as tracking the movement of soldiers on the battlefield.

Second, I studied the neighbor-based pattern mining problem considering sliding window semantics, which is a common semantics for bounding infinite streams for SPJ query processing, while having been barely applied to the area of complex pattern mining. Sliding window semantics assume a window size (either a time interval or a count of objects), with the pattern detection results generated based on the most recent data that falls into the sliding window. However, in previous clustering work [GMMO00, GMM$^+$03], objects with different time horizons are either treated equally or given weights decaying as their recentness decreases. These techniques summarize the accumulative characteristics of the incoming data, while losing the ability to isolate and identify the specific patterns existing in the most recent stream portion. Using my earlier example, the financial analyst may only be interested in the patterns arising in the most recent transactions, for example, those that happened in the last 5 minutes. In such cases, I need the sliding window technique to purge the out-of-date information and form patterns only based on the most recent transactions.

1.2.2 Multiple Mining Query Optimization

Similar to for single query processing, the main focus of previous work on shared execution strategies for multiple queries in streaming environments has been on SPJ [HFAE03, KFHJ04, WRGB06] and aggregation queries [AW04, KWF06]. Little effort has been reported in the literature to date on addressing the problem of multiple query optimization for complex pattern mining queries, such as clusters and outliers. Although some general principles used in previous multiple query optimization work can also be applied in my context, the problem I try to solve in my dissertation is generally more complicated. In particular, the key for multiple query optimization lies in the sharing of meta-information that needs to be maintained by different queries, such as sharing the intermediate results or operator states for SPJ and aggregation queries. The meta-information that needs to be maintained in my context, namely the neighbor-based pattern structures defined by individual tuples as well as their topological interrelationships, is much more complex than that needed for join or aggregation operators, which are usually pair-wise relationships or simply numbers (aggregation results).

1.2.3 Summarization and Matching for Mined Patterns

Summarization techniques have been studied by the database community for decades. A variety of techniques have been developed for effectively summarizing data in relational datasets. The most common techniques are sampling [CMR05, HYJS06], histograms [LLYZ03, MVW98], wavelets [GKMS01, JWK01], and sketches [IKM00]. More recently, researchers have started to look at designing summarization methods for more complex objects in database systems. These techniques can roughly be divided into two categories. The first category considers summarization of spatial objects and the topological relationships among them [LLYZ03, BKS93]. In these techniques, each spatial object is considered as a single entity corresponding to a single record in the database. The goals for these techniques tends to be to effectively represent the external shapes and the topological relationships, such as containment and overlaps, among these objects. More specifically, these techniques usually focus on describing the boundaries of the object and the relationships among the boundaries of different objects. The second category of techniques aims to summarize the objects that are formed by the composition of many smaller granularities of objects, such as the patterns formed by many individual tuples [NRS08, THP08, RLL06], for example graphs, time series and clusters. For the latter techniques, the features of the target objects are determined by the characteristics of the member objects that compose them. Usually, statistical methods are applied to integrate the characteristics of member objects and thus derive a representation of the target objects. For example, the position of a K-means style cluster is commonly represented by a centroid, which is an average value of the positions of all its cluster members. In general, these techniques primarily focus on the “internal features” of the objects, such as internal structures (e.g., connections in graphs) and accumulative statistics (e,g., density).

In my dissertation, I design a model for summarizing neighbor-based patterns that are defined over individual tuples and the neighborships among them. To the best of my knowledge, the summarization of this family of patterns has not yet been accomplished in the literature. It falls into the second category as discussed in the previous paragraph.

One of the most important query types that can be specified on the archive of historical patterns are the “matching queries”. which aim to return identical or similar patterns in the history, given a “to-be-matched” pattern. Such matching queries can be either one-time or continuous in streaming environments. The previous work studying such matching queries usually focus on relatively simple patterns, such as graphs [W09] or time series [GW02], while the problem of matching neighbor-based patterns, such as density-based clusters, has not yet been solved in the literature.

1.3 Proposed Solutions

1.3.1 Solutions for Pattern Extraction Query Execution

Challenges. Efficiently detecting neighbor-based patterns for sliding windows is a challenging problem. Naive approaches that run the static neighbor-based pattern detection algorithms from scratch for each window are not feasible in practice, considering the conflict between the high complexity of these algorithms and the real-time response requirement from streaming applications. Based on my experiments (Chapter 9.2), detecting density-based clusters from scratch in a 50K-object window takes around 100 seconds in my test environment, clearly not meeting real-time response requirements of some time-critical applications.

The incremental approach, which continuously maintains the exact neighbor relationships (henceforth referred to as "*neighborship*") among objects, will also fail in many cases. This is because the potentially huge number of pairwise *neighborships* can easily raise the memory consumption to unacceptable levels. In the worst case, N^2 *neighborships* may exist in a single window, with N the number of data points in the window. My experiments confirm that this solution consumes on average 15 times more memory than the naive approach in real datasets [EFK99].

To overcome this resource strain of a huge memory consumption while still enabling incremental computation, several *neighborship* abstractions, such as cluster membership, can be maintained instead of the exact pairwise *neighborships*. However, designing solutions based on abstracted *neighborships* comes with the shortcoming that the maintenance of abstracted *neighborship* is extremely expensive in terms of CPU resources. More specifically, discounting the effect of expired objects from the abstracted *neighborships* becomes a computation-intensive problem, because such expiration of objects may cause complicated pattern structural changes, such as "splitting", whose detection and handling are almost equally computationally expensive as recomputing clusters from scratch.

Proposed Methods. To make the abstracted *neighborships* incrementally maintainable in a CPU efficient manner, I exploit an important characteristic of sliding windows, namely the "**predictability**" of the expiration of existing objects. Specifically, given a window with a fixed slide size, I can predetermine the "life-span" of any data point in the window, namely the exact future windows it will participate in. I further propose the notion of "**predicted views**". In particular, given the objects in the current window, I can predict the pattern structures that will persist in subsequent windows by considering the objects (in the current window) that are known to also participate in each of these windows only, and abstract these predicted pattern structures into "predicted views" of each future window. This "**view prediction**" technique elegantly discounts the effect of expired objects and thus allows me to efficiently maintain the abstracted *neighborships* by handling the impact of new objects only. Since the computationaly cost of handling the insertion of new objects are way more cheaper than handling object expiration. it brings significant savings on the computational resources.

Finally, I propose a hybrid *neighborship* maintenance mechanism incorporating two forms of neighbor abstraction and dynamically switching between them when needed. This solution achieves not only linear memory consumption, but now also guarantees optimality in the number of the range query searches (the most CPU-expensive operations in neighbor-based pattern detection processes). My experimental studies in Chapter 9 confirmed that my proposed density-based cluster mining algorithm based on this solution takes only 5 seconds to cluster the same 50K data points at each window given a slide of 5K new objects, which is at least 3 times faster than Incremental DBSCAN [EKS$^+$98], the state-of-the-art incremental density-based clustering algorithm designed for data warehouse environments. Also, it is on average 5 times faster than the alternative incremental algorithm using abstract *neighborships* only, while it consumes only 5% of memory space compared to that needed by the method using exact *neighborships*. My proposed distance-based outlier algorithm Abstract-C clearly outperforms the only previous algorithm [AF07] when detecting outliers in time-based windows, while performing equivalently with it when dealing with count-based windows (See Chapter 9.3). Similarly, my proposed algorithm for kNN monitoring, MinTopk, save at least 85% of the CPU time compared with the state-of-the-art solution [MBP06], while using almost negligible memory space, in all my test cases (see Chapter 9.4).

1.3.2 Solutions for Multiple Query Optimization

Challenges. Execution of even a single neighbor-based mining request in streaming environments is expensive in terms of system resource utilization. In particular, the "neighbor-based" property of such pattern mining requests requires a potentially large number of neighbor searches during query execution. Each neighbor search has high system resource costs. More specifically, a complete neighbor search for even just one single object may take a full scan through the window, consuming not only a large amount of CPU processing resources but also forcing the full storage of the whole window. Given such high algorithmic complexity of neighbor-based pattern mining requests, serving a large number of them in a single system is extremely resource intensive. The naive method of executing multiple queries independently has prohibitively high demands on both computational and memory resources. Thus it is not feasible in practice, especially when the number of queries to be executed is large.

Therefore, the key problem that I solve in this work is to design shared execution strategies that achieve effective sharing of system resources among multiple queries. In particular, I aim to not only minimize the total number of neighbor searches by sharing the neighbor search computation among multiple queries but also to share the maintenance effort for the progressive pattern construction among queries. This is a challenging problem, because the meta-information required to be maintained by neighbor-based pattern queries is generally more complex than that for SQL query operators. More specifically, I need to maintain the identified neighbor-based pattern structures, such as clusters and outliers, which are defined by their member tuples and the global topological relationships among tuples. In contrast, SQL operators, such as join or aggregation operators, usually maintain pair-wise relations between two individual tuples (independent from the rest of the tuples) or simply numbers (aggregation results). The techniques introduced previously in the database community regarding sharing among SQL queries [HFAE03, KFHJ04] are thus not adequate to solve my problem.

Proposed Solution. In order to maximize the efficiency of the system resource utilization for executing multiple neighbor-based pattern mining queries simultaneously, I analyze the commonalities of such queries. This helps me to identify several general optimization principles which lead to significant system resource sharing among multiple queries.

As the first step towards sharing, I observe that the range query searches (the process of searching for "neighbors" for each object) can be shared among multiple queries and thus reduce the overall CPU consumption (See Chapter 12). Although this is a straightforward sharing strategy, since the range query searches are frequently needed during neighbor-based pattern mining processes, it constitutes an important multiple query optimization principle for such queries.

However, range query search sharing alone is far from sufficient to achieve the goal of scaling to workloads composed of many such queries within a single system. Therefore, I further analyze the interrelations between the patterns identified by queries with different parameters settings, including both pattern-specific and window-specific parameters. First, I study the conditions under which all queries have the same window parameters. I observe that, if the pattern parameters of a query are "more restricted" than those of another one, a "containment" relationship holds between the patterns identified by them. I exploit this foundation of pattern containment to incrementally organize the patterns identified by multiple queries into an integrated structure. I call it *IntView* (See Chapter 13). As a highly compact structure, *IntView* saves the memory space needed for storing the patterns identified by multiple queries. More importantly, *IntView* also enables integrated maintenance for the progressive patterns of multiple workload queries, and thus effectively saves the computational resources for maintaining them independently.

Second, I proposed a "*meta query* strategy", which uses a single meta query to represent all workload queries whose pattern parameters are the same while their window parameters differ (See Chapter 14). The proposed meta query strategy adopts a flexible window management mechanism to efficiently organize the query windows that need to be maintained by multiple queries. By leveraging the overlap among query windows, it minimizes the number of windows that are actually maintained in the system. I show in Chapter 14 that my meta query technique successfully transforms the problem of maintaining multiple queries into the execution of a single query.

Finally, I combine the three techniques proposed, namely range query search sharing, the *IntView* technique and the *meta query* strategy, to form a proposed comprehensive solution for each specific pattern type, namely the shared execution strategies for multiple density-based clustering, distance-based outlier or kNN queries over sliding windows (see Chapter 15). Computation-wise, these three proposed algorithms require only a single pass through the new objects at each window slide. In particular, they only run one range query search for each new object. Also each new object only communicates with its neighbors once for a group of shared queries. Memory-wise, given the maximum window size allowed, the upper bound of the memory consumption of my solution for a group of shared queries is independent of the number of queries in the group.

My experimental studies (in Chapter 16) show that my proposed solution clearly outperforms all the alternative methods, and has great scalability in the number of queries it can handle. In particular, for density-based clustering queries, the system using my proposed algorithm comfortably handles a workload composed of 100 arbitrary queries under a 1K tuples per second data rate. If the number of workload queries increases to 1K, the system still works stably with a 300 tuples per second input rate. On the same experimental platform, given the 300 tuples per second input rate, the existing execution strategies from the literature, such as *IncDBSCAN* [EKS$^+$98] and *Extra-N* [YRW09], can only handle less than 1.7 and 12 percent of the same 1K query workload, respectively. My performance analysis for distance-based outlier and kNN queries shows that a similar performance can be expected from my proposed strategies for those two pattern types as well (see Chapter 16.10).

1.3.3 Solutions for Pattern Summarization and Matching

Challenges. Summarization and matching of neighbor-based patterns is not only an unsolved but also a challenging problem. To serve real-time streaming applications, the proposed techniques must fulfill the following requirements: 1) Pattern summarization must be sufficiently **descriptive** yet highly **compact**. The pattern structure of neighbor-based patterns can be complex. For example, a density-based cluster is defined by a series of densely populated sub-regions as well as the connections among them (See Figure 2.1). Clearly, simple statistical aggregations, such as the centroid or minimum bounding rectangle of a cluster, are insufficient for describing such complex pattern structure. For example, the simplistic centroid+ radius summarization method preserves no knowledge about how the cluster member objects are distributed in the cluster, not to mention the connections among them. 2) The pattern summarization process has to be highly **efficient**. A system conducting expensive online pattern extraction can hardly afford additional system resources for summarizing patterns in real-time. 3) The summarized pattern representation needs to be effectively **retrievable** (“matchable”). The matching process between pattern summarizations ought to loyally reflect the similarity between the original patterns, yet be computationally efficient.

Proposed Solution. To fulfill the above requirement, I first start with the pattern type that has complex pattern structure among neighbor-based pattern family, namely, density-based clusters. I analyze density-based cluster structures and identify their key characteristics, namely *position*, *shape*, *connectivity* and *density distribution*. To capture these features, I investigate two commonly-used summarization principles, namely the graph-based and the grid-based strategies. I discover that neither of them alone is capable to provide an effective summarization for density-based clusters. Therefore, I propose a hybrid solution, called Skeletal Grid Summarization (SGS). In terms of its descriptive power, SGS is shown to guarantee its fidelity to the original clusters on all key features, including postion, shape, density distribution and connectivity. For compactness, my experimental study in Chapter 23 confirms that even the SGS of the highest resolution achieves on average a 98% compression rate of the full representation of the clusters.

Empowered by the proposed SGS summarization, I design a framework to support both continuous cluster extraction and cluster matching queries. A continuous cluster extraction query in my system does not only extract clusters in their full representation (all cluster member objects) for online monitoring purposes like the other state-of-the-art techniques [CEQZ06, YRW09], but it also concurrently compacts them into the SGS summarization. The full and the summarized (SGS) representation formats are complementary to each other, providing a description of the clusters at the individual tuple and cluster feature level respectively. To extract these two representation formats simultaneously and in a highly efficient manner, I propose an integrated cluster extraction + summarization algorithm, C-SGS. C-SGS incrementally maintains both the full representation and the corresponding SGS of the extracted clusters in an integrated manner. This results in an almost "free" cluster summarization generation by piggy-packing the summarization process into the cluster extraction process itself. My experimental study in Chapter 8.2.2 shows that C-SGS, which returns clusters in both full and summarized representation (SGS), has a neglectable overhead, compared with state-of-the-art algorithm Extra-N [YRW09] computing the full representation of clusters only. In all my test cases, the extra response time of C-SGS compared with Extra-N is consistently less than 6% (Section 8.1).

For any "to-be-matched" cluster specified by the analyst, a cluster matching query identifies similar clusters extracted earlier in the same stream from a pattern archive. To support such queries, my framework first archives the SGS of the extracted clusters into a pattern archive. When executing a cluster matching query, my system deploys a filter-and-refine strategy. First, the filter-phase exploits a feature index to locate the potential matching candidates from the pattern store. Then, the refine-phase conducts a more detailed cluster match against these promising candidates and returns those with similarity above a given threshold. My experimental study shows that, efficiency-wise, my system takes only 3 seconds on average to answer a cluster matching query against 10K archived clusters (Chapter 8.2.2). Quality-wise, my user study, which invites human analysts to visually compare the similarity between matched clusters, shows that human analysts agree with a significant larger percentage of the matched clusters found using my proposed matching mechanism compared to those found by alternatives (Chapter 8.2.2).

For distance-based outlier and kNN, due to their relatively simple pattern structure, one can easily use clustering and histogram based summarization methods for their summarization. Correspondingly, their matching processes are much simpler compared to density-based clusters. Although the potential summarization and matching techniques for distance-based outlier and kNN are discussed in Chapter 22, they are not the focus of this dissertation.

1.4 Where our Proposed Techniques May Not Work

This dissertation proposes several general optimization principles, such as predicted views and pattern representation and maintenance techniques. It shows that these principles can be applied to different pattern types in neighbor-based pattern family with proper customizations. However, same as all the optimization principles, the techniques presented in this work have their own application scopes, and may not be suitable to be used for the areas outside these scopes.

The predicted view technique relies on the overlap of sliding windows

($slide < win$). On one hand, if the slide size $slide$ is equal or larger than the window size win, no streaming object can appear in more than one windows, and thus there will be no need and benefit for predicted view maintenance. On the other hand, if the slide size is too small, the predicted view technique may require a system to maintain too many predicted views, which may cause significant computational and storage overhead. Although, one may be able to reduce this overhead by further compressing the predicted views in different windows (see Chapter 7), it is not always possible, depending on the specific target pattern types and the characteristics of the input streams.

The pattern representation and maintenance techniques proposed in this work, including different formats of pattern structure representation and computation strategies to maintain those pattern structures, are specifically designed for neighbor-based patterns, which are defined based on the pairwise neighbor relationship among data objects. Thus, these techniques may not be able to be applied to other pattern types. For example, the incremental density-based cluster maintenance techniques may not be able to be applied to extract clusters with other definitions in which clusters are defined using other criteria rather than local density, such as defining clusters by fixing the number of clusters in the dataset, or defining clusters hierarchically.

1.5 Research Impact and Open Research Questions

My dissertation is an early research effort in the database community to look at the problem of mining complex patterns in data streams. It extends the traditional database query optimization principles, such as incremental query result update, multiple query optimization and result summarization, to solve the challenging problems for mining complex patterns from streams. My dissertation shows that, with innovative tuning and customization, those well-established query optimization principles, which were designed for executing traditional database retrieval queries, such as SPJ queries, can also be extended to serve complex pattern mining queries. Thus, my work opens a new field of designing solutions for the emerging streaming mining applications based on extending the well-known query optimization principles. Such research practice encourages deep analysis of the characteristics of the targeted complex patterns and designing highly efficient customized mining algorithms for them, while saving the unnecessary effort of "re-inventing the wheel" from scratch.

Although my dissertation studies three different aspects of complex pattern mining in data streams, namely efficient pattern extraction query execution, multiple query optimization and mining result management, it is just the beginning of a general research agenda of mining and managing complex patterns in data streams. Many research problems for mining and managing complex patterns in data streams remain open and are worth to be explored.

First, given the great diversity of stream analysis tasks, besides the neighbor-based patterns studied in this dissertation, there are many other complex pattern types that need to be mined from data streams, ranging from association rules, sequences to text. Each of these pattern types may have its own very unique pattern structures, and thus may require its own customized mining strategies designed for it. Although some research effort has been made towards mining these complex patterns in the stream [Han05], mining of many important pattern types, such as graphs and webs, remains largely unsolved.

Second, unlike traditional data mining in static environments where mining queries are submitted to static datasets for one-time mining results, stream analysis tasks tend to be continuous. That is the later requires a mining system to build a **temporal context** for mining results for analysts, reflecting how the patterns change over time in streams. To achieve this goal, the pattern evolution model, which can effective describe the pattern changes over time, and efficient evolution tracking methods have to be designed. My research work [YGRW11] studied the problem of tracking evolution of density-based clusters in streams anytime, while the same problem for many other complex patterns, such as outliers and graphs, remain largely unsolved.

Third, in streaming environments, not only the data but also the queries can be dynamic, indicating that the mining queries can be adapted (change parameter settings), registered to or removed from the mining systems at any time. This requires a mining system to have the capability of smoothly tuning itself to handle a changing workload. This problem is not covered in my dissertation, and can be an important research direction for future work.

An in-depth analysis of the future work for my dissertation can be found in Chapter 26.

1.6 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 discusses the formal definition of neighbor-based patterns and where streaming neighbor-based pattern mining lies in the overall context of the data mining family. The three research topics of my disseration are discussed in detail respectively in Part I (Chapters 3-11), Part II (Chapters 12-18) and Part III (Chapters 19-25). The discussions of each of the three research topics include the analysis of the problem, initial effort for tackling the problem, description of the final proposed solutions, analysis of alternative solutions, experimental evaluation, and lastly discussions of related work. Chapter 25 concludes this dissertation and Chapter 26 discusses possible future work.

Chapter 2

Preliminaries

2.1 General Notion of Neighbor-Based Patterns

We now demonstrate that the neighbor-based pattern mining queries tackled in my work correspond to a particular subclass of the general class of graph mining. This subclass of graph mining tasks is composed of two phases, namely graph definiton and graph mining.

Graph Definition Phase. First, unlike the traditional graph mining tasks, in which the target graph structures are given as input, neighbor-based pattern mining queries require a *graph definition* step before mining of the graph. In particular, given an input dataset D, a neighbor-based pattern mining query first defines a graph $G = (V, E)$, in which $V = D$ corresponds to the set of vertices and E corresponds to the set of edges, modeling all the pair-wise "neighbor relationships" among the vertices. The edges in E can be either directed or undirected. Such *graph definition* step takes two inputs from the query specification, namely:

1): A user-defined distance function $Dist(v_a, v_b)$, $\forall v_a, v_b \in V$, which returns a value $dist\_v_a\_v_b$ reflecting the distance between v_a and v_b.

2): An edge definition function $M(v_a, v_b, dist\_v_a\_v_b)$, which decides whether (true/false) an edge exits $e(v_a, v_b)$ between v_a and v_b. The distance between v_a and v_b, $dist\_v_a\_v_b$, is the base for $M(v_a, v_b, dist\_v_a\_v_b)$ to make this decision, while the specific edge definition mechanism may vary depending on the specific neighbor-based pattern mining query type. For example, $M(v_a, v_b, dist\_v_a\_v_b)$ for density-based cluster [EKSX96, EKS$^+$98] and distance-based outlier [KN98b, AF07] mining imposes a range threshold value to define an undirected edge $e(v_i, v_j)$ between v_a and v_b, while $M(v_a, v_b, dist\_v_a\_v_b)$ for k nearest neighbor (kNN) [MP07, JOT$^+$05] and reverse k nearest neighbor mining queries [MP07] takes a count threshold to define directed edges $e(v_a, v_b)$ and $e(v_b, v_a)$. We will introduce the precise edge definition function for each specific neighbor-based pattern type in their respective formal definition later in this section.

Graph Mining Phase. Given the graph G defined in the graph definition phase, in the second phase, each neighbor-based pattern mining query type mines for a particular type of sub-graph(s) in G that exhibits certain characteristics. We will explain the specific sub-graph(s) that each neighbor-based pattern mining query type mines for in the formal definition of each pattern type.

2.2 Definitions for Specific Neighbor-Based Patterns

We use the term *data point* to refer to a multi-dimensional tuple (object) in the data stream. To be consistent with the graph mining problem definition given above, we use v_i to represent each data point in the following definitions.

2.2.1 Density-Based Cluster

Definition 1 ***Density-Based Cluster Mining Query:*** *Besides the input dataset D and a distance function $Dist(v_a, v_b)$, density-based cluster mining takes two input parameters, namely a range threshold θ^{range} and a count threshold θ^{cnt}.*

In the ***graph definition phase****, a density-based cluster mining defines an undirected edge $e(v_a, v_b)$ between any v_a and $v_b \in D$, if $Dist(v_a, v_b) < \theta^{range}$. We say that v_a and v_b are neighbors of each other in this situation. It thus defines an undirected graph $G = (V, E)$, with V corresponding to all data points in D and E corresponding to all the undirected edges among data points in V.*

Then, in the ***graph mining phase****, we use the function $NumNei(v_i, \theta^{range})$ to denote the number of neighbors a data point v_i has, given the θ^{range} threshold. A data point v_i with $NumNei(v_i, \theta^{range}) \geq \theta^{cnt}$ is defined as a core point. Otherwise, if v_i is a neighbor of any core point, v_i is an edge point. v_i is a noise point if it is neither a core point nor an edge point. Two core points v_0 and v_n are connected if they are neighbors of each other, or there exists a sequence of core points $v_0, v_1, ...v_{n-1}, c_n$, where for any i with $0 \leq i \leq n-1$, a pair of core points v_i and v_{i+1} are neighbors of each other. Each density-based cluster is a group of connected core points and the edge points attached to them. Density-based cluster* *mining mines for all such clusters in the defined graph G.*

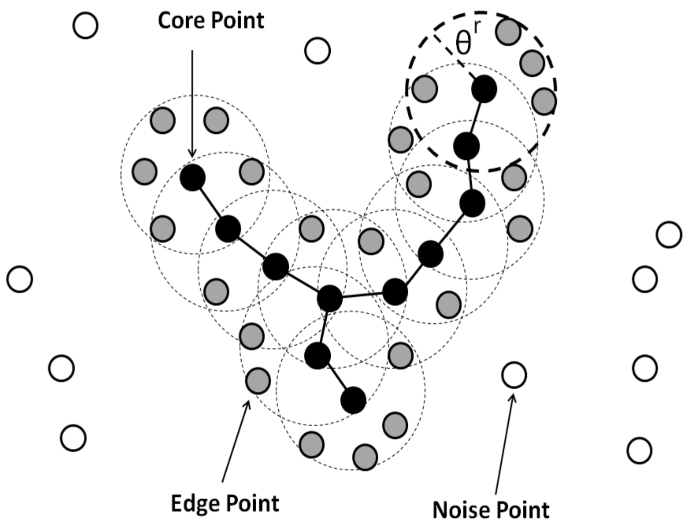

Figure 2.1: An Example of A Density-Based Cluster

Figure 2.1 shows an example of a density-based cluster composed of 11 *core points* (black) and 24 *edge points* (grey).

2.2.2 Distance-Based Outlier

Definition 2 ***Distance-Based Outlier:*** *Besides the input dataset D and a distance function $Dist(v_a, v_b)$, distance-based outlier detection takes two input parameters, namely a range threshold θ^{range} and fraction threshold θ^{fra}.*

In the ***graph definition phase****, distance-based outlier mining defines an undirected edge $e(v_a, v_b)$ between any v_a and $v_b \in D$, if $Dist(v_a, v_b) < \theta^{range}$. We say that v_a and v_b are neighbors of each other in this situation. It thus defines an* *undirected graph $G = (V, E)$, with V corresponding to all data points in D and E corresponding to all the undirected edges among data points in V.*

*Then, in the **graph mining phase**, we use the function $NumNei(v_i, \theta^{range})$ to denote the number of neighbors a data point v_i has, given the θ^{range} threshold. Distance-based outlier mining mines for all data points v_i in the defined graph G, where $NumNei(v_i, \theta^{range}) < |D| * \theta^{fra}$, with N the number of vertices in G.*

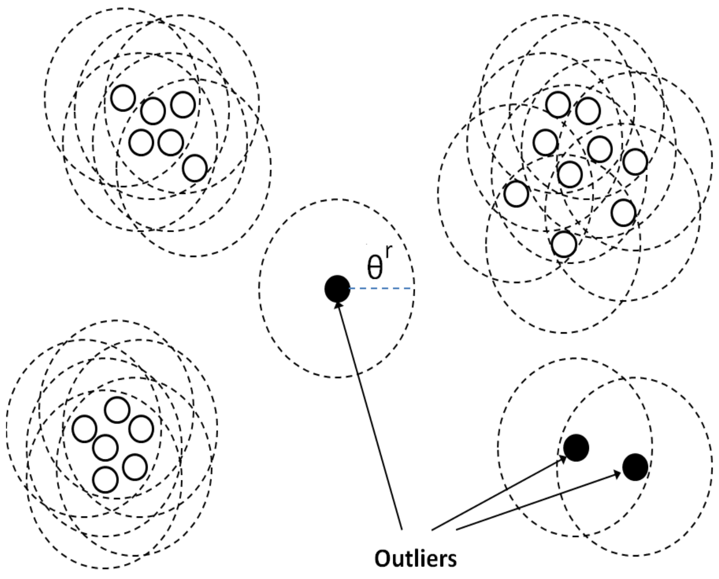

Figure 2.2: An Example of 3 Distance-Based Oultiers

Figure 2.2 shows an example of 3 outliers detected from a 2D dataset containing 25 objects. In this example, $\theta^{fra} = 0.12$, indicating that if an object has less than $25 \times 0.12 = 3$ (the total number of objects in the dataset is 25), it is defined as an outlier.

2.2.3 Top-k Nearest Neighbors (kNN

Definition 3 ***Top-k Nearest Neighbor Query (kNN):*** *Besides the input dataset D and a distance function $Dist(v_a, v_b)$, a kNN mining query takes a query object v_q and a count threshold k.*

*In the **graph definition phase**, a kNN mining query defines a directed edge $e(v_a, v_b)$ from v_a to $v_b \in D$, if there exist less then k data points v_0, v_1, ... ,$v_{k-1} \in D$ that $dist\_v_b\_v_i < dist\_v_b\_v_a (0 \leq i \leq k-1), i \neq a, b$. We say v_b is a neighbor of v_a if there exists an edge $e(v_a, v_b)$ from v_a to v_b. It thus defines a directed graph $G = (V, E)$, with V corresponding to all data points in D and E corresponding to all the directed edges among data points in V.*

*In the **graph mining phase**, a kNN mining query mines for all neighbors for the query object v_q in the defined graph G.*

Figure 2.3 shows an example of 4 nearest neighbors detected for a query object. In this example, the object depicted using the grey dot is the query object. The kNN query identifies four kNN objects (the black dots in the circle) for it.

2.3 Sliding Window Semantics

We focus on periodic sliding window semantics as proposed by Continuous Query Language (CQL) [ABW06] and widely used in the literature [YRW09, AW04, YRW09]. Such semantics can be either time-based or count-based. For both cases, each query Q has a window size $Q.win$ (either a time interval or a tuple count) and a slide size $Q.slide$. The patterns will be gen-

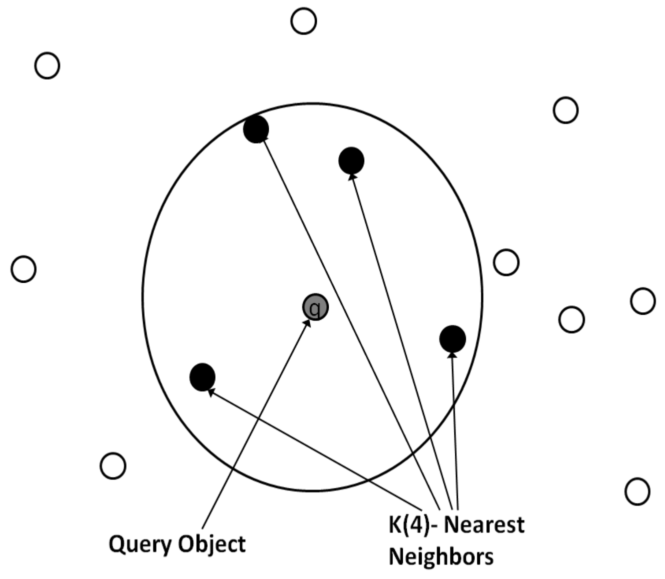

Figure 2.3: An Example of $k = 4$ Nearest Neighbors

erated only based on the data points falling into the window. The query window slides periodically either when a certain number of tuples arrives or a certain amount of time elapses. By sliding, a new window will be built to replace the old window, and thus again cover only the most recent portion of the stream at that moment. The templates of neighbor-based pattern mining queries using this query semantics are shown in Figures 2.4, 2.5 and 2.6. In particular, the templates for density-based cluster detection query over sliding windows is shown in Figure 2.4. The templates for distance-based outlier detection query over sliding windows is shown in Figure 2.5. The templates for kNN detection query over sliding windows is shown in Figure 2.6.

Q_i: ***DETECT Density-Based Clusters FROM*** $stream$
USING $\theta^{range} = r$ ***and*** $\theta^{cnt} = c$
IN Windows WITH $win = w$ ***and*** $slide = s$

Figure 2.4: Template for a Density-Based Cluster Detection Query for Sliding Windows over a Data Steam

Q_i: ***DETECT Distance-Based Outliers FROM*** $stream$
USING $\theta^{range} = r$ ***and*** $\theta^{fra} = f$
IN Windows WITH $win = w$ ***and*** $slide = s$

Figure 2.5: Template for a Distance-Based Outlier Detection Query for Sliding Windows over a Data Stream

Q_i: ***DETECT*** $p_i.kNN$ ***FROM*** $stream$
USING K=k
IN Windows WITH $win = w$ ***and*** $slide = s$

Figure 2.6: Template for a kNN Detection Query for Sliding Windows over a Data Stream

2.4 Where Streaming Neighbor-Based Pattern Mining Lies in Data Mining Family

Data mining, as a general concept for extracting or "mining" knowledge from large amounts of data, covers a rather diverse range of mining tasks. Also, any data mining task may consist of an iterative sequence of the following steps: 1) data cleaning, 2) data integration, 3) data selection, 4) data transformation, 5) pattern extraction, 6) pattern evaluation and 7) knowledge representation [Han05]. In this work, we focus on neighbor-based pattern mining queries in streaming windows. The Parts I and II of this dissertation fall into the pattern extraction step of data mining, which is an essential process where intelligent methods are applied to extract patterns from well prepared data. The Part III of this dissertation are related to both pattern evolution (identifying similar patterns from stream history) and knowledge representation (pattern summarization).

Next, we review a categorization of the common data mining tasks from the literature and show where our target query types lie in this categorization. Traditionally, data mining techniques [ZRL96, EKSX96, ABKS99, KN98b, BKNS00, JOT$^+$05, KOTZ04, NZTK08] are designed for static environments with large volumes of stored data. More recently, as stream applications are becoming prevalent, the problem of mining streaming data is being tackled [AHWY03, YRW09, BDMO03, SPP$^+$06, AF07, MP07]. Based on the dynamics of the input data, we can first divide the data mining tasks into *static data mining* and *stream data mining*. Clearly, our tasks of mining neighbor-based patterns in streaming windows fall into the *stream data min-*

ing category.

Second, for both *static* and streaming data mining, Han and Kamber [Han05] divide the data mining tasks into two categories based on their purposes, namely ***descriptive*** and ***predictive*** mining. In particular, descriptive mining tasks characterize the general properties of the data in the database. Predictive mining performs inferences on the current data in order to make predictions for future data. Typical *predictive* data mining tasks include *classification*, *prediction* and *trend mining* [Han05]. Since our task of mining neighbor-based patterns in data streams aims to find specific patterns in the most recent portion of the stream, which does not necessarily predict anything about the future, our task falls into the ***descriptive data mining*** category.

Among the *descriptive streaming data mining* tasks, they can be further divided into the following categories based on their distinct characteristics [Han05].

1) ***graph mining***: The one-to-one relationships, namely the (directed or undirected, weighted or unweighted) *edges* among objects and the topological *relationships* among objects are the key factors that define the patterns which *graph mining* processes mine for.

2) ***association rule and correlation mining***: *Frequency* is the key factor for association rules and correlations. Namely, the frequency of a certain type of objects to appear together or the frequency of a certain relationship existing among certain attributes of the objects defines association rules and correlations.

3) ***text mining***: In *text mining*, the appearance of certain key words and

relationships among their linguistic meanings of these key words are the key factors that define the patterns in the text.

4) ***sequence mining***: In *sequence mining*, the *time sequences* in which the target events happen or the *time sequences* in which values of an attribute appear are the key factors for the *sequence mining* process.

5) ***clustering***: Cluster mining processes aim to divide the input objects into different *groups*, each having its own characteristics. Maximizing the *similarity* among the objects within the same group and the *disimilarity* among objects within different groups is the goal pursued by clustering algorithms.

6) ***outlier mining***: The *abnormality* of some objects compared to the majority of all objects in the dataset is the knowledge that the *outlier mining* process mines for.

7) ***web page mining***: The *textual content* and the *link structures* among web pages is explored by *web page mining*.

8) ***multimedia mining***: Mining on *images* and *voices* distinguishes *multimedia mining* from other mining tasks.

Given this categorization, if we analyze our neighbor-based pattern mining tasks from the perspective of their general **purpose**, they relate to multiple categories, including cluster and outlier mining. However, as discussed earlier in Chapter 2.1, if we analyze the characteristics of the target **pattern structures** of neighbor-based pattern mining queries, namely how these **pattern strcutures** are defined, all the neighbor-based pattern mining queries can be viewed as subclasses of the *graph mining* category. In conclusion, our target neighbor-based pattern mining queries over streaming windows fall into the *graph mining* category of *descriptive streaming data mining*.

Part I

Efficient Processing of Single Neighbor-Based Pattern Extraction Queries

Chapter 3

Naive vs. Incremental Approach

3.1 Naive Solution–Recompute from Scratch at Each Window

The naive approach for detecting patterns over continuous windows would be to run static pattern detection algorithms from scratch at each window.

For density-based clusters and distance-based outliers the static neighbor-based pattern detection algorithms [EKSX96, KN98b] consume one range query search for every data point in the dataset. In our case, they need N range query searches at each window W_i, with N the number of data points in W_i. Although some minor improvement could be made, such as some range query searches may be terminated earlier when detecting distance-based outliers, $O(N)$ is the lower bound of the number of range query searches needed to detect these patterns in a new dataset (see Lemma 3.1).

Considering the expensiveness of range query searches, such naive approach may not be applicable in practice, specially when N is large. Obviously, without the support of indexing, the complexity of each range query search is $O(N)$. The average run-time complexity of a range query search can be improved by use of index structures, for instance an R-tree could improve it to $O(\log(N))$ [EKSX96]. However, such complexity may still be an unacceptable burden for the streaming applications that require real-time response, not to mention that the high-frequency of data updating in the streaming environments makes the index maintenance expensive. Our experimental study in Chapter 9.2 shows that detecting density-based clusters from scratch in a 50K-object window takes around 100 seconds in our test environment, which is clearly not meeting real-time response requirements of streaming applications.

For kNN detection, recomputing such pattern from scratch at each window takes at least $O(N)$ time [MBP06], with N the number of data points in the query window. If the k nearest neighbors need to be returned in their sorted order, then the CPU cost increases to at least $O(log(k) * N)$. This corresponds to tremendous computational costs, especially for queries with large window sizes and processing high speed input streams. Also, this naive approach forces the system to keep all the tuples in the window, even if some of them may never have a chance to be part of the query results. This causes a huge amount of unnecessary memory utilization, which may harm the applicability of this naive approach in many cases. Our experimental study in Chapter 9.4 shows that recomputing the kNN from scratch in a 1M-object window takes at least 2 minutes in our test environment.

In conclusion, given these limitations, such naive approach of recomputing patterns from scratch is not viable for handling overlapping windows ($Q.slide < Q.win$), where the opportunity for sharing meta-information among windows exists.

3.2 Why Incremental Computation

In sliding window query semantics, query windows overlap when the slide size is smaller than the window size ($slide < size$). In such overlapping-window scenario, the stream objects may fall into (be valid in) multiple query windows before they expire. It serves many application needs, in which the analysts want to detect patterns based on a relatively large query window but would like to see the results updated in a shorter period of time. Using the motivation examples in our introduction, a financial analyst monitoring the intensive-transaction areas (clusters formed in stock transactions) within the last 10-minute stock transactions may want to see the results updated every 1 or 2 minutes. A banker monitoring the potential frauds (outliers) in the last 1-hour credit card transactions may want the results to be updated in every 5 or 10 minutes.

This overlapping-window scenario provides the opportunities to design incremental pattern computation strategies, because the detected patterns in such a scenario may persist and change incrementally in multiple windows. By incremental pattern maintenance, one can re-use the pattern structures detected earlier in the previous windows, and thus significantly reduce the computational costs for extracting patterns from scratch in each window.

3.3 Basic Incremental Computation Strategies

Now we discuss the basic incremental computation strategies for the three types of neighbor-based patterns that we focus on in this dissertation.

Basic Incremental Computation Strategy for Density-Based Cluster.

First, we propose an basic incremental computation strategy for density-based clusters. As we have discussed earlier in Introduction (Chapter 1), we are the first to tackle the problem of density-based cluster detection in sliding windows. We call this method **Exact-Neighborship-Based Solution (Exact-N)**. Exact-N relieves the computational intensity of processing each window by preserving the exact *neighborships* discovered in the previous windows. In particular, Exact-N requires each data point p_i in the window to maintain a list of links pointing to all its neighbors. This allows each data point to access their neighbors directly whenever it needs to.

At each window slide, the expired data points are removed along with the exact *neighborships* they are involved in, namely all the links pointing from or to them. Then Exact-N runs one range query search for every new data point p_{new} to discover the new *neighborships* to be established in the new window. At the output stage, Exact-N constructs the cluster structures by a Depth First Search (DFS) on all *core points* ((Selection, Projection and Join)o less than θ^{count} neighbors) in the window. Exact-N offers the advantage of significantly reducing the amount of range query searches needed at each window slide. In particular, as we discussed in Chapter 3.1, the naive approach requires N (number of objects in the query window) range query searches at each window slide, while this Extra-N algorithm only needs N_{new} range query searches at each window, with N_{new} the number of new data points in the window.

Lemma 3.1 ***For each query window W_i, the minimum number of range query searches needed for detecting density-based clusters in W_i is N_{new}.***

Proof 3.1 *At each new window W_i, each new data point falling into W_i needs a range query search to discover all its neighbors in the window, otherwise we cannot obtain all new neighborships in W_i introduced by the participation of the new data points. This shows the necessity of the N_{new} range query searches. Since we can always preserve all neighborships inherited from W_{i-1}, we will not miss any prior neighborships existing in W_i. This demonstrates the sufficiency of the N_{new} range query searches.*

The pseudo-code for the Exact-N algorithm is shown in Figures 3.1 and 3.2.

Basic Incremental Computation Strategy for Distance-Based Outliers.

The same Extra-N algorithm can be adapted to solve the distance-based outlier detection problem as well. The only difference that distinguishes it from the Extra-N algorithm detecting density-based clusters exists at the output stage. In particular, for distance-based outliers, Exact-N simply outputs the data points with less than $N \times \theta^{fra}$ neighbors at the output stage.

```
Exact-N ($\theta^{range}$,$\theta^{cnt}$ / $\theta^{fra}$)
1 At each window slide
//Purging
2 For each expired data point $p_{exp}$
3   For each $p_i$ in $p_{exp}.neighbors$
4     remove $p_{exp}$ from $p_i.neighbors$;
5   purge $p_{exp}$;
//Loading
6 For each new data point $p_{new}$
7   load $p_{new}$ into index
//Neighborship Maintenance
8 For each new data point $p_{new}$
9   $Neighbors$ =
    RangeQuerySearch($p_{new}, \theta^{range}$)
10      For each $p_j$ in $Neighbors$
11        add $p_j$ to $p_{new}.neighbors$
12        add $p_{new}$ to $p_j.neighbors$
//Output
13    OutputPatterns(pattern type);
```

Figure 3.1: Pseudo-Code for Exact-N Algorithm (Part 1)

```
OutputPatterns(Distance-Based Outliers)
1 For each data point $p_i$ in the window
2   If $p_i.neigbors.size() \leq \theta^{fra} * N$
3     Output $(p_i)$
OutputPatterns(Density-Based Clusters)
1 ClusterId=0;
2 For each $p_i$ with $\leq \theta^{cnt}$ neighbors
3   If $p_i$ is unmarked;
4     OutputCore($p_i$, ClusterId);
5     ClusterId++;
OutputCore($p_c$, ClusterId)
1 mark $p_c$ with ClusterId;
2 output($p_c$);
3 For each data point $p_i$ on $p_c.neighbors$
4   If $p_i$ is unmarked
5     If $p_i.neigbors.size() \leq \theta^{cnt}$
6       OutputCore($p_i$, ClusterId)
7     Else
8       mark $p_i$ with ClusterId;
9       Output($p_i$);
```

Figure 3.2: Pseudo-Code for Exact-N Algorithm (Part 2)

The pseudo-code for this Exact-N algorithm detecting distance-based outliers is shown in Figure 3.1 and 3.2 as well.

Basic Incremental Computation Strategy for kNN.

For kNN detection, [MBP06] presented an incremental computation strategy, SMA, by incrementally maintaining the nearest neighbors for the query object. The key idea of this work is to maintain a "skyband structure", which contains more than k nearest neighbors to the query object in a ranked order. At each window slide, SMA first removes the expired objects from the skyband. Then it inserts the new objects into the skyband if any of them has a smaller distance to the query object compared to any existing objects in the skyband.

At the output stage, if the skyband structure contains at least k objects. SMA simply outputs the first k objects with the smallest distance to the query objects as its k nearest neighbors (kNN). Otherwise, SMA searches against all the objects in the query window looking for the kNN for the query object.

3.4 Limitations of Basic Incremental Strategies

The incremental computation strategies discussed in Chapter 3.3 achieve the basic goal of incremental pattern maintenance, and thus partially or completely avoid the expensive recompute-from-scratch costs suffered by the naive approach mentioned in Chapter 3.1. However, these basic incremental computations still have their own key limitations, and thus do not constitute the ideal solutions for neighbor-based pattern detection in sliding windows.

In particular, Exact-N suffers from two major shortcomings. The first shortcoming is its huge memory consumption. As Exact-N requires storing all exact *neighborships* among data points, its memory consumption may be huge in many cases. In the worst case, the memory requirement may be **quadratic** in the number of data points in the window. Such a tremendous demand on memory may make the algorithms impractical for huge window sizes N, given that the real-time response requirement of streaming applications necessitates main memory resident processing.

The second shortcoming of Extra-N is its expensive CPU costs required for maintaining the exact neighborships among the objects, especially when handling object expirations. As we discussed earlier in Chapter 3.3, at each window slide, Extra-N needs to not only remove the expired objects from the query window but also to remove these expired objects from the neighbor lists of the remaining objects. This could be a very expensive operation computationally, as for each expired object p_{exp}, it needs to search in the neighbor lists of all p_{exp}'s neighbors to identify the positions where p_{exp} are stored and then to remove p_{exp} from these lists.

Both our cost analysis (Chapter 8) and experimental studies (Chapter 9) confirm those two shortcomings of Exact-N for detecting both density-based clusters and distance-based outliers.

Using the SMA algorithm presented in [MBP06] for kNN detection also suffers from both CPU and memory problems. The key limitation of SMA is that it does not eliminate the need of a from-scratch recomputation. In particular, when the skyband maintained by SMA contains less than k objects, it has to search against the whole window content to look for the k nearest neighbors. This causes a serious performance bottleneck. Computationally, when searching for the qualified k nearest neighbors, this process has to look at potentially all objects in the query window. Thus, the computational costs can be close to or even equivalent to calculating the k nearest neighbors from scratch. Memory-wise, as current non-k-nearest-neighbor objects may qualify as k nearest neighbors in future windows, such a recomputation process requires keeping and maintaining all objects alive in the window. This can be a huge burden on memory utilization. For queries that have large window sizes, the number of objects required to be stored can easily reach millions or higher, even when the actual number of k nearest neighbors a user is interested in is fairly small, say $k = 10$ or $k = 100$.

Chapter 4

Predicted View Technique

4.1 Performance Bottleneck – Handle Expiration

Based on our analysis of the limitations suffered by the basic incremental computation strategies discussed in Chapter 3.4, the key performance bottleneck they are facing is caused by handling the object expirations during the window slide process. In particular, at each window slide, any incremental neighbor-based pattern detection algorithm needs to handle the impact to the existing patterns caused by both the insertion of new objects and the deletion of expired objects. Within these two types of computation, handling the impact of a new object insertion to neighbor-based patterns is generally easier, while handling the impact of the expired objects may cause much more system resource utilization. This is because, the expiration of existing objects may cause complex pattern structural changes, ranging from shrinkage, splitting to the termination of the patterns. Handling these complex pattern changes may need tremendous computational and storage resources, which could be as expensive as recomputing the patterns from scratch.

As discussed in Chapter 3.4, the Exact-N algorithm needs tremendous computational effort to remove the expired neighbors for the remaining objects, and the SMA algorithm needs to recompute the kNN from scratch when the object expiration reduces the number of objects in its skyband structure to less than k.

4.2 Predicted View

To solve this serious performance bottleneck of handling object expiration, we now propose a general optimization technique, called "Predicted View". We first highlight the "predictability" property of sliding windows to be exploited for our later algorithm design.

Definition 4 *Given the slide size $Q.slide$ of a query Q and the starting time of the current window $W_n.T_{start}$, the* ***life-span*** *$p_i.lifespan$ of a data point p_i in W_n with time stamp $p_i.T$ is defined by $p_i.lifespan = \lceil \frac{p_i.T - W_n.T_{start}}{Q.slide} \rceil$, indicating that p_i will participate in windows W_n to $W_{n+p_i.lifespan-1}$.*

For example, let's assume that we have a query window with window size $win = 16s$ and slide size $slide = 4s$, and at wall clock time 00:00:00 there are 16 objects falling into the query window with their their time stamps equal to 00:00:00, 00:00:01, 00:00:02, ..., 00:00:15. In this case, as we know that the slide size of the query window is 4s, the predictability property can tell us for sure that the first 4 objects, namely the objects with time stamps equal to 00:00:00, 00:00:01, 00:00:02, 00:00:03 will be expired after the window slide at 00:00:04. For the same reason, after the next window slide at 00:00:08, the next four objects with time stamps equal to 00:00:04, 00:00:05, 00:00:06, 00:00:07 will be expired. This property determines the expiration of current data points in future windows, and thus enables us to pre-handle the impact brought by these expirations on future patterns.

By using this insight for objects' lifespan , we can avoid the computational effort needed for discounting the effect of such expired data points from the detected clusters. The idea is to pre-generate the partial patterns for the future windows based on the data points that are in the current window and known to participate in those future windows (without considering the to-be-expired ones). Then when the window slides, we can simply use the new data points to update the pre-generated patterns in the predicted views and form the up-to-date patterns in each window. As an example, Figures 4.1 and 4.2 respectively demonstrate examples of the "pre-generated" clusters in future windows and the updated clusters after the window slides. The black, grey and white circles represent the core, edge and noise points identified in each predicted view. The lines among any two data points represent the neighborships between them.

In general, the predicted view technique can helps us to eliminate the effort needed for handling the expired objects during incremental neighbor based pattern detection. We will show how this general technique can be applied to the three neighbor-based pattern types that we are focused on in the following sections.

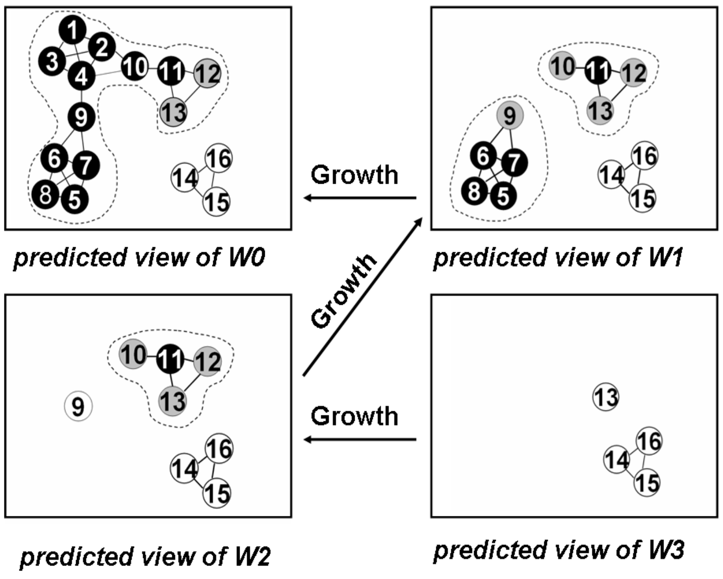

Figure 4.1: Predicted views of four consecutive windows at W_0

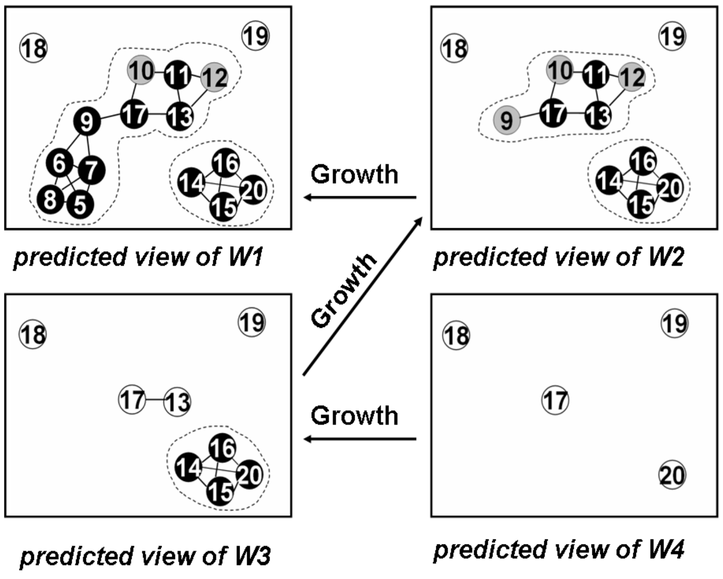

Figure 4.2: Updated predicted views of four windows at W_1

Chapter 5

Proposed Algorithms for Density-Based Clusters

5.1 Abstract Neighborship Maintenance – Abstract-C

Different from Exact-N, we now propose a solution that maintains a compact summary of the *neighborships*, namely the count of the neighbors for each data point. We call it **Abstract-C**. In some cases, these neighbor counts provide sufficient information for generating the patterns.

Challenges. However, maintaining neighbor counts for each data point appears to be not computationally cheaper than the maintenance of their neighbor lists. Since the data points in Abstract-C no longer maintain the exact *neighborships* between each other, they lose the direct access to their neighbors. Thus, expired data points cannot broadcast their expirations to their neighbors without re-running expensive range query searches to figure our who their neighbors are. Obviously, this will largely increase the computational costs required at each window. Therefore, a solution that keeps data points aware of their neighbors' expiration without the help of direct links among them is needed.

Solution. Fortunately, the "predictability" property introduced in Definition 4 provides us with a mechanism to tackle this problem. The key idea is that since we can predict the expiration of any data point p_i, we can pre-handle the impact of p_i's expiration on its neighbors' neighbor counts at the time when they are first identified to be neighbors.

We introduce the notion of a "lifetime neighbor counts" (*lt_cnt*). The "lifetime neighbor counts" of a data point $pi.lt\_cnt$ correspond to a sequence of "predicted neighbor counts", each corresponding to the number of "predicted neighbors" p_i has in a particular future window that p_i will participate in . For example, at a given window W_i, a data point p_i has 3 neighbors in it, which are p_1, p_2 and p_3. By using the "predictability", we could figure out the lifespan of each of these neighbors as well as that of p_i. Let's assume p_1 will expire after W_i. p_2 and p_3 will expire after W_{i+1}. p_i will expire after W_{i+2}. Then, at W_i, $p_i.lt\_cnt$ = (W_i : 3-W_{i+1} : 2-W_{i+2} : 0) indicates that p_i currently has 3 neighbors in W_i, while at (W_{i+1}), 2 of these 3 neighbors, namely p_2 and p_3 will still be its neighbors (p_1 will no longer be p_i's neighbor then as it will expire after W_i). In other words, at W_i, p_i has 2 "predicted neighbors" in W_{i+1}. The length of $p_i.lt\_cnt$ is kept equal to $p_i.lifespan$, and thus decreases by one after each window slide by removing the left most entry. In this example, the W_i : 3 entry will be removed after the window slide. Here we note that all the "predicted neighbor counts" in $p_i.lt\_cnt$ are calculated based on the p_i's neighbors in current window and will later be updated when new data points join its neighborhood. More precisely, each entry on $p_i.lt\_cnt$ records the number of p_i's current neighbors that are known to survive in the corresponding future window.

Lemma 5.1 ***At any given window W_i, the entries in $p_i.lt\_cnt$ obey a monotonic decreasing function pattern.***

The proof of Lemma 5.1 is obvious, because less and less neighbors of p_i in the current window can survive as the window slides.

When later a new data point p_j joins p_i's neighborhood, both $p_i.lt\_cnt$ and $p_j.lt\_cnt$ will be updated. In particular, when p_i and p_j are identified as neighbors, we add 1 to the entries of both $p_i.lt\_cnt$ and $p_j.lt\_cnt$, corresponding to all windows in which both will participate. For example, given $p_j.lt\_cnt$ = ($W_i : 5 - W_{i+1} : 2 - W_{i+2} : 2 - W_{i+3} : 1 - W_{i+4} : 1$) before the update, the *lt_cnt*s of p_i and p_j will be updated to $p_i.lt\_cnt$ = (W_i:4-W_{i+1}:3-$W_{i+2} : 1$) and $p_j.lt\_cnt$ = ($W_i : 6$-W_{i+1}:3-W_{i+2}:3-W_{i+3}:1-$W_{i+4} : 1$). The W_{i+3} and W_{i+4} entries will not be increased as p_i will expire before them. At each window slide, each new data point is associated with a *lt_cnt* with all its entries initialized to zero. Then, each of them runs a range query search to update its own *lt_cnt* and those of its neighbors.

At the output stage, unfortunately, *lt_cnt* does not provide sufficient knowledge to generate the density-based clusters. This is because, although we could know all *core points* in the window, we do not know which of them are within the same clusters. Abstract-C acquires such information by running an extra range query for each *core point* in the window in a Depth First Search manner to reconstruct the clusters. The pseudo code of Abstract-C is shown in Figures 5.1 and 5.2.

Discussion. Abstract-C achieves linear [1] (in the number of data points in the window) memory consumption by maintaining the abstracted *neighborships* (neighbor count) for each data point only. Also, it takes N_{new} (the minimum number) range query searches at each window for neighborship maintenance. However, since Abstract-C takes N_{core} extra range query searches (totally $N_{new} + N_{core}$) for detecting density-based clusters at each window, its performance largely depends on N_{core} the number of *core objects* in the window, which can vary from 0 all the way to N. This instability in CPU performance for the cluster pattern query class is the main shortcoming of Abstract-C. Both our analytical and experimental studies confirm this shortcoming (See Sections 8.1 and 9.2).

5.2 Exact+abstracted *neighborship* Based Solution (Extra-N)

Although Abstract-C achieves linear memory consumption, the extra range query searches may make it inefficient in terms of CPU time when detecting density-based clusters. Hence, we now design the third solution, Abstract-M, which aims to reduce the number of range query searches needed for detecting density-based clusters, while still keeping the linear memory utilization.

[1]The length of *lt_cnt* for each data point is equal to a constant number $C_{ils} = \lceil \frac{Q.win}{Q.slide} \rceil$.

```
Abstract-C ($\theta^{range}$,$\theta^{cnt}$ / $\theta^{fra}$)
1 At each window slide
//Purge
2 For each expired data point $p_{exp}$
3   purge $p_{exp}$;
//Load
5 For each new data point $p_{new}$
6   Initialize_lt_cnt ($p_{new}$)
7   load $p_{new}$ into index
//Neighborship Maintenance
8 For each new data point $p_{new}$
9   $Neighbors$
      = RangeQuerySearch($p_{new}, \theta^{range}$)
10  For each data point $p_j$ in $Neighbors$
11    Updatelt_cnt ($p_{new}, p_j$)
//Output
12 OutputPatterns(pattern type);
Initialize_lt_cnt ($p_i$)
1 For n=1 to $p_i.lifespan - 1$
($p_i.lifespan = \lceil \frac{p_i.T - Window.T_{start}}{Window.Slide} \rceil$)
2 $p_i.lt\_cnt[n] = 0$;
Updatelt_cnt ($p_i, p_j$)
1   For n=1 to $Len(p_j.lt\_cnt)$
2     $p_i.lt\_cnt[n]$++ ;
3     $p_j.lt\_cnt[n]$++ ;
```

Figure 5.1: Pseudo-Code for Abstract-C Part 1

```
OutputPatterns(Distance-Based Outliers)
1 For each data point $p_i$ in the window
2 If $p_i.lt\_cnt[0] \leq \theta^{fra} * N$
  //N is num of tuples in the current window
3   Output($p_i$) ;
4   remove $p_i.lt\_cnt[0]$ ;
OutputPatterns(Density-Based Clusters)
1 ClusterId=0;
2 For each data point $p_i$ in the window
3   If $p_i.lt\_cnt[0] \geq \theta^{cnt}$
4     remove $p_i.lt\_cnt[0]$ ;
5     If $p_i$ is unmarked"
6       OutputCore($p_i$, ClusterId);
7       ClusterId++;
OutputCore($p_c$, ClusterId)
1 mark $p_c$ with ClusterId;
2 output($p_c$);
3 $Neighbors$ =
    RangeQuerySearch($p_c, \theta^{range}$)
4 For each data point $p_j$ in $p_c.neighbors$
5   If $p_j$ is unmarked
6     If $p_j$ has No less than $\theta^{cnt}$ neighbors
7       OutputCore($p_j$, ClusterId)
8     Else
9       mark $p_j$ with ClusterId;
10       output($p_j$);
```

Figure 5.2: Pseudo-Code for Abstract-C Part 2

5.2.1 What Abstract-C Suffers From : "Amnesia"

We note that the extra range query searches needed in Abstract-C are caused by its "amnesia". At every window, Abstract-C requires each new data point to run a range query to determine the *core points* in the window. Then, in addition, it requires each *core point* to run an extra range query search to produce the exact clusters. Unfortunately, the abstracted *neighborship* maintenance in Abstract-C, namely the lt_cnt, does not have the capability to preserve the cluster structures identified in the previous window. In spite of the fact that, in most of the cases, these cluster structures will not only hold for a single window but for multiple continuous windows. For example, a group of data points which comes into the system at the same window may correspond to "life-time neighbors". They may form and thus always belong to the same cluster that will hold throughout their life span even without considering any other data points. Thus, repeatedly running range query searches to re-identify the existing cluster structures is a huge waste in terms of CPU time. So, to relieve the "amnesia" of Abstract-C, we propose to enhance the abstracted *neighborship* maintenance mechanism to capture and preserve the existing cluster structure. We call this enhanced solution Abstract-M.

5.2.2 Enhanced Abstracted *Neighborship*

Abstract-M summarizes the *neighborship* among data points using a higher level abstraction, namely by means of the cluster membership. Specifically, Abstract-M marks the data points found to be in the same clusters with the same cluster IDs, and thereafter preserves such markings for later windows. As an abstracted *neighborship* maintenance mechanism, such marking strategy avoids the "pair-wise" style *neighborship* storing structure applied in Exact-N that may require quadratic memory.

Although marking cluster memberships for data points at the initial window is straightforward, the maintenance of these memberships must now be carefully examined. Here we first identify all the possible changes on density-based cluster structures that may require updates on the cluster memberships of data points. With the aim to make the cluster memberships incrementally maintainable at any single update to the window, we examine change types based on both types of singe updates, namely a removal (expiration) of an existing data point or an addition (participation) of a new data point. We call them *negative changes* and *positive changes* respectively. An update may cause no change on the existing cluster structures.

Negative Changes:

split: The members of an existing cluster now belong to at least two different clusters.

death: An existing cluster loses all its cluster members.

shrink: An existing cluster loses cluster member(s), but no split nor death happens.

Positive Changes:

merge: The cluster members of at least two different existing clusters now belong to a single cluster.

expand: An existing cluster gains at least one new member, but no merge happens.

birth: A new cluster rises, but no merge happens.

These six change types cover all the possible changes that could be caused by any single update to the window. The *negative* and *positive changes* are mutually exclusive to each other, meaning the removal may cause *negative changes* only, while insertion may cause *positive changes* only.

5.2.3 Challenges for Maintaining Cluster Memberships

After a careful examination of the costs of handling each change type, we found that the most expensive operation for incremental cluster membership maintenance lies in handling of *negative changes*. We conclude our analysis by identifying the following challenge:

Observation 5.1 *We have a dilemma on the problem of determining and handling the* ***negative changes*** *on the density-based cluster structures. This is because to determine the specific cluster members affected by the removal of a data point consumes either large of amount of memory or CPU resource. In particular, it needs exact neighborships between the removed data points and their neighbors, which are highly memory-consuming, or large numbers of range query searches, which could be very CPU expensive.*

A key challenge for discounting the effect of expired data points lies in the detection and handling of the *split* of a cluster. In particular, when the expiration of data points causes a cluster to be *split*, the remaining data points in this split cluster need to be relabeled with different cluster memberships as they then belong to different clusters. In Figure 5.3, the transaction from W_0 and W_1 show an example of a split cluster. The expiration of data point 2 causes the cluster composed of *core points*, data points 6, 8 and 12 in W_0 to be split into two clusters, each containing only one *core point*. The expiration of only very few data points may cause a total break of the existing cluster structures into many small pieces, each may continue to persist as a smaller cluster or even may completely degrade to *noise*. Such *split* detection is non-trivial as elaborated upon below.

Observation 5.2 *Given connection information (links) among data points, the problem of detecting a split of a density-based cluster can be mapped to the graph-theoretic problem of identifying "cut-points" in a connected graph [Mun00]. The complexity of this problem is known to be $O(n^2)$, with n the number of vertices in the connected graph (in our case the number of core points in a cluster).*

Moreover, our problem is harder than the problem of identifying the "cut-points", because we do not even have the explicit connection information, namely the exact *neighborships*, among the data points in hand. Without such connection information, we have to again re-run expensive range query searches for every *core point* in the window whenever the window slides. Obviously, this will make Abstract-M no better than Abstract-C and thus defeats the purpose of the Abstract-M design.

5.2.4 View Prediction for Cluster Membership Maintenance

We now demonstrate how the "predictability" property (Definition 4) can once again be exploited to address the dilemma described in Observation 5.1. Specifically, by Observation 5.1, given the data points $W_n.Members$ in the current window W_n, we always know the different subsets of $W_n.Members$

that will participate in each future window. This will enable us to predetermine the cluster structures that will surely appear in each future window based on the data points in the current window. We call such prediction about the characteristics of future windows "predicted views". Figure 5.3 gives an example of the data points falling into the current window W_0. Given these data points in W_0 and the window size $Q.win = 4$ (time units) , the "predicted views" of the subsequent windows of W_0 (until all the data points W_0 expire), namely W_1, W_2 and W_3, are also shown in this figure respectively. Here, the number on each data point indicates its time stamp.

With such "predicted views", we can maintain the cluster structures in each future window **independently**. We call this technique "view prediction". This "view prediction technique is a general principle that can be equally applied to other pattern types, such as graphs, in streaming window semantics.

For density-based clustering, in particular, we "premark" each of the data points with the "predicted cluster membership" for each future window in its life span, if it belongs to any cluster in the corresponding windows. Then, at each window slide, we can simply update the "predicted views" by adding the new data points to each of them and then handling the potential *positive changes* caused by these additions. More specifically, for each window W_i, we update the "predicted cluster memberships" of each data point if they are involved in any *positive change* in this window caused by the participation of the new data points. Figure 5.4 demonstrates the updated views of W_1, W_2, W_3 and W_4, which are computed independently when the new data points join W_1. By doing so, all the expirations are predicted and preprocessed. Based on this foundation provided by the view prediction technique, we develop the following lemma.

Lemma 5.2 *By using the "view prediction" technique to incrementally maintain the cluster memberships for density-based clusters, we eliminate the need to discount the effect of expired data points to extracted clusters. Thus we simplify the incremental density-based cluster detection to the much simpler problem of handling the addition of new data points only.*

Proof 5.1 *We pre-handle the expiration of data points by not using them for cluster formation in the windows that they will not participate in. Therefore, no maintenance to the cluster structures is needed for these windows when those "not-used" objects are purged.*

Fortunately, handling the insertion of new data points is much easier then removal. We will discuss the specific maintenance process for each type of *positive change* after the introduction of the data structure we will use in subsection 5.2.6.

5.2.5 A Stepping-Stone Algorithm: Abstract-M

Before introducing our ultimate solution, we design a stepping-stone algorithm, called Abstract-M. This algorithm uses both the "view prediction" techniques and abstract *neighborship* (cluster membership) maintenance. By maintaining the cluster memberships of data points in predicted views, Abstract-M avoids the expensive cost for handling *negative changes* on density-based clusters. Also, since the cluster memberships maintained

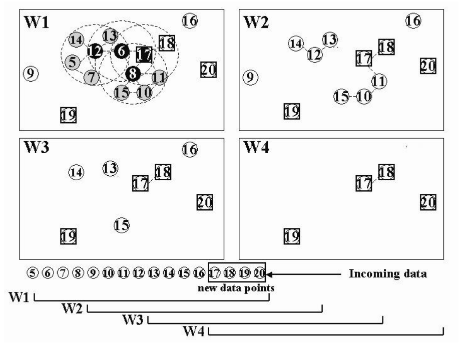

Figure 5.3: "Predicted Views" of 4 Successive Windows at W_0

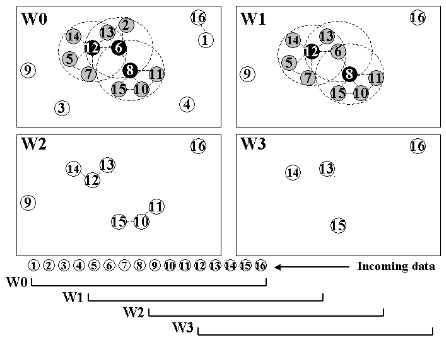

Figure 5.4: Updated "Predicted Views" of 4 Successive Windows at W_1

by Abstract-M indeed capture and preserve the cluster structures, Abstract-M no longer needs one extra query search for each *core point* at the output stage as Abstract-C does to re-build the clusters.

While a significant step forward, Abstract-M does not completely "cure" the "amnesia" suffered by Abstract-C, as it still requires a certain number of extra range query searches, namely $N_{prmtcore}$ (number of existing data points that are "promoted" as new cores) extra range query searches at each window. This is because, as we analyzed earlier, the newly arrived data points may promote the existing *non-core points* to become *core points*. In such cases, the promoted *core points* need to communicate with their neighbors about their new "roles" and thus update the cluster memberships, such as two clusters merge. However, as Abstract-M only maintains cluster membership for each data point, the promoted *core points* have no direct access to their neighbors and thus each of them needs a range query search to broadcast its new role to its neighbors.

Considering the expensiveness of range query searches and the truth that $N_{new} + N_{prmtcore}$ could be as large as N even when N_{new} is very small, Abstract-M does not make the ideal solution that keeps the number of range query searches minimal (N_{new}) and the memory consumption linear. The reason for this is that the enhanced abstracted *neighborship* maintenance mechanism, namely the cluster memberships, still cannot completely represent the *neighborships* among the data points. The data points marked with cluster memberships still have no knowledge about who are their neighbors.

5.2.6 Proposed Solution: Extra-N

By carefully analyzing the strengthes and weaknesses of prior algorithms, we finally propose an ideal solution achieving the merits in terms of both memory and CPU utilization, based on a more capable *neighborship* maintenance mechanism.

Challenges. To achieve the minimum number of range query searches (N_{new}) at each window, we need to completely avoid re-searching for any *neighborships* that have been identified before. This indicates that we have to give data points direct access to their neighbors whenever communication between them is needed. However, this leads to a dilemma in the design of the *neighborship* maintenance mechanism as explained below.

Observation 5.3 *On one hand, to give data points direct access to their neighbors, we have to preserve all the exact neighborships identified in earlier windows. On the other hand, to keep the memory consumption linear, we cannot afford to* *store the exact neighborships in the window.*

Accommodating these two conflicting goals within a single *neighborship* maintenance mechanism is the key challenge that we need to address for our algorithm design.

Solution. We now propose a strategy that successfully tackles this problem by achieving optimality in both memory and CPU consumption. We call this the Exact+abstracted Neighborship based solution (Extra-N). Extra-N combines the *neighborship* maintenance mechanisms proposed in Exact-N, Abstract-C and Abstract-M into one integrated solution. This solution overcomes the shortcomings of the prior solutions while keeping their respective benefits.

We observe that different types of *neighborship* abstractions are most useful during different stages of a data point's life-span. In particular, we need to maintain the exact *neighborships* for a data point in its "non-core point career", while abstracted *neighborships* will be sufficient for its "core point career". More precisely, Extra-N marks each data point p_i by a cluster membership in each window in which it is predicted to be *core point*, while it keeps the exact *neighborships* (links) to all p_i's predicted neighbors for the windows where p_i is predicted to be a *noise* or an *edge point*. Such hybrid *neighborship* maintenance mechanism carries sufficient information to produce the density-based clusters, because all the *core points* in a window W_i are marked with a cluster membership, and all the *edge points* can quickly figure out their cluster memberships by checking those of the *core points* in their neighbor list. We will next demonstrate that Extra-N employs only the minimum number of range query searches while keeping the memory consumption linear.

Data Structure. As mentioned earlier, Extra-N combines the *neighborship* maintenance mechanisms used by all previous three algorithms discussed in this work. In particular, Exact-N inherits the two "life time marks" from Abstract-M, namely lt_cnt and lt_type. In addition, Extra-N introduces a new "life time mark" called "life time hybrid *neighborship* (lt_hn), which stores the "predicted cluster memberships" and the "predicted neighbors" of a data point across different windows in a compact structure. We call the overall data structure composed of lt_cnt, lt_type and lt_hn the Hybrid *neighborship* Mark (*H-Mark)*) for a data point. Figure 5.5 depicts the *H-Marks* of the data points in Figure 5.3. As shown in Figure 5.5, we use

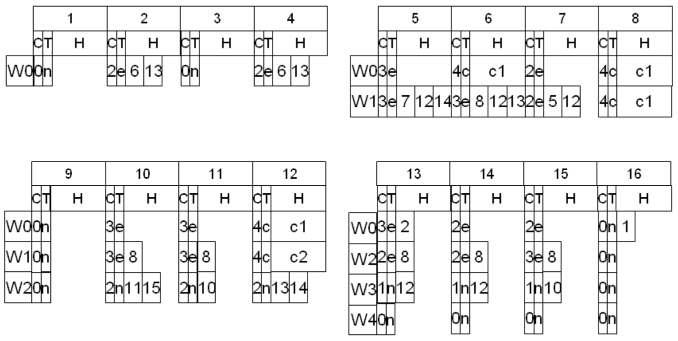

Figure 5.5: The H-Marks of the Data Points at W_0

the columns named C, T and H to present the lt_cnt, lt_type and lt_hn of each data point respectively. Since lt_cnt has been carefully discussed in Chapter 5.1 and lt_type is easy to understand, here we explain lt_hn. For example, at W_0, the *core point* 12 is predicted to be *core point* also in W_1. Thus it is marked by cluster memberships in both windows ($p_{12}.lt\_hn[0]$ = "$c1$",

$p_{12}.lt_hn[0] = $ "$c2$"). Then, as it is predicted to be a *non-core point*, more precisely, a *noise* in W_2, we start to keep the predicted neighbors of p_{12} from this window ($p_{12}.lt\_hn[2] = p_{13}, p_{14}$). Since the number of "predicted neighbors" of a data point follows a monotonically decreasing function (discussed earlier in Chapter 5.1) , the "non core object career" windows of a data point are continuous and right after its "core object career" windows. Here we note that although we maintain the neighbor lists of each data point p_i for all its "non-core point career" windows, the link to each of these neighbors is only physically stored once in lt_hn, no matter how many times it appears in $p_i's$ neighborhood in different windows. This means that the number of predicted neighbors each data point p_i keeps track of is equal to the maximum number of predicted neighbors it has among all its "non-core point career" windows. Given monotonicity, this is equal to the number of predicted neighbors it has in its first "non-core point career" windows. For example, data point 13 in Figure 5.5 has in total 3 predicted neighbors, namely data points 2, 6, and 12, in its first "non-core point career" window W_0. At the same moment, its predicted neighbors in later windows are subsets of these three. For ease of expiration, a predicted neighbor p_j of the data point p_i is stored in the specific row of p_i's *H-Mark* corresponding to the last window in which their *neighborship* will hold.

Lemma 5.3 *Extra-N has the memory consumption linear in the number of data points in the window.*

Proof 5.2 *Since the maximum number of predicted neighbors of each "non core point" p_i is less than the constant θ^{cnt} (otherwise p_i would have been classified* *as core point), and we already know that $p_i.lt\_cnt$, lt_type and lt_hn all have a constant length $\leq C_{ils}$ (defined in Chapter 5.1), H-Mark of any data point is of a constant size. This proves Lemma 5.3.*

Algorithm. Similar to Abstract-M, at each window slide, Extra-N runs a range query search for each new data point to update the "predicted views" of future windows. However, the hybrid *neighborship* maintenance mechanism brings the key advantage to Extra-N of eliminating any extra range query searches from the updating processes. That is when *promotions* happen to the *non core points*, they now have direct access to their neighbors. Thus the *promoted cores* no longer need to run any range query search to re-collect their neighbors.

Lemma 5.4 *Extra-N achieves the minimum number of range query searches needed for detecting density-based clusters at each window.*

Proof 5.3 *First, since Extra-N inherits the neighborship maintenance mechanism from Abstract-M, it needs at most N_{new}+$N_{prmtcore}$ range query searches at each window as Abstract-M does. Second, we know that the $N_{prmtcore}$ range query searches are caused by the handling of promotions. Lastly, no range query search is needed when the promotions happen in Extra-N. Thus, Extra-N only needs N_{new} queries at each window. This proves Lemma 5.4.*

The pseudo code of Extra-N is shown in Figure 5.6.

Conclusion.

Finally, we conclude the optimalities in terms of both CPU and memory utilization achieved by Extra-N algorithm in the following theorem.

```
Extra-N ($\theta^{range}$,$\theta^{cnt}$)
1  For each Window Slide
2      For each expired data point $p_{exp}$ // Purge
3          purge $p_{exp}$;
4      For each new data point $p_{new}$ // Load
5          InitializeHMark ($p_{new}$)
6          load $p_{new}$ into index
7      For each new data point $p_{new}$ // neighborship Maintenance
8          $Neighbors$ = RangeQuerySearch($p_{new}, \theta^{range}$)
9         UpdateHMark ($p_{new}, Neighbors$)
10     OutputPatterns(PatternType); // Output
InitializeHMark ($p$)
1  $length := \lceil \frac{p.T - Window.T_{start}}{Window.Slide} \rceil$ ;
2  set the lengthes of $p.lt\_cnt$, $lt\_type$ and $lt\_hn$ to $length$;
3  For n:=1 to $length$ do
4      $p.lt\_cnt[i] := 0$;
5      $p.lt\_type[i] :=$ "$n$";
6      $p.lt\_hn[i] :=$ "$empty$";
UpdateHMark ($p, Neighbors$)
1  For i:=1 to $Len(p.lt\_hn)$
2    For j:=1 to $Len(Neighbors)$
3      If $Len(Neighbors[j].lt\_hn) < i$
4        remove $Neighbors[j]$ from $Neighbors$
5      Else If $Neighbors[j]$ is NOT New
6        $Neighbors[j].lt\_cnt[i] + +$ ;
8        add $p$ to $Neighbors[j].lt\_hn$ if not added ;
9        add $Neighbors[j]$ to $p.lt\_hn$ if not added ;
9        If $Neighbors[j].lt\_cnt[i] \geq \theta^{cnt}$
10         Mark($Neighbors[j]$, $i$);   11   $p.lt\_cnt[i] := Len(Neighbors)$;
12   If $p.lt\_cnt[i] \geq \theta^{cnt}$
13     Mark($p$,$i$);
Mark($p$,$i$)
1  $p.lt\_type[i] :=$ "$c$";
2  $tempH =$ "$empty$";
3  For each $p$'s predicted neighbor $p_j$;
4      If $p_j.lt\_type[i] =$ "$c$" AND $tempH \neq p_j.lt\_hn[i]$
5          equalize $tempH$ with $p_j.lt\_hn[i]$ ;
6      $tempH := p_j.lt\_hn[i]$;
7  If $tempH = unmarked$
8     $tempH := ClusterId[i]$;
9     $ClusterId[i] + +$;
10 For each $p$'s predicted neighbor $p_j$;
11      If $p_j.lt\_type[i] =$ "$n$";
12          $p_j.lt\_type[i] :=$ "$e$" ;
12          $p_j.lt\_hn[i] := tempH$;
13 remove all the pointers in $p.lt\_hn[i]$ (if any);
14 $p.lt\_hn[i] := tempH$;
OutputPatterns(Density-Based Clusters)
1  For each data point $p_i$ in the window
2      If $p_i.lt\_type[1] \neq$ "$n$"
3          output($p_i$);
4      remove first elements on $p_i.lt\_cnt$, $p_i.lt\_type$ and $p_i.lt\_hn$;
```

Figure 5.6: Psuedo Code of Extra-N Algorithm

Theorem 5.1 *For detecting density-based clusters, Extra-N requires the minimum number of range query searches needed by this problem at each window (by Lemma 5.4), while keeping the memory consumption linear in the number of data points in the window (by Lemma 5.3).*

These properties make Extra-N a very efficient solution for detecting density-based clusters over sliding windows in terms of both CPU and memory resource utilization.

Chapter 6

Proposed Algorithms for Distance-Based Outliers

The same Abstract-C algorithm that we propose in Chapter 5.2 can be slightly adapted to detect distance-based outliers as well. In particular, the only change that we need to make is at the output stage of the Abstract-C algorithm. Instead of repeatedly running range query searches to construct the cluster structures, we can simply output the data points with less than $Winfra$ as the outliers. More specifically, at each window slide, Abstract-C only requires one range query search for each new data point to update the neighbor counts of the data points in the window. Then, at the output stage, it only needs a single scan to the data points in the window to output the outliers. The pseudo-code for the Abstract-C algorithm is also shown in Figures 5.1 and 5.2

In conclusion, the efficient neighborship maintenance process together with the simple output stage make Abstract-C a very efficient algorithm for distance-based outlier detection in sliding windows.

Chapter 7

Proposed Algorithms for kNN

7.1 Equivalence to Top-k Problem

Here, we first explain that the kNN detection in sliding windows that we tackle in this work (defined in Definition 3 in Chapter 2) is equivalent to the top-k object mining problem in sliding windows as defined in my work [YSRW11].

In particular, in the top-k object mining problem in sliding windows [YSRW11], the top-k mining query continuously detects the k objects that have the highest ranking among all the valid objects in the window based on an user-specified preference function F. This preference function F can be any complex function based on the attribute values of each object p_i. It gives each object p_i a preference score $F(p_i)$ based on the attribute values of p_i, which does not change over time.

In the kNN detection problem defined in Definition 3, we continuously detect the k objects that are closest to the query object p_q, whose position is

static, from the valid objects in the query window. Same as the preference function F in top-k query, the distance function $Dist(p_i, p_q)$ used to calculate distance between an object p_i to the query object q_i in the kNN query are user specified and can be any complex function based on the attribute values of p_i. Since the position of p_q is static, indicating that the attribute value of p_i are only variables in function $Dist(p_i, p_q)$, $Dist(p_i, p_q)$ are actually a function F' of p_i only. So, for any kNN query, given the distance function $Dist(p_i, p_q)$ and the position of p_q, we can define a preference function F calculating the distance between p_i and p_q as p_i's preference score. In other words, $F(p_i)$ in top-k query and $Dist(p_i, p_q)$ in kNN query are mutually transferable. Thus, since both top-k query and kNN query seek for the k objects having highest ranking (by either $F(p_i)$ or $Dist(p_i, p_q)$). Therefore, these two problems are in fact equivalent to each other, both seeking for the k objects having highest ranking in the query window.

Given this fact, we will discuss our proposed solutions for top-k mining queries in the following sections. These solutions can be equally used to process kNN queries.

7.2 Theoretical Foundations

Minimal Top-k Candidate Set As we have analyzed in Chapter 3.3, the basic incremental computation strategy for top-k (kNN) queries suffer from the from-scratch-recomputation problem. To solve this problem, we can again use the predicted view technique proposed in Chapter 4. Using the predicted view technique, we can pre-determine the specific subset of the current objects that will participate (be alive) in each of the future windows. This enables us to pre-generate the partial query results for future windows, namely the "predicted" top-k result for each of the future windows. They would be based on the objects in the current window but have already taken the expiration of these objects in future windows into consideration.

Figure 7.1 shows an example of a top-k result for the current window and predicted top-k results for the next three future windows ($Q.win = 16$, $Q.slide = 4$). Each object is depicted as a circle labeled with its object id. The position of an object on the Y-axis indicates its F score, while the position on the X-axis indicates the time it arrived at the system.

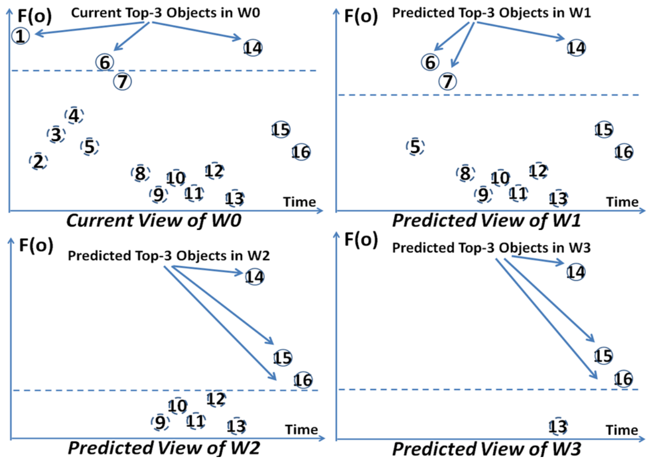
F(o)
Current Top-3 Objects in W0
Current View of W0
Predicted Top-3 Objects in W1
Predicted View of W1
Predicted Top-3 Objects in W2
Predicted View of W2
Predicted Top-3 Objects in W3
Predicted View of W3
Time

Figure 7.1: (Predicted) Top-k Results Four Consecutive Windows at time of W_0 (slide size = 4 objects)

Based on the objects in W_0, we not only calculate the top-k (k=3) result in W_0, but we also pre-determine the potential top-k results for the next three future windows, namely W_1, W_2 and W_3, until the end of life spans of all objects in W_0. In particular, the top-k results for the current window W_0 are generated based on all 16 objects in W_0, namely objects o_1 to o_{16}. While the predicted top-k results for future windows are calculated based on smaller and smaller subsets of objects in W_0, namely the predicted top-k results for W_1, W_2 and W_3 are calculated based on object o_5 to o_{16}, o_9 to o_{16} and o_{13} to o_{16} respectively. All other objects belonging to W_0 but determined to not fall into the (predicted) top-k results for any of these (future) windows (from W_0 to W_3 in this case) can be discarded immediately.

In the example shown in Figure 7.1, only the 7 objects within the predicted top-k results for the current and the three future windows (depicted using circles with solid lines) are kept in our system, while the other 9 objects (depicted using circles with dashed lines) are immediately discarded. The latter are guaranteed to have no chance to become part of top-k result throughout their remaining life spans. On the left of Figure 7.3, we list the predicted top-k results maintained for these four windows. Although these pre-generated top-k results are only "predictions" based on our current knowledge of the objects that have already arrived so far, meaning that they may need to be adjusted (updated) when new objects come in, they guarantee an important property as described below.

Theorem 7.1 *At any time, the objects in the predicted top-k result constitute the* ***"Minimal Top-K candidate set" (MTK)****, namely the minimal object set that is both necessary and sufficient for accurate top-k monitoring.*

Proof 7.1 *We first prove the* ***sufficiency*** *of the objects in the predicted top-k results for monitoring the top-k results. For each of the future windows W_i (the ones* *that the life span of any object in the current window can reach), the predicted top-k results maintain k objects with the highest F scores for each W_i based on the objects that are in the current window and are known to participate in W_i. This indicates that any other object in the current window can never become a part of the top-k results in W_i, as there are already at least k objects with larger F scores than it in W_i. So, they don't need to be kept. Then, even if no new object comes into W_i in the future or all newly arriving objects have a lower F score, the predicted top-k results would still have sufficient (k) objects to answer the query for W_i. This proves the sufficiency of the predicted top-k results.*

Next we prove that any object maintained in the predicted top-k results are ***necessary*** *for top-k monitoring. This would imply that this object set is the minimal set that any algorithm needs to maintain for correctly answering the continuous top-k query. Any object in the predicted top-k result for a W_i may eventually be a part of its real top-k result. This would happen if no new object comes into W_i or all new objects have a lower F score. Thus discarding any of them may cause a wrong result to be generated for a future window. This proves the necessity of keeping these objects.*

Based on the sufficiency and necessity we have just proved, the objects in the predicted top-k results constitute the "Minimal Top-K candidate set" (MTK), namely the minimal object set that is necessary and sufficient for accurate top-k monitoring.

7.3 An Initial Approach: PreTopk

Now we first introduce the first step toward solving this problem, which is based on incremental maintenance of MTK. We call this approach PreTopk. When the window slides, the following two steps update the predicted top-k results. At step 1, we simply purge the view of the expired window. For example, as shown in Figure 7.3, the top-k result of W_0 in Figure 7.2 is removed, and W_1 becomes the new current view. This simple operation is sufficient for handling the object expiration. At step 2, we create an empty new predicted top-k result for the newest future window to cover the whole life span of the incoming objects. Using our example in Figure 7.3, the newest future window in this case is W_4. Therefore, each new object will fall into the current window and all future windows that we are currently maintaining. We thus update these predicted top-k results by simply applying the addition of each new object.

In particular, when a new object o_{new} comes in, we attempt to insert it into the predicted top-k result of each window. If the predicted top-k result of a window has not reached the size of k yet, we simply insert o_{new} into it. Otherwise we also remove the existing top-k object with the smallest F score once o_{new} is inserted. If it fails, namely the predicted top-k result sets of all future windows maintained have reached size k and $F(o_{new})$ is no larger than the F score of any object in them, o_{new} will be discarded immediately. Again, such computation is straightforward. Figure 7.2 shows the updated predicted top-k results of our running example (Figure 7.1) after the insertion of four new objects.

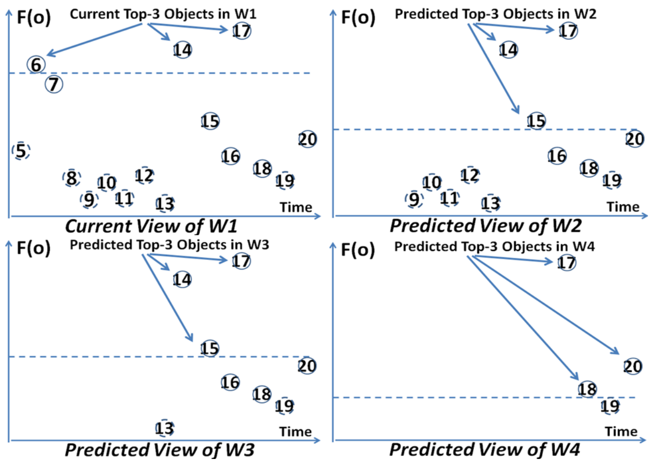
F(o)
Current Top-3 Objects in W1
Current View of W1
Predicted Top-3 Objects in W2
Predicted View of W2
Predicted Top-3 Objects in W3
Predicted View of W3
Predicted Top-3 Objects in W4
Predicted View of W4
Time

Figure 7.2: Updated Predicted Top-k Results of Four Consecutive Windows at time of W_1 (slide size = 4 objects)

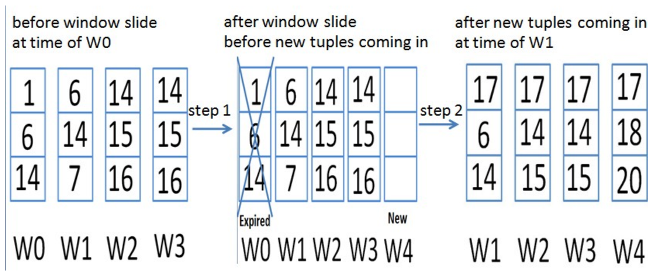

Figure 7.3: Update Process of Predicted Top-k sets From Time of W_0 to W_1

Conclusion. PreTopk solves the recomputation bottleneck suffered by the state-of-the-art solutions [MBP06]. Memory-wise, it only keeps the minimal number of objects necessary for top-k query monitoring, which is shown to be independent of the potentially very large window size (in Theorem 8.1). Computation-wise, the processing costs for generating the top-k result in each window are no longer related to the window size. This is a significant improvement over the state-of-the-art solution [MBP06], because both the processing and memory costs of any solution that involves recomputation are related to the window size, which is usually an overwhelming factor compared to k or the slide size. A in-depth cost analysis of PreTopk can be found in Chapter 8.2.2.

However, the **limitations** of PreTopk are obvious. Its performance is significantly affected by a constant factor, namely C_{nw}, the number of predicted top-k results to be maintained. More precisely, since PreTopk maintains the predicted top-k result for each window independently, both its CPU and memory costs increase linearly with the number of predicted top-k results to be maintained (C_{nw}). Although C_{nw} is a constant, as it is fixed given the query specification and will never change during query execution, it can be large in some cases. For example, if a query Q has a window size $Q.win = 10000$ and a slide size $Q.slide = 10$, PreTopk maintains predicted top-k results for 1000 different windows. Our experimental studies in Chapter 9.4 confirm this inefficiency of PreTopk as the ratio between $Q.win$ and $Q.slide$ increases.

7.4 Proposed Solution: MinTopk

7.4.1 Properties of Predicted Top-k Results

To design a solution whose CPU and memory costs are independent not only from the window size but also the number of future windows to be maintained, we analyze the interrelationships among the predicted top-k results maintained by PreTopk.

Property 1: Overlap. We observe that the predicted top-k results in adjacent windows tend to partially overlap, or even be completely identical, especially when the number of predicted top-k results to maintain (C_{nw}) is large. We now explain the overlap property of the predicted top-k results across multiple windows.

Lemma 7.1 *At any given time point, the predicted top-k result for a future window W_i is composed of two parts: 1) $K_{inherited}$, a subset of predicted top-k objects inherited from the previous window W_{i-1} ; 2) K_{new}, the "new" top-k objects which qualify as top-k in W_i but not in W_{i-1}. $|K_{inherited}| \cap |K_{new}| = \emptyset$ and $|K_{inherited}| + |K_{new}| = k$. Then the following property holds:* ***For any object*** *$o_i \in K_{new}$* ***and*** *$o_j \in K_{inherited}$, $F(o_i) < F(o_j)$.*

Proof 7.2 *If there exists an $o_i \in K_{new}$ with $F(o_i)$ larger or equal to $F(o_j)$ of any object $o_j \in K_{inherited}$, o_i will be in the predicted top-k results for the previous window W_{i-1} and thus $o_i \in K_{inherited}$. As $|K_{inherited}| \cap |K_{new}| = \emptyset$ by definition, this is a contradiction. Thus, there cannot exist any $o_i \in K_{new}$ and $o_j \in K_{inherited}$ such that $F(o_i) > F(o_j)$.*

In the earlier example in Figure 7.1, at time of window W_0, objects 6 and

14 belong to $K_{inherited}$ of W_1, while object 7 belongs to K_{new} of W_1.

When C_{nw} is large, implying that the window moves a small step (compared to the window size) at each slide, only a small percentage of the objects will expire after each window slide. Then, the majority of the predicted top-k result of a window come from $K_{inherited}$. In the previous example, where $Q.win = 10000$ and $Q.slide = 100$, if k=500 objects, at least 80 percent of the predicted top-k objects in a window will be the same as those in the previous window (worst case). On average, this percentage will be even higher, as the expired top-k objects should be only a small portion of all the expired objects. In the example shown earlier in Figure 7.3, at W_0, the current top-k of W_0 and predicted top-k results of W_1 only differ in one object, and those of W_2 and W_3 are in fact exactly the same.

Property 2: Fixed Relative Positions. The relative positions between any two objects o_i and o_j in the predicted top-k result sets of different windows remain the same.

Since the F score for any object is fixed and the predicted top-k objects in any window are organized by F scores, o_i will always have a higher rank than o_j in any window in which they both participate, if $F(o_i) > F(o_j)$.

7.4.2 Solution: Integrated View Maintenance

Given the two properties identified in Section 4.1, we now propose an integrated maintenance mechanism for the sequence of predicted top-k results in future windows. As shown in the cost analysis in Chapter 8.2.1, the major processing costs for PreTopk to maintain top-k results lie in positioning each new object into the predicted top-k results of all future windows.

Thus, our objective is to share the computation for the positioning each new object into multiple predicted top-k results (multiple future windows).

To achieve this goal, instead of maintaining C_{nw} independent predicted top-k result sets, namely one for each window, we propose to use a single integrated structure to represent the predicted top-k result sets for all windows. We call this structure the *super-top-k list*. At any given time point, this *super-top-k list* includes all distinct objects in the predicted top-k results of the current as well as all future windows. The *super-top-k list* is sorted by $F(o)$. Figure 7.4 shows an example of the *super-top-k list* containing the objects in predicted top-k results for four windows.

Next, we tackle the problem of how to distinguish among and maintain top-k results for multiple windows in this single *super-top-k list* structure. As a straightforward solution, for each object, we could maintain a window mark (a window Id) for each of the windows in which the object is part of its predicted top-k result set. We call this the *complete window mark strategy*. Using the example in Figure 7.1, at the time of W_0, object 14 needs to maintain four window marks, namely W_0, W_1, W_1 and W_3, as it is in the predicted top-k results for all these four windows. When an object is qualified for or disqualified from the predicted top-k result set of a window, we would respectively need to add a new or remove an existing window mark from it for the corresponding window.

This solution suffers from a potentially large number of window marks being maintained for each object. Thus, both the addition and removal process of window marks may require traversing the complete window mark lists. This is clearly not desirable.

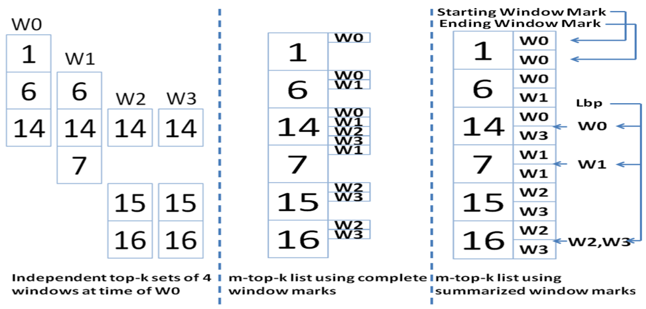
W0
W1
W2
W3
Independent top-k sets of 4 windows at time of W0
m-top-k list using complete window marks
Starting Window Mark
Ending Window Mark
Lbp
W2,W3
m-top-k list using summarized window marks

Figure 7.4: Independent Top-k Result Sets vs. *Super-top-k* Structure using Complete and Summarized Window Marks

7.4.3 Optimal Integration Strategy based on Continuous Top-k Career

To overcome this shortcoming, we observe the following.

Lemma 7.2 *At the time of the current window W_i, the minimal F score of the predicted top-k objects in a future window $W_{i+n}(n > 0)$ is smaller than or equal to that of any window $W_{i+m}(0 \leq m < n)$, $W_{i+n}.F(o_{min\_topk}) \leq W_{i+m}.F(o_{min\_topk})$.*

Proof 7.3 *Since some objects may expire after each window slide, the objects in the current window W_i that will participate in W_{i+n}, D_W_{i+n}, is a subset of those will participate in W_{i+m}, D_W_{i+m} $(m < n)$. Thus, the minimal F score of the top-k objects selected from the object set D_W_{i+n} in W_{i+n} cannot be larger than the minimal F score of the top-k objects selected from a super set of D_W_{i+n}, namely the object set D_W_{i+m} in W_{i+m}.*

Based on Lemma 7.2 we derive the lemma below.

Lemma 7.3 *At any given moment, if an object is part of the predicted top-k result for a window W_i, then at that moment it is guaranteed to be in the predicted top-k* *results for all later windows W_{i+1}, W_{i+2} ... W_{i+j} ($j \geq 0$), until the last window in its life span.*

Proof 7.4 *Since some objects may expire after each window slide, the objects in the current window W_i that will participate in W_{i+n}, D_W_{i+n}, is a subset of those will participate in W_{i+m}, D_W_{i+m} $(m < n)$. Thus, the minimal F score of the top-k objects selected from the object set D_W_{i+n} in W_{i+n} cannot be larger than the minimal F score of the top-k objects selected from a super set of D_W_{i+n}, namely the object set D_W_{i+m} in W_{i+m}.*

This continuous top-k career property in Lemma 7.3 establishes the theoretical foundation for an innovative design of the *super-top-k* list. Namely, we design a more compact encoding for window marks of each object. In particular, for each object, as its "top-k career" is continuous, we simply maintain a *starting* and an *ending window mark*, which respectively represent the first and the last windows in which it is predicted to belong to top-k. As shown on the right of Figure 7.4, the first (upper) window mark maintained by each object is its starting window mark and the second (lower) one is its ending window mark. Clearly, the number of window marks needed for each object is now constant, namely 2, no matter in how many windows it is predicted to belong to the top-k result.

To enable us to efficiently decide in which windows a new object is predicted to make top-k result set when it arrives at the system, we also maintain one *Lower Bound Pointer (lbp)* for each window pointing at the top-k object with the smallest $F(o)$. When a new object arrives, we simply need to compare it with the object pointed by the *lbp* of each window. In the example shown in Figure 7.4, the *lbp* of W_0 and W_1 point to objects 14 and 7 respectively, while those of W_2 and W_3 both point to object 16. We call this the *summarized window mark strategy*. This is not only an important improvement in terms of memory usage but it significantly simplifies the top-k update process, as demonstrated below.

7.4.4 Super-Top-K List Maintenance

Handling Expiration. Logically, we simply need to purge the top-k result for the expired window from the *super-top-k* list. This task seems to be no longer as trivial as in the independent view storage solution (PreTopK), because now the top-k objects for different windows are interleaved within one and the same *super-top-k* list. We may need to search through the list to locate the top-k objects of the expired window. However, the following observation bounds such cost, indicating that searching is not needed.

Lemma 7.4 *At any given time, the top-k objects of the current to-be-expired window are guaranteed to be the first k objects in the super-top-k list, namely the ones with the highest F scores.*

Proof 7.5 *First, since the objects in the super-top-k list are sorted by F scores, the first k objects in super-top-k list are those objects with highest F scores within the whole list. Second, the top-k objects of the current window are selected from all the objects in the current window, while the predicted top-k objects for any future window are selected based on a subset of objects in the current window. Therefore, the top-k objects of the current window must be the ones with highest F scores among the current window. Their F scores cannot be lower than those of the other* *objects belonging to the predicted top-k results of future windows, which constitute the later part of the super-top-k list.*

Thus we can "purge" the first k objects on our *super-topk-list* without search. Note that purging here is only a logical concept to convey that these objects will no longer serve as top-k for this to-be-expired window. However, some of the top-k objects for the expired window may continue to be part of the predicted top-k results in future windows. We cannot simply delete them from the *super-top-k list* without considering their future role.

Instead, when the window slides, we implement purging of the expired window by updating the window marks of the first k objects in *super-top-k* list. More specifically, we increase the starting window mark by 1 for each of these objects. As the "top-k careers" of an object is continuous (Lemma 7.3), such update conveys that these objects will no longer serve as top-k for the expired window and their "top-k career" are predicted to start at the next window. If the starting window mark of an object becomes larger than its ending window mark, with the ending window mark the latest window in which it can survive, we know that this object will have no chance to be part of the top-k result in its remaining life-span. We can thus physically remove it from the *super-top-k* list.

Handling Insertion. For the insertion of a new object o_{new}, we take two steps to update the *super-top-k* list. First, we position it into the *super-top-k* list. Second, we remove the object with the smallest F score from the windows that the new object is predicted to be part of their top-k results.

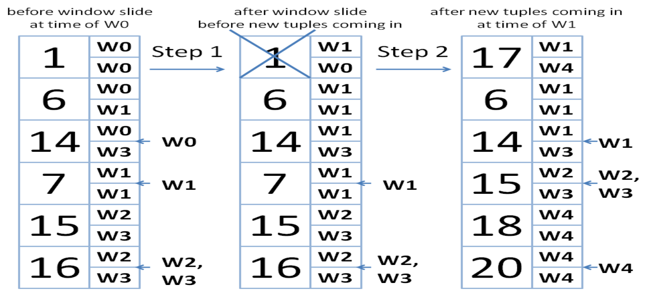

Figure 7.5: Update Process of *super-top-k list* From W_0 to W_1

For the first step, the positioning process has become fairly simple due to the support from the summarized window marks. In particular, for each object, if the predicted top-k result set of any future window represented by the *super-top-k list* has not reached the size of k yet, or if its F score is larger than that of any object in the *super-top-k* list, we insert it into the *super-top-k list* based on its F score. Otherwise it will be discarded immediately. If the new object is inserted into *super-top-k* list, which indicates that it is in the predicted top-k results of at least the last window in its life span, its ending window mark is set to be the window Id of this last window. The starting window mark of a new object is simply the oldest window on the *super-top-k list* whose $F(o_{min\_topk})$ (the F score of the object pointed by its lower bound pointer) is smaller than $F(o_{new})$. We find this window by comparing $F(o_{new})$ with $F(o_{min\_topk})$ of the oldest window on the *super-top-k* list, and keep comparing $F(o_{new})$ with that of the younger ones (with larger window Ids), until we find the oldest window whose $F(o_{min\_topk})$ is smaller than $F(o_{new})$, ($F(o_{min\_topk})$ monotonically decreases as window Id increases in Lemma 7.2). In the example shown in Figure 7.5, the F score of the new object 17 is larger than $F(o_{min\_topk})$ of all windows, from W_1 to

W_4. Thus its starting window mark is set to W_1, indicating that its "top-k career" is predicted to start at W_1.

Second, for each window in which the new object is inserted into its predicted top-k result, one previous top-k object becomes disqualified and thus must be removed. Given that we have an *lbp*, for each window pointing to its top-k object with smallest F score, locating such disqualified object is trivial. For such a disqualified object, as it now serves in one less window as a top-k object, we simply increment its starting window mark by 1. Same as in the purging process, if its starting window mark now becomes larger than its ending window mark, we physically remove it from *super-top-k* list. Objects 7 and 16 are such examples in Figure 7.5.

7.4.5 Final Move Towards Optimality

Now we have the last but also the most challenging maintenance task left. We must design a strategy to efficiently redirect the lower bound pointer (*lbp*) of each window from which the object with the smallest $F(o)$ has just been removed. To do this, we need to locate the object that currently has the smallest $F(o)$ for each of those affected windows on the *super-top-k list*. On first sight, this task seems to require at least one complete search through the *super-top-k list* for each affected window. This is because the objects belonging to different windows are now interleaved in this integrated structure. Thus, we would have to search and locate the objects whose $F(o)$ scores used to be the second smallest one in each window. If so, the searches would make the redirecting process very expensive computationally and thus would significantly affect the overall performance of the MinTopk algorithm. To solve this problem, we carefully analyze the characteristics of the *super-top-k list* and discover the following important property.

Lemma 7.5 *For each window W_i whose predicted top-k result is represented by the super-top-k list, the object with the second smallest F score in its predicted top-k result, $W_i.o_{sec\_min\_topk}$, is always directly in front of (adjacent to) the object with the smallest F score in its predicted top-k result, $W_i.o_{min\_topk}$.*

Proof 7.6 *This lemma can be proven by an exhaustive examination of all possible scenarios. By Lemma 7.1, we know that, at any given moment, the predicted top-k result set for a future window W_i, $W_i.topk$, is composed of two parts: 1) $K_{inherited}$, a set of inherited top-k objects from the previous window W_{i-1} ; 2) K_{new}, a set of "new" objects that qualify as top-k in W_i but did not in W_{i-1}. By Lemma 7.2, any object $o_i \in W_i.K_{new}$ has a lower F score than any object $o_j \in W_i.K_{inherited}$. For the current window W_i, the proof is straightforward. Since the top-k objects of the current window W_i are always the first k objects in super-top-k list (Lemma 7.4), and the objects on the super-top-k list are sorted by F scores, $W_i.o_{sec\_min\_topk}$ is in the $(k-1)^{th}$ position of super-top-k list, and $W_i.o_{min\_topk}$ is in the k^{th} position.*

Now let us consider the next window right after W_i, namely W_{i+1}. There are four possible situations. 1) $W_{i+1}.o_{sec\_min\_topk}$, $W_{i+1}.o_{min\_topk} \in W_{i+1}.K_{inherited}$. It is easy to see that, in this case, $W_{i+1}.K_{new}$ is empty and the top-k objects of W_{i+1} are exactly the same as those in W_i. Thus these two objects are simply the same two top-k objects with the lowest F scores in W_i, and have been shown to be adjacent to each other in the case above. 2)

$W_{i+1}.o_{sec\_min\_topk}, W_{i+1}.o_{min\_topk} \in W_{i+1}.K_{new}$. Since any object $o_i \in W_i.K_{new}$ has a lower F score than any object $o_j \in W_i.K_{inherited}$, we know that these two objects with lowest F scores in $W_{i+1}.K_{new}$ definitely have lower scores than any object in $W_{i+1}.K_{inherited}$. Thus no top-k objects in the previous window can be in between of them two. Also, any "new" predicted top-k object in the next window W_{i+2}, namely any object in $W_{i+2}.K_{new}$, must have a smaller F score than these two objects do, otherwise it would have already made top-k in W_{i+1} and thus would not be in $W_{i+2}.K_{new}$ but in $W_{i+2}.K_{inherited}$. So, no predicted top-k object of any later window can be in between of them. This proves the case for the second situation. 3) $W_{i+1}.o_{sec\_min\_topk} \in W_{i+1}.K_{inherited}$ and $W_{i+1}.o_{min\_topk} \in W_{i+1}.K_{new}$. This case is possible only if exactly one top-k object will expire from W_i. In this case, $W_{i+1}.o_{sec\_min\_topk}$ must be the last one in $W_{i+1}.K_{inherited}$, and $W_{i+1}.o_{min_topk}$ must be the only one in $W_{i+1}.K_{new}$. They are thus also adjacent to each other. 4) $W_{i+1}.o_{min\_topk} \in W_{i+1}.K_{new}$ and $W_{i+1}.o_{sec\_min\_topk} \in W_{i+1}.K_{inherited}$. This case is simply impossible, because any object $o_i \in W_i.K_{new}$ has lower F score than any object $o_j \in W_i.K_{inherited}$ (Lemma 7.1), but clearly $F(W_{i+1}.o_{min\_topk}) > F(W_{i+1}.o_{sec\_min\_topk})$. Now we have covered all four possible situations for W_{i+1}. We thus can prove that, at the same moment, $W_{i+j}.o_{sec\_min\_topk}$ and $W_{i+j}.o_{min\_topk}$ $(j > 1)$ in any future window are also in adjacent positions using the same method.

Using Lemma 7.5, we can now conduct the redirection procedure effortlessly. We simply move the lower bound pointer of each affected window by one position up in the *super-top-k* list. Lastly, after the insertion of all new objects, the first k objects on the *super-top-k list* correspond to the top-k results for the current window and can be output directly. We call this proposed algorithm MinTopK. The pseudo code of MinTopK can be found in Figure 7.6.

```
$o_i$: an object. $o_i.T$: object $o_i$'s time stamp.
$o_i.start\_w/.end\_w$: starting/ending window mark of $o_i$.
$W_i.T_{end}$ : ending time of a window $W_i$.
$W_{exp}$: the window just expired.
$W_{new}$: the newest future window.
$W_i.lbp$: lower bound pointer of $W_i$.
$W_i.tkc$: top-k object counter of $W_i$.
$o_{w_{i.lbp}}$: object pointed by lower bound pointer of $W_i$.
$o_{min\_suptopk}$ : object with smallest $F$ score on super-top-k list.

MinTopk ($S, win, slide, F, k$)
1 For each new object $o_{new}$ in stream $S$
2   if $o_{new}.T > W_{cur}.T_{end}$
      //slide window
3     OutputTopKResults();
4     PurgeExpiredWindow();
    // super-top-k list maintenance
5   UpdateSuperTopk ($o_{new}$)

OutputTopKResults()
1   output first k objects on super-top-k list;

PurgeExpiredWindow()
1   For first k objects ($o_{exp}$) on super-top-k list
2     $o_{exp}.start\_w++$;
3     If $o_{exp}.start\_w > o_{exp}.end\_w$
4       remove $o_{exp}$ from super-top-k list;
5   remove $W_{exp}$;
6   create a new future window $W_{new}$;
7   $W_{new}.tkc := 0$;
8   $W_{new}.lbp := o_{min\_suptopk}$ ;

UpdateSuperTopk ($o_i$)
1 If $F(o_i) < F(o_{min\_suptopk})$ AND All $W_i.tkc == k$
2   discard $o_i$ immediately;
3 Else position $o_i$ into super-top-k list;
4   For each $W_i$ that $F(o_{w_{i.lbp}}) < F(o_i)$
5     If $W_i.tkc < k$
6       $W_i.tkc++$;
5     Else $o_{w_{i.lbp}}.start\_w++$;
6     If $o_{w_{i.lbp}}.start\_w > o_{w_{i.lbp}}.end\_w$ ;
9       remove $o_{w_{i.lbp}}$ from super-top-k list;
10      move $W_i.lbp$ by one position in super-top-k list;
```

Figure 7.6: Proposed Solution: MinTopk Algorithm

Chapter 8

Cost Analysis for Proposed Algorithms

In this chapter, we first discuss the cost analysis for the algorithms detecting density-based clusters and distance-based outliers in Chapter 8.1. This is because the those two neighbor-based pattern types share the same neighbor (graph) definition mechanism as defined in Chapter 2.2, and thus have a similar detection process. Then, we will discuss the cost analysis for the kNN detection in Chapter 8.2.

8.1 Cost Analysis for Density-Based Cluster and Distance-Based Outlier Detection

To thoroughly analyze and compare the algorithms for density-based cluster and distance-based outlier detection, we first design cost models for modeling both the CPU and memory consumption for those two pattern types in sliding window semantics.

8.1.1 Cost Models

We establish a CPU cost model capturing the response time of each algorithm to answer the query in each individual window. Such response time includes all the time consumed by the four stages of the neighbor-based pattern detection process, namely the purging, loading, *neighborship* maintenance and output. The memory cost model is designed to describe the memory space utilized by each algorithm. Such memory utilization includes the memory space for storing both the raw data and also the meta-information in each window.

These cost models are built based on several common assumptions for continuous query execution with sliding window scenarios. They are: 1) Pattern detection for each window W_n starts after all the data points belonging to this window arrive at the system. In particular, we buffer all the new data points until the system (wall clock) time reaches $W_n.T_{end}$ (for time-based windows) or the $Q.win^{th}$ arrives to the system (for count-based windows), and then start the pattern detection process for W_n. 2) The system is running in a stable state, meaning we always have sufficient computational power to finish the pattern detection for a window W_n within the allocated time period, namely between $W_n.T_{end}$ and $W_{n+1}.T_{end}$. 3) To achieve the real-time response for continuous query execution, we avoid the performance penalty of I/O operations by keeping all the raw data and meta-data in the main memory.

We first define the symbols used in our cost models in Table 8.1. These symbols are used to indicate the information for a single window.

Average number of expired data points	N_{exp}
Average number of new data points	N_{new}
Average number of "core points"	N_{core}
Average number of "promoted core points"	$N_{prmtcore}$
Number of neighbors for a specific data point p_i	$N_{(p_i)}$
Average number of data points	N
Average initial life-span for each data point	C_{ils}

Table 8.1: Symbols Used In The Cost Models.

CPU Costs of Alternative Algorithms.

Now we define the CPU costs of primitive operations in the neighbor-based pattern detection processes in Table 8.2.

CPU cost of purging a data point	c_p
CPU cost of loading a data point into index	c_l
CPU cost of removing/establishing an exact *neighborship* (single-directional)	c_n
CPU cost of updating a integer attribute	c_i
CPU cost of running a range query search	c_{rqs}
CPU cost of examine a data point during the output	c_o
CPU cost of updating the *lt_cnt* of a data point	$c_{lt\_cnt} = \frac{C_{ils}}{2} c_i$
CPU cost of updating the M-Table of a data point	$c_{mt} = \frac{3C_{ils}}{2} c_i$
CPU cost of updating the H-Marks of a data point	$c_{hm} = C_{ils} * c_{int} + \frac{\theta^{cnt}}{2} c_i$

Table 8.2: CPU Cost of Individual Operations.

We use the average costs in the estimation of c_{lt_cnt}, c_{mt} and c_{hm}. This is because such cost for a specific data point is decided by its life-span, which indicates the number of "predicted views" that will need to be updated. We assume the life-spans of the data points in each window are uniformly distributed from 1 to C_{ils}. Thus we use $\frac{C_{ils}}{2}$ to present the average. This assumption holds for all count-based window cases and also for time-based window cases if the input rate is stable. The same estimation is used for memory costs of lt_cnt, mt and hm later.

Given the costs of individual operations, we now design models for the CPU cost of each algorithm. Again, the CPU cost is the sum of the cost for the four stages of purging, loading, *neighborship* maintenance and output. We use the symbols, C_{purge}, C_{load}, C_{nei_main} and C_{output} to indicate the cost of each algorithm for these four stages respectively. Also, we use superscript to denote the cost of a specific algorithm and the target pattern type along with these symbols. For example, $C_{purge}^{Exact-N(c)}$ indicates the cost of purging for Exact-N to detect density-based clusters, while $C_{nei\_main}^{abs(o)}$ indicates the cost of *neighborship* maintenance for Abstract-C to detect distance-based outliers. For a given algorithm, if the cost of a certain stage is the same for both of the pattern types (density-based clusters and distance-based outliers), we omit the pattern type part of the superscript and only use the stage name as well as the algorithm name to generalize the cost for this stage. For example, $C_{purge}^{Exact-N}$ indicates the cost of purging for Exact-N to detect either of the two pattern types. Note, algorithms Exact-N, Abstract-C and the naive approach handle both cluster and outlier detection, while Abstract-M and Extra-N support clustering only. The specific costs of each alternative algorithm at each query processing stage are listed in Table 8.3.

C^{naive}_{purge}	$N_{exp} * (c_p)$
C^{naive}_{load}	$N_{new} * c_l$
$C^{naive}_{nei\_main}$	$N_{new} * c_{rqs}$
C^{naive}_{output}	$N * c_o$
$C^{exact-N}_{purge}$	$\sum_{1 \leq i \leq N_{exp}} (\sum_{1 \leq j \leq N_{p_i}} c_n + c_p)$
$C^{exact-N}_{load}$	$N_{new} * c_l$
$C^{exact-N}_{nei\_main}$	$\sum_{1 \leq i \leq N_{new}} (c_{rqs} + \sum_{1 \leq j \leq N_{p_i}} 2 * c_n)$
$C^{exact-N(c)}_{output}$ $C^{exact-N(o)}_{output}$	$N * c_o + \sum_{1 \leq i \leq N_{core}} (\sum_{1 \leq j \leq N_{p_i}} c_o)$ $N * c_o$
C^{abs-C}_{purge}	$N_{exp} * (c_p)$
C^{abs-C}_{load}	$N_{new} * c_l$
$C^{abs-C}_{nei\_main}$	$\sum_{1 \leq i \leq N_{new}} (c_{rqs} + \sum_{1 \leq j \leq N_{p_i}} c_{lt\_cnt})$
$C^{abs-C(c)}_{output}$ $C^{abs-C(o)}_{output}$	$N * c_o + \sum_{1 \leq i \leq N_{core}} c_{rqs}$ $N * c_o$
$C^{abs-M(c)}_{purge}$	$N_{exp} * (c_p)$
$C^{abs-M(c)}_{load}$	$N_{new} * c_l$
$C^{abs-M(c)}_{nei\_main}$	$\sum_{1 \leq i \leq N_{new}} (c_{rqs} + \sum_{1 \leq j \leq N_{p_i}} c_{mt}) + \sum_{1 \leq i \leq N_{prmtcore}} (c_{rqs} + c_{mt})$
$C^{abs-M(c)}_{output}$	$N * c_o$
$C^{extra-n(c)}_{purge}$	$N_{exp} * (c_p)$
$C^{extra-n(c)}_{load}$	$N_{new} * c_l$
$C^{extra-n(c)}_{nei\_main}$	$\sum_{1 \leq i \leq N_{new}} (c_{rqs} + \sum_{1 \leq j \leq N_{p_i}} c_{hm}) + \sum_{1 \leq i \leq N_{prmtcore}} (c_{hm})$
$C^{extra-n(c)}_{output}$	$N * c_o$

Table 8.3: CPU Costs of Alternative Algorithms at Four Stages

Naive Approach: 1) Purge Cost: remove all expired data points from the window. 2) Load Cost: load all new data points into the index. 3) *Neighborship* Maintenance Cost: for all data points in the window, run a range query search to form the patterns. 4) Output for Clusters: check each data point in the window, and output each data point associated with a cluster Id.

Exact-N Algorithm: 1) Purge Cost: remove all expired data points. Then for each data point remaining in the window, remove the expired neighbors from its neighbor list, 2) Load Cost:load all new data points into the index. 3) *Neighborship* Maintenance Cost: for each new data point p_i, run a range query search. Then for each of its neighbor p_j found, add p_i and p_j to each others' neighbor list. 4.1) Output for Clusters: check each data point in the window and run depth first search on all *core points*. 4.2) Output for Outliers: check each data point in the window, .

Abstract-C Algorithm: 1) Purge Cost: remove all expired data points in the window. 2) Load Cost: load all new data points into the index. 3) *Neighborship* Maintenance Cost: for each new data point p_i, run a range query search. Then for each of p_i's neighbor p_j found, update p_i and p_j's lt_cnt. 4.1) Output for Clusters: check each data point in the window and run one range query search for each *core point* to form clusters. 4.2) Output for Outliers: check each data point in the window, output those with not enough neighbors (not enough neighbors on their neighbor lists)as outliers.

Abstract-M Algorithm: 1) Purge Cost: remove all expired data points in the window. 2) Load Cost: load all new data points into the index. 3) *Neighborship* Maintenance Cost: for each new data point p_i, run a range query search. Then for each of p_i's neighbor p_j found, update p_i and p_j's lt_mt. Also, for each promoted *core point*, run a range query search and update its lt_mt. 4) Output for Clusters: check each data point in the window, output those with not enough neighbors (neighbor counts are not not large enough) as outliers.

Extra-N Algorithm: 1) Purge Cost: remove all expired data points in the window. 2) Load Cost:load all new data points into the index. 3) *Neighborship* Maintenance Cost: for each new data point p_i, run a range query search. Then for each of $p_i's$ neighbor p_j found, update p_i and $p_j's$ H-Marks. 4) Output for Clusters: check each data point in the window.

Memory Costs of Alternative Algorithms.

Similarly, we define the memory cost of individual data structures in Table 8.4 before we discuss the memory cost of each algorithm.

Memory cost of a data point	m_p
Memory cost of an exact *neighborship* (single-directional)	m_n
Memory cost of an integer attribute	m_i
Memory cost of the lt_cnt of a data point (used by Abstract-C)	$m_{lt\_cnt} = \frac{C_{ils}}{2} m_i$
Memory cost of the M-Table (mt) of a data point (used by Abstract-M)	$m_{mt} = \frac{3C_{ils}}{2} m_i$
Memory cost of the H-Marks (hm) of a data point (used by Extra-N)	$m_{hm} = \frac{5C_{ils}}{4} m_i + \frac{\theta^{cnt}}{2} m_n$

Table 8.4: Memory Costs of Individual Data Structures.

Again, we use $\frac{C_{ils}}{2}$ to represent the average length of each data point's life-span, namely the number of "predicted views" that the data point needs to be maintained in. Since lt_cnt for each point is simply composed by $\frac{C_{ils}}{2}$

neighbor counters (integers), its memory cost is $\frac{C_{ils}}{2}m_i$. For mt, as each data point needs to maintain 3 integers, namely a neighbor counter (integer), a type indicator (integer) and a cluster membership (integer), for each "predicted view" its memory cost is $\frac{3C_{ils}}{2}m_i$. For hm of each data point, it needs to maintain 2 integers, namely a neighbor counter (integer) and type indicator (integer) for one "predicted view". Also, it needs to maintain certain numbers of cluster memberships (between 0 and $\frac{C_{ils}}{2}$) and exact neighbors (between 0 and θ^{cnt}). We use half of the maximum numbers for both of their estimations, which would be the case when the time stamps of the input data are uniformly distributed or queries are using count-based windows. Thus its memory cost is $\frac{5C_{ils}}{4}m_i + \frac{\theta^{cnt}}{2}m_n$.

Then we give the memory cost of each algorithm in Table 8.5. Here we note that since the three algorithms, namely the Exact-N, the Abstract-C, and the naive solution have the same memory costs when detecting density-based clusters and distance- based outliers, we do not distinguish between them in our cost model. While we put a (c) after each algorithm designed to detect density-based clusters only. Basically, the memory costs of each algorithm are composed of the utilization for storing the raw data ($N * m_p$) and the utilization for storing the corresponding meta-data about each data point.

8.1.2 Performance Analysis

Analysis for Density-Based Clusters Algorithms. We first analyze the costs of the algorithms for detecting density-based clusters in Table 8.6.

Memory Cost of the Naive Solution	$N * (m_p + m_{int})$
Memory Cost of Exact-N	$\sum_{1 \leq i \leq N}(m_p + \sum_{1 \leq j \leq N_{p_i}} m_n)$
Memory Cost of Abstract-C	$N * (m_p + m_{lt\_cnt})$
Memory Cost of Abstract-M(c)	$N * (m_p + m_{mt}$
Memory Cost of Extra-N(c)	$N * (m_p + m_{hm})$

Table 8.5: Memory Costs of Each Algorithm.

	Exact-N	Abstract-C	Abstract-M	Extra-N	Naive
Num of rqs	N_{new}	$N_{new}+N_{core}$	$N_{new} + N_{prmtcore}$	N_{new}	N
Worst Case Memory Overhead	$N^2 * m_n$	$N * \frac{C_{ils}}{2} * m_{int}$	$N * \frac{3C_{ils}}{2} * m_{int}$	$N * (C_{ils} * m_{int} + \theta^{cnt} * m_n)$	$N * m_i$
Performance Factors	$\bar{N}_{(p_i)}$ N_{new}	$\bar{N}_{(p_i)}$ N_{new} N_{core} (C_{ils})	$\bar{N}_{(p_i)}$ N_{new} $N_{prmtcore}$ (C_{ils})	$\bar{N}_{(p_i)}$ N_{new} $(C_{ils}$ $\theta^{cnt})$	$\bar{N}_{(p_i)}$ N
More Efficient If	$\bar{N}_{(p_i)} \Downarrow$ $N_{new} \downarrow$	$\bar{N}_{(p_i)} \downarrow$ $N_{new} \downarrow$ $N_{core} \downarrow$ $(C_{ils} \downarrow)$	$\bar{N}_{(p_i)} \downarrow$ $N_{new} \downarrow$ $N_{prmtcore} \downarrow$ $(C_{ils} \downarrow)$	$\bar{N}_{(p_i)} \downarrow$ $N_{new} \downarrow$ $(C_{ils} \downarrow)$ $(\theta^{cnt} \downarrow)$	$\bar{N}_{(p_i)} \downarrow$ $N \downarrow$ $(N_{new} \uparrow)$

Table 8.6: Cost Analysis of Each Algorithm (↓=small, ⇓=very small, ↑=large, we use () if impact is minor).

There are several observations that can be made based on our analysis shown in Table 8.6.

1) There are two major factors affecting the performance of all the algorithms, namely $\bar{N_{(p_i)}}$ the average number of neighbors each data point has and N_{new} the average number of new data points in each window (except that the performance of the naive solution depends on N instead of N_{new}). $\bar{N_{(p_i)}}$ has a great influence on the cost for *neighborship* maintenance, as it decides on the amount of *neighborships* that exist in each window. The increase of $\bar{N_{(p_i)}}$ will cause the increases of the costs for all algorithms, as the range query searches become more expensive and each data point needs to communicate with more neighbors to update its meta-data. N_{new} is the proven lower bound for the minimum number of range query searches needed at each window for neighbor-based pattern detection (see Chapter 5.1). Obviously, the larger N_{new} is, the more range query searches are needed for all algorithms. The only exception is the naive solution that needs N range query searches in all cases.

2) Only Exact-N and Extra-N guarantee the minimal number of range query searches at each window, namely N_{new}, and thus avoid the most expensive operations to the best level. All the other alternative algorithms may need extra range query searches depending on the characteristics of the input data.

3) The performance of Exact-N is very sensitive to $\bar{N_{(p_i)}}$, namely its memory overhead becomes quadratic in N when $\bar{N_{(p_i)}}$ approaches N. This makes it work well only when $\bar{N_{(p_i)}} \Downarrow$, meaning $\bar{N_{(p_i)}}$ is very small. All three predictability-based solutions, Abstract-C, Abstract-M and Extra-N have linear memory overhead in all the cases.

4) The performance of Abstract-M is largely decided by $N_{prmtcore}$, which may significantly increase the number of range query searches that Abstract-M needs to run at each window. As $N_{prmtcore}$ can potentially be as large as the size of all data points inherited from the previous window $N - N_{new}$, it makes Abstract-M suffer from the risk of having an even worst performance than the naive solution. The same problem may happen to Abstract-C as well, when N_{core} gets close to N.

5) The cost of the predictability-based solutions drops even lower as the constant C_{ils} decreases, because it decides on the number of "predicted views" to be stored and updated. Besides C_{ils}, the performance of Extra-N is also affected by another constant θ^{cnt}, which works as the upper bound for the number of exact *neighborships* each data point stores. Although we list these two factors for the completeness of our analysis, they are not the key factors deciding algorithms' performance. This is because they are unrelated to the number of range query searches needed and usually very small constants compared with N.

Analysis for Distance-Based Outlier Algorithms. For detecting distance-based outliers in sliding windows, Abstract-C achieves both the minimal number of range query searches and linear memory requirement, while Exact-N suffers from a potential quadratic memory overhead. The naive solution does not take advantage of incremental computation.

Conclusion. Based on our cost analysis, we conclude that Extra-N and Abstract-C are the best solutions for detecting density-based clusters and distance-based outliers over sliding windows respectively. To validate our claims derived from this analytical evaluation, a thorough experimental study using both real and synthetic data streams is presented in Chapter 9.

8.2 Cost Analysis for kNN Detection Algorithms

8.2.1 Cost Analysis for PreTopk Algorithm

The predicted top-k result for each future window can be organized using different data structures, such as a sorted list supported by a tree-based index structure or a min-heap organized on the F score of the objects. No matter which data structure is chosen, the best possible CPU costs for inserting a new object into a top-k object set and keeping the size of the top-k object set unchanged has complexity $O(log(k))$. More precisely, $log(k)$ for positioning the new object in the top-k object set, and $log(k)$ for removing the previous top-k object with the lowest F score. Thus the overall processing costs for handling all new objects for each window slide is $O(N_{new} * C_{ave\_topk} * log(k))$, with N_{new} the number of new objects coming to the system at this slide, and C_{ave_topk} [1] the average number of windows each object is predicted to make top-k when it arrives at the system. As the object expiration process is trivial, this constitutes the total cost for updating the top-k result at each window slide,

Memory-wise, PreTopk maintains predicted top-k results for
C_{nw} [2] windows. The memory consumption of PreTopk is composed of two parts: first, the number of distinct objects stored in memory; second,

[1]When data is uniformly distributed, $C_{ave\_top-k} = \frac{2k}{3slide}$.
[2]$C_{nw} = \lceil \frac{Q_i.win}{Q_i.slide} \rceil$ which is equal to the maximum number of windows a new object can be alive.

the number of references to the objects in the predicted top-k results of all future windows. An object may appear in the predicted top-k results for multiple windows and thus needs multiple references. In the example shown in Figure 7.4, object 14 is predicted to be part of the top-k results in four windows, and thus four references to it are needed.

For the first part, PreTopk achieves the minimal number of objects to maintain for continuous top-k monitoring (Theorem 7.1). The size for this minimal set in the average case is analyzed below.

Theorem 8.1 *In the average case* [3]*, the number of distinct objects in the predicted results for all future windows is* $2k$*.*

For the second part, the number of references stored by PreTopk is simply $C_{nw} * k$, as there are C_{nw} windows and k objects in each of them. The size of an object reference (Ref_{size}) is typically significantly smaller than the size of the actual object (Obj_{size}), especially when the object contains a large number of attributes. In summary, the average memory cost for PreTopk is $2k * Obj_{size} + C_{nw} * k * ref_{size}$.

Conclusion. PreTopk successfully over comes the recomputation bottleneck suffered by the state-of-the-art solutions [MBP06]. Memory-wise, it only keeps the minimal number of objects necessary for top-k query monitoring, which it is shown to be independent of the potentially very large window size (in Theorem 8.1). Computation-wise, the processing costs for generating the top-k result in each window are no longer related to the window size. This is a significant improvement over the state-of-the-art

[3]Data is uniformly distributed on $F(o)$, indicating that the objects with different F scores have equal opportunity to expire after the window slides.

solution [MBP06], because both the processing and memory costs of any solution that involves recomputation are related to the window size, which is usually an overwhelming factor compared to *k* or the slide size.

However, there are also clear **limitations** for PreTop. The above cost analysis reveals that the performance of the PreTopk algorithm is affected by a constant factor, namely C_{nw}, the number of predicted top-k results to be maintained. More precisely, since PreTopk maintains the predicted top-k result for each window independently, both its CPU and memory costs increase linearly with the number of predicted top-k results to be maintained (C_{nw}).

8.2.2 Cost Analysis for MinTopk Algorithm

CPU and Memory Costs in the General Case. The CPU processing costs of MinTopk to handle object expiration are $O(k)$, as we simply need to update the window marks of the first k objects on the super-top-k list. For handling the new objects, the cost for each object p_{new} is $P^{intopk} * (log(MTK.size) + C_{nw\_topk}) + (1 - P^{intopk}) * 1$ ($0 \leq P^{intopk} \leq 1$), with $MTK.size$ the size of MTK (the number of objects maintained in the *super-top-k list*) and P^{intopk} the probability that p_{new} will make the MTK set. In general, when p_{new} makes the MTK set (with P^{intopk} probability), the cost for positioning p_{new} into the *super-top-k list* is $log(MTK.size)$ with the support of any tree-based index structure. The cost for redirecting the lower bound pointers is simply equal to $C_{nw\_top\_k}$ [4], the number of windows that are affected by its insertion,

[4]$C_{nw\_top\_k}$ is bounded by the constant C_{nw}, namely it is at most equal to C_{nw}, the total number of windows maintained.

because we only need to move that pointer for each affected window by one position (Lemma 7.5). Otherwise, with $1 - P^{intopk}$ probability, it will be discarded immediately with the cost of just a single check (comparing its F score with the minimal F score on the *super-top-k* list).

Lemma 8.1 *The CPU complexity for MinTopk to handle each new object is* $O(log(MTK.size))$*.*

Therefore, the CPU complexity of MinTopk to process each window is $O(N_{new} * (log(MTK.size))$ in the general case, with N_{new} the number of new objects coming in that window slide.

Memory-wise, MinTopk only needs a constant memory size to maintain each object in the MTK set.

Lemma 8.2 *The memory size required by MinTopk to maintain each object* p_i *in super-top-k list is of constant size, in particular, it is* $(Obj_{size} + 2Ref_{size})$*.*

Therefore, the memory complexity of MinTopk is $O(MTK.size)$ in the general case.

From the analysis above, we can observe that the size of the MTK, $MTK.size$, is a key factor affecting both CPU and memory costs of MinTopk. In the **best case**, $MTK.size$ equals to k. This would happen when the predicted top-k results for all future windows are identical. In the **worst case**, it is equal to the max size of each predicted top-k result set (*k*) times the number of windows maintained (C_{nw}), namely $C_{nw} * k$. This would mean that all predicted top-k objects expire after each window slide. However, this special case is highly unlikely in real streams. It could only happen when the F scores of the objects in a stream monotonically decrease across time and the slide size is at least as large as k.

Clearly, the average case is the most important one. In the **average case**, the size of the *super-top-k* is only $2k$, as we have proven that the average number of distinct objects in MTK is $2k$ in Lemma 8.1. This is comparable to the size of the final top-k result, which is equal to k. Thus, the average-case CPU complexity of MinTopk for generating the top-k results at each window is $O(N_{new} * log(k))$. The average-case memory complexity of MinTopk is $O(k)$.

Optimality of MinTopk. Now we prove the optimality of our proposed MinTopk algorithm in both CPU and memory utilization.

Theorem 8.2 *MinTopk achieves optimal memory complexity for continuous top-k monitoring in sliding windows. MinTopk also achieves optimal CPU complexity for continuous top-k monitoring in sliding windows, when the top-k results are returned in a ranked order based on preference scores.*

Proof 8.1 ***Memory-Optimality:*** *We have proven that the MTK set is the minimal object set that is necessary for any algorithm to accurately monitor top-k objects in sliding windows in Lemma 7.1. We emphasize that this minimality holds in the general case, namely, given any unknown arrival rate distribution and preference score distribution of the input stream. Thus the optimal memory complexity of any top-k monitoring algorithm in sliding windows is at least $O(MTK.size)$.*

Now we show that MinTopk achieves this optimal memory complexity. First, MinTopk only maintains one reference for each object in MTK set. Then, in Lemma 8.2, we have shown that the memory space needed by MinTopk for each object in the *super-top-k list is $Obj_{size} + 2 * Ref_{size}$. Denoting the size of MTK by $MTK.size$, then the memory cost of MinTopk is $MTK.size * (Obj_{size} + 2 * Ref_{size})$. Since Obj_{size} and Ref_{size} are both of constant size, the memory complexity of MinTopk is $O(MTK.size)$. This proves that MinTopk has optimal memory complexity in the general case.*

CPU-Optimality. *To prove that MinTopk algorithm achieves the optimal CPU complexity for generating the ranked top-k results at each window slide, we formalize this problem as P_{newk}.*

***Problem** P_{newk}**:** Given two datasets D_{new} and D, which respectively represent the new object set for a window slide and the objects inherited from the previous window, $|D_{new}| = N_{new}$ and $|D| = N$. Each object o_i in D_{new} or D has a unique $F(o_i)$ score [5]. The objects in D_{new} and D are not sorted on $F_(o_i)$ score. The goal is to return a dataset K which is composed of k objects from $D_{new} \cup D$ which have the largest $F(o_i)$ scores in $D_{new} \cup D$ in ranked order of $F(o_i)$.*

Next we show that the problem P_{newk} is at least as hard as the following problem $P_{newk'}$.

***Problem** $P_{newk'}$**:** Given two datasets D_{new} and D_k, which respectively represent the new object set coming with a window slide and the existing top-k object set of D ($D_k \subseteq D$), $|D_{new}| = N_{new}$ and $|D_k| = k$. Each object o_i in D_{new} or D_k has a unique $F(o_i)$ score. The objects in D_{new} are unsorted on $F_(o_1)$ score. The goal is to return a dataset K which is composed of k objects from $D_{new} \cup D_k$ which have the largest $F(o_i)$ scores in $D_{new} \cup D_k$ in ranked order of $F(o_i)$.*

[5]The assumption on uniqueness of preference scores is a common assumption for top-k and most sorting related problems [CSRL01]. It is mainly for simplifying the problem definition. While our proposed techniques do not require such uniqueness of preference scores in query processing.

The problem P_{newk} is at least as hard as $P_{newk'}$, because any algorithm that solves P_{newk} has to consider any object $o_i \in D$, as any of them may be part of the new top-k results. However, given that D_k is the top-k object set in D, thus for any object o_i, if $o_i \in D$ but $o_i \notin D_k$, it cannot be in the new top-k results, because there are already k objects in D_k having larger $F_(o)$ scores than this o_i. Thus, any algorithm solving P_{newk} can solve $P_{newk'}$ also, indicating that P_{newk} is at least as hard as $P_{newk'}$.

Now we prove the lower bound of $P_{newk'}$ by showing that $P_{newk'}$ can be reduced to the sorting based on comparison problem [CSRL01]. In particular, first, Let A denote any algorithm that solves this problem. Then we give the following inputs to A, namely the input datasets D_{new} and D_k, in which for any $o_i \in D_{new}$ and $o_j \in D_k$, $F(o_i) > F(o_j)$ and $|D_{new}| = |D_k|$. It is easy to see that if A solves P_{newk} with the inputs above, A sorts D_{new}. This implies that $P_{newk'}$ can be reduced to sorting based on comparison for D_{new}, namely any algorithm A solving $P_{newk'}$ can be used to solve sorting based on comparison problem. It is well known that the lower bound for sorting based on comparison problem on a dataset of size n is $O(n * log(n))$ [CSRL01]. Therefore the lower bound of the $P_{newk'}$ is $O(N_{new} * log(k))$.

Since we have shown that any algorithm that solves the top-k problem in sliding windows with ranked top-k results returned is dealing with a problem at least as hard as $P_{newk'}$, we now have proven that the lower bound for any top-k monitoring algorithm for generating the top-k results in ranked order for each window is $O(N_{new} * log(k))$.

As we have shown in Lemma 8.1, MinTopk takes only $O(log(MTK.size))$ to process each new object that arrives within a window slide.

Its CPU complexity to process each window is $O(N_{new} * (log(MTK.size))$. *Now we must determine what is the size of MTK in the general case. In Section 4.6, we have shown that in the average case* $MTK.size = 2k$, *and even in the worst case,* $MTK.size$ *is bounded by a constant factor* $C_{nw} = \frac{win}{slide}$. *This indicates that no matter what the input rate and preference score distributions of the input stream are, the design of MTK guarantees that it contains at most* $C_{nw} * k$ *objects. Namely,* $MTK.size = C_{nw} * k$ *in the worst case. Therefore the CPU cost of MinTopk is* $O(N_{new} * log(C_{nw} * k))$ *in the worst case. Since* C_{nw} *is a constant that is known and will not change once the query is specified, the CPU complexity of MinTopk is* $O(N_{new} * log(k))$. *This proves that MinTopk achieves the optimal CPU complexity for generating ranked top-k objects at each window in the general case.*

Chapter 9

Experimental Study

9.1 Real and Synthetic Streaming Datasets

9.1.1 Real Streaming Datasets

The first real streaming dataset we use is the Stock Trading Traces data (STT) from [INE]. This dataset has about one million transaction records throughout the trading hours of a day. Each transaction contains the name of the stock, time of sale, matched price, matched volume, and trading type.

The second real streaming dataset, GMTI (Ground Moving Target Indicator) is provided by MITRE Corp. modeling troop movement in a certain area. It captures the information of moving objects gathered by different ground stations or aircraft in a 6-hour time frame. It has around 100,000 records regarding the information on vehicles and helicopters (speed ranging from 0-200 mph) moving in a certain geographic region.

For the experiments that involve data sets larger than the sizes of these two datasets, we augment them to the required sizes by appending similar data after them. In particular, we append multiple rounds of the original data varied by setting random differences on all attributes, until it reaches the desired size.

9.1.2 Synthetic Streaming Datasets

For the evaluation of density-based cluster detection, we built a synthetic data generator to generate the datasets containing controlled numbers of clusters and noise. Each synthetic dataset is composed of one thousand stream segments. Each segment of data contains certain percentage (as an input parameter) of random noise and a set of clusters, each following a Gaussian distribution but each with different randomly selected *mean* and *variance*.

For the evaluations of distance-based outlier detection algorithms, we use the Gauss Data Set, which is a synthetically generated time sequence of 35,000 one dimensional observations. It consists of a mixture of three Gaussian distributions with uniform noise. This is the dataset used by the only previous work [AF07] detecting distance-based outliers in continuous windows.

9.2 Experimental Studies for Density-Based Cluster Extraction

9.2.1 Experimental Platform

Our experiments for density-based cluster detection algorithms are conducted on a HP Pavilion dv4000 laptop with Intel Centrino 1.6GHz processor and 1GB memory, which runs Windows XP professional operating system. The algorithms are implemented with VC++ 6.0.

9.2.2 Experimental Methodologies

We run all the experiments using both synthetic and real data for 10K windows. We measure two key metrics for stream processing algorithms, response time and memory footprint. Those two metrics are also evaluated by our cost models in Chapter 8.1. In particular, we measure the average response time (referred as CPU time henceforth) it takes to answer a pattern detection query at each window. This response time includes the time consumed by all the four stages of pattern detection at each window. So, the average response time reflects the average time an algorithm needs from "input data is ready" to "results are output" for each single window. The response time is averaged over all the windows in each experiment. The memory footprint, which indicates the peak memory space consumed by an algorithm, is recorded over all the windows.

9.2.3 Overall Evaluation

To compare the performance of all five algorithms discussed in this work, namely Exact-N, Abstract-C, Abstract-M, Extra-N and the naive solution, we conduct a comprehensive experiment with a wide range of the synthetic data generated by our data generator. These experiments cover all the important combinations of the two major cost factors identified in our cost analysis (Chapter 8.1), namely $N^{-}_{(p_i)}$ and N_{new}.

To avoid the performance fluctuations caused by different base sizes, namely different number of data points in the window, we use count-based windows (equal in concept to time-based windows with uniform data rates). Thus, N_{new} is equal to the slide size $Q.slide$, and $N^{-}_{(p_i)}$ is controlled by adjusting two input parameters of the data generator. More specifically, we can increase $N^{-}_{(p_i)}$ by expanding the size of each cluster *CluSize* while decreasing the *variance* of its Gaussian Distribution.

To cover all the major combinations of these two factors, we vary $N^{-}_{(p_i)}$ from 1% to 50%, and $Q.Slide$ from 10% to 100% , both in terms of the percentage to window size $Q.win$ and both with 7 different settings. In particular, the 7 different $N^{-}_{(p_i)}$ settings represent the data from "very sparse" ($N^{-}_{(p_i)} = 1\%$), "medium dense" ($N^{-}_{(p_i)} = 20\%$) and finally to "very dense" ($N^{-}_{(p_i)} = 50\%$). The 7 different $Q.Slide$ settings, which are 10% to 50% with 10% increments plus 80% and 100%, cover all the increments from "mostly remaining" ($Q.Slide = 10\%$), "half-half" ($Q.Slide = 50\%$), "mostly new" ($Q.Slide = 80\%$) and finally to "all new" ($Q.Slide = 100\%$). We measure the CPU time (shown in Figure 9.1) as well as the memory footprint (shown in Figure 9.2) of the five algorithms for all $7 \times 7 = 49$ combinations. Other settings of this experiment include window size $Q.win$ = 5K, $\theta^{range} = 0.003$ and $\theta^{cnt} = Q.win \times 5\% = 250$.

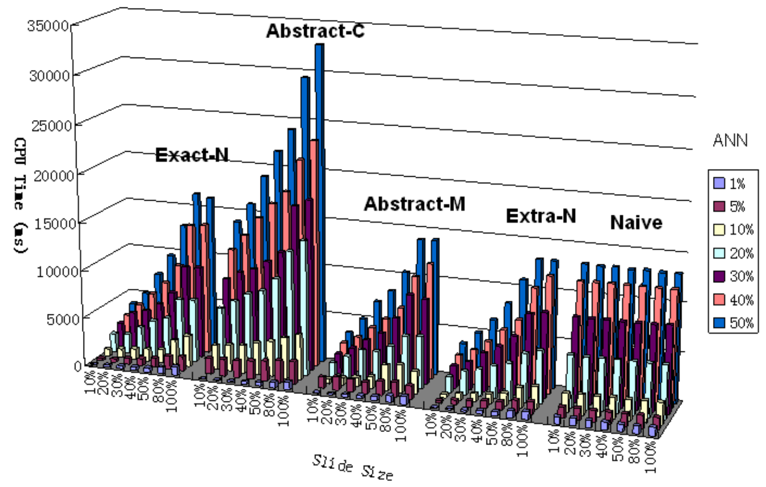

Figure 9.1: Comparison on CPU Performances of Five Algorithms

From Figures 9.1 (CPU) and 9.2 (memory), we observe that Abstract-M and Extra-N clearly outperform the other three algorithms, namely the Exact-N, Abstract-C and the naive solution, in most of the test cases. Besides the naive solution which does not take advantage of incremental computation, the other two incremental algorithms, Exact-N and Abstract-C suffer from a huge consumption of either memory space or CPU time in many cases.

In particular, as shown in Figure 9.2, the memory consumption of Exact-N is at least 80% percent higher than the naive solution as the later can be considered as having no memory overhead. More importantly, such 80% percent gap only happens when the data is very sparse ($\bar{N_{(p_i)}} = 1\%$ of

$Q.win$). It increases to more than 4000 percent when $\bar{N_{(p_i)}}$ reaches 50% of the window size, indicating that the data stream contains very dense sub-regions. Such results agree with our earlier analysis, considering the large number of links each data point has to store. This experiment confirms that Exact-N is not an efficient algorithm in terms of memory consumption. In addition, the CPU time it uses in all 49 cases is on average 25 present higher than that used by Extra-N. This is calculated by summing the difference percentage in all 49 cases and divided it by 49. This fact eliminates it from the set of plausible candidates even in terms of CPU-efficiency.

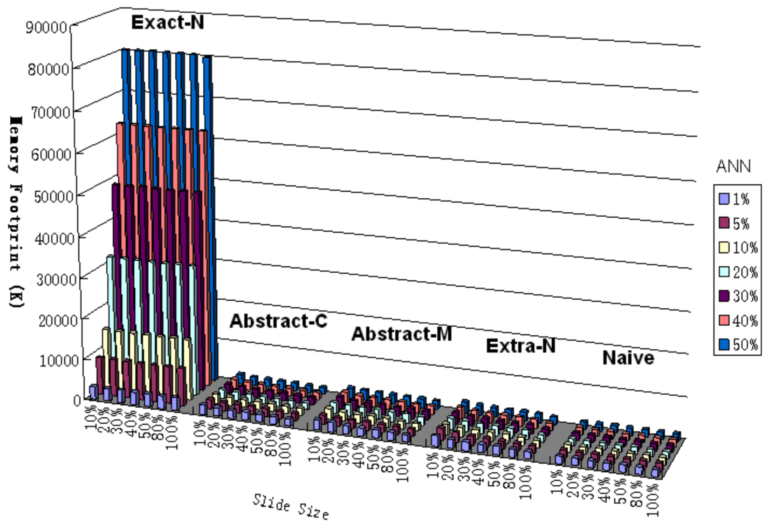

Figure 9.2: Comparison on Memory Performances of Five Algorithms

Abstract-C, an incremental algorithm which does not maintain the exact neighborships, has good memory efficiency. However, since the time efficiency of Abstract-C is highly sensitive to the number of *core points* N_{core} in the window, the performance of Abstract-C is largely decided by the interrelationship between two variables, namely $N^{-}_{(p_i)}$ and θ^{cnt}. More specifically, when $N^{-}_{(p_i)}$ is far below the θ^{cnt} and thus there exist very few or no *core points* in the window, Abstract-C is actually a very efficient algorithms in terms of both CPU and memory. As shown in Figure 9.1, it is even faster than Abstract-M and Extra-N for $N^{-}_{(p_i)} = 1\%$ cases in terms of CPU time. However, as $N^{-}_{(p_i)}$ increases and eventually surpasses θ^{cnt}, indicating that more and more data points become *core points*, the time efficiency of Abstract-C drops dramatically. In many cases shown in Figure 9.1, it is not only much slower than our proposed algorithms. Abstract-M and Extra-N, but also slower even than the naive solution. This experiment illustrates that Abstract-C is very inconsistent in terms of CPU time and it performs well only if N_{core} is very very low. Given the limited scope of applicability, Abstract-C is not a general attractive solution.

Our proposed algorithms, namely Abstract-M and Extra-N, take advantage of incremental computations while successfully avoiding the huge overhead on both memory and CPU. Compared with the naive solution, both Abstract-M and Extra-N need more memory space as they need to maintain a certain amount of meta-information for future windows. However, such overhead is much smaller than that of Exact-N and in fact is always being kept at very acceptable levels. In particular, for Abstract-M the memory consumption in all 49 cases is on average 33 percent higher than that of the naive solution. This is calculated in the same way as we compared the CPU time of Exact-N and Extra-N earlier. For Extra-N, this number becomes 36 percent, which is slightly higher but still quite modest. These facts confirm that our proposed algorithms, Abstract-M and Extra-N, have very good and consistent memory-efficiency.

The negligible CPU overhead of our proposed algorithms is also confirmed by this experiment. As shown in Figure 9.1, Abstract-M and Extra-N saved CPU time substantially compared to the naive solution in all the cases where $Q.Slide \leq 50\% \times Q.win$. Even in the cases when $Q.Slide$ is very close (80%) or even equal to $Q.win$ (typically the limit of the incremental algorithms), Abstract-M and Extra-N exhibit comparable performances with those of the naive solution. Actually, both Abstract-M and Extra-N can be taken as variances of the naive solution when the windows are non-overlapping, because they only detect the patterns based on the "view" of the current window and no "predicted view" would be generated nor maintained. This indicates that our proposed algorithms have very small CPU overhead in all cases and thus are viable candidates for system's only implementation, regardless of the input data and queries.

9.2.4 Abstract-M vs. Extra-N

We first discuss the equivalence of Abstract-M and Extra-N shown in many of our above test cases, which on first sight does not appear as one would have expected in our cost analysis. The main reason for this is that the number of *promoted core points* $N_{prmtcore}$ stayed small in many cases and thus did not entangled the performance of Abstract-M. Actually, we found that $N_{prmtcore}$ tends to be small, unless a large number of data points who have a "boundary" (close to θ^{cnt}) number of neighbors exist. However, such situations are not found to be often in our experiments on both synthetic and real data.

Although Abstract-M and Extra-N work equivalently well in many of our test cases, they do behave quite differently when $N_{prmtcore}$ turns to be an unneglectable factor. To better understand their performance in these special cases, we zoom into the cases with $\bar{N_{(p_i)}} = 5\%$ in our comprehensive experiment (Figure 9.1 and Figure 9.2). Figures 9.3 and 9.4 show the zoomed in subparts of the same experiment results discussed in the earlier part of this subsection.

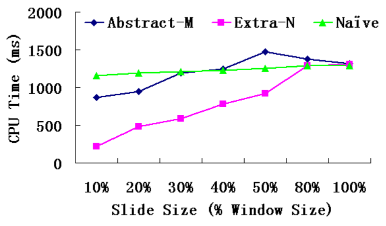

Figure 9.3: Comparison on CPU Time of Abstract-M and MPS in $\bar{N_{(p_i)}} = 5\%$ cases

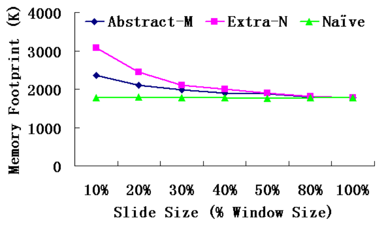

Figure 9.4: Comparison on Memory Usage of Abstract-M and MPS in $\bar{N_{(p_i)}} = 5\%$ cases

In the cases shown in Figures 9.3 and 9.4, Abstract-M tends to use more CPU time while Extra-N consumes more memory space. This is as expected because of the existence of a large number of *promoted core points* in each window. In particular, since in the $\bar{N_{(p_i)}} = 5\%$ cases, the number of neighbors each data point has is quite close to the population threshold, $\theta^{cnt} = 5\%$ of the window size, many *core points* may be demoted to become *edge points* or even *noises* after losing some of their neighbors as the window slides. For the same reason, the *non-core points* have a good chance to be promoted to become *promoted core points* after gaining some new neighbors at the window slide. Corresponding to our analysis in Sections 7 and 8, each *promoted core point* charges an extra range query search from Abstract-M, while it charges Extra-N for the memory space to store the links to its neighbors in its "*non core point* career" before its promotion. On the one hand, Extra-N guarantees the minimum number of range query searches and thus time efficiency in all cases, but it may consume more memory, especially when $N^{-}_{(p_i)}$ and θ^{cnt} are both large and close to each other. On the other hand, Abstract-M never stores the exact *neighborships* and thus is more memory-efficient. However, it usually takes extra range query searches and thus consumes more CPU time. Such preferences of Abstract-M and Extra-N for CPU time versus memory space utilization can be observed in most of the test cases, although they are most apparent in the cases we zoomed into. Thus, in general, a system can choose to implement Abstract-M when the memory space is its key limit, while implementing Extra-N if CPU time is its major resource concern.

9.2.5 Scalability Analysis

We now look at the scalability in terms of the base size, meaning that how many data points the algorithms can cluster at each window. So, in this experiment, we test count-based windows sized from 10K to 50K with a fixed slide size 5K. Other settings of this experiment are equal to those from the previous comprehensive one, except that we fixed $N^{-}_{(p_i)}$ at $1K$.

As shown in Figures 9.5 and 9.6 both our algorithms, Abstract-M and Extra-N show very good scalability in window sizes in terms of both CPU and memory, while others failed in either or both of them. In particular, both Abstract-M and Extra-N only need 5 seconds to cluster 50K data

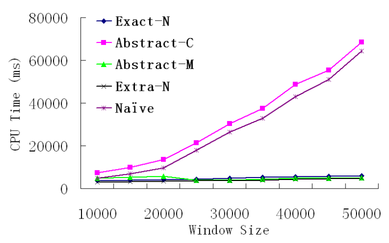

Figure 9.5: Comparison of CPU Scalability on Base (Window) Size

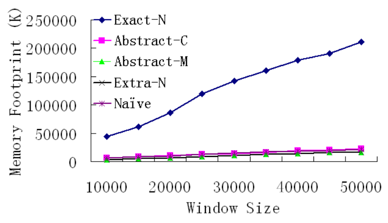

Figure 9.6: Comparison of Memory Scalability on Base (Window) Size

points at each window given 5K new data points. On the other words, both algorithms can comfortably handle a data rate of 1K per second with a 50K window. Also, the memory usage of both algorithms increases very modestly with the growth of the window size.

Second, we investigate the effect of dimensionality on the performance of our algorithms. As shown in Figure 9.7, the CPU time of both our proposed algorithms, especially Extra-N, increases only modestly with the number of dimensions. This demonstrates that our algorithms have an even better than linear scalability in the dimensionality. This is because the number of dimensions will only affect the CPU time needed for the range query searches but has no impact on the *neighborship* maintenance costs. As we have already largely reduced the number of range query searches needed in these two algorithms, and even achieved the minimal for Extra-N (see our cost analysis in Chapter 8.1), they both are expected to have excellent scalability in the number of dimensions. Our experimental results confirmed this.

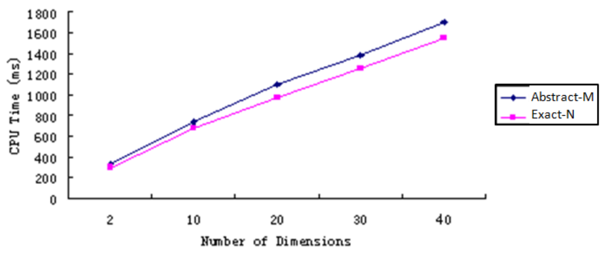

Figure 9.7: Comparison of CPU Scalability on Dimensionality

9.2.6 Evaluation with Real Data

We first evaluate the performance of all five competitors with the GMTI data, which is a representative for moving object monitoring applications. We varied the slide size $Q.Slide$ from 10% to 100% of the $Q.win$. Given these query parameters, we find there are 6 to 11 clusters in the window at different time horizons, and $N_{(p_i)}^{-}$ in the windows varies from 9% to 11% of the number data points in the windows.

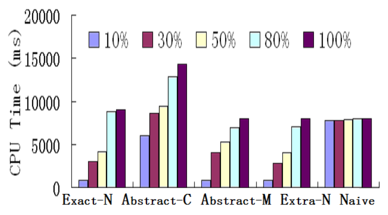

Figure 9.8: Comparison on CPU Time with GMTI data

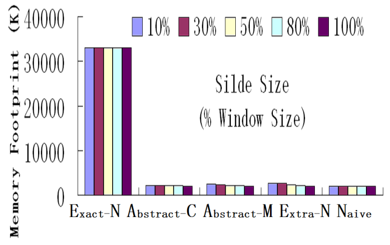

Figure 9.9: Comparison on Memory Usage with GMTI data

As depicted in Figures 9.8 and 9.9, Extra-N has the best time efficiency compared with all other methods. The memory usage of Extra-N is on average 16% higher than the naive solution in the five cases. This memory overhead is a little bit higher than that of Abstract-M (11% higher than the naive solution) but still very acceptable.

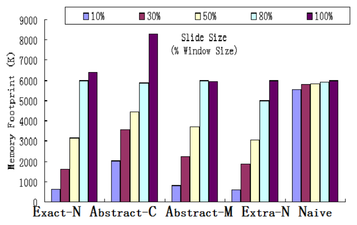

Figure 9.10: Comparison on CPU Time with STT data

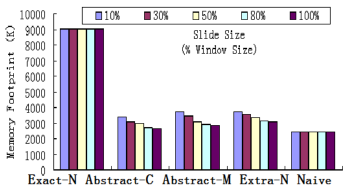

Figure 9.11: Comparison on Memory Usage with STT data

For STT data, by using the query parameters learned from our pre-analysis of the data, the number of clusters existing in the windows ranges from 17 to 26, and the number $\bar{N_{(p_i)}}$ in the windows varies from 6% to 9% to the number data points in the windows. Similar behaviors can also be observed using STT data (as shown in Figures 9.10 and 9.11).

Thus, generally, our experiments on real data also confirm that our proposed algorithms Abstract-M and Extra-N outperform other alternative methods and thus are the preferred solutions for density-based cluster detection in sliding windows over all other alternatives.

9.3 Experimental Studies for Distance-Based Outlier Extraction

9.3.1 Experimental Platform

Our experiments for distance-based outlier detection algorithms are conducted on a HP Pavilion dv4000 laptop with Intel Centrino 1.6GHz processor and 1GB memory, which runs Windows XP professional operating system. The algorithms are implemented with VC++ 6.0.

9.3.2 Experimental Methodologies

We run all the experiments using both synthetic and real data for 10K windows. We measure two key metrics for stream processing algorithms, response time and memory footprint. Those two metrics are also evaluated by our cost models in Chapter 8.1. In particular, we measure the average response time (referred as CPU time henceforth) it takes to answer a pattern detection query at each window. This response time includes the time consumed by all the four stages of pattern detection at each window. So, the average response time reflects the average time an algorithm needs from "input data is ready" to "results are output" for each single window. The response time is averaged over all the windows in each experiment. The memory footprint, which indicates the peak memory space cons umped by an algorithm, is recorded over all the windows.

9.3.3 Alternative Methods

We compare the performance of our outlier algorithm Abstract-C with two alternatives, namely the naive solution and the exact-STORM [AF07], which is the only previous work in the literature we are aware of that detects distance-based outliers in sliding windows. Before we discuss the experiment results, we first describe the exact-STORM algorithm.

Generally, exact-STORM is designed to incrementally detect distance-based outliers over count-based windows. Similar with Abstract-C, exact-STORM requires one range query search for every new data point in each window. However, it uses a different *neighborship* maintenance mechanism. In particular, the exact-STORM algorithm requires every data point p_i in the window to maintain two data structures. The first one, called $p_i.nn\_before$, is a list containing the identifiers of the most recent preceding neighbors of p_i. $p_i.nn\_before$ is similar with the "neighbor list" we use in Exact-N that gives p_i direct access to its neighbors. However, it has two special characteristics. First, $p_i.nn\_before$ only stores the preceding neighbors of p_i, whose arrival and expiration are earlier than those of p_i. Second, for "count- based" based windows, the length of $p_i.nn\_before$ has an upper bound $k = N \times \theta^{range}$, which equals the number of neighbors p_i needs to have to be a "safe inlier". This is because the number of data points in the count-based window is fixed. So, if a data point already has $k = N \times \theta^{range}$ neighbors, it cannot be an outlier in the current and future windows until any of them expire. The second data structure $p_i.count\_after$ is a counter of the number of succeeding neighbors of p_i. The succeeding neighbors denote the neighbors of p_i whose arrival and expiration are later than that of p_i. At each window slide, exact-STORM runs one range query search for every new data point, and updates nn_before and $count\_after$ for each of them and their neighbors. At the output stage, exact-STORM outputs the outliers based on the information in each data point's nn_before and $count\_after$. More details about the algorithm can be found at [AF07].

In count-based windows, since exact-STORM achieves both the minimum number of range query searches and also the linear memory consumption (it stores at most k neighbors for each data point), it is equivalent to our proposed algorithm Abstract-C, while using different *neighborship* maintenance mechanisms. However, being designed specifically for the count-based window scenario, exact-STORM would tend to perform badly in the time-based window scenario. This is because the "safe inlier" property, which it relies on to limit the length of nn_before in the count-based scenario, no longer holds for time-based windows. In particular, for time-based windows, since the number of data points in the window is not fixed, the number of neighbors a data point needs to be an "inlier" may change as well. So, no matter how many neighbors a data point already has, it can never be viewed as a "safe inlier" in future windows and has to keep the "identifiers" of all its "preceding" neighbors. So in the time-based window, exact-STORM would suffer from the same problem as Exact-N does, namely the huge number of exact *neighborships* (links) that must be stored.

9.3.4 Performance Evaluation

In our experiments, we compare the performance of exact-STORM and Abstract-C in both count- and time-based window scenarios using the Gauss Dataset. In both scenarios, we follow the implementation of exact-STORM presented in [AF07], except for breaking the upper bound on the length of nn_before as required in the time-based window scenario. The experimental results are shown in the Figures 9.12 to 9.15.

As shown in Figures 9.12 and 9.13, for count-based windows, exact-STORM and Abstract-C perform equivalently well and clearly outperform the naive solution in terms of CPU time.

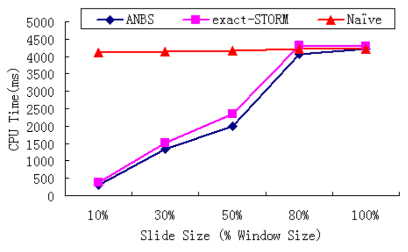

Figure 9.12: Comparison on CPU Time for Count-Based Window Scenario

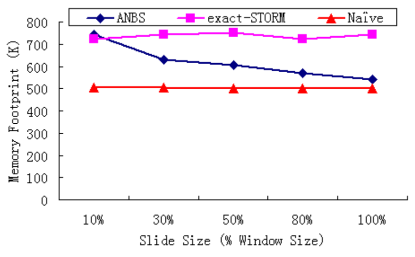

Figure 9.13: Comparison on Memory Usage for Count-Based Window Scenario

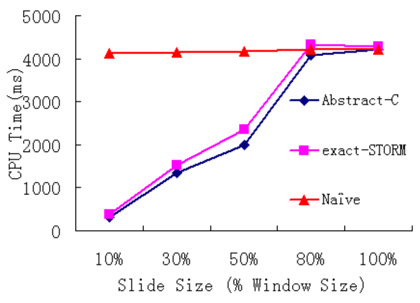

Figure 9.14: Comparison on CPU Time for Time-Based Window Scenario

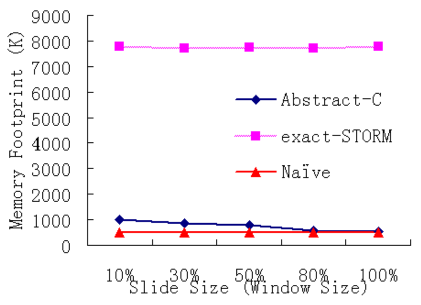

Figure 9.15: Comparison on Memory Usage for Time-Based Window Scenario

However, as shown in Figures 9.14 and 9.15, Abstract-C clearly outperforms both the naive solution and exact-STORM in time-based window scenarios. This is because the naive solution does not take any advantage of incremental computation and Exact-STORM suffers from the huge memory overhead for storing the neighbors in time-based window. In contrast, Abstract-C does not have either of these two problems and thus shows a much better performance.

9.4 Experimental Studies for kNN (top-k) Extraction

9.4.1 Experimental Platform

We conducted our experiments for kNN detection on an HP G70 Notebook PC with an Intel Core(TM)2 Due T6400 2.00GHz processor and 3GB memory, which runs Windows Vista operating system. We implemented the algorithms in C++ using Eclipse.

9.4.2 Alternative Methods

We compare our proposed algorithm MinTopk's performance with two alternative methods, namely the state-of-the-art solution SMA [MBP06] (Section 2.2), and the basic algorithm we presented in this work, PreTopk (Chapter 7.2).

9.4.3 Experimental Methodologies

For all alternatives, we measure two common metrics for stream processing, namely average processing time for each object (CPU time) and the memory footprint, indicating the maximum memory space required by an algorithm. We run each experiment 10 times (runs). Within each run, we process each query for 10K windows (slides for 10K times). The statistics results are averages from the 10 runs. To thoroughly evaluate the alternative algorithms, we compare their performance under a broad range of parameter settings and observe how these settings affect their performance.

9.4.4 Evaluation for Different k Cases

This experiment is to evaluate how the number of preferred objects, k, affects the performance of the three algorithms. We use the STT data. We fix the window size at 1M and slide size at 100K, while varying k from 10 to 10K. As shown in Figures 9.16 and 9.17, both the CPU and memory usage of all three alternatives increases as k increases. This is expected, because the sizes of the key meta-data, namely the predicted top-k results for PreTopk and MinTopk (organized differently though) and the skyband for SMA, all increase linearly with k. However, both the CPU and memory usage of PreTopk and MinTopk are significantly less than those utilized by SMA.

CPU-wise, both PreTopk and MinTopk saved at least 85% of the processing time for each object compared with that used by SMA in the four test cases. By further analyzing the specific components of the processing time, we found that SMA used a large portion (around 60%) of its process-

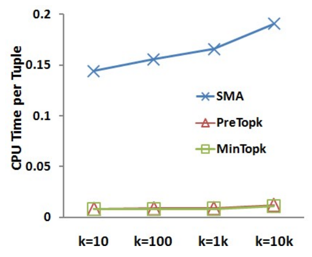

Figure 9.16: CPU time used by three algorithms with different k values

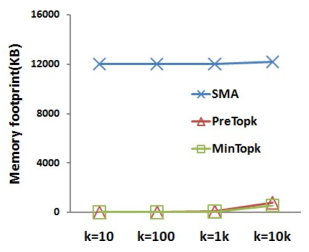

Figure 9.17: Memory space used by three algorithms with different k values

ing time on loading and purging large numbers of objects from the grid file. Such cost is completely eliminated by both PreTopk and MinTopk, as they do not maintain any index for the whole window and the "useless" objects are discarded immediately upon their arrival at the system. Also, we found that SMA used around $10-30\%$ of the processing time for top-k recomputation in different cases. More specifically, its recomputation rate (number of recomputations divided by the number of window slides) increases from 23% to 56%, as k increases from 10 to 10K. This indicates that the recomputation process is more frequently needed in SMA when the ratio between k to the slide size increases. This is because, when k is large, it is more likely that the same amount of new objects cannot re-fill the skyband to reach the size of at least k. Again, such recomputation process is needed by neither PreTopk nor MinTopk. In general, the huge CPU time savings of PreTopk and MinTopk are achieved by eliminating the need for expensive index maintenance and top-k recomputation.

The memory consumption of both PreTopk and MinTopk is negligible compared with that used by SMA in all test cases, and especially when k

is small. The reason is obvious. Namely, both PreTopk and MinTopk only keep the "necessary" objects (MTK), whose size is only $2k$ on average (see Theorem 8.1 in Chapter 7.2), while SMA needs to keep all 1M objects alive in the window. Our measurement of the size of MTK, namely the length of *super-top-k list*, also confirms Theorem 8.1, as in all four test cases, the size of MTK never exceeds $3.5k$ and is $2.4k$ on average.

The performance of MinTopk is also better than PreTopk in all these test cases. In particular, MinTopk uses on average 23% less processing time and 33% less memory. Such comparable performance of these two algorithms is caused by the relatively large $slide/win$ rate adopted in this experiment, which makes the number future windows maintained by both algorithms small (only 10 for all cases). We will further analyze this issue in experiment 3.

9.4.5 Evaluation for Different win Cases

Next, we evaluate the effect of the window size win on the algorithms. We use the GMTI data for this experiment. We fix the value of k at $1K$ and the *slide/window* rate at $\frac{1}{10}$, while varying win from $1K$ to $1M$. As shown in Figures 16.11 and 16.12, both the CPU and memory usage of PreTopk and MinTopk are significantly less than those utilized by SMA in all test cases, and especially when win is large.

In particular, both the CPU and memory usage of SMA increase dramatically as the window size increases. This is expected, because it requires full storage of all objects alive in the window and thus its memory usage increases almost linearly with the window size. The increase of the CPU time for SMA is also obvious, while less sharp than that of its memory utilization. This is because to process the same number of objects, no matter what the window size is, SMA needs to load and purge each object once into the index. Thus this part of the processing time is the same for all test cases. The increase of CPU time for SMA is primarily caused by the increasing cost of top-k recomputation in larger windows.

Both the CPU and memory usage of PreTopk and MinTopk are not affected by the window size. This confirms our cost analysis in Sections 7.2 and 7.4, namely the costs of PreTopk and MinTopk are independent from the window size.

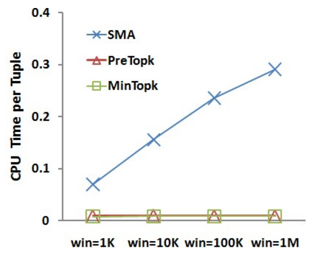

Figure 9.18: CPU times for varying window sizes

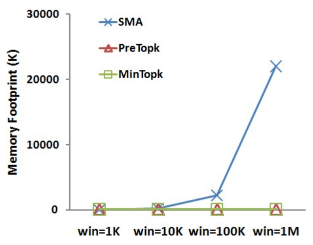

Figure 9.19: Memory space for varying window sizes

9.4.6 Evaluation for Different $slide$ Cases

In this case, we evaluate the effect of the window size win on the algorithms. We use the STT data for this experiment. We fix the value of k at $1K$ and window size at 1M, while varying the slide/window ratio from $0.01\% - 10\%$, namely the slide size from 100 to $100K$.

As shown in Figures 16.7 and 16.8, both the CPU and memory usage of MinTopk are still significantly less than those utilized by SMA in all test

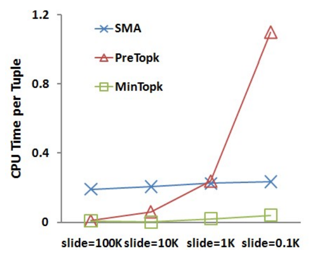

Figure 9.20: CPU times for varying slide sizes

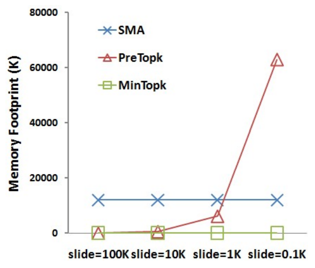

Figure 9.21: Memory space for varying slide sizes

cases. In particular, in the $slide = 100k$ case, MinTopk only takes 0.097 ms to process each object on average, while SMA needs 0.172 ms for each object. In terms of the response time needed for processing each window, this means that our method only takes around 10 seconds to update the query result for each window slide (100K new objects), while SMA needs more than 2 minutes, This is as expected and can be explained by the same reasons as in the previous test cases. However, an important observation made from this experiment is that the performance of PreTopk can be strongly affected by the *slide/win* rate. In particular, both the CPU and memory usage of PreTopk increase dramatically as the slide/window rate decreases. Its performance is comparable with MinTopk when the $slide/window$ rate is 10%, while it gets even worse than SMA when it decreases to 0.01%. This is as expected. Since PreTopk maintains the predicted top-k results for each future window independently, its resource utilization increases linearly with the number of future windows maintained, which is equal to $\lceil \frac{win}{slide} \rceil$ (see cost analysis in Chapter 7.2).

The performance of both SMA and MinTopk are not affected by the change of *slide/win* rate. Their average processing time even drops a little bit, because to process the same amount of objects, the larger slide size will cause less frequent output and thus requires less output cost.

In conclusion, although PreTopk shows comparable performance with MinTopk in the cases in which the *slide/win* rate is modest, its performance can be very poor when the *slide/win* rate is small. In short, the performance of MinTopk has been shown to be stable under any parameter settings.

9.4.7 Evaluation for Non-Uniform Arrival Rate Cases

In this experiment, we evaluate the algorithms' performance under non-uniform arrival rate. Namely, the number of objects that arrive at each window slide varies in this case. We use the STT dataset and use the real time stamp of each transaction to present the arrival rates of the stream. The average arrival rate of the transactions in this data is around 400 transactions each second, while the actual arrival rate at each window slide varies significantly depending on the choice of slide size and the particular time period.

As in this experiment the number of objects needed to be processed for each window varies significantly, instead of measuring the average CPU time for processing each tuple, we measure the response time for processing each window, namely the accumulative time for answering the query at each window slide from all objects arrived to results outputted. We evaluate three different cases with slide size $slide$ equal to 1, 10 and 100 seconds respectively, while we fix the window size at 1000 seconds and k equal to 1K for all three cases. For each case, we run the query for 10K windows and we measure the minimum, maximum, average and standard deviation of the response time at each window.

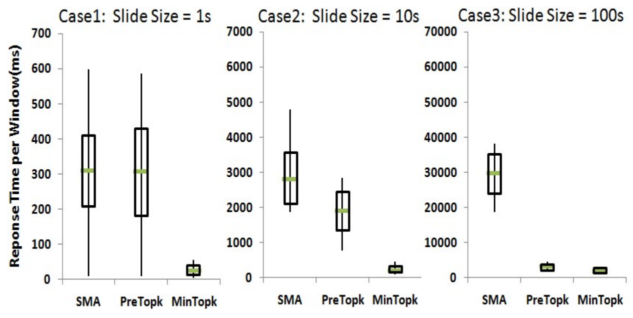

Figure 9.22: Response time for processing each window given non-uniform arrival rate.

As shown in Figure 9.22, we observe that given the non-uniform arrival rate, the response time of all three algorithms varies by slide sizes. In general, when the slide size is small, as in the $Slide = 1s$ case, the variations of the response time tend to be very large. This is because given very short time period granularities, the number of objects arriving at each window can vary significantly. As shown in the Case 1 in Figure 9.22, the minimum response time of all three algorithms are very close to zero. This is because in some windows very few objects arrived (less than 20), and thus they only required limited computation. While when slide size increases, as in the $Slide = 10s$ and $Slide = 100s$ cases, we can observe that the variations of the response time of all algorithms tend to be smaller. This is because the unevenness of the arrival rates in short time periods is now averaged in the larger time frames.

However, no matter what kind of variations exist in the arrival rate, our proposed MinTopk algorithm still shows obvious superiorities to the other two alternatives. Since the overall trends for all three competitor algorithms observed are similar to those trends for the uniform arrival rate cases, the results can again be explained using similar reasoning as given in the previous experiments.

Chapter 10

Related Work for Part I

Within our target neigbor-based pattern types, density-based clustering was first proposed in [EKSX96] as DBSCAN algorithm for static data.

Traditionally, the pattern detection techniques are designed for static environments, where large amounts of data are being collected and stored. In this work, our target pattern types are density-based clusters [EKSX96, EKS$^+$98], distance-based outliers [KN98b, KN99] and kNN patterns. They are all popular pattern types defined by local neighborhood properties. Well-known algorithms for detecting those patterns from static datasets include [ZRL96, HW, GRS98, EKSX96, ABKS99] for density-based cluster, [Ord96, RR96, BKNS00, KN98b, KN99] for distance-based outliers and [TWS04, CwH02] for kNN (top-k) pattern. These techniques designed for the static environment are based on two assumptions. First, all relevant data is a priori available, either locally or on distributed servers, before query execution. Second, the pattern detection queries are ad-hoc queries executed only one single time. However, both assumptions do not hold for streaming environments in which data are continuously coming in and the top-k queries are continuously re-executed. Thus, clearly, these techniques cannot be directly used to solve our problem, namely continuous neighbor-based pattern detection in streaming environments.

More recently, pattern detection on streaming data began to be studied. The earlier clustering algorithms applied to data streams [GMMO00, GMM$^+$03] are global clustering algorithms adapted from the static k-means algorithm. They treat the data stream clustering problem as a continuous version of static data clustering. They treat objects with different time horizons (recentness) equally and thus do not reflect the temporal features of data streams. Later, [AHWY03] presented a framework of clustering streaming data using a two stage process. At the first stage, the online component works on the streaming data to summarize it into micro-clusters. At the second stage, an offline component clusters the micro-clusters formed earlier to form final clustering results using a static k-means algorithm. In this framework, a subtraction function is used to discount the effect of the earlier data on the clustering results. Several extensions have been made to this work, focusing respectively on clustering distributed data streams [BGM$^+$06], multiple data streams [DHYC06], and parallel data streams [BH06]. None of these works described above deal with the arbitrarily shaped local clusters, nor do they support sliding window semantic. [BDMO03] is the only work we are aware of that discusses the clustering problem with sliding windows. However, it again is a global clustering algorithm maintaining approximated cluster centers only.

[EKS$^+$98] presented techniques to incrementally update density-based clusters in data warehouse environments. Since all optimizations in this work are designed for a single update (a single deletion or insertion) to the data warehouse, this fits well for the relatively stable (data warehouse) environments but is not scalable for sustainable environments. [CT07, CEQZ06] also studied the problem of detecting density-based clusters over streaming data. However, [CT07, CEQZ06] do not identify the individual members in the clusters as required by the application scenarios described earlier in the introduction. Also, to capture the dynamicity of the evolving data, they both use decaying factors derived from the "age" information of the objects. These decaying factors put lighter weights on older objects during the clustering processes. This approach emphasizes the recent stream portion more compared to the older data, still it does not enforce the discounting of old data's effect from the pattern detection results. So, they are not suitable for the applications requiring sliding window scenarios discussed in this work, which can only consider the most recent data only.

The problem of detecting outliers in data streams has been studied by [YiTWM00, Agg05, SPP$^+$06, AF07]. Among these works, [YiTWM00, Agg05] work with outliers with different definitions from ours. Thus, these techniques cannot be applied to detect distance-based outlier as targeted by our method. [SPP$^+$06] study the detection of distance- and MEDF-based outliers in hierarchically structured sensor networks. Also, the outliers detection is based on approximated data distribution. So, this work has a different goal from us. Most similar to our work, [AF07] introduced an algorithm to detect the distance-based outliers within sliding window scenario. However, this work only deals with count-based windows, where the number of objects in the window is aprioir known and fixed. Both our analytical and experimental studies reveal that this method is not suitable for answering queries with the time-based windows, where each window may have different numbers of objects. Our experimental studies confirm that our work instead is efficient in handling both cases.

Researchers have also started to look at the problem of processing **top-k queries in streaming environments** [MBP06, HMA09, JYC$^+$10, JYC$^+$08]. Among these works, [MBP06] is the closest to our work in that it also tackles the problem of exact continuous top-k query monitoring over a single stream. This work presents two techniques. First, the TMA algorithm computes the new answer of a query whenever some of the current top-k points expire. Second, the SMA algorithm partially precomputes the future changes in the result, achieving better running time at the expense of slightly higher space requirements. More specifically, as the key contributions of this work, SMA maintains a "skyband structure" which aims to contain more than k objects. This idea is similar to the one used in [YY$^+$03] for materialized top-k view maintenance. However, unfortunately, neither of these two algorithms eliminate the recomputation bottleneck (see Chapter 7.2) from the top-k monitoring process. Thus, they both require full storage of all objects in the query window. Furthermore, they both need to conduct expensive top-k recomputation from scratch in certain cases, though SMA conducts recomputation less frequently than TMA. While our proposed algorithm eliminates the recomputation bottleneck, and thus realizes completely incremental computation and minimal memory usage.

[HMA09, JYC$^+$08, JYC$^+$10] handle incomplete and probabilistic top-k models respectively in data streams. while we work with a complete and non-probabilistic model. Thus, they target different problems from ours. In particular, the key fact affecting the top-k monitoring algorithm design is the meta information maintained for real-time top-k ranking and the corresponding update methods , which vary fundamentally by specific top-k models. For example, due to the characteristics of the incomplete top-k model, [HMA09] proves that maintaining a object set with its size linear in the size of the sliding window is necessary for incomplete top-k query processing. While in our (complete) top-k model, we maintain a much smaller object set, whose size is independent from the window size but linear in the query parameter k only (see *Lemma* 8.1). Similarly, due to the characteristics of uncertain top-k models, [JYC$^+$08, JYC$^+$10] maintain a significantly larger amount of meta information, namely a series of candidate top-k object sets (they call Compact Sets) that contain more objects than the Minimal Top-k Candidate Set (MTK) identified and maintained in this work. These candidate objects are organized and updated in different manners from us to serve their specific models. In general, those specific data structures and the corresponding update algorithms designed for other top-k methods are not optimized for our problem and thus do not achieve the optimal complexities in system resource utilization for our problem.

Part II

Multiple Neighbor-Based Pattern Extraction Query Optimization

Chapter 11

Problem Definition

We provide specialized shared query processing strategies for workloads of the same query type. The queries in each workload must be specified on a commom input stream but with arbitrary pattern and window parameters. We call all the queries submitted to the system together a Query Group QG, and each of them a Member Query of QG. All the queries within a same query group should have the same query type, be it density-based clustering, distance-based outlier detection or kNN queries. We use a common assumption that all the member queries are registered to and pre-analyzed by our system before the arrival of the input stream, indicating that all the member queries will be started simultaneously [1]. Our goal is to minimize the overall CPU- and memory- resource consumption for executing all the member queries registered to our system.

[1]In my dissertation, I do not study the problem of handling the dynamic queries, which means the queries may be added or removed from the workload during the execution. Instead, this topic is an important part of my future work (See Chapter 26.3)

Chapter 12

A Preliminary Sharing Effort: Share Range Query Searches

The basic strategy to share the computations among multiple neighbor-based pattern mining queries is to share the range query searches. Generally, to execute a query group QG with $|QG| = N$, we can execute N single query algorithms, each for a member query, independently (with each query maintaining its own progressive patterns independently), yet share the computations needed by the range query searches. Specifically, at each new query window, the single query algorithms require every new data point p_{new} to run a range query search to identify its neighbors, and communicate with them to update progressive patterns in the window. This means that, if executed independently, for a query group QG with $|QG| = N$, we need to run N range queries for each new data point p_{new}. However, by using range query search sharing, we could instead run just one range query search for each p_{new}, even if the queries in QG have different range thresholds θ^{range}.

In particular, we run the range query search for each p_{new} using $Q_i.\theta^{range}$, with $Q_i.\theta^{range}$ larger or equal to any $Q_j.\theta^{range}$ in QG. Using the result set of this "broadest" range query search, we then gradually filter out the results for the other queries with smaller and smaller θ^{range}. Clearly, for a given data point, the result set of a range query search using smaller θ^{range} is always a subset of that using a larger one. Also, since the range query search with largest θ^{range} is in any case needed for the particular query, no extra computation is introduced by this process. This general principle of sharing range query searches can be applied to any neighbor-based pattern mining request that requires neighbor searches for data points, such as distance-based outlier detection. Sharing range query searches can be very beneficial for optimizing the system resource utilizations, especially when the window size is large.

Discussion. However, sharing range query searches alone is not sufficient for handling a heavy workload containing hundreds or even thousands of queries. Two critical problems still remain: 1) Since every member query still stores its progressive patterns independently, the memory space needed by executing a query group QG grows linearly with $|QG|$. 2) Because of the independent pattern storage, the pattern maintenance computation of different queries cannot be shared. To solve these two problems, we need to further analyze and exploit the commonalities among the member queries. Our goal is thus to design an integrated pattern maintenance mechanism that effectively shares both the storage and computational resources needed for multiple queries.

In our experimental studies shown in Figures 30, 31 and 32 in Chapter 16 we demonstrate how much performance gains that can be achieved by using this range query search sharing strategy alone. Also, in the same experiments we compare such gains with those can be achieved by our complete proposed solutions, which include more sophisticated sharing strategies introduced in the following Chapters 13 and 14.

Chapter 13

Sharing Among Queries with Arbitrary Pattern Parameters

In this Chapter, we discuss the shared processing of multiple queries with arbitrary pattern parameters. We first assume that the queries have the same window parameters, namely the same window size win and the same slide size $slide$. In such cases all the member queries will always detect patterns from exactly the same portion of the data streaming data (those fall into the current query window). This assumption will later be relaxed in Chapter 15 to allow completely arbitrary parameters.

To solve this problem, we analyze the relationships between the pattern sets identified by neighbor-based pattern mining queries with different pattern-specific parameter setting. In particular, we characterize the conditions under which one query is "more restricted" than the other, and discover that a "containment" relationship holds between the pattern sets identified by the queries following such "strictness order". By exploiting this containment relationship, we incrementally organize the patterns identified by multiple queries into an integrated structure, and thus manage to maintain them in a shared manner. Such shared execution strategy leads to significant savings in both CPU time and memory utilization.

13.1 "Containment" among Neighbor-Based Pattern Sets

The definition of "containment" between neighbor-based pattern sets is generally more complex than the traditional "containment relationship" between the result sets of SPJ queries. In particular, such containment among neighbor-based pattern sets is not restricted to simple super- or subset relationships. Here we first use density-based clusters, which have one of the most complicated pattern structures and complex containment relationships among the neighbor-based pattern family, to explain this concept.

13.1.1 "Growth Property"

We call the specification of such containment relationship among density-based cluster sets the "Growth Property". We now first define the "containment" between two density-based clusters.

Definition 5 *Given two density-based clusters C_i and C_j (each cluster is a set of data points, which are called cluster members of this cluster), if for any data point $p \in C_i$, $p \in C_j$, we say that C_i is **contained** by C_j, denoted by $C_i \subset C_j$.*

We now give the definition for the "growth property" between two density-based cluster sets.

Definition 6 *Given two cluster sets Clu_Set1 and Clu_Set2 with for $i = 1, 2$, $Clu\_Set_i = \bigcup_{1 \leq x \leq n} C_x$, and for any $y \neq z$, $C_y \cap C_z = \emptyset$. If for any C_i in Clu_Set1, there exists exactly one C_j in Clu_Set2 that $C_i \subset C_j$, Clu_Set2 is defined to be a "**growth**" of Clu_Set1. We say the growth property holds between Clu_Set1 and Clu_Set2.*

Beyond this definition, we now characterize all the possible interrelationships between the clusters belonging to Clu_Set1 and Clu_Set2.

Observation 13.1 *Given Clu_Set1 and Clu_Set2 with Clu_Set2 a **growth** of Clu_Set1, then any cluster C_j in Clu_Set2 must either be a **new** cluster (for any $p \in C_j$, $p \notin C_i$, if C_i is in Clu_Set1), an **expansion** of a single cluster in Clu_Set1 (there exists exactly one C_i in Clu_Set1 such that $C_i \subset C_j$), or a **merge** of multiple clusters in Clu_Set1 (there exist C_i, C_{i+1},...C_{i+n}($n > 0$) in Clu_Set1 with C_i, C_{i+1},...$C_{i+n} \subset C_j$.*

Figures 13.1 and 13.2 give an example of two cluster sets between which "growth property" holds.

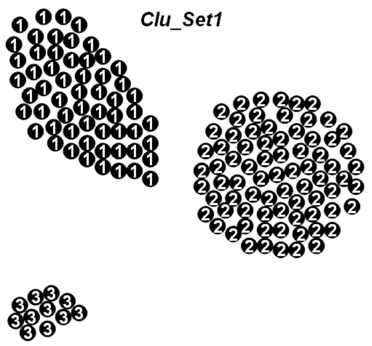

Figure 13.1: Cluster Set 1 containing 3 clusters

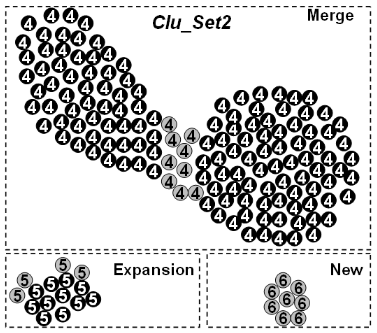

Figure 13.2: Cluster Set 2 containing 3 clusters, which is a growth of Cluster Set1

The black spots in the figures represent the data points belonging to both cluster sets, while the gray ones represent those belonging to Clu_Set2 only. As depicted in the figures, the cluster C_4 in Clu_Set2 is a "merge" of clusters C_1 and C_2 in Clu_Set1, while the cluster C_5 and cluster C_6 in Clu_Set2 are an "expansion" of cluster C_2 in Clu_Set1 and a "new" cluster respectively. Generally, if Clu_Set2 is a "growth" of Clu_Set1, any two data points belonging to the same cluster in Clu_Set1 will also be members of the same cluster in Clu_Set2.

13.1.2 Hierarchical Pattern Representation

If the "growth property" transitively holds among a sequence of cluster sets, a hierarchical cluster structure can be built across the clusters in these cluster sets. The key idea is that, instead of storing cluster memberships for different cluster sets independently, we incrementally store the cluster "growth information" from one cluster set to another. Figures 13.3 and 13.4 respectively give examples of independent and hierarchical cluster membership structures built for the two cluster sets shown in Figures 13.1 and 13.2.

As shown in Figure 13.3, if we store the cluster memberships for cluster members in these two cluster sets independently, each cluster member (black squares) belonging to both clusters has to store two cluster memberships, one for each cluster set. However, if we store them in the hierarchical cluster membership structure as depicted in Figure 13.4, we no longer need to repeatedly store the cluster memberships for these "shared" cluster members. Instead, we simply store cluster memberships for each cluster

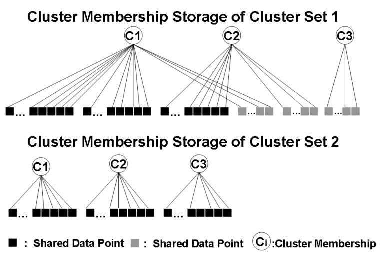
Cluster Membership Storage of Cluster Set 1
C1
C2
C3
Cluster Membership Storage of Cluster Set 2
C1
C2
C3
■ : Shared Data Point ■ : Shared Data Point Ci :Cluster Membership

Figure 13.3: Independent Cluster Membership Storage for Cluster Sets 1 and 2

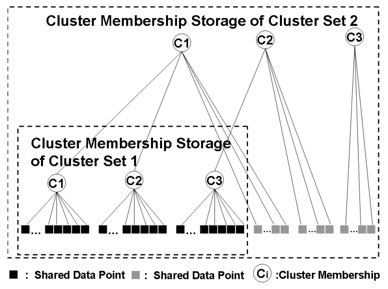
Cluster Membership Storage of Cluster Set 2
C1
C2
C3
Cluster Membership Storage of Cluster Set 1
C1
C2
C3
■ : Shared Data Point ■ : Shared Data Point Ci :Cluster Membership

Figure 13.4: Hierarchical Cluster Membership Storage for Cluster Sets 1 and 2

member belonging to Clu_Set1, and then store the cluster "growth" information from Clu_Set1 to Clu_Set2. In particular, we just need to correlate each cluster C_i in Clu_Set1 with a cluster in Clu_Set2 that contains it, and thereafter each cluster member can easily find its cluster membership in a specific cluster set by tracing to the corresponding level of the hierarchical cluster membership structure. Such "growth" information is that now based on the **granularity of complete clusters** rather than the **granularity of individual cluster members**. Generally, for a sequence of cluster sets for which the "growth property" transitively holds, the hierarchical cluster structure can largely save the memory space needed for storing them.

Lemma 13.1 *Given a query group QG for which the growth property transitively holds among the cluster sets identified by all its member queries, the upper bound of the memory space needed for storing the cluster memberships using hierarchical cluster structure is* $2 * N_{core}$ *(independent from* $|QG|$*), with* N_{core} *the number of distinct data points that are at least once identified as core point in any member query of QG.*

Proof 13.1 *The relationship between the number of cluster memberships stored and N_{core} is equal to the relationship between the total size of a binary heap and the number of leaf nodes of this heap. This is because a higher level cluster membership will only be stored if a merge of the cluster memberships happened at the lower level.*

Besides the benefit of potentially huge memory savings, such hierarchical cluster structure can also help us to realize the integrated maintenance for multiple cluster sets identified by different queries, and thus save computational resources from maintaining them independently. In the later parts of this work, we will carefully discuss how this general principle can be used to benefit our multiple query optimization strategy.

13.1.3 Incremental Representation for Other Pattern Types

The containment relationship of other neighbor-based pattern types, such as distance-based outliers and top-k nearest neighbors are simpler than density-based clusters. In particular, as the outlier set or nearest neighbor set identified by a query is simply an object set, the containment relationship between any two outlier sets or nearest neighbor sets is simply the super- or sub-set relationship. Thus, the incremental representation of such pattern sets is simple as well. More precisely, we can store the smallest sets first and then incrementally store the extra objects for larger and larger sets.

13.2 Integrated Maintenance for Multiple Density-Based Clustering Queries

Now we discuss, for density-based clusters, in which cases such containment relationship holds and how it can help us to conduct shared execution for multiple queries. For a group of density-based clustering queries, they can vary on both pattern parameters, namely θ^{range} and θ^{cnt}. We first look at the cases in which the variations are only allowed on one parameter.

Arbitrary θ^{cnt}/θ^{range} Cases

In the first case, all queries have the same θ^{range} but arbitrary θ^{cnt}. Here, we make a straightforward observation.

Observation 13.2 *Given all queries in a query group having the same θ^{range}, the neighbors of each data point identified by these queries are the same.*

This observation indicates that for all our member queries, the neighborships identified in each specific window are exactly the same. However, this does not mean that the cluster structures identified by all queries are same and we can store them in the same way. This is because the different $\theta^{cnt}s$ of the member queries may assign different "roles" to a data point. For example, a data point with 4 neighbors is a "core point" for query Q_1 having $Q_1.\theta^{cnt} = 3$, while it is a "none-core point" for Q_2 having query $Q_2.\theta^{cnt} = 10$. As the hybrid neighborship abstraction (discussed earlier in Chapter 5.2.6) requires each none-core point to store the links to its exact neighbors, while the core points store the cluster memberships only, a data point may need to store different types of neighborship abstractions depending on its roles identified by different queries.

To solve this problem, we turn to the "growth property" of density-based cluster structure discussed in Chapter 13.1.

Lemma 13.2 *Given two queries Q_i and Q_j specified on the same dataset, with $Q_i.\theta^{range} = Q_j.\theta^{range}$ and $Q_i.\theta^{cnt} \leq Q_j.\theta^{cnt}$, the cluster set identified by Q_i is a "growth" of the cluster set identified by Q_j (see **growth property** as defined in Definition 6).*

Proof 13.2 *First, since $Q_i.\theta^{cnt} \leq Q_j.\theta^{cnt}$, the "core point" set identified by Q_j is a subset of that identified by Q_i. Second, since all the neighborships identified by Q_i and Q_j are exactly the same, all the "connections" in any cluster structure identified by Q_j will also hold for Q_i. This indicates that the cluster structure identified by Q_j will also be identified by Q_i (although it may be further expanded or merged). Finally, the "additional" core points identified by Q_i may only cause the birth of new clusters or expansion or merge of the clusters identified by Q_j, because they either extend these cluster structures when they are "connected" to one or more of them (causing expansion or union) or form new clusters by themselves when they are not "connected" to any (causing birth). This indicates that the cluster set identified by Q_i is a "growth" of that identified by Q_j (by Observation 13.1). This proves the lemma 13.2.*

Figure 13.5 demonstrate an example of the cluster sets identified by three queries having the same θ^{range} but different $\theta^{cnt}s$.

Integrated Representation of Predicted Views across Multiple Queries with Arbitrary θ^{cnt}. As we discussed earlier in Chapter 13.1, once the

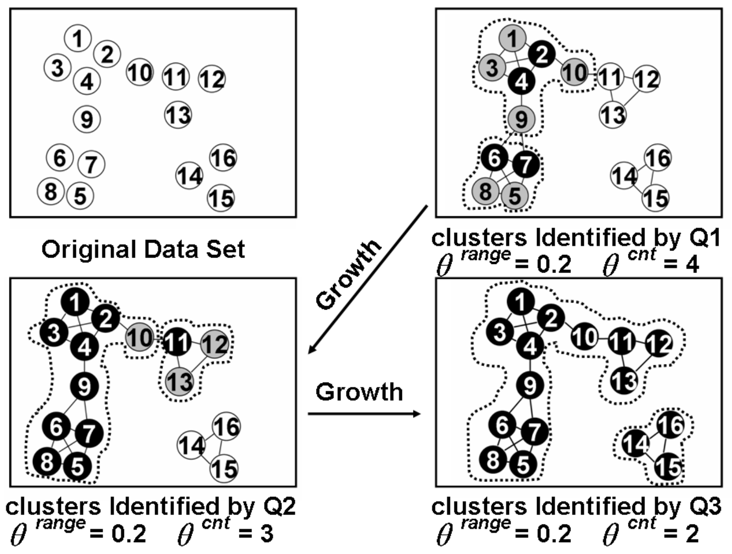
Original Data Set
clusters Identified by Q1
θ^{range} = 0.2 θ^{cnt} = 4
Growth
Growth
clusters Identified by Q2
θ^{range} = 0.2 θ^{cnt} = 3
clusters Identified by Q3
θ^{range} = 0.2 θ^{cnt} = 2

Figure 13.5: Cluster sets identified by three different queries

"growth" property holds among the cluster sets, we can build the hierarchical cluster structure for them. We thus build an integrated hierarchical structure to represent multiple predicted views identified by different queries for the same corresponding predicted window. We refer to such Integrated Representation of Predicted Views across Queries with arbitrary θ^{cnt} by $IntView\_\theta^{cnt}$. For each predicted window, $IntView\_\theta^{cnt}$ starts from the predicted view with the most "restricted clusters". In this context, this corresponds to the predicted view maintained by Q_i with the largest θ^{cnt} among QG. Then, it incrementally stores the cluster "growth information", namely the "merge" of existing cluster memberships and the new cluster memberships, from one query to the next in the decreasing order of θ^{cnt}. Figure 13.6 gives an example of an $IntView\_\theta^{cnt}$, which represents the predicted views (shown in Figure 13.5) identified by three different queries.

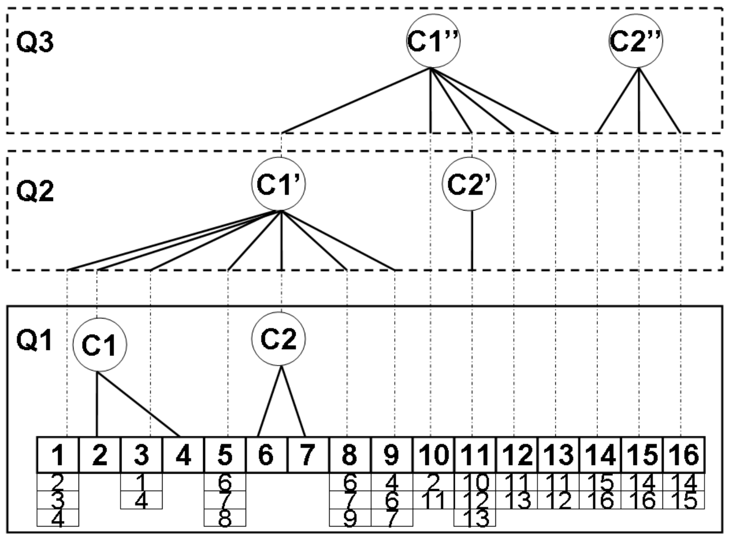

Figure 13.6: $IntView\_\theta^{cnt}$: Integrated Representation for density-based clusters identified by three different queries

$IntView\_\theta^{cnt}$ successfully integrates the representations of multiple "predicted views" into a single structure, thus saving the memory space otherwise needed to store them independently.

Lemma 13.3 *Given the maximum window size allowed, the upper bound of the memory space needed by $IntView\_\theta^{cnt}$ is independent of $|QG|$, the cardinality of the query group.*

Proof 13.3 *First, there are two types of meta-information that need to be stored by $IntView\_\theta^{range}$, namely the cluster memberships and the exact neighbors of the data points. Since $IntView\_\theta^{range}$ uses the hierarchical structure described in Chapter 13.1 to store the cluster memberships for the data points, the upper bound of the memory space used for storing cluster memberships is independent from $|QG|$ (as proven in Lemma 13.1). Second, $IntView\_\theta^{cnt}$ only stores the* *exact neighbors for "non-core" data points, and the maximum number of exact neighbors a "non-core" point can have is a constant (namely, $max(Q_i.\theta^{cnt}) - 1$). Thus, the upper bound of the memory space used for storing exact neighbors is again independent from $|QG|$. This proves Lemma 13.3.*

Without using $IntView\_\theta^{cnt}$, the memory space needed for independently storing the cluster memberships identified by all member queries in QG will increase linearly with $|QG|$. Our method now makes it independent from $|QG|$ (as proven in Lemma 13.3).

Maintenance of $IntView\_\theta^{cnt}$. Besides the memory savings, we can also incrementally update multiple predicted views represented by a $IntView\_\theta^{cnt}$, thus saving computational resources. In particular, for each new data point p_{new}, we start the update process from the bottom level of $IntView\_\theta^{cnt}$, namely the predicted view identified by the query with largest θ^{cnt}. Then we incrementally propagate the effect of inserting this new data point to the next higher level of predicted views. Using the example utilized earlier in Figure 13.6, a new data point identified to have 3 neighbors in the window is a "none-core" in the bottom (most restricted) level predicted view, where $\theta^{cnt} = 4$. So, at the bottom level, we simply add all its neighbors to its neighbor list. However, its effect to upper level predicted views may differ, as this data point may be identified as a "core point" by a more "relaxed" query, say when $\theta^{cnt} = 3$. Then, we need to generate a cluster membership for it at that predicted view and merge it with those cluster memberships (if any) belonging to its neighbors.

The pseudo-code for the maintenance algorithm of $IntView\_\theta^{cnt}$ can be found in Figures 15.2 and 15.3, which is a special case of our final solution *Chandi*. In this special case, besides the exact same predicted windows built for all queries, the neighbor sets of a new data point identified by all queries are exactly same. We emphasize that the maintenance process is efficient for the following two reasons: 1) No extra range query search is needed when a data point is found to be a "core point" in an upper level predicted view and thus needs to communicate with its neighbors. This is because as a "none core point" in the lower level predicted views, it would already have stored the links to all its exact neighborships and thus would have direct access to them. 2) As the "growth" of cluster sets identified in predicted views is incremental, less and less maintenance effort will be needed as we handle the higher level predicted views.

We also found that the "growth property" holds between two queries with the **same** θ^{cnt} **but different** θ^{range}**s**.

Lemma 13.4 *Given two queries Q_i and Q_j specified on the same data set with $Q_i.\theta^{cnt} = Q_j.\theta^{cnt}$ and $Q_i.\theta^{range} \geq Q_j.\theta^{range}$, the cluster set identified by Q_i is a "growth" of that identified by Q_j.*

The shared execution strategy for this case is very similar to the previous case in which all queries have the same θ^{range}s. Thus, the detailed discussion for this case is omitted here.

Arbitrary θ^{range}, Arbitrary θ^{cnt} Case

Now we discuss the shared processing for a query group QG with queries having totally arbitrary pattern parameters, namely arbitrary θ^{range} and arbitrary θ^{cnt} values. Although the "growth property" holds between the cluster sets identified by two queries Q_i and Q_j, if Q_i and Q_j share at least one query parameter, it does not necessarily hold if both query parameters of Q_i and Q_j differ. To again take advantage of the compact structure of the Integrated Representation of Predicted Views, we need to explore when the "growth property" holds between two queries in the most general cases.

Lemma 13.5 *Given two queries Q_i and Q_j specified on the same dataset, with $Q_i.\theta^{cnt} \leq Q_j.\theta^{cnt}$ and $Q_i.\theta^{range} \geq Q_j.\theta^{range}$, the cluster set identified by Q_i is a "growth" of that identified by Q_j.*

Proof 13.4 *Lemma 13.5 can be proven by the transitivity of the "growth property". Given a query Q_k with $Q_i.\theta^{cnt} \leq Q_k.\theta^{cnt} \leq Q_j.\theta^{cnt}$ and $Q_k.\theta^{range} = Q_j.\theta^{range}$, the cluster set identified by Q_k is a "growth" of that identified by Q_j (by Lemma 13.2). This means that for any cluster C_a identified by Q_j there exists a cluster C_b identified by Q_j such that $C_a \subseteq C_b$. Also, the cluster set identified by Q_i is a "growth" of that identified by Q_k (by Lemma 13.4). This means that for any cluster C_b identified by Q_j there exists a cluster C_c identified by Q_i that $C_b \subseteq C_c$. So for any cluster C_a identified by Q_j there exist a cluster C_c identified by Q_i such that $C_a \subseteq C_c$. Thus, the cluster set identified by Q_i is a "growth" of that identified by Q_j (by Definition 6).*

To more intuitively describe the relationship between any two queries in a query group, we give the following definition.

Definition 7 *Given two queries Q_i and Q_j specified on the same dataset, if $Q_i.\theta^{cnt} \leq Q_j.\theta^{cnt}$ and $Q_i.\theta^{range} \geq Q_j.\theta^{range}$. we say Q_j is a "more restricted" query than Q_i, and Q_i is a "more relaxed" query than Q_j.*

Integrated Representation of Predicted Views across Multiple Queries with Arbitrary Pattern Parameters. We aim to build a single structure which represents the "predicted views" identified by all member queries of QG in the same window. However, given the "growth property" only holds between two queries if one is more restricted than the other, we can no longer expect to put all member queries into a single hierarchy.

Our solution is to build a "**Predicted View Tree**", which integrates multiple linear predicted view hierarchies into a single tree structure. In this tree structure, each predicted view (except the root) only needs to store and maintain the incremental information (cluster "growth") from its parent, much like the predicted views in $IntView\_\theta^{range}$ and $IntView\_\theta^{cnt}$. In particular, such a "Predicted View Tree" starts from the predicted view that represents "the most restricted query" among QG. "The most restricted query" here corresponds to the member query that has both the smallest θ^{cnt} and the largest θ^{range} among QG. If such a "most restricted query" does not naturally exist in QG, we build a "virtual" one by generating a query with the smallest θ^{cnt} and the largest θ^{range} among QG. The predicted view representing this "most restricted query" will be the "root" of our "Predicted View Tree". If the most restricted query is a virtual query, its predicted view will be used for "Predicted View Tree" maintenance but it will never generate any output. Then the predicted views representing more relaxed queries will be iteratively put on the next higher level (farther from the root) of the tree. More specifically, after picking "the most restricted query" as the root of the tree, we iteratively pick (and remove) "the most restricted queries" remaining in QG and put their predicted views as the next level of the tree. Here, a member query Q_j is one of "the most restricted queries" remaining in QG, if there does not exist any other member query Q_i in QG, which is "more restricted" than Q_j.

For example, given $QG = \{Q_1(\theta^{range} = 0.5, \theta^{cnt} = 5), Q_2(\theta^{range} = 0.4, \theta^{cnt} = 7), Q_3(\theta^{range} = 0.2, \theta^{cnt} = 10), Q_4(\theta^{range} = 0.3, \theta^{cnt} = 7), Q_5(\theta^{range} = 0.4, \theta^{cnt} = 8)\}$. The root of the "Predicted View Tree" is the predicted view representing "the most restricted query", namely Q_3 in this case. Then, the second level "most restricted queries" are Q_4 and Q_5, which are more relaxed than Q_3 but more restricted than Q_1 and Q_2. Finally, the third level "most restricted queries" are Q_1 and Q_2. This process of figuring out "the most restricted queries" at each level is equal to the problem of iteratively calculating the "skyline" [YLL$^+$05, ZMC09, SIK07] in the two dimensional space of θ^{range} and θ^{cnt}. Since this process of building "Predicted View Tree" can be conducted offline during query compilation, any existing skyline algorithm can be plugged into our system to solve this problem.

The predicted views on the lower level of the tree always represent more restricted queries than those on the higher levels. Then, the "growth information", namely the evolution of cluster memberships and the "additional exact neighbors", will be stored from one predicted view to each of its "children" on the higher level. Such building process guarantees an important property of "Predicted View Tree" as described in the following lemma.

Lemma 13.6 *Given a cluster set Clu_Set_m identified by a query Q_m on the i^{th} level of the "Predicted View Tree", and a cluster set Clu_Set_n identified by a query Q_n on the $(i-1)^{th}$ level, the "growth information" between Clu_Set_m and Clu_Set_n is no more than that between Clu_Set_m and any cluster set Clu_Set_o identified by a query Q_o on the $(i-j)^{th}(i > j > 1)$ level.*

Proof 13.5 *Since the queries on the $(i-j)^{th}$ level are always more restricted than those on the $(i)^{th}$ level, we know that Clu_Set_n is a growth of Clu_Set_o, Clu_Set_m is a growth of Clu_Set_n and Clu_Set_m is also a growth of Clu_Set_o. This means the "growth information" from Clu_Set_o to Clu_Set_m can actually be divided into two parts, namely the "growth information" from Clu_Set_o to Clu_Set_n and that from Clu_Set_n to Clu_Set_m. This proves that the "growth information" from Clu_Set_n to Clu_Set_m is no more than that from Clu_Set_o to Clu_Set_m.*

This property assures that each predicted view in the "Predicted View Tree" maintains the smallest increments and represent multiple predicted views as compact as possible.

To finalize the tree structure, for each query Q_n on the i^{th} level of the tree, we need to determine its "parent" on the $(i-1)^{th}$ level. We aim to find such a "parent" query Q_m that is most similar to Q_n, indicating that there exists the least "growth information" from the cluster set identified by itself to that identified by Q_n. Since the queries with similar $\theta^{range}s$ tend to identify similar neighborships in the window, this indicates that the difference on $\theta^{range}s$ has a larger influence on cluster changes compared with $\theta^{cnt}s$. So, when we determine the parent predicted view, although we consider the similarity between both pattern parameters, more "weight" is given to that between $\theta^{range}s$. To unify the names of the hierarchical structures representing multiple predicted views, we henceforth call this the "Predicted View Tree" $IntView\_\theta$.

Although $IntView\_\theta$ is a tree structure, instead of a linear sequence like $IntView\_\theta^{cnt}$ and $IntView\_\theta^{range}$, they share the core essence that each predicted view is incrementally built based on the predicted view most similar with it, and the "growth property" holds between them. We call the member queries on each path of $IntView\_\theta$ a group of ***shared queries***.

Lemma 13.7 *The upper bound of the memory space needed by $IntView\_\theta$ for any group of shared queries is independent from the number of queries in this group.*

Since all these queries are on the same path of $IntView\_\theta$ structure, indicating that the growth property transitively holds among the cluster set identified by them. The independency between the upper bound of the memory space and the number of queries can be proven using the same method as we used for proving Lemma 13.3.

The maintenance process of $IntView\_\theta$ is also similar with that for $IntView\_\theta^{cnt}$ and $IntView\_\theta^{range}$. For each new data point, we always start the maintenance from the root of the $IntView\_\theta$, namely the predicted view representing the most restricted query. Then we incrementally maintain the predicted views on the next higher level of $IntView\_\theta$. Again, this maintenance process is a special case of our final uniform solution *Chandi* (pseudo code shown in Figures 15.2 and 15.3). In this special case, the predicted windows built for all queries are exactly same.

Now we conclude with the contribution of $IntView\_\theta$.

Theorem 13.1 *For a given density-based clustering query group QG with member queries having arbitrary pattern parameters, $IntView\_\theta$ achieves full sharing of both memory space and query computation.*

Proof 13.6 *First, the storage mechanism of $IntView\_\theta$ is completely incremental. In particular, since each predicted view in $IntView\_\theta$ only stores the increments from its "parent", no duplicate information is ever stored among any two predicted views. This proves that $IntView\_\theta$ achieves full sharing of memory space. Second, since the maintenance process of $IntView\_\theta$ is incremental as well, indicating that each new data point only communicates with each of its neighbors once on each path of the tree structure, no matter how many different predicted views their neighborship appears in. This proves that $IntView\_\theta$ achieves full sharing of the computation of multiple queries.*

13.3 Integrated Maintenance for Multiple Distance-Based Outlier Detection Queries

For distance-based outlier detection over sliding windows, the solution for processing a single request is discussed in the literature [AF07, YRW09]. Using the most up-to-date technique, namely the Abstract-C algorithm [YRW09], the meta-information that needs to maintained to update the outliers is simple. In particular, for each data point in the window, its neighbor count will be sufficient to determine whether it is an outlier or not. Thus, for an individual query, the meta-information it maintains for each predicted view is the potential outlier set in the corresponding predicted window, namely the data points that are known to have less than $win * \theta^{fra}$ neighbors in that window, if no new data points were to join its neighborhood. Also, a predicted neighbor count will be maintained for each potential outlier in each predicted window. Such neighbor counts will be updated when the new data points come in. This then helps us to decide whether the data points should still be kept in the potential outlier set. More specifically, each new data point updates the predicted neighbor count of its own and of all its neighbors in each future window. The data points with too many neighbors (neighbor count larger or equal than $win * \theta^{fra}$) will be removed from the potential outlier set of a particular window, since they can no longer be identified as outliers.

Figure 13.7 shows the predicted views built for two queries Q_1 and Q_2 on a 2D dataset. That is $Q_1.\theta^{range} = 0.1, \theta^{fra} = 25\%, win = 8, slide = 2$ (for Q_1, a data point with 2 or less neighbors will be identified as an outlier), and Q_2 with $Q_2.\theta^{range} = 0.15, \theta^{fra} = 15\%, win = 8, slide = 2$ (for Q_2, a data point with 1 or less neighbors will be identified as an outlier) . Figure 13.8 (top 2 lines) shows the corresponding meta-information Abstract-C maintains for these two queries independently. More details of Abstract-C can be found in [YRW09].

By analyzing the semantics of distance-based outliers, we observe that the containment relationship holds between the outlier sets identified by different queries under certain conditions. First, let us fix the distance threshold θ^{range}, while varying the fraction threshold θ^{fra} among the member queries in a query group QG. We then note that the outliers identified by a query Q_i with the largest θ^{fra} among QG will always cover (be a superset of) all the outliers that are identified by other member queries. For any two queries Q_i and Q_j, if $Q_i.\theta^{range} \leq Q_j.\theta^{range}$, the outlier set identified by Q_j is a subset of that identified by Q_i. In other words, the identified outlier set "grows" monotonically as θ^{fra} increases. So, in this case, for each predicted window, we can store and maintain the potential outlier sets identified by different queries incrementally for all member queries. As shown in Figure 13.8, we can just store a single copy of the largest potential outlier set and use a flag (depicted with different levels of darkness) to distinguish by which queries each outlier is identified. Here a single integer flag will be sufficient to distinguish among all the possible combinations, because the "strictness" of the queries are exactly ranked. In other words, a data point can only be identified as outlier by Q_i, if it can be identified by all queries that have $Q_j.\theta^{fra} > Q_i.\theta^{fra}$. Also, since the neighbor count of each data point identified by all queries will always be same, we can simply maintain a single predicted neighbor count for each data point and use it to answer all queries. Thus, in this case both the storage and computation of the meta-information maintained by all queries are fully shared.

Second, if we fix the fraction threshold θ^{fra}, while varying the distance threshold θ^{range} among a query group QG, the outliers identified by a query Q_i with smallest θ^{range} among QG will always cover all the outliers that should be identified by the other member queries. For any two queries Q_i and Q_j, if $Q_i.\theta^{range} \geq Q_j.\theta^{range}$, then the outlier set identified by Q_j is a subset of that identified by Q_i. In other words, the identified outlier set "grows" monotonically as θ^{fra} decreases. This case is very similar to the previous case just discussed in the last paragraph. Thus, the same incremental maintenance mechanism of potential outlier sets is also applicable here. The only difference between this case and the previous case is now the neighbors identified by different queries for the same data point may be different. However, as the neighbor counts identified for any data point monotonically increase with the θ^{range} parameter, we simply maintain the increments on the neighbor counts for each query, if there are any. See data point 4 in Figure 13.8 for an example.

Finally, we allow arbitrary settings on both distance and fraction thresholds. In this more general case, we can observe that the outlier set identified by a "stricter" query Q_i is a subset of that identified by a more "relaxed" query Q_j, if $Q_i.\theta^{range} \leq Q_j.\theta^{range}$ and $Q_i.\theta^{fra} \leq Q_j.\theta^{fra}$. Thus we can group the member queries into several non-overlapping subgroups to ensure that such containment relationship transitively holds among the queries in each of the subgroups. Then we can use the same techniques discussed in the previous cases to maintain the outliers integrally for all queries in each subgroup.

13.4 Integrated Maintenance for Multiple kNN Queries

This pattern type takes only one pattern-specific parameter, namely the input k. Thus, a group of such queries with different input k settings are all querying the nearest neighbors of the same given object but are asking for

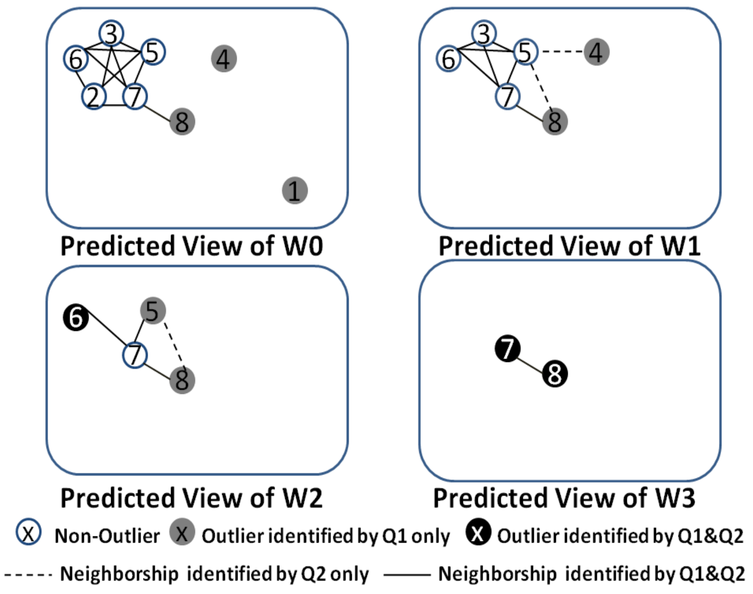
Predicted View of W0
Predicted View of W1
Predicted View of W2
Predicted View of W3
Non-Outlier
Outlier identified by Q1 only
Outlier identified by Q1&Q2
Neighborship identified by Q2 only
Neighborship identified by Q1&Q2

Figure 13.7: Distance-Based Outliers Identified by Q_1 and Q_2

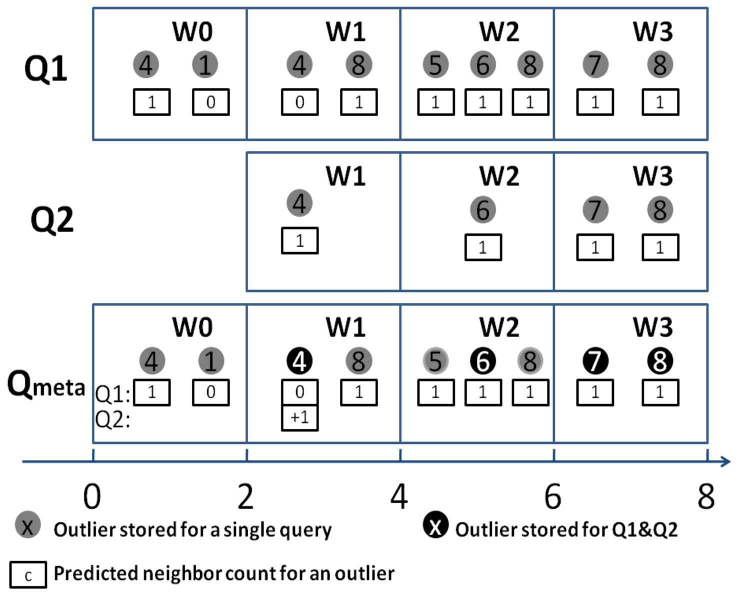
Q1
Q2
Qmeta
W0
W1
W2
W3
Q1:
Q2:
+1
0
2
4
6
8
Outlier stored for a single query
Outlier stored for Q1&Q2
Predicted neighbor count for an outlier

Figure 13.8: Independent vs. Integrated Representation for Distance-Based Outliers Identified by Q_1 and Q_2

different numbers of such nearest neighbors. In this case, using the incremental pattern maintenance mechanism is quite straightforward. Basically, in any predicted window, the k nearest neighbors (kNN) identified by a query Q_i with the largest k among the query group will cover all the kNN that should be identified by other queries. For any two queries Q_i and Q_j, if $Q_i.k \geq Q_j.k$, the kNN identified by Q_j is a subset of that identified by Q_i. So, we can again incrementally store and maintain the k nearest neighbors identified by different queries in an integrated manner.

In particular, in any predicted window, for a group of kNN queries, rather than maintaining one kNN set for each query, we only maintain a single "KNN set" of the query object, namely its K Nearest Neighbors, with K the largest k setting among all queries in the query group. This single KNN set will represent the kNN of all queries in the query group. This KNN set can be simply implemented as a sorted list based on their distance to the query object.

When a new data point p_{new} comes into the system, we only compare the distance between p_{new} and the query object with the distance between the query object and its Kth nearest neighbor in KNN. If the p_{new} is closer to the query object compared to its Kth nearest neighbor, it qualifies for the KNN set, indicating that it will be in kNN set for at least one query in the group. Otherwise p_{new} is discarded for this predicted window, as it has no chance to make kNN for any query in this predicted window.

If p_{new} qualified the KNN set, we use p_{new} to update the KNN set, We only need two operations to process the update. First, we put p_{new} into the KNN set and then remove the previous Kth nearest neighbor (the farthest one) from KNN. During the output, we simply scan the kNN set from the 1st to kth nearest neighbors. The nearest ones will be reported as kNN for all queries, while the farther ones will be reported to less and less queries depending on the k settings of the particular queries. Clearly, the cost of our integrated maintenance strategy for multiple kNN queries is almost equal to the cost of executing the single kNN query with the largest k setting.

Chapter 14

Sharing Among Queries with Arbitrary Window Parameters

In this chapter, we discuss our proposed memory and CPU sharing strategy among multiple neighbor-based pattern queries with different window parameters, namely variations in the window size win and the slide size $slide$. During this discussion, we assume that all these queries have the same pattern parameters. The techniques proposed in this section are general, and can be equally applied to all the neighbor-based pattern mining queries discussed in this work and other query types, such as graph mining. This is because the optimizations introduced here are at the window level, namely, regarding to the planning and organization of predicted windows but independent from the specific pattern maintenance within each window. To explain the proposed ideas in detail, we pick density-based clusters as the specific pattern type in our running examples.

14.1 Same win, Arbitrary $slide$ Case.

In this case, all member queries have the same window size win, while their slide sizes may vary. First, we assume that all queries start simultaneously. The equality of window sizes implies that all queries always query on the same portion of the data stream. More specifically, at any given time the data points falling into the windows of different queries are the same. Then, the only difference among multiple queries is that they may need to generate output at different moments, due to their varying slide sizes. For example, given three queries Q_1, Q_2 and Q_3, with $Q_1.win = Q_2.win = Q_3.win = 10(s)$, $Q_1.slide = 2(s)$, $Q_2.slide = 3(s)$ and $Q_3.slide = 6(s)$, the query windows of them cover exactly the same portion of the data stream at any given time, while they are required to output the clusters at every 2, 3 and 6 seconds respectively. So, to serve the different output time points, they need to build predicted windows starting at different times, each serving a future output time point. In this example, assuming all three queries start at wall clock time 00:00:00, they all need to build a predicted window starting at 00:00:00 for generating the output at 00:00:10, which is their first and shared output time point. Then Q_1 needs to build predicted windows starting at 00:00:02, 00:00:04, etc, to serve the output time points at 00:00:12, 00:00:14, while Q_2 and Q_3 need to build predicted windows starting at 00:00:03, 00:00:06, etc, and 00:00:06, 00:00:012, etc, respectively.

To solve this problem, for a given group QG, we build a single meta query Q_{meta} which integrates all the member queries of QG. In particular, this meta query Q_{meta} has the same window size with all member queries in QG, while its slide size is no longer fixed but adaptive during the execution. More specifically, the slide size of Q_{meta} at a particular moment is decided by the nearest moment which at least one member query of QG needs to be answered. The specific formula to determine the next output moment is:

$$T_{nextoutput} = Min(\lceil \frac{T - win}{Q_i.slide} \rceil + 1) * Q_i.slide + win)$$

With T the current wall-clock time and win the common window size among all queries. Using the earlier example, for the query group having three member queries, we build a meta query Q_{meta} for it with $win = 10s$. So, at wall-clock time 00:00:10, the slide size of Q_{meta} should be 2s, as 00:00:12 will be the nearest time at which a member query (Q_1) needs to be answered. Then its slide size is adapted to 1s, 1s and 2s at 00:00:02, 00:00:03 and 00:00:04 respectively for the same reason.

Such adaptive slide size strategy is compatible with the "view prediction" technique. This is because, although the slide size of Q_{meta} may keep changing, these changes are still predictable and periodic. In particular, given the slide size of all the member queries, we always know at which moments which member queries need to be answered. The interval between any two successive output moments is actually changing periodically. So, we can construct an output schedule (with a lookahead a of finite number of output time points) for Q_{meta}, which predetermines the slide size of Q_{meta} at any given moment.

Knowing the slide sizes of Q_{meta}, we can just build predicted windows for Q_{meta} based on the output time points. Still using the earlier example, at wall-clock time 00:00:10, we would have built eight "predicted windows" for Q_{meta} , which start from 00:00:00, 00:00:02, 00:00:03, 00:00:04, 00:00:06, 00:00:08, 00:00:09 and 00:00:10 respectively, as each of them corresponds to an output time point for at least one member query. Among these eight "predicted windows", many of them are actually serving multiple queries. For example, the "predicted windows" starting at 00:00:00 and 00:00:06 will be used to answer Q_1, Q_2 and Q_3 as they correspond to the output time points that are shared by all three queries. This also means that if we were to maintain the predicted windows for these queries independently, four more predicted windows would need to be maintained at this given moment. In particular, Q_2 and Q_3 would have needed to maintain their own predicted windows starting at 00:00:00 and 00:00:06 separately, although they are exactly the same as those maintained by Q_1. In this example, 33 percent of the "predicted windows" are saved from the independent maintenance mechanism. This means that 33 percent of storage space and computational resources are saved in this case.

In conclusion, by building a meta query representing all member queries in a query group, we can save both the memory space and CPU processing time for answering the query group for the following reasons: 1) No overhead, in particular, no extra predicted views will be introduced, as a predicted window is built only if at least one member query needs output at that moment. In other words, all the predicted windows built in our integrated solution need to be maintained by individual member queries anyways. 2) Many predicted views can be shared as several member queries may require output at the same time. The specific amount of sharing depends on the percentage of overlaps of member queries' output time points.

14.2 Same $slide$, Arbitrary win Case

In this case, although the window size may vary among the member queries, we hold the slide size steady, indicating that their output schedules are identical. Here we first work with a common assumption that all the window sizes of the member queries are multiples of their common slide size. We observed that, given a query group with member queries having the same slide size but different window sizes, all the member queries require output at exactly the same moments. Based on this observation an important characteristic can be discovered for such query groups.

Lemma 14.1 *Given a query group QG with member queries having the same slide size $slide$ but arbitrary window sizes (multiples of $slide$), the "predicted windows" maintained for Q_i, with $Q_i.win$ larger or equal to any other $Q_j.win$ in QG, will be sufficient to answer all member queries in QG.*

Proof 14.1 *This is because the "predicted windows" maintained for Q_i will cover all the "predicted windows" that need to be maintained for all the other queries. More specifically, at any given moment, say wall-clock time T, the "predicted windows" that need to be maintained for a member query Q_n include all those starting at $T - n * slide$ ($1 \leq n \leq \frac{Q_n.win}{slide}$). As $Q_i.win$ is larger or equal than any $Q_j.win$, the "predicted windows" maintained for Q_i cover all those needed by* *other queries. At time T, any member query Q_j can be answered by the "predicted window" starting from $T - Q_j.win$.*

For example, given three queries Q_1, Q_2 and Q_3, with $Q_1.slide = Q_2.slide = Q_3.slide = 5s$, $Q_1.win = 10$, $Q_2.slide = 15s$ and $Q_3.slide = 20s$, at wall clock time 00:00:20, the "predicted windows" built by Q_3 start from 00:00:00, 00:00:05, 00:00:10 and 00:00:15 respectively, while those need to be maintained by Q_1 and Q_2 start from 00:00:10, 00:00:15 and 00:00:05, 00:00:10, 00:00:15 respectively. The later all overlap with those built by Q_3. At this moment, the "predicted window" starting from 00:00:00 can be used to answer Q_3, while the predicted windows starting from 00:00:10 and 00:00:05 can be used to answer Q_1 and Q_2 respectively.

In summary, we only need to maintain the predicted windows for a single member query, namely the query with the largest window size, and then can answer all the member queries in the query group with different predicted windows it maintains. Clearly, full sharing is achieved. Here, we also note that although we made the common assumption in Lemma 14.1 that the window sizes are multiples of $slide$, to make the problem easier to understand, it is not crucial for our solution. Our solution can easily be relaxed to handle the cases where window sizes of member queries are completely arbitrary.

14.3 Arbitrary $slide$, Arbitrary win Case

We now give the solution for the more general case that both window parameters, namely win and $slide$, are arbitrary. Generally, the solution for this case is a straightforward combination of the two techniques introduced in the last two subsections. In particular, we simply build one single meta query that has the largest window size among all the member queries and uses an adaptive slide size. These two techniques are fully compatible, because they were both designed to make sure correct predicted windows (start and end as required by query semantics) are created to answer the member queries.

Here we use an example to demonstrate our solution. Given three queries Q_1, Q_2 and Q_3, with $Q_1.win = 10$, $Q_1.slide = 4$, $Q_2.win = 9$, $Q_2.slide = 5$, $Q_3.win = 6$ and $Q_3.slide = 2$, and all starting at wall clock time 00:00:00, we build a meta query Q_{meta} with $Q_{meta}.win = max(Q_i.win)_{(1 \leq i \leq 3)} = 10$. Then we adaptively change its slide size based on the next nearest output time point required by (at least) one of these three queries. For instance, at wall clock time 00:00:10, six predicted windows would have been built, which start from 00:00:00 (serving Q_3 for output at 00:00:10), 00:00:01 (serving Q_2 for output at 00:00:10), 00:00:04 (serving Q_1 for output at 00:00:12 and Q_3 for output at 00:00:10), 00:00:06 (serving Q_2 for output at 00:00:13 and Q_3 for output at 00:00:12), 00:00:08 (serving Q_1 for output at 00:00:18 and Q_3 for output at 00:00:14) respectively. Figure 14.1 shows the predicted views that need to be maintained by each of these three queries independently, versus those would instead be maintained by the meta query at wall clock time 00:00:10.

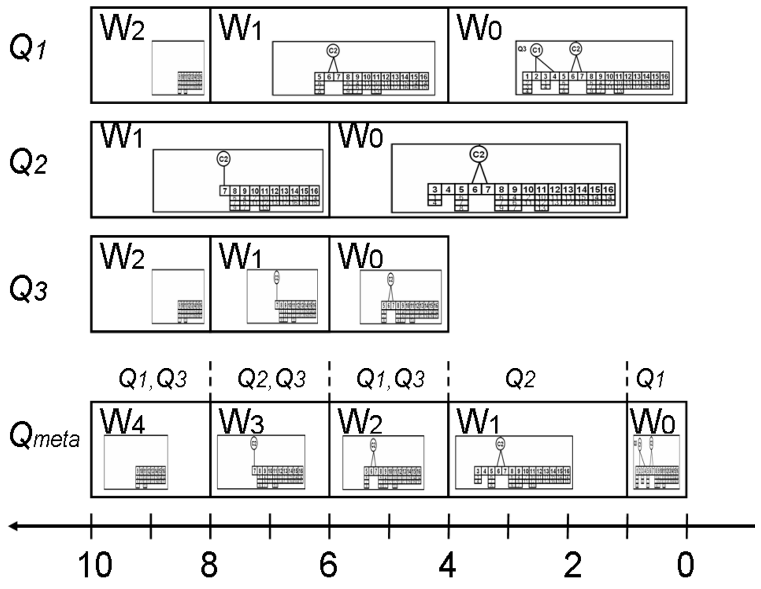

Figure 14.1: Predicted Views Maintained By Three Queries Q1, Q2 and Q3 Independently versus Those Maintained By a Single Meta Query

Chapter 15

Putting IT All Together: The General Case

Finally, we now discuss the case that the pattern and window parameters are both arbitrary for the queries in a query group. Although sharing among a group of totally arbitrary queries is a hard problem if we had to solve it from scratch, we now can easily handle it by combining the two techniques introduced in last two sections, namely the incremental pattern representation technique and the meta query technique. These two techniques are orthogonal to each other, and can thus be easily combined. In particular, the integrated pattern representation technique (introduced in Chapter 13) is designed to share among a group of queries that are specified on the same dataset, which in our case is each predicted window. So, we can consider this here as an "**intra-predicted-windows**" sharing technique. On the other hand, the meta query technique (introduced in Chapter

14) is designed to make sure that the predicted windows, which need to be maintained by different queries, start and end properly and share across the different predicted windows. So, it is an "**inter-predicted-windows**" sharing technique. Thus, these two orthogonal techniques can be easily applied together to realize the full potential of sharing of the member queries on both the inner- and inter-predicted window level.

Here we use an example to demonstrate such a combination. Given three queries Q_1, Q_2 and Q_3 starting at 00:00:00, with $Q_1(win = 10, slide = 4, \theta^{range} = 0.2, \theta^{cnt} = 5)$; $Q_2(win = 9, slide = 5, \theta^{range} = 0.3, \theta^{cnt} = 4)$ and $Q_3(win = 6, slide = 2, \theta^{range} = 0.2, \theta^{cnt} = 3)$, we first use the meta query technique to build the predicted windows they need to maintain. At wall clock time 00:00:10, the required predicted windows are the same as those shown in Figure 14.1. Then, for each predicted window built, we apply the $IntView\_\theta$ technique to build an "Predicted View Tree" to integrate the predicted views (of different queries) in this window. For the predicted window starting from 00:00:04, which is serving Q_1 and Q_3, we build a "Predicted View Tree" representing both Q_1 and Q_3. Now the "Predicted View Tree" structures built for different windows may no longer be all the same as those in the example we demonstrated in Figure 15.1. This is because the predicted view of a particular query will appear on a "Predicted View Trees" only if this predicted window needs to be maintained by this query, indicating this predicted window corresponds to an output time point for it. Using the same example, Q_2 has no predicted view in W_4, as W_4 is not a predicted window that needs to be maintained by it.

We call this ultimate hierarchical structure $IntView$. Figure 15.1 depicts the final $IntView$ built for the three queries mentioned in the earlier example.

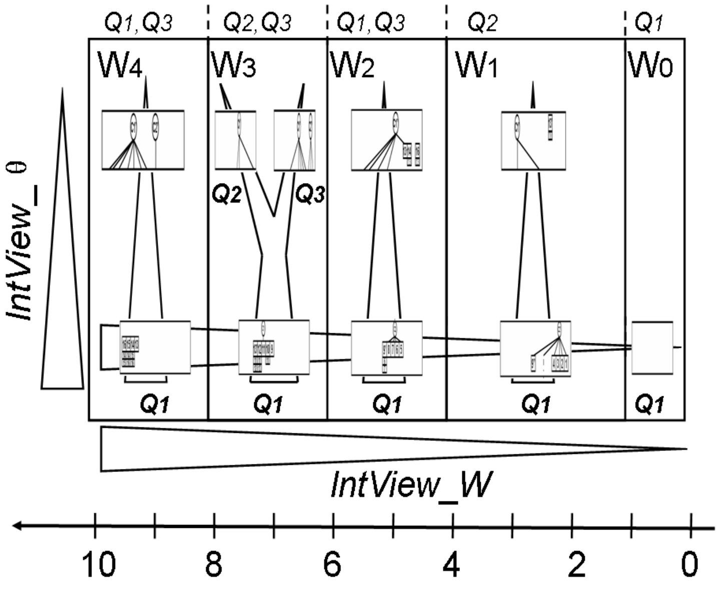

Figure 15.1: $IntView$: Integrated Representation for Predicted Views Identified by 3 Queries in 5 Predicted Windows

In particular, $IntView$ is a tree structure that starts from the predicted view acting as the root (r_{newest}) of the "Predicted View Tree" in the newest predicted window (with the largest window number). Thus, each root predicted view in an older predicted window is now incrementally built based on that in the next window. This indicates that, as subtrees for $IntView$, each "Predicted View Trees" in an older window is now built based on the incremental information from the next (the newer) window (as its root itself now is incremental). We call the final solutions for density-based clustering, distance-based outlier detection and kNN queries *Chandi*, *SDOD* and *SkNN* respectively. We give the pseudo-code of them in Figures 15.2, 15.3, 15.4 and 15.5.

p_i: a data point. p_{new}:a new data point. $p_i.T$:p_i's time stamp. clu_mem:cluster membership. W_i : predicted window. $W_{oldest}/_{newest}$: oldest/newest W on $IntView$. $W.T_{end}$: ending time of W. $W_i.root$: the root predicted view of in W_i. $IntView$: the overall IntView structure. PV : a predicted view.
Q_i : a member query. $Q_i.PV$: a predicted view built for Q_i.

```
Chandi ($QG$)
1 For each new data point $p_{new}$
2   If $p_{new}.T > W_{oldest}.T_{end}$
3     Purge($W_{oldest}$); //purge the oldest predicted window // purge
4   load $p_{new}$ into index // load
5   $neighbors$:=RangeQuerySearch($p_{new}, max(Q_i.\theta^{range})$)
6   UpdateIntView ($p_{new}, neighbors$) // IntView Maintenance
7   If $p_{new}.T == T_{output}$
8     Output(); // output
9     add new window $W_{newest}$ to $IntView$

Purge($W_i$)
1   purge any $p_i$ from index If $p_i.T < W_i.T_{end}$
2   remove $W_i$ from $IntView$

UpdateIntView ($p, neighbors$)
1 For i:=1 to $neighbors.size()$
2     DistributeNeighbor($p, neighbors[i], W_{newest}.root$);
3 UpdatePredictedView($p, W_{newest}.root$);
```

Figure 15.2: *Chandi*: Proposed Algorithm for Multiple Density-Based Clustering Queries (Part 1)

As shown in our pseudo-code in Figures 15.2, 15.3 and 15.4, when a new data point arrives at the system, the *Chandi* and *SDOD* algorithms, which handles density-based clusters and distance-based outliers respectively, first run a range query search using the largest θ^{range} among the

```
**DistributeNeighbor**($p_{new}, p_i, PV$)
1 If $dist(p_{new}, p_j) \leq Qi.\theta^{range}$
2     add $p_i$ to PV.neighbors (neighbors distributed to PV)
3 Else For each $Q_j.PV$ at higher level
4     DistributeNeighbor ($p$,neighbor, $Q_j.PV$);

**UpdatePredictView** ($p, PV$)
1 $p.neighborcount = PV.neighborsinthisview.size()$;
2 For i:=1 to $PV.neighbor.size()$
3     $PV.neighbors[i].neighborcount++$;
4     If $PV.neighbors[i]$ becomes a new core
5         HandleNewCore($PV.neighbors[i]$);
6 If $p.neighborcount \geq Q_i.\theta^{cnt}$
7     HandleNewCore($p, PV$);
8 For each $Q_j.PV$ at higher level
9     UpdatePredictView ($p, Q_j.PV$);

**HandleNewCore**($p, PV$)
1 $p.type = core$;
2 $p.clu\_mem$=new $clu\_mem$ (generate a new cluster membership);
3 For i:=1 to $PV.neighbors.size()$
4     If $PV.neighbors.type == core$
5         Merge $PV.neighbors[i].clu\_mem$ and $p.clu\_mem$;
6     If $PV.neighbors[i].type == noise$
7         $PV.neighbors[i].type := edge$;
8         $PV.neighbors[i].clu\_mem := p.clu\_mem$;
9 For each $Q_j.PV$ at higher level
10     PropagateNewCore($p, Q_j.PV$);
```

Figure 15.3: *Chandi*: Proposed Algorithm for Multiple Density-Based Clustering Queries (Part 2)

p_i: a data point. p_{new}:a new data point. $p_i.T$:p_i's time stamp.
W_i : predicted window. $W_{oldest}/_{newest}$: oldest/newest W on $IntView$.
$W.T_{end}$: ending time of W. $PV.p\_outlier$: the potential outliers in PV.
$IntView^{outlier}$: the overall IntView structure. $W_i.PV$: predicted view built for W_i.
Q_i : a member query. $Q_i.PV$: a predicted view built for Q_i.

```
SDOD ($QG$)
1 For each new data point $p_{new}$
2   If $p_{new}.T > W_{oldest}.T_{end}$
3     Purge($W_{oldest}$); //purge the oldest predicted window // purge
4   load $p_{new}$ into index // load
5   $neighbors$:=RangeQuerySearch($p_{new}, max(Q_i.\theta^{range})$)
6   UpdateIntView ($p_{new}, neighbors$) // IntView Maintenance
7   If $p_{new}.T == T_{output}$
8     Output(); // output
9     add new window $W_{newest}$ to $IntView$

Purge($W_i$)
1   purge any $p_i$ from index If $p_i.T < W_i.T_{end}$
2   remove $W_i$ from $IntView$

UpdateIntView ($p, neighbors$)
1 For each $W_i$ on $IntView^{outlier}$
2   UpdatePredictedView($p, W_i.PV$,$neighbors$);

UpdatePredictView ($p, W_i.PV, neighbors$)
1 For each $Q_i$ that maintains $W_i$ (in ascending order of "strictness")
2    $p.neighborcount := 0$
3    For i:=1 to $neighbor.size()$
4       If $Distance(p, neighbor[i]) < Q_i.\theta^{range}$
5          $p.neighborcount++$;
6          $neighbors[i].neighborcount++$;
7       If $neighbors[i] \in W_i.PV.p\_outliers$ and
            $neighbors[i].neighborcount \geq Q_i.\theta^{fra} \times win$
8          mark $neighbors[i]$ as safe non-outlier for $Q_i$;
9       If $neighbors[i]$ marked as safe non-outlier for all queries;
10         remove $neighbors[i]$ from $W_i.PV.p\_outliers$;
11    If $p \notin W_i.PV.p\_outliers$ AND $p.neighborcount < Q_i.\theta^{fra} \times win$;
12       put $p$ in $W_i.PV.p\_outliers$;
```

Figure 15.4: *SDOD*: Proposed Algorithm for Processing Multiple Distance-Based Outlier Detection Queries

p_i: a data point. p_{new}:a new data point. $p_i.T$:p_i's time stamp.
W_i : predicted window. $W.T_{end}$: ending time of W.
$W_{oldest}/_{newest}$: oldest/newest W on $IntView$.
p^Q: the query object. $PV.KNN$: the KNN (see Chapter 13.4) in PV.
$PV.p^Q.K^{th}\_NN$: the K^{th} nearest neighbor of p^Q in PV.
$IntView^{kNN}$: the overall IntView structure for kNN queries.
$W_i.PV$: predicted view built for W_i. Q_i : a member query.
$Q_i.PV$: a predicted view built for Q_i.
$Q_i.k$: the k parameter of Q_i .
$Q_i.T_{output}$: the next output time for Q_i.

```
SDOD ($QG$)
1 For each new data point $p_{new}$
2   If $p_{new}.T > W_{oldest}.T_{end}$
3     Purge($W_{oldest}$); //purge the oldest predicted window // purge
4   load $p_{new}$ into index // load
5   UpdateIntView ($p_{new}$) // IntView Maintenance
6   If $p_{new}.T == T_{output}$
7     Output($QG$, $T_{output}$); // output
8     add new window $W_{newest}$ to $IntView$

Purge($W_i$)
1   purge any $p_i$ from index If $p_i.T < W_i.T_{end}$
2   remove $W_i$ from $IntView$

UpdateIntView ($p$)
1 For each $W_i$ on $IntView^{kNN}$
2   UpdatePredictedView($p, W_i.PV$);

UpdatePredictView ($p, W_i.PV$)
1 If $W_i.PV.KNN.size() < K$
2   insert $p$ into $W_i.PV.KNN$;
3 Else
4   If $Dist(p^Q, p) < Dist(p^Q, W_i.PV.K^{th}\_NN)$
5     insert $p$ into $W_i.PV.KNN$;
6     remove $W_i.PV.p^Q.K^{th}\_NN$ from $W_i.PV.KNN$;

Output($QG$, $T_{output}$)
1 For each $Q_i \in QG$  2  If $Q_i.T_{output} == T_{output}$  3    output the first
$Q_i.k$ objects on $W_i.PV$;  4    $Q_i.T_{output} += Q_i.slide$;
```

Figure 15.5: *SkNN*: : Proposed Algorithm for Processing Multiple kNN Queries

query group to collect all its potential neighbors. Then it distributes each of them to the first predicted view on each path of $IntView$, in which their "neighborship" truly exists. Then, it starts the $IntView$ maintenance process from the root of $IntView$, namely the root predicted view of the newest predicted window in $IntView$, and then incrementally maintains those at higher levels of $IntView$. During the maintenance of each predicted view, they only needs to communicate with the neighbors assigned to that particular view. The computation needed by *SkNN* is simpler. For each predicted window, it first decides whether the new object is qualified for the KNN set in that window. If the answer is "yes", then it updates the KNN in that window using the new object, otherwise the new object will not affect the KNN in that window.

Computation-wise, all three proposed algorithms, namely *Chandi*, *SDOD* and *SkNN* only require a single pass through the new data points at each window slide. In particular, *Chandi* and *SDOD* only require one range query search for each new object, and each new object only communicates with its neighbors once for all shared queries. *SkNN* only requires each new object to update the KNN in each window at most once.

Memory-wise, for all three proposed algorithm, as they all maintain the pattern sets identified by the multiple queries in a single $IntView$ structure, the upper bound of the memory consumption of them for a group of shared queries on the same path is independent from the "length" of this path, namely the number of shared queries in this group. This can be proven using the same method as we used for proving Lemma 13.7. In conclusion, our proposed algorithms, namely *Chandi*, *SDOD* and *SkNN* achieve full sharing for multiple density-based clustering, distance-based outlier queries and kNN queries respectively over the same input stream in terms of both CPU and memory resources.

Chapter 16

Experimental Study

16.1 Experimental Platform

All our experiments in this part of this dissertation are conducted on a HP Pavilion dv4000 laptop with Intel Centrino 1.6GHz processor and 1GB memory, which runs Windows XP operating system. We implemented all algorithms with VC++ 7.0.

16.2 Real and Synthetic Streaming Datasets

labelp2-exp-datasets We use the same real streaming datasets, GMTI [EFK99] and STT [INE], as we used for experiments in Part I of my dissertation (see Chapter 9.1.1).

16.3 Alternative Algorithms

As we discussed earlier in Chapter 4, density-based cluster has one of the most complex pattern structure within the neighbor-based pattern family. Also, given the same window size and input rate, each individual density-based clustering query is much more system-resource consuming compared with an distance-based outlier or kNN query. Therefore our experimental evaluation concentrates on thoroughly evaluate the performance of our proposed algorithm *Chandi* which handles multiple density-based clustering queries. This includes evaluations on its performance under a broad range of parameter settings, how it compares with method using a straightforward sharing method, and test to its scalability on the number of queries that can be handled. Beyond that, a theoretical analysis of the performance of the other two pattern types will be given later in Chapter 16.10.

To evaluate our proposed *Chandi* algorithm, for any input *QG*, we compare *Chandi*'s performance of two major alternative methods, executing *QG* with four alternative methods, namely executing one *Extra-N* algorithm [YRW09] for each member query with and without sharing of range query searches (henceforth referred as *Extra-N with rqs* and *Extra-N*), and executing one *IncDBSCAN* algorithm [EKS$^+$98] for each member query with and without sharing of range query searches (referred as *IncDBSCAN with rqs* and *IncDBSCAN*). The reasons why we choose them are: 1) *Extra-N* algorithm is the only algorithm we are aware of in the literature solving density-based clustering over sliding windows; 2) *IncDBSCAN* algorithm is the most well known method for incremental density-based clustering (but not designed for sliding window semantics).

16.4 Experimental Methodologies

We measure two common metrics for stream processing algorithms, namely average processing time for each tuple (CPU time) and memory footprint, indicating the peak memory space required by an algorithm.

As we know, each density-based clustering query using sliding window semantics has four input parameters, namely two pattern parameters: θ^{cnt}, θ^{range}, and two window parameters: win and $slide$. In many cases, the domain knowledge or specific requirements of the analysis tasks may restrict some of them to particular values. For example, a moving object monitoring task may require the θ^{range} to be the maximum distance that two objects can keep wireless communication, and the window size to be the time interval between two successive reports of a single object. Thus the queries submitted by different analysts may only differ on a subsets of these parameters. In our experiments, we first evaluate the four test cases, each has only one of the four parameters different among the member queries.

16.5 Evaluation for One-Arbitrary-Parameter Cases

For each test case, we prepare a query group QG with $|QG| = 20$ by randomly generating one input parameter (in a certain range) for each member query, while using common parameter settings on the other three parameters. The parameter settings in our experiment are learned from a pre-analysis of the datasets. In particular, we pick parameter ranges that allow member queries to identify all the different major cluster structures that could be identified in the datasets. In all our test cases, the largest number of clusters identified by a member query is at least five times the smallest number of clusters identified by the other, indicating that the cluster structures identified by different queries vary significantly. In each test case, we use different subsets of QG (sized from 5 to 20) to execute against GMTI data.

16.5.1 Arbitrary θ^{cnt} case

We use $\theta^{range} = 0.01$, $win = 5000$ and $slide = 1000$, while varying θ^{cnt} from 2 to 20. In this test case, at most 16 clusters are identified by the most restricted query with $\theta^{cnt} = 20$, while at least 3 clusters are identified by the most relaxed one with with $\theta^{cnt} = 3$. As shown in Figures 16.1 and 16.2, both the average processing time and the memory space used by all five alternatives increase as the number of member queries increases. This is because more meta-information needs to be computed and stored by all of them. However, the utilization of CPU resources by *Chandi* is significantly lower than those consumed by other alternatives, especially when the number of the member queries increases, and its memory consumption is almost equal to *IncDBSCAN* and much lower than *Extra-N*. This matches our analysis in Chapter 13, because in this test case, the predicted windows need to be maintained by *Chandi* for different queries completely overlap. Also, since no "extra neighborships" exists in any window, the cluster growth information that needs to be maintained by *Chandi* among the queries is relatively simple. Thus, the system resource consumption of *Chandi* increases very modestly when the number of member queries increases. While since other alternative methods maintain the progressive clusters independently for different queries, their consumption of system resources increases dramatically when the number of member queries increases.

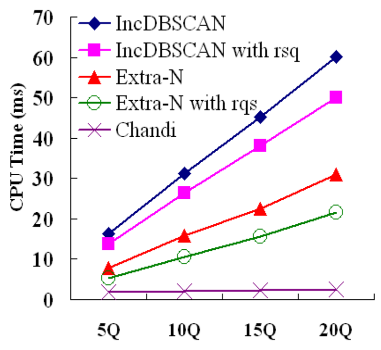

Figure 16.1: CPU Time used by Five Competitors in Arbitrary θ^{cnt} Cases

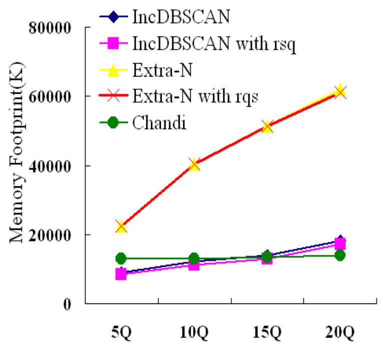

Figure 16.2: Memory Space used by Five Competitors in Arbitrary θ^{cnt} Cases

16.5.2 Arbitrary θ^{range} case

In this case, we use $\theta^{cnt} = 10$, $win = 5000$ and $slide = 1000$, while varying θ^{range} from 0.01 to 0.1. In this test case, at most 10 clusters are identified by the most restricted query with $\theta^{range} = 0.1$, while at least 2 clusters are identified by the most relaxed one with $\theta^{range} = 0.1$. As shown in the Figures 16.3 and 16.4, similar situations can be observed that *Chandi* uses significantly less CPU and memory resources than other alternatives. In this test case, the system resource consumption of *Chandi* increases more as the number of queries increases compared with the previous test cases. This is for of two main reasons. 1) Since the θ^{range} parameters vary among the queries, the range query search cost increases along with the increase of the number of queries even with the range query sharing (each data point needs to figure out its neighbors defined by different queries). 2) As the neighborships identified by different queries differ, such "extra-neighborships" are more likely to cause cluster structure changes and thus requires *Chandi* to maintain more meta-information in $IntView$. The performance of other competitors, especially for *IncDBSCAN*, is affected by the increasing cost of range query searches as well. This is because the performance of *IncDBSCAN* (with rqs or not), which consumes large numbers of range query searches during the purging process, largely relies on the cost of range query searches.

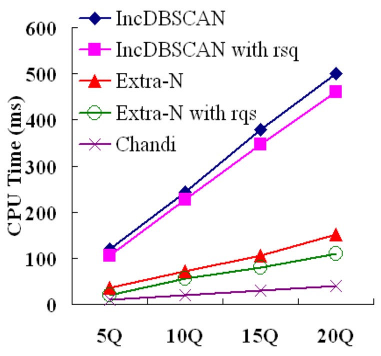

Figure 16.3: CPU Time used by Five Competitors in Arbitrary θ^{range} Cases

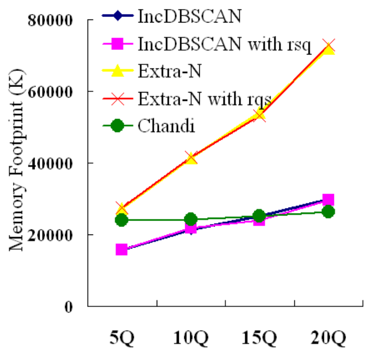

Figure 16.4: Memory Space used by Five Competitors in Arbitrary θ^{range} Cases

16.5.3 Arbitrary win case

In this case, we use $\theta^{cnt} = 10$, $\theta^{range} = 0.01$, $slide = 500$, while varying win from 1000 to 5000 (we use 500 as granularity for any window parameter). As shown in Figures 16.11 and 16.12, we can observe that the performance of *Chandi* is even better compared with the previous test cases. In particular, its resource utilizations for both CPU and memory are almost unchanged as the number of queries increases. This is expected, because in this case *Chandi* only maintains the meta-information for a single query, which is sufficient to answer all the member queries. Thus, the cost of *Chandi* in this case only depends on the query with the largest win, which is independent of the number of queries in the query group.

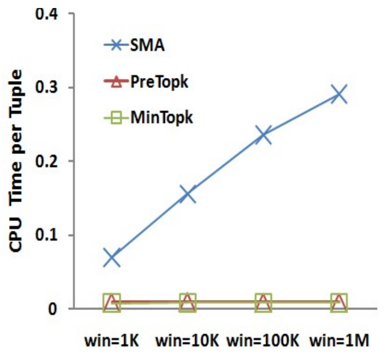

Figure 16.5: CPU Time used by Five Competitors in Arbitrary win Cases

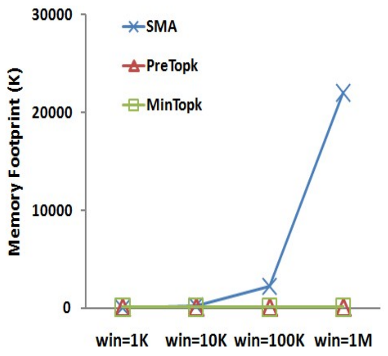

Figure 16.6: Memory Space used by Five Competitors in Arbitrary win Cases

16.5.4 Arbitrary $slide$ case

In this case, we use $\theta^{cnt} = 10$, $\theta^{range} = 0.01$, $window = 5000$, while varying $slide$ from 500 to 5000. As shown in Figures 16.7 and 16.8, the performance of *Chandi* is similar with that in the arbitrary win case. This is because the cost of *Chandi* in this case depends on the number of predicted windows that need to be maintained, which is decided by the query with smallest slide size but does not necessarily increase with the number of queries in the query group.

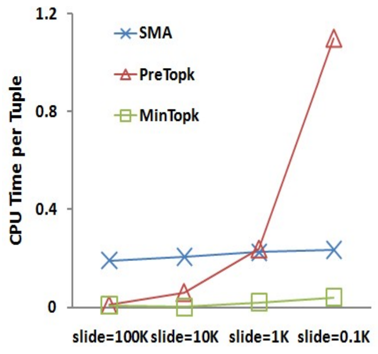

Figure 16.7: CPU Time Used by Five Competitors in Arbitrary $slide$ Cases

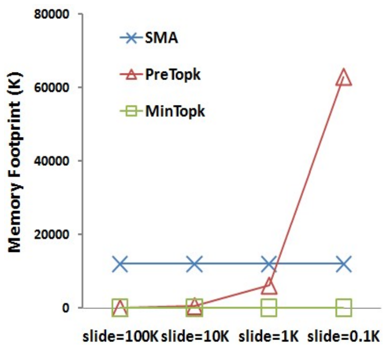

Figure 16.8: Memory Space used by Five Competitors in Arbitrary $slide$ Cases

16.6 Evaluation for Two-Arbitrary-Parameter Cases

labelp2-exp-two-arbitrary We evaluate two test cases, each has two of the four parameters different among the member queries. In the first test case, member queries have arbitrary pattern parameters but common window parameters, indicating that they may have different definition to the clusters but always have the same query window. In the second test case, member queries have arbitrary window parameters but common pattern parameters, indicating they may have different query windows but have the same definition to the clusters.

16.6.1 Arbitrary Pattern Parameters

In this case, we use $win = 5000$, $slide = 1000$, while vary θ^{cnt} from 2 to 20 and θ^{range} from 0.01 to 0.1. As shown in Figures 16.9 and 16.10, *Chandi* still consumes significantly less CPU time compared with the other alternatives, although the increase of CPU consumption caused by the increase of member queries is more obvious. This is because totally arbitrary pattern parameters lead to an even larger difference in the clusters identified by different queries, and thus increase the maintenance costs of *Chandi*. In particular, in this test case, the largest number of clusters identified by the number query with $\theta^{range} = 0.01$ and $\theta^{cnt} = 14$ reaches 35, while the smallest number of clusters identified by the query (with $\theta^{range} = 0.1$ and $\theta^{cnt} = 3$) is only 2. The memory space used by *Chandi* in this case is much less than *Extra-N* while being only slightly higher than *IncDBSCAN*. Again, this is caused by the more incremental information existing among the predicted views maintained by *Chandi*. However, as the CPU performance of *IncDBSCAN* is much worse than *Chandi*, the overall performance of *Chandi* is still much better.

16.6.2 Arbitrary Window Parameters

In this case, we use $\theta^{cnt} = 10$ and $\theta^{range} = 0.01$, while varying win from 1000 to 5000 and $slide$ from 500 to 5000 (for any query Q_i, $Q_i.slide < Q_i.win$). As shown in Figures 16.11 and 16.12, the performance of *Chandi* is similar with that observed in the arbitrary win or $slide$ case. This is because, although the queries now have arbitrary settings on both parameters, such

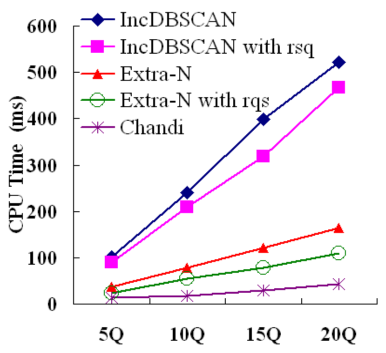

Figure 16.9: CPU Time used by Five Competitors in Arbitrary Pattern Parameter Cases

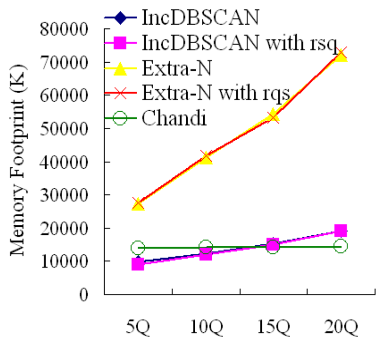

Figure 16.10: Memory Space used by Five Competitors in Arbitrary Pattern Parameter Cases

fact does not affect the principle of how the "meta query" strategy works. In particular, the cost of answering a query group still only depends on the largest *win* in the query group and the number of predicted views that need to be maintained. Both do not necessarily increase along with the number of queries.

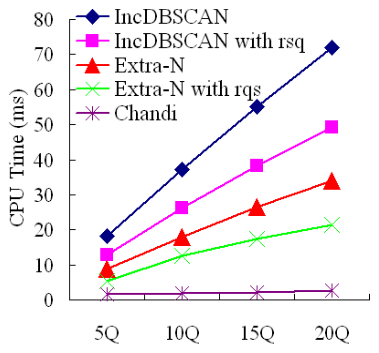

Figure 16.11: CPU Time used by Five Competitors in Arbitrary Window Parameter Cases

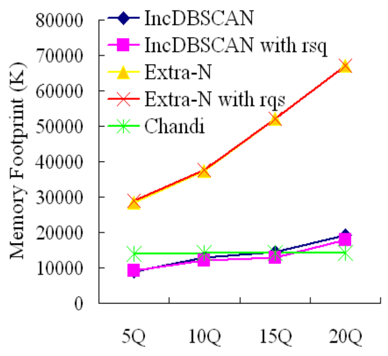

Figure 16.12: Memory Space used by Five Competitors Arbitrary Window Parameter Cases

16.7 General Case: Four Arbitrary Parameters.

labelp2-exp-general-case Finally, we evaluate the general case, with all four parameters being arbitrary. We divide this experiment into three cases, each measuring the performance of the algorithms when executing different numbers of queries. In particular, for each test case, we generate 30 query groups each with N member queries (N equals to 20, 40 and 60 for three cases respectively). Each query group is independently generated, and the member queries in each group are randomly generated with parameter settings: $\theta^{cnt} =$ 2 to 20, $\theta^{range} =$ 0.01 to 0.1, $win =$ 1000 to 5000, and $slide =$ 500 to 5000. For each test case, we measure the average cost of each algorithm for executing all 30 query groups. Beyond that, we zoom into the overall average cost of each algorithm, and measure the cost caused by each specific subtask. In particular, the CPU measurement is divided into two parts, namely the CPU time used by range query searches and that used by cluster maintenance. For the memory space consumed, we distinguish between the memory used by raw data (for storing actual tuples) and the memory used for meta-data.

As shown in charts C1, C2 and C3 in Figure 16.13, we observe that the average CPU time used by *Chandi* is 70, 76, and 85 percent lower then the best alternative method, *Extra-N with rqs*, in the three cases respectively. In particular, the CPU time used by *Chandi* to conduct range query searches is always less than 10% compared with that needed by *IncDBSCAN* with rqs. This is because *Chandi* only requires each new data point to run one range query search when it arrives at the system, while *IncDBSCAN* relies

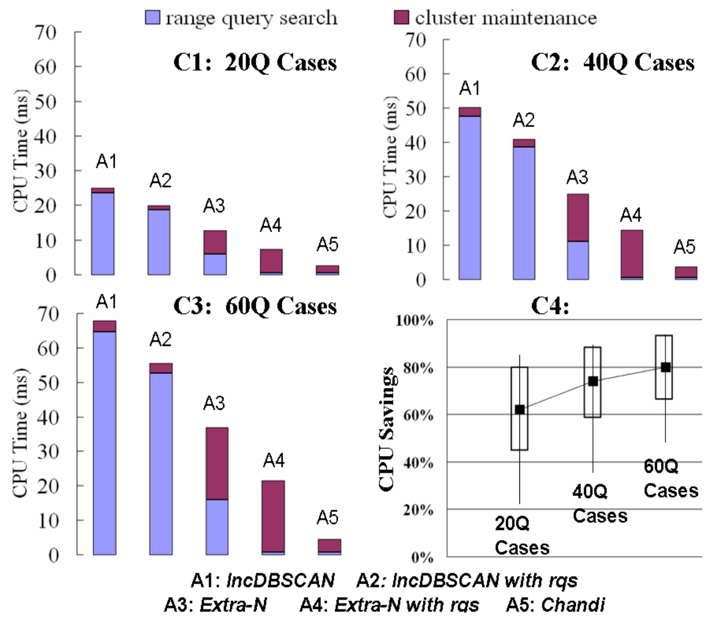

Figure 16.13: Detailed Comparison on CPU Time Consumption of Five Algorithms

on repeated range query searches to determine the cluster changes. The CPU time used by *Chandi* to maintain meta-information is at least 62% less than that used by *Extra-N with rqs*. This is because *Chandi* updates the meta-information for different queries integrally, while *Extra-N* maintains them independently.

Besides the comparison of the average system resource consumption, we also measure the savings of *Chandi* for each individual query group in all three test cases. In particular, for each query group, we measure the difference in resource utilization between *Extra-N with rsq* and *Chandi*, which corresponds to the difference between executing them using the best existing technique and our proposed strategy. More specifically, for each group, we first calculate the difference on CPU (or memory) utilization between two *Chandi* and *Extra-N*. Then, we use the difference to divide that used by *Extra-N with rqs* to get the saving percentage achieved by *Chandi*. As shown in C4 of Figures 16.13, *Chandi* never performs worse than *Extra-N with rqs* for any query group. For the first test case (each query group has 20 queries), the average savings achieved by *Chandi* in terms of CPU time are 62%. Although the minimum savings in this case among the 30 groups is 23%, the maximum savings reach 84% , and the standard deviation is only 19% . As the number of queries in each group increases, the savings achieved by *Chandi* are even higher in the other two test cases. In particular, the average savings achieved by *Chandi* of CPU time increases to 80% when the number of queries in each group increases to 60. The minimum and maximum savings on CPU time increases to 45% and 92% respectively in this case, and the standard deviation of the savings decreases to 12%. This shows the promise of *Chandi* that, for a query group with 60 queries, it can achieve savings between 73% to 92% of CPU time in most of the cases. Among the 30 queries in this query group, 23 of them fall into this range. The average savings achieved by *Chandi* on memory space in this 60-query cases is 89%.

16.8 Evaluation for Scalability

Now we evaluate the scalability of the algorithms in terms of the number of queries they can handle under a certain data rate. In this experiment, we use *Extra-N*, *Extra-N with rqs* and *Chandi* to execute query groups sized from 10 to 1000 against GMTI data. Similar with the earlier experiment, the member queries in the query group are randomly generated with the arbitrary parameter settings in certain ranges. In particular, the parameters settings in this experiment are $\theta^{cnt} = 2$ to 30, $\theta^{range} = 0.001$ to 0.01, $win =$ 1000 to 5000, and $slide =$ 500 to 5000.

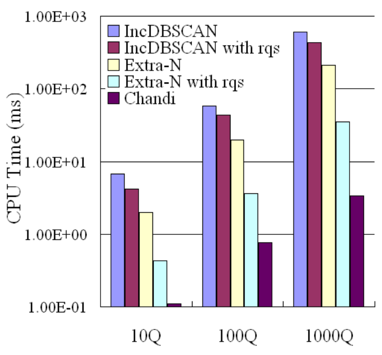

Figure 16.14: CPU Time used by Five Competitors in Logarithmic Scale

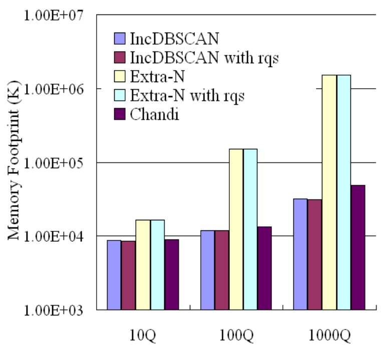

Figure 16.15: Memory Space used by Five Competitors in Logarithmic Scale

As shown in Figures 16.14 and 16.15, both the CPU time and the memory space used by *Chandi* increase modestly as the number of member queries increases. In particular, the CPU time consumed by *Chandi* increases around 6 times when the number of queries grows from 10 to 100 (increased 9 times), and then it increases less than 4 times when number of queries grows from 100 to 1000. Thus totally the CPU time consumed by *Chandi* increases 33 times when the number of queries increased from 10 to 1000, which is 100 times. Such increase for *Extra-N* and *Extra-N with rqs* are 105 times and 89 times respectively. More specifically, in our test cases, the average processing time (CPU) for each tuple used by *Chandi* to execute the 100-query and 1000-query query groups are 0.76ms and 3.3ms respectively. This indicates that our system can comfortably handle 100 queries under a 1000 tuples per second data rate, and handle 1000 queries under a 300 tuples per second data rate. For the memory space used, *Chandi* has even better performance as its utilization of memory space only increases 5 times when the number of queries increases from 10 to 1000, while such increase for *Extra-N* and *Extra-N with rqs* are both 98 times.

16.9 Conclusion for Experimental Study of Density-Based Cluster Detection Algorithms

Generally, *Chandi* is more efficient than other alternative methods in terms of both CPU and memory utilization when executing multiple density-based clustering queries specified on the same input stream. *Chandi* achieves most sharing when only one of the four parameters differ among the member queries. Among the four one-arbitrary-parameter cases, *Chandi* achieves most sharing in the arbitrary win case, while least is achieved in the arbitrary θ^{range} case. For the two-arbitrary-parameter cases, *Chandi* performs better when the member queries have arbitrary window parameters rather than arbitrary pattern parameters. For the general cases, where the member queries have arbitrary parameter settings on all four parameters, *Chandi* still clearly outperforms the other alternative methods by achieving on average 60 percent savings for CPU time and 84 percent savings in memory space. Lastly, *Chandi* shows a good scalability in terms of handling a large number (hundreds or even thousands) of queries under a high data rate.

16.10 Performance Analysis for Distance-Based Outlier and kNN Detection Algorithms

Now we discuss the potential performance of our proposed methods for other neighbor-based pattern types [1]. The savings expected for our proposed methods for the other neighbor-based pattern types are similar to those observed from our above comprehensive case study with the clustering-pattern type. Thus we only briefly review these expected savings below.

Performance Analysis for Distance-Based Outlier Queries. For individual **distance-based outlier queries**, the CPU processing resources for query execution are composed of two major parts, namely running range query searches for each new object arriving at the system and updating the neighbor-counts for each data point whose neighborhood is affected by those new objects. This is identical to the situation observed for density-based clustering queries. In our experiments for density-based clusters, for a single query execution, the cost for neighbor searches constitutes around 40 percent of the overall CPU processing costs, while the remaining costs are primarily consumed by updating the cluster structures. For distance-based outlier queries, this percentage of CPU utilization for conducting neighbor searches will surely be even much higher, as the pattern structures to be updated for maintaining outliers are simpler and thus clearly need much less computational resources for processing updates on them.

Same as clustering queries, the cost of the neighbor searches can be

[1] This dissertation does not cover experimental studies for evaluating *SDOD* and *SkNN* algorithms. A thorough discussion and experimental studies for multiple kNN query optimization can be found in my collaborative work with Avani Shastri in [SYRW11]

completely shared among all queries using our method. In particular, we simply need to run one single range query search for each new data point (using the largest range, of course) to collect all the point's neighbors for all participating queries. This indicates that, for executing a large group of queries, the costs for conducting neighbor searches can almost be completely saved. This is because the cost of running a single range query search for each data point is neglectable compared with the cost required for updating the neighbor counts for potentially large number of queries upon the arrival of each data point.

For the second part of the cost, namely the cost associated with the neighbor count maintenance, the savings depend on the overlap of the stream windows and on the number of queries that identify the same number of neighbors for each data point. Thus, the savings that can be achieved in this component may vary somewhat. However, as indicated above we can completely save the costs associated with neighbor searches (at least 40 percent). In addition, we can expect to have fairly significant amount of sharing among the pattern update processes for the different patterns, especially for larger query groups (where similar queries tend to be more likely). Thus one can safely expect that savings achieved by our method for distance-based outlier queries will be comparable or even outperform those for the cluster-based queries.

Memory-wise, the cost for individual distance-based outlier query execution is composed of both raw and meta data storage. In particular, each query needs to store all the valid data points in the window and the neighbor counts for each data point. For this pattern type, the major memory savings that could be achieved by our method should come from the storage of the actual raw streaming objects, as now using incremental pattern representation, we only need to store one reference for each data point for all queries. The storage for meta information, namely the neighbor counts, may not be saved significantly. This is because each neighbor count maintained by a query is just an integer. Storing a new count number for a query or only storing the increments from a "stricter" query will not make any difference in terms of memory usage.

Performance Analysis for kNN Queries. Computation-wise, the major cost for executing individual kNN queries comes from updating the k nearest neighbors when the new data points arrive. As we discussed in Chapter 13, the k (the largest k setting among all queries) nearest neighbors of the query object are incrementally stored, and thus the updating effort can be completely shared. In particular, if a new data point qualifies for the kNN of the query object, we only need two operations to first put the new data point into a single kNN set and remove the previous Kth nearest neighbor (the farthest one). This cost is much cheaper than placing the new data point into the kNN sets for all queries, and more importantly, will almost not be affected by the number of queries in the query group. Therefore, we envision that the percentage of savings achieved by our method in terms of CPU time will increase linearly in the number of queries.

Memory-wise, as discussed in Chapter 15, using our shared execution method, the information (raw and meta) needs to be stored by a group of query is the same as what needs to be stored by a single query (the query with the largest k setting). Therefore the memory cost of our method is independent from the number of queries in the query group.

Chapter 17

Related Work for Part II

Algorithms for density-based clustering queries over streaming data include [YRW09, CT07, CEQZ06]. Among these works, [CT07] and [CEQZ06] have goals different from ours, because they are neither designed to identify the individual members in the clusters nor enforce the sliding window semantics for the clustering process. Thus these two algorithms cannot be applied to solve the problem we tackle in this work. [YRW09] is the only algorithm we are aware of that detects density-based clusters in sliding windows. Our experimental study conducted in Chapter 16 shows that our shared execution strategy largely outperforms the strategy of using this algorithm independently for each query. [YGX$^+$10] builds a visual system to allow analysts to interactively explore density-based clusters in streaming environments.

[AF07] and [MP07] discuss the problem of detecting distance-based outliers and top-k nearest neighbors in data streams respectively. Again, these works concentrate on single query execution only. We borrow the basic ideas of maintaining meta-information, such as potential outlier sets and k nearest neighbors from them. However, instead of maintaining such meta-information independently for each query, we developed integrated maintenance strategies for shared execution among multiple queries, and thus achieve significant savings on both CPU and memory resources.

As a general query optimization problem, multiple query optimization has been widely studied for not only static but also streaming environments. Such techniques can be roughly divided into two different groups, namely "plan level" and "operator level" sharing. "Plan level" sharing techniques [LRY08, CDTW00, CDN02] aim to allow the different input queries to share the common operators across their query plans, and thus lower the overall costs for multiple query execution. Operator level sharing studies the sharing problem on a finer granularity, namely within the individual operators. In particular, they aim to share the operator state as well as the query processing computation within a single operator, when multiple queries have similar yet not identical operator specifications. For example, two queries may calculate aggregations for the same input stream but using different window sizes. The problem we solve in this paper falls into the operator level sharing category.

Previous research efforts discussing such operator level sharing techniques focus on simple operators, such as selection and join operators [MSHR02, HFAE03, KFHJ04, WRGB06, ZKOS05], and aggregation operators [KWF06, AW04, ZKOS05]. To our best knowledge, none of them discuss the sharing for clustering operators. Some general principles used in these works, such as query containment [HFAE03], can also be applied in our context (used in sharing range query searches for our solution). However, the key problem we address in this work, namely the integrated maintenance of density-based cluster structures identified by multiple queries, is different from the optimization effort required by selection, join or aggregation sharing. In particular, the meta-information we need to maintain, namely the cluster structures defined by individual cluster member objects as well as their interrelationships, is much more complex than those for selection, join or aggregation operators, which are usually pair-wise relations or simply numbers (aggregation results). Efficient maintenance of such meta-information requires thorough analysis of the properties of density-based cluster structures, which is a key contribution of our work. This has not been studied in any of these works.

Part III

Summarization and Matching of Neighbor-Based Patterns

In Part III of this dissertation, we discuss the problem of summarization and matching of neighbor-based patterns in streaming environment. It extends the scope of this dissertation from real-time pattern extraction as covered in Parts I and II to also include long-term pattern analysis and retrieval.

Within the three neighbor-based pattern types discussed in this work, density-based clusters have the most complicated pattern structures and thus are most challenging for pattern summarization and matching. So, in the third part of my dissertation, I mainly focus on studying these problems for density-based clusters. In addition, I will discuss matching and summarization for the other two pattern types in each component of our framework.

Chapter 18

Supported Queries and System Overview

We aim to support two types of analytical queries:

18.1 Continuous Pattern Extraction Queries

A *Continuous Pattern Extraction Query* returns both *full* and *summarized representation* of the extracted patterns (Figure 18.1). The design of our proposed pattern summarization formats will be introduced in Chapter 19.1.

DETECT $NeighborBasedPattern^{f+s}$ ***FROM*** $stream$
USING $pattern\_para\_1 = x_1$ ***and ...*** $pattern\_para\_2 = x_2$
IN Windows WITH $win = w$ ***and*** $slide = s$

Figure 18.1: Continuous Pattern Extraction Query Returning Full *(f)* and Summarized *(s)* Representations of Neighbor-Based Patterns

18.2 Pattern Matching Queries

Given a user specified to-be-matched pattern P_i, a pattern matching query finds patterns similar to P_i that reside in the historical pattern archive. We show a template of such a query in Figure 18.2.

GIVEN $NeighborBasedPattern^s\ P_i$
SELECT $NeighborBasedPattern^s\ P_j$ ***FROM*** $History$
WHERE $Distance(P_i, P_j) \leq sim\_threshold$

Figure 18.2: Pattern Matching Query finding Patterns Similar to the To-Be-Matched Cluster Based on Cluster Summarization

The to-be-matched pattern can be any pattern specified by an analyst. Typically, it may be a pattern detected in the most recent portion of the stream that represents the newest characteristics of the stream. The matched patterns, if any, will be found in the historical pattern store, which archives the patterns extracted by applying the *Continuous Clustering Query* against earlier portions of the stream.

18.3 System Overview

To support these two types of analytical queries, we design a framework composed of four major components (Figure 18.3). Here we give a brief overview of the functionalities of each component, while in-depth technical details are discussed later in Chapters 21 to 23.

The **Pattern Extractor** executes the *Continuous Pattern Extraction Query* (Figure 18.1) against the input stream. It outputs both *full* and *summarized representations* of the extracted patterns. Both representations are returned

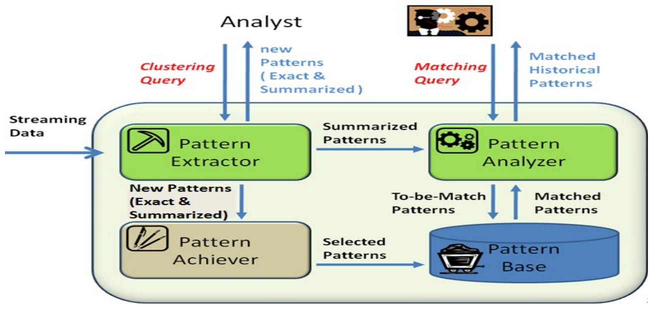

Figure 18.3: System Overview

to the analyst for real-time monitoring. Meanwhile, the extracted patterns are also passed to the Pattern Archiver for storage, and to the Pattern Analyzer for pattern matching.

The **Pattern Archiver** selectively archives the newly detected patterns into the Pattern Base. These archived patterns constitute the *Stream History* available for subsequent *Pattern Matching Queries* (Figure 18.2). The Pattern Archiver controls which extracted patterns should be kept in the Pattern Base and at which resolution they should be archived.

The **Pattern Base** organizes the archived patterns. To facilitate pattern matching against historical patterns, it employs multiple feature indices to organize the archived patterns. This helps the *Cluster Matching Queries* to quickly locate any potential matching candidates.

The **Pattern Analyzer** executes the *Cluster Matching Queries* (Figure 18.2). If an analyst is interested in any newly extracted pattern and would like to learn whether similar patterns had been detected before in the *Stream History*, she can submit her *Cluster Matching Query* to the Pattern Analyzer to search for matches against the Pattern Base.

Chapter 19

Pattern Summarization

19.1 Summarization for Density-Based Clusters

In this section, we present our proposed *summarized representation* of density-based clusters, which is critical for both cluster storage and matching.

19.1.1 Features of Density-Based Clusters

Based on our analysis, we identify four key features that define each density-based cluster, which can be divided into two categories, namely external and internal features.

External Features:

Location: The location of a cluster indicates its position in the data space. It provides basic information about each cluster, such as where a congestion area (a cluster) arises in the traffic, or in which price range an intensive-transaction area, a cluster based on price, volume and transaction time, is detected in the stock transaction stream.

Shape: Density-based clusters can have arbitrary shapes. The shape is a key feature, because a certain shape of the cluster may convey specific meaning for an application. For example, for the clusters representing intensive-transaction areas in stock transactions, a cluster having a long spread on the transaction price but a short range on transaction time conveys that a large number of transactions of a certain stock happened in a short time period while its price fluctuates dramatically within this time period.

Internal Features:

Connectivity: The connectivity of a density-based cluster describes how sub-regions within the cluster are connected. It is an important feature for density-based clusters for both definition and application reasons. First, it defines the internal structure of each cluster. The definition of the density-based cluster (see section 2.2.1) relies on the connectivities among sub-regions to define each cluster. Second, the connectivities among sub-regions may be relevant to applications. For example, if two sub-regions within a single cluster representing a group of moving troops are not directly connected, then this may indicate the units in these two sub-regions cannot directly communicate with each other, because there are no connected "Head Nodes" (core objects) in these two sub-regions of their wireless network.

Density Distribution: Although the definition of density-based clusters imposes a minimal density requirement on objects in a cluster, the density of each cluster can be rather diverse across its sub-regions. The density distribution within each cluster may be of an analyst's interest in many applications. Using the earlier example, even in a single congestion area, the level of congestion (density of vehicles) may vary among sub-regions. Therefore, the density distribution in each sub-regions may be the key for working out a congestion relief plan, as the super dense sub-regions may be the key points that cause the congestion.

19.1.2 Skeletal Point Summarization

In general, any effective *summarized representation* for density-based clusters has to be able to capture these four key features. Given that any density-based cluster may have an arbitrary shape, connectivity and also density distribution, it is clear that using any single aggregation method to represent any of these features will end up with a poor descriptiveness. Therefore, we propose an alternative summarization principle for density-based clusters. Namely we divide each cluster into sub-regions to pursue better homogeneity on these features in each sub-region. Then we describe the features in each sub-region and also the interrelationships among the sub-regions.

First, we investigate a summarization method based on the representative points and the connections among them. In general, we divide each cluster into sub-regions and use one representative point to represent each sub-region. In particular, for a given cluster C_i, we try to find a subset of the cluster member objects in C_i to represent it. We call this subset *skeletal point set* and each object in this set a *skeletal point*. The *skeletal point set* should have the following property:

1) *coverage:* To ensure every sub-region in a cluster is covered, the neighborhood of the *skeletal point set* needs to cover all the cluster member object

in a cluster C_i. In particular, every cluster member objects in C_i has to be a *neighbor* (see definition in Chapter 19.1) of at least one *skeletal point*.

2) *connectivity:* To capture the connectivity within a cluster, the *skeletal point set* needs to be self-connected. In particular, any two *skeletal points* in a *skeletal point set* need to be *connected* (see definition in Chapter 19.1). If two *skeletal points* are neighbors to each other, we say there is an "edge" between them.

3) *minimality:* To ensure the compactness of the summarization, the *skeletal point set* of a cluster C_i should have the smallest cardinality among all possible subsets of C_i, which have the two properties above.

A *skeletal point set* and the edges among them constitute a *summarized representation* of a cluster. We call it the *skeletal point graph*. A *skeletal point graph* could effectively capture most of the features of a density-based cluster, namely the *position*, the *connectivity* and the *shape*. However, it suffers from the following limitations.

Limitations. First, the *skeletal point graph* has a limited descriptiveness for clusters' density distributions. This is because using this summarization method the sub-regions within a cluster, each represented by a *skeletal point's* neighborhood, potentially overlap . Therefore, there is no succinct expression that can be employed to clearly describe the density distribution within each cluster. On the one hand, the naive approach of ignoring the overlaps and expressing the density in each sub-region separately may cause ambiguities, as each object may be counted in the density calculation for multiple sub-regions. On the other hand, if one would like to quantify and explicitly express the overlaps, this would require complex expressions

Second, such *skeletal point graph* is not easily computable (identifiable). Generally, finding such a *skeletal point graph* for a cluster is equal to the problem of identifying the ***minimal connected dominant set*** in an undirected graph. In particular, we could model any density-based cluster C_i as an undirected graph G_i, with each cluster member object being a vertex. In this graph, an edge exists between two cluster member objects if they are neighbors to each other. Given this graph G_i as input, the *minimal connected dominant set* identified for G_i will be equal to the *skeletal point set* of C_i.

Unfortunately, the problem of identifying a *minimal connected dominant set* has been proven to be NP-complete [GK96]. The state-of-the-art approximation techniques can find an approximate set with a cardinality that is no larger than $3 + ln(\theta)$ times the minimal cardinality in the worst case. However, their CPU complexity is at least $O(n^2)$, with n the number of objects in each cluster for our problem. And even worse, they need to take this graph as their input. Namely, this requires the cluster generation process to not only provide the cluster member objects of each cluster, but also the edges (pair-wise neighbor relationship) among the cluster member objects. Based on our best knowledge, none of the state-of-the-art density-based clustering algorithms provides such ”edge ”information. Even if one could design such a clustering algorithm that were to provide such information, it would cause a huge memory consumption as the number of edges existing among the cluster member objects may be very large, n^2 in the worst case.

Third, the *skeletal point graph* is not easily matchable. The key difficulty for matching such *skeletal point graphs* is caused by their indeterminacy. In particular, the *skeletal point graph* of a cluster is non-deterministic, indicating that a single cluster C_i may not have a unique, but instead multiple alternative *skeletal point graphs* with very different graph structures. Thus, the matching result between the *skeletal point graphs* of two clusters may not effectively tell us about the similarity between these two clusters.

In addition, the problem of matching two *skeletal point graphs* is equal to the problem of calculating the *graph edit distance* between two graphs. This problem is known to be very expensive in terms of CPU utilization [NRB06], and thus not suitable for real-time query processing.

Conclusion. As a good starting point, the Skeletal Point Summarization (SKPS) has a good descriptiveness for most of the features of density-based clusters. However, it suffers from the limitations above and thus does not constitute an ideal solution for summarizing density-based clusters in streaming environments. As we have observed, these limitations are caused by its overlapping and non-deterministic sub-region division strategy. Thus, we propose to improve it by adjusting the sub-region division strategy in the following section.

19.1.3 Proposed Solution: Skeletal Grid Summarization

Basics of Grid-Based Summarization. The limitations suffered by SkPS are caused by its overlapping and non-deterministic sub-region division strategy. Thus, we now propose to adapt SkPS by dividing each cluster into non-overlapping and uniformly sized sub-regions. In particular, we divide the whole data space into uniformly sized grid cells. For each cluster, its sub-region division is now determined by the grid cells into which its members fall. Therefore, a cluster C_i can now be represented by all the grid cells which contain at least one $C_i's$ cluster member objects.

Connectivity Preservation. However, this simplistic grid-based summarization method clearly lacks one key capability of the SkPS solution, namely it does not capture the connectivity within clusters. In SkPS, the connectivity information of each cluster is well preserved. Such connectivity information can be divided into two categories, namely the inner- and inter sub-region connections. First, each sub-region in SkPS itself is "well connected", as all objects in a sub-region are neighbors of the same *skeletal point*. Second, the inter-connections among different sub-regions are explicitly expressed by the "edges" in SkPS. While in this simplistic grid-based summarization, neither of these two types of connectivity information are available. First, we have no knowledge about whether the objects within a sub-region (grid cell) are connected. Second, although we know the topological adjacency among the grid cells, we do not know whether objects in adjacent grids are connected.

Connectivities In Grid Cells. To solve this problem, we propose to integrate the concept of "connectivities" into the grid-based solution. As foundation, we first introduce the concept of *status* of a grid cell. That is, we divide the grid cells in each cluster's summarization into two categories, namely "*core grids*" and "*edge grids*".

Definition 8 ***Core cells****: a core cell of a cluster C_i contains at least one core object (See Def. 2.2.1) of C_i.*

Edge cells*: an edge cell of a cluster C_i contains no core object, but at least one* *edge object (See Def. 2.2.1) of C_i.*

***noise cells**: a noise cell contains neither core objects nor edge objects of any cluster* [1].

For **inner-sub-region connections**, we follow the basic principle for the sub-region division strategy, which is to pursue homogeneity in each sub-region. In particular, we pick a fine grid size to guarantee that the objects that fall into the same grid cell are neighbors of each other. More precisely, the diagonal of each grid should be equal to the range threshold θ^r in the given clustering query (see Section2.2.1). This grid cell size selection will be relaxed later in our discussion of the multi-resolution cluster summarization (Chapter 21). Under this fine grid size selection, the *core* and *edge girds* can be shown to have the following properties.

Lemma 19.1 *All objects in a core cell belong to the same cluster.*

Proof 19.1 *Since each core cell contains at least one core object and all the objects in each core cell are now neighbors of each other, it implies that now all objects in the same core cell are neighbors of at least one common core object. Based on the definition of density-based cluster (see Def. 2.2.1), the neighbors of a core object belong to the same cluster.*

Lemma 19.2 *The number of objects in an edge cell must be less than the count threshold θ^c in the clustering query.*

Proof 19.2 *We prove this lemma by contradiction. Given that all objects in a grid cell are neighbors of each other, if there are at least θ^c objects in an edge cell, those*

[1] *noise grid* will not appear in our proposed cluster summarization for any cluster. They are only used in cluster computation stage (see Chapter 20)

objects would be core objects, as they all have at least θ^c neighbors. This contradicts the definition of edge grid (Def. 2.2.1).

Given these properties, each grid cell is "well-connected" and constitutes a basic unit for the inter-grid connection expression, as defined below.

For the **inter-sub-region connection**, we now define the "connections" between grid cells.

Definition 9 *Two core cells ccl_1 and ccl_2 are **directly connected**, if there exists at least one core object p_i in ccl_1 and one core object p_j in ccl_2 that are neighbors of each other. Two core cells ccl_0 and ccl_n are **connected**, if they are directly connected to each other, or there exists a sequence of core cells $ccl_0, ccl_1, ...ccl_{n-1}, ccl_n$, where for any i with $0 \leq i \leq n-1$, each pair of core cells ccl_i and ccl_{i+1} are directly connected with each other.*

*An edge cell ecl_i is **attached** to a core grid ccl_j, if there exists at least one object p_i in ecl_i and one core object p_j in ccl_j that are neighbors of each other.*

Two edge cells are neither connected nor attached to each other.

Given the connection definition for grid cells above, all *core cells* of a cluster are *connected* to each other, and all *edge cells* are *attached* to at least one *core cell* of C_i.

Skeletal Grid Summarization. Based on the status and connections of grid cells, we now give the definition of our proposed Skeletal Grid Summarization (SGS) model.

Definition 10 *A **Skeletal Grid Summarization** (SGS) of a density-based cluster C_i is composed of all grid cells that contain at least one cluster member object of C_i. We call each grid cell in a SGS, a **Skeletal Grid Cell** (Sc) of C_i.*

$SGS = \{Sc_0, Sc_1, ...Sc_n\}$. Each Sc_i has five attributes, namely $SG_i = (location[], size, population, status, connection[])$.

1) location vector: a sequence of values, each indicating the minimum value on one of the dimensions covered by Sc_i.

2) side length: the range of values on each dimension.

3) population: the number of objects contained by Sc_i

4) status: whether Sc_i is a core or an edge cell.

5) connection vector: a sequence of boolean connection indicators, each indicating Sc_i's connection to one of its adjacent skeletal grid cells. For any edge or noise cell, all connection indicators are "false". For any core grid, a connection indicator is "true" if the corresponding adjacent skeletal grid cell Sc_j is a core cell and Sc_i and SG_j are directly connected, or if SG_j is an edge cell attached to SG_i.

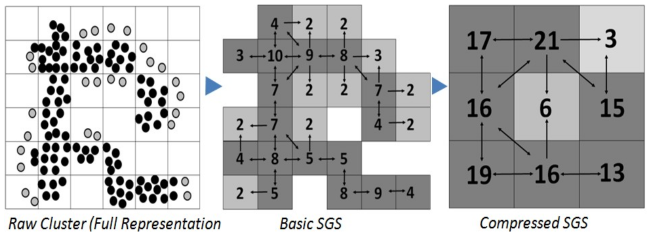

Figure 19.1: Example of full representation, basic SGS and compressed SGS of a 2D cluster

Figure 19.1 shows an example of our proposed Skeletal Grid Summarization (SGS) for a 2D cluster. SGS achieves our goal of preserving all four features, as shown below.

Lemma 19.3 ***Fidelity to Location and Shape:*** *The data space covered by $C_i.SGS$ is larger than that covered by the cluster member objects of C_i by a bounded error.*

Namely, any point in the data space covered by $C_i.SGS$ is at most θ^r away from a cluster member object in C_i.

Proof 19.3 *The data space covered by $C_i.SGS$ is composed of the union of that covered by all its skeletal grid cells. Since all member objects of C_i fall into these grid cells, the data space covered by $C_i.SGS$ is larger than that covered by C_i's member objects. Since each skeletal grid cell in $C_i.SGS$ contains at least one member of C_i, and the diagonal of each cell is θ^r, any point in data space covered by a skeletal grid cell is at most θ^r away from a member of C_i.*

Lemma 19.4 ***Fidelity to Density Distribution:*** *For any sub-region in a cluster C_i, which is composed of n grid cells, $C_i.SGS$ can accurately express its density.*

Proof 19.4 *Since the skeletal grid cells in $C_i.SGS$ don't overlap, the population recorded by each skeletal grid cell accurately reflects the number of objects in it. Therefore, for any sub-region in a C_i composed of n skeletal grid cells, we can always accurately calculate its density by dividing its totally population by its total volume.*

Lemma 19.5 ***Fidelity to Connectivity:*** *If there are two sub-regions in C_i connected through a connected core object path composed of n core objects, there must exist a core grid path connecting these two sub-regions with* ***at most*** *n core cells on this path.*

Proof 19.5 *Since any skeletal grid cell containing at least one core object is a core cell, if there exists a core object path between two sub-regions, there must exist a core cell path between them. In the worst case, each core grid on this core grid path* *contains only one core object. Thus the length of the core grid path is at most equal to the length of the core object path.*

In conclusion, SGS effectively captures all key features of density-based clusters using a compact description.

Dimensionality Concern. A concern for such a grid-based data space division is that it may suffers from the dimensionality problem, namely, that the number of grids needed to represent a cluster may be huge for clustering high dimensional data. However, in practice, this problem does not significantly affect the performance of our summarization method for the following reasons: First, even for high dimensional data, the clustering is usually applied to only a subset of the data's attributes. Second, as density-based clusters are "overly dense" areas in the whole data space, it has been observed that they tend to be identified only in compact subareas and less likely to spread across the whole data space [CT07]. Third, as our proposed summarization strategy supports multiple resolutions, the analyst can choose the resolution level based on the system budget.

19.2 Summarization for Distance-Based Outliers

When each individual distance-based outlier is viewed as a single pattern, the pattern structure of each outlier is very simple. In particular, each outlier is a single object (tuple). In this case, further summarizing the outliers may not be necessary, as the amount of information needed to describe each outlier is already very small.

An alternative way of viewing distance-based outliers is to identify the potential “outlier groups” among the detected outliers. In particular, each outlier group is a group of outliers which appear in a same area of the data space. The outliers in a same outlier group are usually identified as outliers for the same or similar reasons. For example, in the stock transaction stream, few transaction records with (similar) high volumes may be identified as outliers, as they are all “far away” from the common transactions with normal volumes. The two outliers on the lower right corner of Figure 2.2 are example for a potential outlier group.

To identify these outlier groups, we can apply clustering algorithms on all detected outliers. Then for each detected outlier group, we can summarize its characteristics using the summarization techniques for clusters. To keep the summarization succinct, we can use simple aggregative summarization techniques, such as Centroid + Radius + Population, combination to convey the key characteristics of each outlier group.

19.3 Summarization for kNN

For k nearest neighbors of an object, when the constant k is small, further summarizing them may not be necessary as well, as the amount of information needed to describe them is already small. Summarization for k nearest neighbors may be needed when k is very large. In this case, we can view the k nearest neighbors of an object as a dataset and summarize it using dataset summarization techniques, such as histograms [BRV11, dAMFH08] or clustering methods [JMF99].

Chapter 20

Pattern Extractor

In this chapter, we introduce the pattern extractor that executes the *Continuous Pattern Extraction Query*, and thus extracts the patterns for each window in real-time. For subsequent cluster analysis, it outputs the patterns in both full and summarized representations.

20.1 Extracting and Summarizing Density-Based Clusters

20.1.1 A Two Stage Strategy and Its Limitations

For density-based clusters, to provide such functionality, a straightfoward methodology would be a two-stage process, namely clustering first followed by summarizing. In particular, at the first stage, we could employ the state-of-the-art clustering algorithm [YRW09] to extract the clusters. Then, at the second stage, we could design a summarization algorithm to summarize the extracted clusters into our proposed SGS summarization format (see Chapter 19.1).

However, this strategy will suffer from significantly lower performance compared to conducting clustering only (without summarization), as the summarization stage is expensive in terms of both CPU and memory consumption. Most notably, summarizing a cluster into SGS needs extra information beyond the regular clustering results, which is not provided by state-of-the-art clustering algorithm. Namely, same as in the case of the *skeletal point graph*, to form SGS of clusters, one needs not only the objects in each cluster, but also the connections (pair-wise neighborships) among the objects, because both the status and connections of each *skeleton grid cell* needs to be determined by the connections which the objects contained by it has. We have noted that requiring a clustering algorithm to provide such connectivity information in the output will cause serious system overhead for the clustering process itself. The alternative to obtain such connectivity information is to re-search for all the neighbors for each cluster member during the summarization process. However, this alternative is obviously even more computationally expensive and constitutes a significant waste of CPU time, as the connections among the objects would already have been identified before during the clustering process.

To solve this problem, we instead propose an integrated strategy that incorporates cluster extraction and summarization into a single process. The key observation that motivates this integrated computation method is given below.

Observation 20.1 *The main tasks for both density-based cluster extraction and SGS computation are the same, namely to first identify the connections (neighborships) among the objects and analyze them to form the cluster structures in either the full or a summarized representation.*

This observation reveals the key commonality among the cluster extraction and summarization processes. Based on it, we design an integrated extraction+summarization method to effectively share the *neighborship* identification and cluster formation processes.

20.1.2 Incremental Computation and Challenges

To avoid conducting the prohibitively expensive clustering process from scratch at each window, our proposed method incrementally maintains the cluster structures across the windows. To realize incremental computation, we need to find appropriate meta-data that can be maintained for both the full and summarized cluster representations. Our proposed solution is that, besides the raw data falling into each window, which needs to be maintained for cluster extraction in any case, we incrementally maintain the *skeletal grid cells* in the data space. With updated *skeletal grid cells*, we can easily output both the summarized and full representations of detected clusters. First, based on connections among the *skeletal grid cells*, we can easily determine the summarized representation SGS (a group of connected *skeletal grid cells*) for each cluster. Second, given the SGS of a cluster C_i, $C_i.SGS$, we can figure out the cluster member objects of C_i based on the objects falling into the respective *skeletal grid cells* belonging to $C_i.SGS$.

However, incrementally maintaining *skeletal grid cells* in an efficient manner is a challenging task. In particular, tracking the changes to the *skeletal grid cells* caused by expired objects can be expensive in terms of system resource utilization, and thus constitutes the key performance bottleneck for *skeletal grid cell* maintenance.

When an object p_{exp} expires, it needs the connections at the object level, to update the connections among the *skeletal grid cells*. For example, when p_{exp} expires, we first need to know which objects are neighbors of p_{exp}, as their *neighborships* with p_{exp} will end from now on. This may break the connections between the *skeletal grid cell* Sc_i in which p_{new} resides and those in which p_{exp}'s neighbors reside. However, considering the large amount of pair-wise *neighborships* that may exist among the objects, maintaining all of them has been shown to be extremely expensive in terms of system resource utilization, analytically and experimentally [YRW09]. Therefore, the straightforward incremental maintenance method, which updates *skeletal grid cells* corresponding to each insertion and deletion, is not practical.

20.1.3 "Lifespan" Analysis

To solve this computation bottleneck, we present a *skeletal grid cell* maintenance method using "lifespan" analysis. This method elegantly eliminates the need for handling the impact of expired objects on the *skeletal grid cells*. The solution is based on the observation that in the sliding window semantics the lifespan of any object as well as the *neighborships* among objects are deterministic. Therefore, at the insertion stage, when we handle the impact of new objects on the *skeletal grid cells*, we take the lifespans of the objects into consideration. In particular, we pre-determine the changes that will happen to the *skeletal grid cells* when these objects expire later. Then at the expiration stage, no further update is needed to handle the impact of expired objects. Thus we avoid the bottleneck discussed above.

Among the five attributes of a *skeletal grid cell*, except *location* and *side length* that are fixed over time, the other three namely *population*, *status* and *connections* are changing over time as the objects come and go with each window slide. The *population* of each *skeletal grid cell* is easily trackable with a simple object counter. Thus, we focus on the lifespan analysis of the *status* and the *connections*.

Basics for Lifespan Analysis. First, we start with analyzing the lifespan of individual objects.

Observation 20.2 *Given the slide size $Q.slide$ of a query Q and the starting time of the current window $W_n.T_{start}$, the* ***lifespan*** *of an object p_i in W_n with time stamp $p_i.T$ is $p_i.lifespan = \lceil \frac{p_i.T - W_n.T_{start}}{Q.slide} \rceil$, indicating that p_i will participate in windows W_n to $W_{n+p_i.lifespan-1}$.*

The number of windows that an object p_i can survive in is determined by after how many window slides that $p_i's$ time stamp will still be greater than the starting time of the window. Based on the lifespan of individual objects, we analyze the lifespan of *neighborship* between two objects.

Observation 20.3 *Given two objects p_i and p_j, the neighborship between them, $Neighbor(p_i, p_j)$ will hold for*
$Neighbor(p_i, p_j).lifespan = Min(p_i.lifespan, p_j.lifespan)$ windows, namely,

it will exist in all windows from W_n to $W_{n+Neighbor(p_i,p_j).lifespan-1}$ until either p_i or p_j expires.

Based on these observations, we can further analyze the lifespan of different stages of an object's "*career*".

Observation 20.4 *Given an object p_i and all its neighbors objects p_{nb1} to p_{nbk}, the number of windows in which p_i will be a core object $p_i.core\_lifespan = Min(p_i.lifespan, win\_\theta^c\_nei)$, with $win\_\theta^c\_nei$ the number of windows in which at least θ^c objects within p_{nb1} to p_{nbk} will participate. The number of windows in which p_i will be edge object $p_i.edge\_lifespan = Min[p_i.lifespan - p_i.core\_lifespan, Max_{1 \leq j \leq k}(p_{nb_j}.core\_l$*

Basically, an object will be a *core object* in all the windows that it has at least θ^c neighbors. It will be an *edge object* when it *core object career* ends (no longer has enough neighbors) but at least one of its neighbors is still a *core object*.

Lifespan Property at Grid Cell Level. To tackle *skeletal grid cell* maintenance, we now extend the concept of lifespan from the object level to the grid cell level. In particular, we analyze how the lifespan of objects, their *neighborships* and their *career* affects the lifespan of *skeletal grid cell*s' *status* and *connections*. For each *skeletal grid cell* Sc_i, we maintain one lifespan indicator for $Sc_i.status$ and one for each $Sc_i.connections[i]$. Each lifespan indicates that, based on the objects in the current window, in how many future windows the value of this attribute will persist. These indicators are updated whenever new objects arrive.

Lemma 20.1 ***Status Lifespan.*** *Given a skeletal grid cell Sc_i, all the objects p_0 to p_n in Sc_i , the number of windows in which Sc_i will be a core cell $SG_i.core\_lifespan = Max_{0 \leq i \leq n}(p_i.core\_lifespan)$.*

Lemma 20.1 can be deduced from the definition of a *core cell* (Def. 8). Namely, Sc_i is a *core cell* if it contains at least one *core object*.

Lemma 20.2 ***Connection Lifespan.*** *Given two skeletal grid cells Sc_i and Sc_j, and all objects in Sc_i, $p_0^{sc_i}$ to $p_n^{sc_i}$, and all objects in Sc_j, $p_0^{sc_j}$ to $p_m^{sc_j}$, the number of windows in which Sc_i and Sc_j will be connected is defined as $Connection(Sc_i, Sc_j).lifespan = Max[Min(p_a^{sg_i}.core\_lifespan, p_b^{sg_j}.core\_lifespan, Neighbor(p_a^{sg_i}, p_b^{sg_j}).lifespan)]$, $\forall a \in [0, n]$, $b \in [0, m]$.*

This indicates that two *skeletal grid cells* remain connected if at least one pair of *core objects*, each from one of the two *skeletal grid cells*, are neighbors to each other.

Auxiliary Meta-Data. To insure that we only run one range query search (rqs) for each new object and never re-run rqs for existing objects, we maintain an auxiliary meta information for each object in the window. In particular, we maintain a “non-core-career neighbor list” for each object p_i to store all p_i's neighbors in its “non core career”. For example, p_i currently may have 100 neighbors. Based on the lifespan analysis, it will be a *core object* for 3 windows and then due to most of its neighbors expiring, it will become a *edge object* for 2 windows before expiration. In this case, the “non-core-career neighbor list” of p_i only contains its neighbors in the last 2 windows of its lifespan, say 5 objects.

The “non-core-career-neighbors” of each object are maintained in a dynamic hash table. The hash table of each object p_i is initialized to have n buckets, with n the number of windows that p_i can survive. The hash key of the table is the number of windows that a neighbor object can survive. For example, when a data point p_i finds a "non-core-career-neighbor" p_j, p_j will be added to the k^{th} bucket of the hash table, with k the number of windows p_j can still survive (if k is larger than the number of buckets remained on p_i, p_j is put in the last bucket). At each window slide, we can simply remove the whole first bucket of each remaining object p_i, as all the neighbors in this bucket must be expired after the window slide. Then, the second bucket becomes the new first bucket and so on. This removal process also indicates that the numbers of windows that the other neighbors of p_i can survive decrease by one, as each remaining bucket now is now one position closer to the first bucket.

The number of neighbors in such "non-core-career neighbor list" is bounded by the constant θ^c. Namely an object can never have more than θ^c neighbors in its non-core career, otherwise it would instead be a *core object* in those windows. This theoretical bound guarantees the "lightness" of this auxiliary meta-data. On the other hand, it provides all necessary access to the objects' neighbors needed in our cluster extraction process (see Chapter 5.2.6) It thus guarantees that we only run the minimum number of range query searches (one for each new object) during the clustering.

20.1.4 Proposed C-SGS Algorithm

We call our proposed algorithm based on the maintenance of Skeletal Grid Cells (SGS), C-SGS.

Initialization. For a continuous clustering query, at the initialization stage, C-SGS builds a grid-based index whose grid cell size is equal to the size of the finest *skeletal grid size* for this query (see Chapter 19.1). We assign to each grid cell in this index the same attributes as the *skeletal grid cells*, while we set their status to be *noise*, density to be "0", and connections to be all "false" initially.

Handling Insertions. For each new object p_{new} inserted into the window, C-SGS first loads it into its corresponding *skeletal grid cell* based on its position in the data space. Then, we run a range query search for p_{new} to identify p_{new}'s neighbors. Based on the lifespan of p_{new} and its neighbors (Lemma 20.2), we can determine the lifespan of the *neighborships* among p_{new} and its neighbors (Lemma 20.3), as well as the lifespan of different stages of $p_{new}'s$ "career" (Lemma 20.4). Using this information, we now update the *status* and *connections* of the *skeletal grid cells* in which p_{new} falls into and in which its neighbors reside.

For ***status*** of *skeletal grid cells*, the insertion of a new object may only cause two types of changes. Namely, it may "promote" the *skeletal grid cells* to become *core cells* or "prolong" their *core cell lifespans*.

status promotion: A new object p_{new} may promote the *skeletal grid cell* Sc_i that it resides in to become a *core cell*, if it becomes the first *core object* in Sc_i. In this case, we set the *status* of Sc_i to *core cell* and set its core cell lifespan equal to the core objects lifespan of p_{new}. An example of this case is shown by Case 1 of status promotion in Figure 20.1.

p_{new} may also cause a *status* change of a *skeletal grid cell* by upgrading its non-core-object neighbors, which reside in these affected *skeletal grid cells*, to *core objects*. In this case, for each upgraded neighbor p_{upg} of p_{new}, we first determine the lifespan of p_{upg}'s career by analyzing itself and its neighbors. As every p_{upg} was a non-core object, the "non-core-career neighbor list" will help us to quickly access all its neighbors without running any range query search again. Thus, we update the *status* of the *skeletal grid cells* in which p_{upg} resides to *core cell* and set its core grid lifespan equal to the core object lifespan of p_{upg}. Correspondingly, the "non-core-career neighbor list" of each p_{upg} also needs to be updated to exclude those objects that will only be neighbors of p_{upg} in its core object career. An example of this case is shown in Case 2 of status promotion in Figure 20.1.

status prolong: A new object p_{new} may prolong the core cell lifespan of the *skeletal grid cell* Sc_i in which it resides, if $p_{new}'s$ core object lifespan is longer than that of any existing object in Sc_i. In this case, we set $Sc_i's$ core cell lifespan equal to the core object lifespan of p_{new}. An example of this case is shown in Case 1 of status prolong in Figure 20.1.

p_{new} may also prolong the core cell lifespans of the *skeletal grid cells* by extending p_{new}'s neighbors' core object lifespan. For each p_{new}'s neighbor whose core object lifespan is extended because of p_{new}'s arrival, p_{cole}, we first determine how long its core object lifespan is extended, by analyzing it would have at least θ^c neighbors in how many more windows after p_{new} joining its neighborhood. Then, we update the core cell lifespan of the *skeletal grid cell* in which each p_{cole} resides to the core object lifespan of the corresponding p_{cole}, if the later is longer. An example of this case is shown in Case 2 of status promotion in Figure 20.1.

For ***connections*** of *skeletal grid cells*, the insertion of a new object may

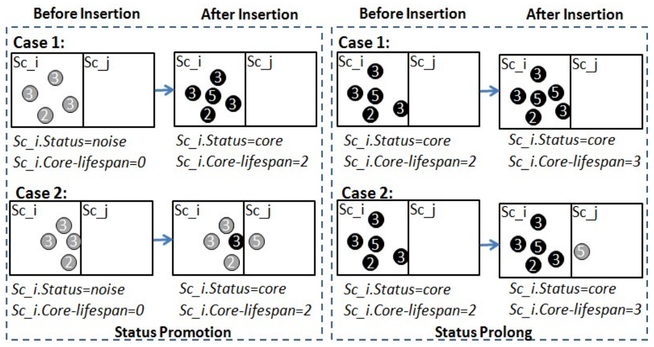

Figure 20.1: Examples of updating cell status. $\theta^c = 4$, grey circle=*edge point*, black circle=*core point*, number on each object= number of windows the object can survive.

also only cause two types of changes. Namely, it may build new connections between *skeletal grid cells* or prolong the life span of existing connections.

connection build-up: A new object p_{new} may build the connection between the *skeletal grid cell* Sc_i that it resides in as a core object to another *skeletal grid cell* Sc_j, if it finds a *core object* neighbor p_j in SG_j. In this case, we first set the corresponding connection indicator of both Sc_i and Sc_j to "true". Then we set the life span of this connection to the smaller of the two core objects life spans of p_{new} and p_j.

p_{new} may also build new connections between *skeletal grid cells* by upgrading its non-core object neighbors to *core objects*. In this case, for each upgraded neighbor p_{upg}, we first determine its core object life span. Then, we check whether there already exists any *core objects* in $p'_{upg}s$ "non-core-career neighbor list" that are in the adjacent grid cell. If yes, we build the connections between these grid cells.

connection life span prolong: A new object p_{new} may prolong the *life span* of the connections between two grid cells, if $p'_{new}s$ builds a new *core object neighborship* across them that will last longer than any other *core object neighborship* existing before across these two grid cells. In this case, we update the life span of this grid cell connection to the life span of this new *core object neighborship*.

p_{new} may also prolong the life spans of the connections between affected grid cells by extending its neighbors' *core object career*. We again use the same methodology introduced above to update the life spans of these connections.

Handling Expirations. By using the lifespan analysis technique introduced above, the impact on the *skeletal grid cells* that could be caused by expiring objects has been pre-handled when objects arrive. Therefore, no maintenance effort is needed for handling cluster structure changes when individual objects expire. After the window slides, the only update needed for the attributes of *skeletal grid cells* is to check whether the new window is out of its lifespans. If the new window is out of its *core cell lifespan*, its *status* needs to be set back to *edge cell*. If the new window is out of the lifespan of any of its connections, the corresponding connection needs to be set back to "false".

Output Stage. At the output stage, the updated *skeletal grid cells* can be viewed as the vertices V in a graph G, and the connections among them can be viewed as the edges E among the vertices. Therefore, we simply conduct a depth first search on all the *core cells* to collect different groups of connected *core cells* and the *edge cells* attached to them. Each connected group of *skeletal grid cells* constitutes the SGS summarization of a cluster C_i, $C_i.SGS$. Given $C_i.SGS$, the *full representation* of C_i can be easily figured out by collecting all objects covered by *core cells* in $C_i.SGS$ and those covered by the *edge cells* in $C_i.SGS$ and connected to at least one *core object* in $C_i.SGS's$ *core cells*.

20.2 Extracting and Summarizing Distance-Based Outliers

Given the simple pattern structure of distance-based outliers, the summarization process as discussed in Chapter 19.2 is much simpler and computationally cheaper than summarizing density-based clusters. In particular, if we choose to view each individual outlier as a pattern, then no summarization process is necessary, as each outlier is simply a stream object. In this case, we can simply use the same Abstract-C algorithm that we proposed in Chapter 6 to extract the outliers. If we choose to view the outliers extracted from a same query window as a pattern and would like to further summarizae them, we can group them again by their similarities. To do so, we can use the same Abstract-C algorithm to extract the outliers, and use the grid index (same as we used in C-SGS algorithm for density-based clustering) to support the neighbor search processes in Abstract-C. Similar to SGS, the idea is to summarize the outliers in the same areas using grid cells. More specifically, we can use each grid cell to represent the outliers falling into it, or using a set of adjacent grid cells to represent all the outliers falling into the data space covered by them. This method has very little overhead compared to executing Abstract-C alone. This is because the only extra computation effort needed is to locate and connect (if needed) the grid cells containing outliers. These processes can be very efficient. The locating process can be almost free, as we can register each "outlier containning grid cell" when each outlier is discovered. The connecting process needs a depth first search on all the "outlier containning" grids, which needs N_{ocgc}^2 computation time, with N_{ocgc} the number of "outlier containning grid cells". However, as N_{ocgc} tends to be small, if not trivial, compared to the total number of grid cells covering the data space, this cost is again very limited.

20.3 Extracting and Summarizing kNN

Similar as density-based outliers, kNN has simple pattern structures. Thus, the same extraction+summarization two-phase strategy can be used to solve the extracting and summaring problem for kNN. In particular, we can run our proposed kNN extraction algorithm, MinTopk, (see Chapter 7.1) to extract the kNN, and also use grid index to hold all the valid objects in the window. Then, if we need further summarization for the k nearest neighbors that have been found, we can summarize them using the grid cells as we discussed above in Chapter 20.2.

Chapter 21

Pattern Archiver

In our system, the pattern archiver handles two major tasks, namely pattern compression and selective pattern archival.

21.1 Pattern Compression

First, depending on the system-resource budget and the specific analytical tasks, the pattern archiver controls at which resolution the patterns will be archived into the Pattern Base. Based on the output from the Pattern Extractor, we have three options, namely archiving clusters in the full representation or in the summarized representation or both. Beyond that, to allow more flexibility in the balance of system-resource budget and accuracy of pattern analysis, the Pattern Achiver can further compress the SGS of clusters provided by the Pattern Extractor.

21.1.1 Cluster Summarization in Multi-Resolutions.

Our proposed cluster summarization SGS of clusters supports multiple resolutions. In general, the SGS in different levels of resolution follows the design as presented in Chapter 19.1. In particular, an SGS of any resolution is composed of a sequence of *skeletal grid cells*, and each *skeletal grid cell* has the same 5 attributes, namely the *location vector*, *side length*, *population*, *status*, and *connection vector*.

For any cluster C_x, the SGS of C_x formed in the Pattern Extractor is based on the finest granularity, namely the smallest *skeletal grids cells*. Thus it is of the highest resolution. We call such SGS the "*Basic* SGS" of C_x. The *Basic SGS* of C_x is the base to form the compressed SGS at C_x in lower resolutions. The SGS in lower resolutions are built based on hierarchically compressing the *Basic* SGS. For a cluster C_x, we say that the *Basic* SGS of C_x is at the *Level 0* of the resolution hierarchy, noted as $C_x.SGS^{L_0}$. Any SGS in a lower resolution is at a higher *Level* n, where $n > 0$ and n increases as the resolution decreases, noted as $C_x.SGS^{L_n}$.

Each *skeletal grid cell* in $C_x.SGS^{L_n}$ $(n > 0)$, $C_x.Sc_i^{L_n}$, is formed by combining the *skeletal grid cells* within a certain (θ) sized hypercube space in $C_x.SGS^{L_{n-1}}$. As an example shown in Figure 19.1, a 2-dimensional cluster C_x has SGS in three resolutions. They are at *Levels 0, 1 and 2*. In this example, $\theta = 2$, indicating that each *skeletal grid cell* of SGS at *Levels 1 or 2* is made by combining 2×2 adjacent *skeletal grid cells* at *Levels 0 or 1* respectively. Both the number of resolutions allowed and parameter (θ) are part of the configuration of our system.

This compression process of building $C_x.SGS^{L_n}$ can be processed with in a single scan to the *skeletal grids* in $C_x.SGS^{L_{n-1}}$. In particular, given $C_x.SGS^{L_{n-1}}$ and to build $C_x.SGS^{L_n}$, we first generate a sequence of *skeletal grid cells* for $C_x.SGS^{L_n}$ to cover the whole data space occupied by *skeletal grids* in the $C_x.SGS^{L_{n-1}}$. Then we set the five attributes for any $C_x.Sc_i^{L_n}$ based on the *skeletal grid cells* covered by it at *Level n-1*. The side length of any $C_x.SG_i^{L_n}$ is simply equal to the side length of a *skeletal grid cell* at *Level n-1* times θ. The other three attributes, namely the *status*, *population* and *connections* of a $C_x.Sc_i^{L_n}$ is decided by the $C_x.Sc^{L_{n-1}}s$ covered by it. In particular, any $C_x.Sc_i^{L_n}$ is a *core cell* if at least one $C_x.Sc_i^{L_{n-1}}$ covered by it is a *core cell*. Otherwise, it is an *edge cell*. The population of any $C_x.Sc_i^{L_n}$ is equal to the sum of the population of the $C_x.SG^{L_{n-1}}s$ covered by it The connection vector of a $C_x.Sc0_i^{L_n}$ is decided by the connections between the "boundary" $C_x.Sc^{L_{n-1}}s$ covered by it and those covered by its adjacent $C_x.SG^{L_n}$s. More precisely, a $C_x.SG_i^{L_n}$ is *connected* to an adjacent *skeletal grid* $C_x.Sc_j^{L_n}$, if at least one pair of $C_x.Sc^{L_{n-1}}$s covered by them respectively are *connected*.

21.1.2 Distance-Based Outlier Summarization in Multi-Resolutions

For distance-based outliers, if we use the clustering method to further summarize the extracted outliers, we can use hierarchical clustering methods, such as single-link clustering method or Birch [JMF99], to balance the amount of information needed for storing the outliers and the accuracy of outlier expression.

In particular, if we use the Centroid + Radius+ Density (CRD) to express each cluster formed by outliers, the information needed for storing each cluster is fixed, and the key factor that determines the amount of information needed for expressing the outliers is the number of clusters formed in the detected outlier set. To achieve more compact pattern storage, the hierarchical clustering algorithms can incrementally merge the clusters and thus form less and less clusters. In general, the different numbers of clusters formed based on detected outliers constitute the distance-based outlier summarization in multi-resolutions.

21.1.3 kNN Summarization in Multi-Resolutions.

For kNN, the same multi-resolution summarization principle can be applied. In particular, no matter we adopt histogram or clustering methods to summarize the kNN of an object, we can also pursue high compactness of kNN expression by using large granularities in the pattern summarization.

In particular, if we adopted histogram based summarization, we can pick histogram bins with different sizes for different summarization resolution. If we adopted clustering methods for summarization purpose, the hierarchical clustering algorithms can again provide us summarization at different resolution.

21.2 Budget- and Accuracy-Aware Resolution Selection.

Given the multiple resolution choices, the Pattern Archiver can decide in which resolution to archive the patterns based on both the system-resource budget and the accuracy required by the specific analytical tasks. Our proposed multiple resolution summarization methods for neighbor-based patterns provide perfect support for such decision making. This is because for a pattern summarization at certain resolution, both its space consumption and conciseness are deterministic and easily calculable.

Taking density-based clusters as example, for space consumption, given the basic SGS of a cluster extracted from the Pattern Extractor, we can easily determine the number of *skeletal grid cells* needed in any other resolution for the same cluster, by calculating how many *skeletal grid cells* at the certain resolution level is needed to cover the same data space. Since the SGS in different resolutions have the same design, the amount of information carried by each *skeletal grid cell* in any resolution is fixed and foreknown. Thus, we can easily determine how much storage space is needed exactly for this cluster if using a certain resolution. For accuracy, as the side length of the *skeletal grid cells* at all resolutions are foreknown, the analysts knows exactly the granularity that their analytical task will be working on, if picking a certain resolution.

In this work, while we propose to provide such functionalities of efficiently summarizing density-based clusters into different resolutions, the specific methods of deciding in which resolution to archive a given cluster is not the focus of this work. Instead, I leave this question for my future work.

21.3 Selective Pattern Archiving

The Pattern Archiver also selectively picks which clusters to archive. Currently, our system supports several simple but useful cluster selection mechanism, including using sampling techniques to select certain numbers of clusters to archive in a period of time and using feature selection to only archive clusters with certain features (e.g. only archive the clusters reaching a certain population or volume). More sophisticated pattern selection techniques, such as evolution driven techniques, will be studied in our future work.

Chapter 22

Pattern Storage and Match

In this Chapter, we discuss the storage and matching of neighbor-based patterns in our system.

22.1 Storage and Matching for Density-Based Clusters

22.1.1 Cluster Organization in Pattern Base

Our proposed cluster summarization method SGS empowers us to easily organize the extracted clusters based on their features. In particular, we build two indices for the archived clusters. One is based on the position of each cluster, and the second is based on all other features of each cluster captured in SGS.

We call the first index the *locational feature index*. As multi-dimensional objects, we express the position of each cluster using its minimum bounding rectangle (MBR). In our system, we employ one of the most widely used indices for MBRs, namely the R-tree index to organize them. The second index, called the *non-locational feature index*, organizes the clusters based on their non-locational features. We use a four-dimensional grid index to organize the clusters' SGS, with the four dimensions: the volume (number of *skeletal grid cells*, the status count (number of *core cells*), the average density and the average connectivity of each cluster.

22.1.2 Cluster Matching Process

The *Cluster Matching Queries* (see Figure 18.2) are executed by the Pattern Analyzer. To execute such queries, we first provide a distance metric (between 0-1) to measure the distance between two clusters. The metric is user-customizable based on application semantics.

$$Dist(C_a, C_b) = ps * Dist_{location} + \sum w_i * Dist_{nlf\_i}(C_a, C_b)$$

$$ps, Dist_{location} = 0 \| 1, \forall w_i, Dist_{nlf\_i} = [0, 1], \sum w_i = 1)$$

In this distance metric, $Dist_{location}$ indicates whether two clusters overlap (1) or not (0). ps indicates whether the matching is "position-sensitive" (1) or not (0). $Dist_{nlf_i}$ represents the distance of two clusters on a specific non-locational feature and w_i represents the analyst-specified weight on this feature.

To use this distance metric, the analyst needs to first specify whether the matching required by her application is position-sensitive, namely whether the matched clusters have to overlap in the data space. For the position-sensitive applications, we set $ps = 1$. If two clusters do not overlap, $Dist_{location}(C_a, C_b) = 1$, the largest possible distance between two clusters, indicating that the two clusters are not similar and no further comparison on other features will be needed. For the non-position-sensitive applications, since $ps = 0$, the locational distance between two clusters is considered to be 0.

The second part of the distance metric measures the distance between two clusters on the four non-locational features, namely volume, status, population and connectivity. The distance on these features are used in both the match candidate search and detailed cell level cluster match.

Candidate Search. Given a to-be-matched cluster, a customized distance metric and a distance threshold specified by the analyst, our system first searches the potential match candidates in the Pattern Base. In the positional-sensitive case, the Pattern Analyzer first searches the *locational feature index* for the candidate clusters. If any overlapped clusters are found, it will calculate their non-locational distance with the to-be-matched clusters, and returns similar clusters if their distances are smaller than the given threshold. In the non-position-sensitive case, the Pattern Analyzer directly searches the non-locational feature index for the candidates. Given the distance metric and the distance threshold, the Pattern Analyzer can determine the range of the search on each dimension (feature). For example, given the volume of the to-be-matched cluster equal to 20, the weight on size distance is 0.20, the overall distance threshold is 0.2, the volume of the candidate clusters have to be between 14 and 30. This is because any other number $x < 14 \| x > 30$ will make $abs(x - 20)/min(x, 20) > (0.2/0.4)$, which will definitely not fulfill the search criteria. The same principle can be used on other features to determine the range of search. Given the search ranges on all dimensions, the Pattern Analyzer can quickly narrow down the candidate clusters to a small subset by searching the feature index.

Grid Cell Level Cluster Match. Given a to-be-matched cluster and a match candidate cluster for it, grid cell level cluster match compares the features of two clusters in their corresponding sub-regions (skeletal grid cells). In particular, grid cell level match uses the same customizable distance metric introduced earlier, while the distance between two clusters is now measured by aggregating the differences between all the corresponding *skeletal grid cell* pairs in these two clusters. More precisely, given a certain alignment between two clusters C_a and C_b, [1] each *skeletal grid cell* Sc_i in C_a may have a corresponding *skeletal grid cell* in Sc_j, depending on whether its corresponding sub-region is also covered by Sc_j. If Sc_i has a corresponding *skeletal grid cell* Sc_j in C_b, their difference can be measured by comparing their status, density and connectivity features. Otherwise, Sc_i is assigned the maximum difference with its corresponding sub-region, which is not a part of C_b and thus can viewed as an empty grid. When calculating the distance between two clusters C_a and C_b. we sum the difference between each Sc_i in C_a and its corresponding sub-region in C_b to form the overall distance between the two clusters.

In the position-sensitive cases, no alignment is needed, or in other words, the alignment vector is always equal to [0,0,...,0]. This is because such applications require any *skeletal grid cell* Sc_i in C_a to be matched with the *skeletal grid cell* Sc_j in C_b that have the same absolute position in the data space.

[1] An alignment for two Skeletal Grid Summarizations (SGS) is a location shifting vector. For example, given two three dimensional clusters C_a and C_b, an alignment equal to [1,2,1] indicates that any *skeletal grid cell* in C_a with location vector equal to [x,y,z] corresponds to a *skeletal grid cell* in C_b with location vector equal to [x+1,y+2,z+1], if any.

Therefore in such cases, we only need a single scan of the *skeletal grid cells* in two clusters to calculate the distances between them.

In the non-position-sensitive case, one or more alignments that minimize the distance between two clusters may exist. When given sufficient computation time, such as in an offline computation, one could apply an exhaustive search to find such an optimal alignment. In our system, for online computation, we use an A* style anytime search algorithm to search for the best alignment within a certain computation time budget. In particular, we start with an alignment that makes two clusters well overlapped. Then we continuously search along the direction of the most promising "nearby" alignment, which gives the smallest distance so far. When the given computation time budget is reached, we stop searching and return the smallest distance found so far as the distance between the two clusters.

22.2 Storage and Matching for Distance-Based Outliers and kNN

For distance-based outliers, the straightforward matching process is to match two individual outliers. Since each individual outlier is composed by a single stream tuple, such matching process is equivalent to comparing two individual tuples. Any distance function measuring the distance between two multi-dimensional objects, such as well-known Euclidean Distance or Manhattan Distance, can be used to match two outliers. To support such matching, we can simply organize the individual outliers using a standard multi-dimensional index, such as multi-dimensional grid index or R-tree index.

A more interesting outlier matching scenario is to match two outlier sets detected on different time points. Such matching process can reveal how the distribution of outliers in the stream changes over time. The matching process can be conducted directly on the two outlier sets using the subset matching algorithm presented in our work [YRW07]. This algorithm takes two datasets and a distance function measuring the distances between any two objects in the datasets as input. As output, it returns the distance (similarity) between two datasets. In this case, we can organize the outlier sets based on some simple statistics of them, such as the number of outliers in each outlier set.

Also, the outlier sets matching process can be conducted on the two summarized outlier sets. In particular, if we use a set of clusters in their CRD expression (as we discussed earlier) to present each cluster set, the problem of matching two outlier sets becomes equivalent to matching two cluster sets. This problem can also be solved by using the subset matching algorithm mentioned above. Namely, we can treat each cluster set as a dataset, and each cluster in the cluster set as a multi-dimensional object in the detest. The CRD expression of each cluster, namely the position of its centroid, radius and density constitute the values on the different dimensions of its corresponding objects. Then, similar as density-based cluster matching, we can allow analysts to specify a distance function with customized weight on each dimension to measuring distance between any two clusters' CRD expression. The two cluster sets and the customized distance function will be used as the distance function in the subset matching algorithm.

The matching process of the kNNs of two objects can be handled similarly as distance-based outliers. Namely, we can either match them as two datasets directly or summarize them first using histogram or clustering methods and match the corresponding histograms or cluster sets.

Chapter 23

Experimental Study

23.1 Experimental Platform

We conducted our experiments on a Dell desktop with an Intel Core2 2.2GHz processor and 3GB memory, which runs Windows 7 professional. We implemented the algorithms in VC++ 7.0.

23.2 Real and Synthetic Streaming Datasets

We use the same real streaming datasets, GMTI [EFK99] and STT [INE], as we used for experiments in Part I and II.

23.3 Alternative Summarization Formats

In our experiment, we compare both the effectiveness and efficiency of our proposed Skeletal Grid Summarization with three alternative cluster summarization formats. 1) The traditional "Centroid-Radius-Density" summarization (CRD). 2) Random sampling summarization (RSP). RSP for each cluster is generated by sampling the cluster members at a certain sampling rate R. To compare RSP with our proposed SGS summarization, for each specific cluster in the experiment, R is always set to make its RSP have the same memory consumption with the SGS for the same cluster. 3) Skeletal Point Set (SkPS) summarization (See Chapter 19.1).

23.4 Performance of Cluster Extraction and Summarization

First, we evaluate the performance of the alternative cluster summarization in terms of how many system resources are needed to generate them respectively. In particular, since our proposed solution, C-SGS, incorporates the cluster extraction and summarization in a single process, we compare its performance with the following alternatives. 1) Extra-N: Extract clusters using state-of-the-art algorithm Extra-N [YRW09] only but do not generate any cluster summarization. 2) Extra-N + CRD: Extract clusters using Extra-N first and then generate CRD for each extracted cluster. 3) Extra-N + RSP: Extract clusters using Extra-N first and then generate RSP for each extracted cluster. 4) Extra-N + SkPS: Extract clusters using Extra-N algorithm and then generate (approximated) SkPS for each cluster using MG algorithm proposed in [GK96]. [1].

[1]The problem of generating exact SkPS has been shown to be NP-complete (Chapter 19.1), and thus clearly not to be affordable in streaming environments.

We first run each alternative method against the STT stream to extract clusters based on four dimensions, namely the transaction type (buy/sell), price, volume and time. To compare the performance of the alternatives when handling clusters with different characteristics, we use three different query parameter settings, namely case 1: ($\theta^r = 0.05$, $\theta^c = 10$), case 2: ($\theta^r = 0.1$, $\theta^c = 8$), case 3: ($\theta^r = 0.2$, $\theta^c = 5$). Also, for each case, we use three different window parameter settings, namely we fix the window size (win) for all three settings at 10K tuples, while letting slide size $slide$ equal to 0.1, 1K and 5K tuples respectively for three settings.

For each case, we first verify the **correctness** of our proposed C-SGS cluster extraction method by comparing the clusters extracted by it in full representation with those extracted by the state-of-the art technique Extra-N. In all the test cases, we found that the clusters extracted by C-SGS are identical with those extracted by Extra-N.

For **efficiency** evaluation, we measure two major performance metrics for stream processing: 1) The average response time for each window (denoted as Response Time). For each window, we measure the average CPU time elapsed from the time when the new data arrive until the time when all clusters are output and cluster summarization is generated. The average processing time shown in all cases is all averaged among runs for 10K windows. 2) The memory footprint, namely the peak memory utilization of each alternative, among the 10K windows.

As shown in Figure 23.1, compared to Extra-N, which extracts clusters only but does not generate any cluster summarization (the baseline case), the other four alternatives, each generating a specific type of cluster summarization, exhibit some overheads in terms of CPU time utilization. However, the overhead caused by C-SGS, Extra-N + CRD, and Extra-N + RSP, is very modest, if not neglectable. The reason for such modest overhead caused by Extra-N + CRD and Extra-N + RSP is obvious. This is because CRD and RSP are both very simple summarization formats that can easily be generated by at most two scans of the cluster members of each cluster. The overhead caused by our proposed solution C-SGS is comparable with those two simple summarization methods. This is because the major computation needed for generating the SGS cluster summarization, namely determining the status and connection among *skeletal grid cells*, is elegantly piggy-backed by the cluster extraction process itself in our C-SGS method. The CPU overhead of Extra-N + SkPS is significantly higher than that of the other alternatives. This is because generating SkPS is very expensive computationally [GK96]. For different window parameter settings, C-SGS has lower overhead in the settings that have a larger $win/slide$ rate. This is because the performance of Extra-N is affected by the increasing $win/slide$ rate [YRW09], while the performance of C-SGS is not sensitive to this ratio.

Memory-wise, the overhead caused by Extra-N + SkPS is also significant, while those caused by other alternatives are very modest (Figure 23.1). For Extra-N + SkPS, since it requires a cluster to be expressed as a graph when it is output for the SkPS summarization generation, a large amount of extra memory space is needed to store the connections among objects in both cluster extraction and summarization generation stages. For Extra-N + RSP and Extra-N + CRD, their little overheads on memory space are expected. This is because they almost do not need extra memory space to compute their very simple cluster summarization formats. For our proposed method C-SGS, low memory overhead is also expected, because the process of generating SGS happen in place with the clustering process. It only needs to maintain a very limited amount of extra meta-data, mainly the connections between *skeletal grid cells*, which can be efficiently expressed as bit strings.

The overall CPU and memory costs for all alternatives increase in the test cases with larger θ^r and smaller θ^c. This is because such parameter settings tend to let a query identify more "connections" (neighbor relationships) among the objects, which in general consumes more system resources. As expected, they tend to form larger clusters in terms of both population and volumes. In our experiment, the largest clusters (population-wise) identified in the three test cases are composed of around 0.5K, 3.2K and 9.2K objects. This increases the cost of the clustering process itself, and the extracted clusters with larger population and spread will also cause more resource utilization for generating summarization formats for them.

Similar performances are also observed in our experiments using GMTI data. As shown in Figure 23.2, our proposed method C-SGS has very modest overheads in both CPU and memory utilization compared to the baseline.

Evaluation for Time-Based Windows. To further evaluate the performance of the alternative methods using time-based windows and under fluctuate input rates, we conduct the following experiments. In particular, we run three queries using the same pattern parameter settings (test cases 1-3) as used in previous experiments against both STT and GMTI

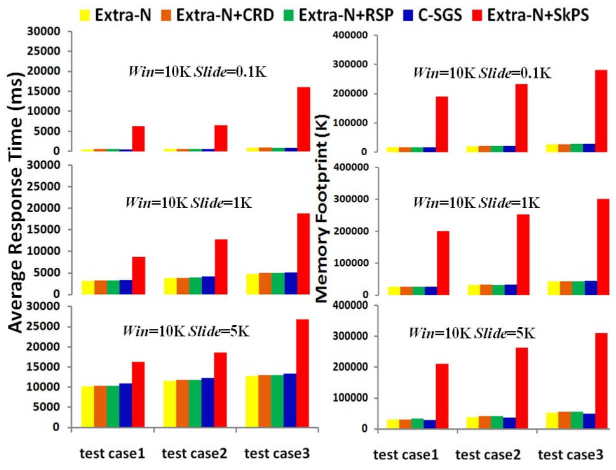

Figure 23.1: CPU Time and Memory Comparison for Generating Alternative Summarizations on STT Stream

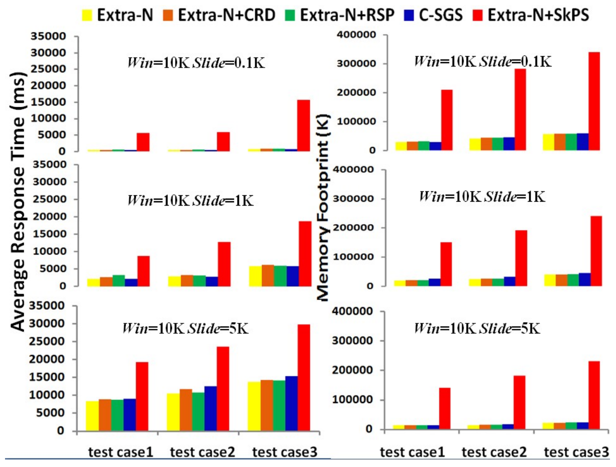

Figure 23.2: CPU Time and Memory Comparison for Generating Alternative Summarizations on GMTI Stream

streams but now give them a time-based query window ($Win = 5min$, $Slide = 0.5min$). In these experiments, we measure not only the average response time of all windows but also the lowest and the highest response time for each window during the query execution.

For the STT stream, the average data rate is around 50 tuples per second, and the minimum and maximum data rates are 6 and 188 tuples per second respectively. Thus, within a 5 min query window, the average number of objects valid in the window is around 15K. As we are using 0.5 min as slide size, the average number of objects for each window slide is around 1.5K. The minimum and maximum number of objects in each window slide are 0.68K and 2.9K respectively.

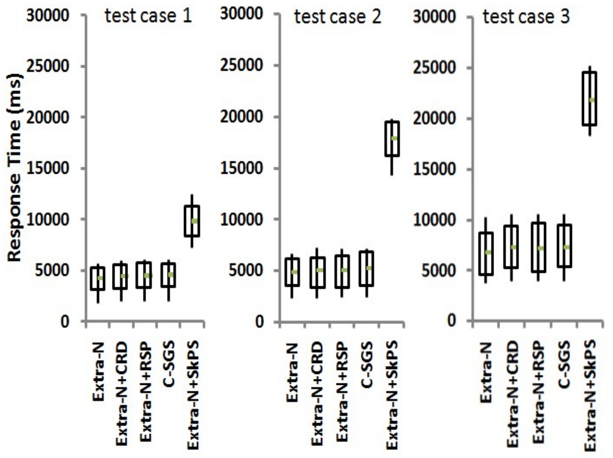

Figure 23.3: CPU Time Comparison Using Time-Based Window (STT Stream)

As shown in Figure 23.3, similar performance of the alternatives can be observed compared with previous experiments using count-based query windows. Namely, the overhead of our proposed method C-SGS compared to the baseline (Extra-N alone) is very modest in terms of the lowest, the highest and the average response time. For average response time, the overheads of C-SGS are only 6%, 9% and 7% in three test cases respectively compared to Extra-N, which are again comparable to generating other two simple summarization formats, namely CRD and RSP. The overhead of Extra-N + SkPS is significant higher than that of the others.

In our experiment, the fluctuate input rate shows a strong but similar impact on four alternatives, namely Extra-N, Extra-N + CRD and Extra-N + RSP and C-SGS. For the first three Extra-N algorithm based alternatives, large variations on their response time are caused by the different execution times of the Extra-N algorithm when very different number of objects arrive at each window slide. This is because the Extra-N algorithm execution constitutes the key computation costs for those method, and the followed summarization step is simple and thus cost little CPU time. For C-SGS, the variation on its response time is caused by the similar reason, as its computation costs are also decided by the number of objects arriving at each window. For Extra-N+ SkPS, the variations in response times are also similar to those of the other alternatives in terms of absolute amount but relatively insignificant. This is because, in Extra-N + SkPS, a large percentage of CPU time is used for computing SkPS (61%, 72% 69% respectively for test cases 1, 2 and 3), which is related to the characteristics of the clusters extracted but independent from the input rate.

As shown in Figure 23.4, similar performance of the alternatives is also observed when running the same queries against the GMTI stream. In the GMTI stream, the average data rate is around 10 tuples per second, and the minimum and maximum data rates are 0 and 32 tuples per second respectively. Within the 5 min query window, the average number of objects valid in the window is around 3K. The average number of objects arriving at each window slide is around 0.3K. The minimum and maximum number of objects arriving at each window slide are 0.1K and 0.8K respectively.

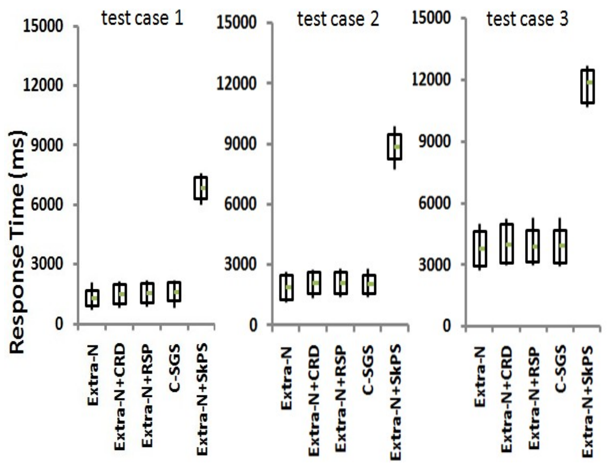

Figure 23.4: CPU Time Comparison using Time-Based Window (GMTI Stream)

In conclusion, using our proposed C-SGS solution, we can efficiently generate the Skeletal Grid Summarization (SGS) for extracted clusters during online clustering process, with very limited system resource overhead. Its performance is comparable with generating very simple cluster summarization formats, namely CRD and RSP. The system overhead needed for generating SkPS is significantly higher than those of the other alternatives in terms of both CPU and memory utilization. We run the same experiments on the GMTI stream as well. Similar performances are observed for all the alternatives, which confirm the conclusions that we have drawn above.

23.5 Effectiveness for Cluster Matching Queries

Now we study the effectiveness of the alternative cluster summarization formats in terms of matching clusters through a user study. Namely, we evaluate whether our proposed summarization format SGS and other alternatives can provide good quality in terms of cluster matching. Before our experiment, we remind the readers that our proposed cluster summarization format SGS guarantees to preserve the four key features of each density-based cluster (Chapter 19.1), while none of the other alternatives achieves this. Also, our proposed cluster matching mechanism for SGS (Chapter 22) takes all four features into consideration when deciding on the similarity between clusters. This indicates that similar clusters found using our proposed SGS summariztion and the corresponding similarity metrics are guaranteed to be similar on these four features to a certain level, depending on the similarity threshold setting.

To confirm the superiority of our proposed solution against other cluster summarization formats in terms of cluster matching quality, we ask human analyst to visually analyze the similarity between the matched clusters found using our method and other alternatives. The reasons why we use human analysts to visually confirm the similarity between clusters are: 1) Since matching density-based clusters is still an open research problem, there does not exist any proven analytical metric in the literature, which can be used as the "golden criteria" for measuring cluster similarity. 2) The capability of human analysts to visually observe and compare data patterns has been proven by the visualization community [War94, BW96], especially when proper visualization tools are provided.

In particular, we run the three queries as we used earlier for performance analysis against the STT data using our proposed C-SGS method. During the online clustering process of each query, we archive all the clusters detected in the stream into the Pattern Base until the number of clusters in the pattern base reaches 1K. Also, for each archived cluster, we generate and keep the other three alternative cluster summarization formats, namely RCD, RSP and SkPS. The full representation (cluster member tuples) of each archived cluster is also kept for later visual analysis.

Then, we stop cluster archiving but keep the clustering process running, and randomly pick 10 new clusters detected as the to-be-matched cluster for each query. For each of the to-be-matched cluster, we run four cluster matching queries for it, using one alternative cluster summarization format. In particular, we implement a subtraction function to measure the distance between the CRD of two clusters, which gives equal weight to the three cluster features captured in CRD, namely the centriod, range and density. We use the subset matching algorithm presented in [YRW07] to calculate the distance between the SkPS of two clusters. We use the graph edit distance algorithm presented in [NRB06] to calculate the distance between the SkPS of two clusters. We give equal weight to all four features when measuring the distance between the SGS of two clusters. For all the alternative summarization methods, we take the top three similar clusters (with smallest distances to the to-be-matched cluster) found by each method for evaluation. We call them the “matched clusters”.

To measure the quality of cluster matching, we invite 20 human analysts, all graduate students at WPI, to visually compare each to-be-matched cluster C_i with all matched clusters returned by four alternative summarization formats. The analysts are asked to equally consider all four key features of the density-based clusters, namely the position, shape, density distribution and connectivity. The visual analysis process of the analysts is supported by ViStream [YGX+10], a freeware multivariate data visualization tool, which has been shown to be effective in helping human analysts to observe and understand mutil-dimensional clusters in streaming environments. For each to-be-matched cluster, the analysts are asked to rate all the matched clusters found for it into three categories, namely “very similar”, “similar”, “not similar”. The analysts are not told which “matched” clusters are found using which cluster summarization format.

As shown in Figure 23.5, the analysts rated a high percentage of matched clusters found using our proposed solution SGS as “very similar” or “similar”, while the three alternatives have significantly lower rate in these two categories. In particular, given the total 90 matched clusters found for the 30 to-be-matched clusters, the 20 analysts gave $20 \times 90 = 1800$ ratings in total. Within these ratings, 486 (27%) of them are “very similar”, 972 of them are (54%) of them are “similar” and only 19% are “not similar”. Within the other alternatives, the one received the highest “similar rate” is the SkPS, which received 12% “very similar”, 25% “similar” and 65% “not similar”. The other two alternatives, namely CRD and RSP, got even lower ratings.

By analyzing the cases in which “not similar” ratings are given, we found that the majority of such cases for our SGS method happened on very large in population but sparsely distributed clusters. Namely, when the clusters are spread to very large areas, usually in more than 100 *skeletal grid cells*, the matched clusters may not be recognized as similar by visual analysis. This is because, when the SGS are composed of many *skeletal grid cells*, our distance metric may decide that the distance between two clusters is small, because they are similar in a large percentage of the *skeletal grid cells*. However, the difference between two clusters on those "not similar" *skeletal grid cells* may appear to be significant for human analysts visually, as the absolute numbers of them are no longer trivial compared to the compact clusters. However, such cases are found to be not common in our experiments. The lower ratings of simple summarization formats CRD and RSP are expected, because they are only capable of capturing the difference between clusters on simple statistical features. The ineffectiveness of SkPS is also expected. As our analysis in Chapter 19.1.2 reveals, the key problem of SkPS for cluster matching lies in its undeterminancy and its lack of ability for expressing density distributions.

In conclusion, our proposed cluster summarization demonstrates good effectiveness for cluster matching queries in our experiments, while none of the other alternatives achieves comparable effectiveness for cluster matching.

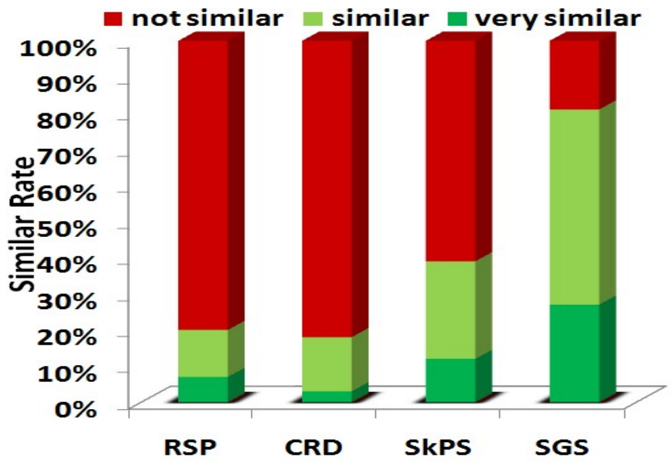

Figure 23.5: Similar Rating by Users for Matched Clusters found by Alternative Summarization Methods

23.6 Efficiency of Cluster Matching Queries

Now we study the efficiency processing cluster matching queries in our system. In particular, we measure the resource utilization for running the cluster matching queries using our proposed summarization SGS, and compare it with running cluster matching queries for the same to-be-matched clusters but using an alternative cluster summarization. Same as in the previous experiment for effectiveness, we run the same three queries against the STT data using our proposed C-SGS method. To measure the performance of matching queries against the Pattern Base with different sizes, for each query, we evaluate three test cases. Namely, the number of clusters in the Pattern Base equal to 0.1K, 1K and 10K respectively. In each test case, we run a clustering query and archive all the clusters detected into the Pattern Base until the size of the pattern base reaches the required number. Also, for each archived cluster, we generate and keep the other three alternative cluster summarization formats for evaluating other matching method. Once the required number of clusters are archived, we stop archiving and randomly pick 100 clusters detected to conduct the cluster matching queries. Same as in the previous experiment, we run four matching queries for each to-be-matched cluster using one alternative cluster summarization method and the corresponding distance metric.

In this experiment, we measure the average response time for each cluster matching query and memory space consumed by storing cluster summarizations. To accurately measure the response time for the cluster matching queries, the online clustering process is ceased during the cluster matching in this experiment. Also, to be fair to other alternatives, a population filter are used for all of them to filter out the candidates with significant population difference with the to-be-matched clusters.

As shown in Figure 23.6, our proposed summarization SGS has the second lowest average response time among all alternative methods. In particular, when matching against 0.1K archived the clusters, the average response time for each cluster matching query using SGS is less than 0.1 second. For the 1K and 10K cases, the average response time for our solution is only around 0.5 seconds and 3 seconds. Such efficiency is comparable with cluster matching using CRD, which is very fast because of the simple matching mechanism (simply three subtraction operations, one on each feature). This thanks to the design of SGS, which effectively summarize the key features of each cluster on both cluster and grid levels. In particular, by using our proposed two-phase matching strategy, the majority of the candidates in the pattern base are filtered out in the cluster level statistic matching phase. Thus, the more expensive grid level matching is only needed for a very small portion of the candidates. In our experiment, we found that for each to-be-matched cluster only 6% of the candidate clusters necessitated the grid level match on average during the cluster matching process.

The average response time for the other two alternatives, namely RSP and SkPS, are significantly higher. This is because of the high complexity of the algorithms that are needed for matching these two summarization formats. In particular, the both algorithms for matching RSP and SkPS [GK96] have $O(n^3)$ CPU complexity, with n the number of objects in the connected dominant set and sampling set respectively.

Memory-wise, our proposed method SGS consumes only 0.12M, 1.38M and 12.24M memory space to store 0.1K, 1K and 10K clusters respectively (Figure 23.7). In particular, each 4-dimensional *skeletal grid cell* only consumes 23 bytes, position: 16 bytes (4 integers), status: 1 byte (1 boolean), density: 4 bytes (1 integer), connection: 2 bytes ($2^4 = 16$ booleans). In all test cases, the average number of *skeletal grid cells* in each cluster is 68. Therefore, only 1.5K memory is needed to store the SGS of each cluster on average. Compared with the memory space needed for storing the full representation of the clusters, which need 6.4M, 75.2M and 680.2M to store 0.1K, 1K and 10K clusters respectively, the average compression rate of SGS in our experiment is around 98%. Such compactness of SGS is comparable with SkPS, which consumes a similar amount of memory space for expressing each cluster. In our experiment, since we always set the sampling rate of RSP to make its memory consumption equal to that of SGS, their consumptions are always equal. As very simple summarization method, CRD has even lower memory consumption. However, its matching quality has been shown to be far below our proposed summarization in the previous experiment.

In conclusion, our proposed solution demonstrates high efficiency for cluster matching queries, which is significantly better than matching SkPS or RSP. Its performance is comparable with that of matching simple CRD cluster summarizations. However, as we have shown earlier in the previous experiment, CRD is not able to provide satisfactory matching quality. Therefore, our solution is the best alternative method in terms of both effectiveness and efficiency for cluster matching queries.

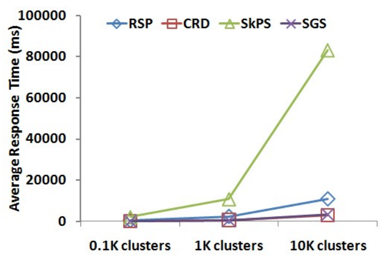

Figure 23.6: CPU Time Comparison for Cluster Matching Queries using Alternative Cluster Summarization Methods

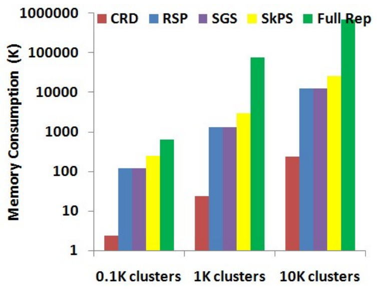

Figure 23.7: Memory Comparison for Cluster Matching Queries using Alternative Cluster Summarization Methods (in Logarithmic measure)

23.7 Evaluation for SGS at Multiple Resolutions

In this experiment, we evaluate the effectiveness and efficiency of our proposed SGS method, when lower resolution SGS are used for matching queries.

In particular, we use the same experimental methods applied in experiment 2 (Chapter 23.5) and 3 (Chapter 23.6), while now using the GMTI stream as data source. For cluster extraction, we also use 4 dimensions in GMTI stream as in the previous experiments, including the latitude and longitude of the objects. We pick 20 large-spread clusters to evaluate the quality and performance of the cluster matching queries for them when SGS of different resolutions are used. More precisely, we run the same three queries against the GMTI data, and archive all detected clusters until the pattern base reaches the size of 1K. Different from previous experiments, we run each query three times and each time we archive SGS of the detected clusters using SGS at different resolutions. Namely, besides basic SGS with highest resolution, in the other two runs, we archive cluster using "half-" and "quarter-" resolution SGS, whose Skeletal Grids' length on each dimension are two and four times to that of the basic SGS Skeletal Grids respectively. Then, we pick the same clusters detected in each run to conduct the cluster matching queries using these. The criteria of picking those clusters is that they have to reach certain volume. In particular, each of the selected to-be-matched cluster occupies at least 200 basic skeletal grid cells, which needs to be expressed by at least 13 "half" resolution Skeletal Grids and 2 "quarter" resolution Skeletal Grids. This is because, only in such cases, those lower resolution SGS may be necessary, otherwise the basic SGS is already very compact and efficient. For each test case, namely for SGS at each resolution, we compare both its matching quality and performance with other alternatives.

As shown in Figure 23.8, the performance of the matching queries using SGS increases as the resolution of SGS decreases. In particular, when using half and quarter resolution SGS, the average response time of each cluster matching query are 38% and 18% of that of using the basic SGS. Such performance gain is significant, while it is not proportional with the ratio between the number of *skeletal grid cells* in the SGS with different resolutions. In particular, for these 4 dimensional clustering queries, the half and quarter resolution SGS respectively only consume around 6% and 0.4% *skeletal grid cells* for each cluster compared with the basic SGS. However, the response time for matching clusters using these lower resolution SGS constitute higher percentages compared with such ratio. This is because, although the CPU time needed for grid level matching is directly decided by the number of Skeletal Grids in the cluster summarization, a large portion of the CPU time needed for the cluster matching query is consumed by feature index searching, which is independent from the number of *skeletal grid cells* in SGS. Also, the cluster level matching using lower resolution SGS tend be less selective. This explains why the response time gain is not proportional with the resolution ratio.

Compared to other alter naives, the average response time using half and quarter resolution SGS are better than SkPS and RPS in all test cases, and even comparable with CRD in the quarter resolution case. Since we pick the same to-be-matched clusters for all test cases, the performance and matching quality of CRD and SkPS in these cases are the same. Besides SGS, the only alternative that has a varying performance and matching quality in different test cases is RSP. This is because the sampling rate of RSP in each test case is specifically set to make its memory consumption equal to the corresponding SGS at a certain resolution. The response time for cluster matching using RSP also decreases.

Memory-wise, as shown in Figure 23.9, the amount of memory space needed for archiving the same number (1K) of clusters decreases significantly as the resolution of SGS decreases. Unlike the response time, such performance gain on memory space is proportional to the resolution ratio. This is because, the memory space needed by SGS is strictly proportional to its resolution, namely the number of Skeletal Grids in the SGS. In the half and quarter resolution cases, the memory space needed by SGS is even less than that needed by SkPS. In the quarter resolution cases, the memory consumption of SGS is even comparable with CRD. This is because in the quarter resolution case only very few, usually less than 5, Skeletal Grids are needed for expressing each cluster in SGS. Such very limited number of Skeletal Grids only needs a very modest amount of memory space.

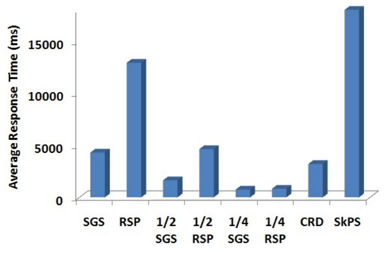

Figure 23.8: CPU Time Comparison for Cluster Matching Queries using Alternative Cluster Summarization Methods

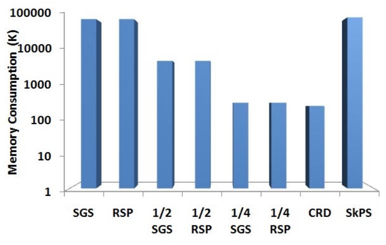

Figure 23.9: Memory Comparison for Cluster Matching Queries using Alternative Cluster Summarization Methods

As shown in Figure 23.10, the quality of cluster matching using SGS decreases as the resolution goes down. This is as expected, because the lower the resolution of SGS, the less descriptiveness SGS can provide to the cluster. In particular, in our experiment, using the finest resolution, SGS received a 76% "similar rate", including "similar" and "very similar" ratings. While when using half and quarter resolution SGS, such "similar rate" decreases to 62% and 29% respectively. Compared to other alternatives, the half resolution SGS still shows significantly better matching quality. Namely, its "similar rate" 62% is much higher than that of the second best alternative, SkPS 38% However, the quarter resolution SGS does not show significantly better matching quality to other alternatives. The "similar rate" received by quarter resolution SGS (29%) is lower than that of SkPS (38%). But in general, SGS is he best method among all alternatives in terms of cluster matching quality, especially when higher resolutions are adopted.

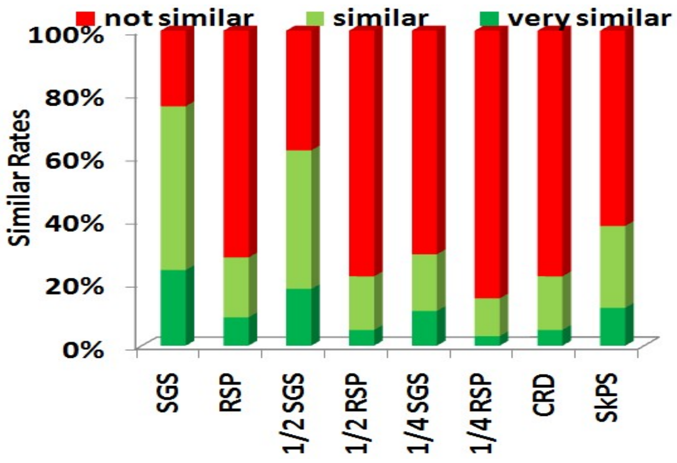

Figure 23.10: Similar Ratings by Users For Matched Clusters using Multiple Resolutions

In conclusion, our experiment shows that the demands of SGS in terms of system resource utilization decrease significantly as lower resolutions are used. This provides flexibility for the applications that needs different query response time and the systems that have different resource budget. As trade-off, the cluster matching quality provided by SGS also decreases as lower resolutions are used. However, given the same resource utilization, SGS consistently provides the best cluster matching quality among all alternatives in our experiment.

Chapter 24

Related Work for Part III

Pattern summarization is a general topic for database community. Many previous works have studied the problem of summarizing various kinds of pattern types in both static and streaming environments. For example, [NRS08, LDS11, LT10] study the problem of summarizing graphs; [AU11, LLL11, WL10] discuss the summarization of documents; [VvLS11, GRDG11] present techniques for summarizing frequent itemsets. Each category of these effective summarization mechanisms are designed based on deep understandings to the unique pattern structure of each pattern type.

However, the problem of summarizing neighbor-based pattern patterns has not been studied in the literature yet. Without an effective yet compact summarization method, each neighbor-based pattern has to be expressed by its full representation. For example, each density-based cluster has to be expressed by all its cluster member objects. Obviously, such full representation is neither succinct nor does it explicitly reflect the features of each pattern. It causes serious inconvenience for both storing and analysis of neighbor-based patterns, especially in streaming environments in which real-time responsiveness may be required.

As, we discussed earlier in this Chapter, density-based clusters have one of the most complex pattern structures in neighbor-based pattern family. Traditional clustering methods [HW, ZRL96], such as k-mean style clustering, usually treat clusters as statistical phenomena. Therefore, many key features of the clusters, such as their shapes and densities, are summarized using a rather simplistic description. In particular, first, these works assume clusters are spherically shaped. Therefore, the shape of a cluster is usually described using a simple "$centroid + radius$" formula. Second, the previous work do not capture the internal features of the clusters, such as how its density is distributed. For example, the density of the cluster is either treated as uniform or varying along the radius only. Obviously, such simple formula cannot well describe the complex cluster structure of density-based clusters. This is because the shapes of density-based clusters can be arbitrary and the objects within each cluster can be arbitrarily distributed as well, not to mention the complex sub-region connectivities in each cluster. To the best of our knowledge, no summarization method has been specifically designed for density-based clusters.

For computing cluster summarization in streaming environments, if the clusters are treated as statistical phenomena, they are considered to be "aggregatable" over time [AHWY03, DHYC06]. For example, the centroid of a cluster (the average of values on cluster member objects) is acquired by continuously aggregating the values of each new cluster member object) . In particular, [AHWY03] used one *cluster feature vector* to represent each micro-cluster detected in the stream. They rely on the *additivity property* of the *cluster feature vectors* to aggregate the cluster features over time and compare the features of a same cluster at different time points by subtracting its *cluster feature vectors* on corresponding time points.

However, the complex cluster structure of density-based clusters are not simply aggregatable over the sliding windows. The continuous expiration of old objects and arrival of new objects at each window may cause complex cluster structural changes, such as merge and split and connectivity changes within the clusters, which cannot be simply captured by aggregation results. Thus, these techniques cannot effectively capture the features of density-based clusters within sliding window scenario.

Pattern matching is also a general topic for database community [NRB06, MCH11, WFZZ10]. However, to our best knowledge, these does not exist any previous work studying the specific problems of matching neighbor-based patterns.

Chapter 25

Conclusions of This Dissertation

In this dissertation, I tackle the problem of mining and managing neighbor-based patterns in streaming environments. My disseration solves the problems in three major aspects of neighbor-based pattern mining and management, namely efficient neighbor-based pattern extraction, multiple query optimization for neighbor-based pattern mining queries and summarization and matching for neighbor-based patterns. It extends the traditional stream processing systems, which mainly focused on efficient processing of SPJ queries, to now have the capabilities of extracting and managing complex patterns in data streams.

In the first part of my dissertation work, I study the problem of efficient extraction of neighbor-based patterns from sliding windows over streaming data. I first identify that the major difficulty of incremental detection of the neighbor-based patterns exists in the handling of expired objects. For this reason, my two primitive incremental algorithms Exact-N and Abstract-C suffer from either massive CPU or memory consumption for detecting density-based clusters. Then, I design the third algorithm Abstract-M based on a proposed "view prediction" technique, which elegantly discounts the effect of expired data points from the patterns. Finally, the combination of the "view prediction" technique and a proposed hybrid *neighborship* maintenance mechanism leads to our proposed solution Extra-N, achieving both linear memory consumption and the minimum number of range query searches. Both my analytical and experimental studies confirm that: 1) My proposed algorithms Extra-N and Abstract-M are near-optimal in detecting density-based clusters over sliding windows in terms of CPU time, memory space and also scalability. 2) My proposed algorithm Abstract-C is a CPU- and memory-efficient algorithm for distance-based outlier detection in sliding windows. It clearly outperforms the only previous algorithm [AF07] when detecting outliers in time-based windows, while performing equivalently with it when dealing with count-based windows. 3) My proposed algorithm for kNN detection, MintTopk, by identifying and elegantly updating the minimal object set (MTK) that is necessary and sufficient for top-k monitoring, not only minimizes the memory utilization for executing top-k queries in sliding windows, but also achieves optimal CPU complexity when returning the top-k results in a ranked order.

In this second part of my disseration work, I present the first framework for the efficient shared processing of a large number of neighbor-based pattern mining requests over streaming windows. It is the first step of applying multiple query optimization principles from the field of databases to process large numbers of data mining requests in stream environments. I propose several general optimization principles that are applicable to different (at least three) neighbor-based pattern mining query types. Both my analytical and experimental studies show that these principles can bring significant system resource sharing among multiple queries. In particular, my proposed algorithms *Chandi*, *SDOD* and *SkNN*, which are based on these optimization principles, respectively achieve full sharing of both CPU and memory utilization when simultaneously executing multiple density-based clustering, distance-based outlier and kNN queries. My experimental study shows that, my proposed solution *Chandi* that handles the density-based clustering queries, which has the most complex pattern structure within neighbor-based pattern family, is on average four times faster than the best alternative method while using 85% less memory space. More savings can be achieved if the queries have similar parameter settings. *Chandi* also exhibits excellent scalability in terms of being able to handle large numbers of queries under high speed input streams in my experiments. My performance analysis for distance-based outlier and kNN queries shows that the similar performance can be expected from my proposed strategies for those two pattern types as well.

In the third part of my disseration work, I present a framework to support summarization and matching of neighbor-based patterns in streaming environments. First, my work solves several open problems for density-based cluster analysis, namely, designing a descriptive yet compact summarization method for such clusters. Second, I present an efficient computation strategy to quickly summarize the detected clusters into SGS during the online clustering. Lastly, I design a pattern archiving and matching mechanism, which allows the analysts to submit cluster matching queries to find similar pattern detected earlier in the stream history. My experimental study demonstrates the clear superiority of my proposed methods on both the efficiency and effectiveness for pattern summarization and matching to other alternative methods.

As I have concluded above, my dissertation does not only present specific techniques, but also propose several general optimization principles for neighbor-based pattern mining and management in streams. This includes the predicted view, meta query composition for multiple sliding window queries, incremental pattern maintenance across multiple queries, and etc.

Analysts in stream mining fields, such as financial analysts, traffic controllers and bank managers, can directly use our proposed techniques through our to-be-released freeware interactive stream mining system, ViStream [YGX$^+$10], by plugging in their own data sources. Researchers and engineers can further improve our proposed techniques by editing ViStream system, or puttting their own implementations to my proposed techniques together. Also, as many of my proposed optimization strategies are general, namely they are not restricted to neighbor-based patterns but applicable to other pattern types as well, researchers and engineers can design their customized mining strategies based on those proposed principles.

Chapter 26

Future Work

As an early effort in the database community to tackle the problem of mining and managing complex patterns in data streams, my dissertation work opens a broad area for future research in streaming data mining and management, as described below.

26.1 Efficient Pattern Extraction for Other Complex Pattern Types

Clearly, although the neighbor-based pattern family covers a group of important pattern types, there exist many other complex pattern types worth mining in streaming environments, such as mining graphs [BHPG11, AW10], text [WZJS09, WEC09] or association rules [LWC11]. Same as extracting neighbor-based patterns, the efficient extraction strategies of these complex patterns constitute the foundation for analyzing them in the streaming environments. Based on the characteristics of each pattern type, specific extraction algorithms may need to be designed to extract each type of them efficiently. However, the general optimization principles that I presented in this dissertation, such as the "predicted view" technique and the "integrated pattern representation" may be leveraged for assisting the optimization process for those queries. For example, when a pattern extraction query uses sliding window semantics and the object expiration constitute the major resource-consuming bottleneck for pattern extraction, the "predicted view" technique may be applied to efficiently handle the object expiration. In general, the neighbor-based pattern extraction techniques that I presented in this work is a good resource for optimization strategies for extracting other complex pattern types in data streams.

26.2 Pattern Evolution Model and Evolution Tracking

Pattern evolution is also an important future work direction for my dissertation. Due to the temporal characteristics of streaming environments, an important type of knowledge that the analysts would like to learn about the streams is how the patterns in the streams change over time. Such evolution knowledge may reveal a lot of valuable information for streaming applications. For example, in traffic monitoring applications, an analyst does not only need to know the major traffic congestions (clusters) just detected in the traffic streams, but also needs to keep track of how these clusters evolve over time. This is because the change of the congestions may be the key indicator of whether the congestion-relief strategy applied is effective or not.

To provide such knowledge to the analysts, one first needs to study and define evolution model for the target pattern type, which can effective describe the pattern changes over time. Such definition of a pattern evolution model takes careful analysis of both the target pattern structure and the specific analytical tasks. My work [YGRW11] on density-based cluster evolution in sliding windows provides ideas for modeling the evolution of complex patterns in sliding windows. In particular, a pattern evolution model can model both the 1-to-1 evolution, such as a single pattern preserves or expands in a certain period of time, and the 1-to-N (or N-to-1) evolution, such as a single pattern splits into several smaller patterns (or several smaller patterns merge to become a larger pattern). The similar pattern evolution semantics may also be applied to other complex pattern type.

After defining the evolution model, another important task for the evolution study is to efficiently track the pattern evolution in the stream. Depending on the specific pattern structures and evolution model, different evolution tracking algorithms can be designed. However, a general optimization strategy, which I used in [YGRW11] for tracking evolution of density-based clusters, may benefit such algorithm design. That is to integrate the pattern evolution tracking process within the incremental pattern extraction process, if the target patterns are incrementally maintainable. As the key task for both evolution tracking and incremental pattern extraction is to track the pattern structure changes over time, combining them into a single process may significantly save both the computational and storage resources, that otherwise may need to be spent twice for each of them.

26.3 Dynamic Stream Mining Queries

Another important future research direction for stream data mining is to study the optimization problem for dynamic stream mining queries. Unlike the static continuous queries studied in this work, which have fixed mining parameter settings and are assumed to be running from the start to the end of the applications, the dynamic stream mining queries may change their parameter settings during their execution and be added or removed from the mining system at any time. For this research topic, one needs to tackle the research problem in two major aspects. First, how to realize smooth query parameter migration. In particular, when the parameter settings of existing queries change, one needs to design query migration techniques to minimize the time needed for switching the query execution from the old parameter settings to the new parameter settings. Second, how to coordinate the relationships between all the mining queries in the system as queries come and go, especially when mining queries are executed in the shared manner as we discussed in the second part of my dissertation.

Those problems are challenging problems, while some of the optimization principles presented in my dissertation, such as maintaining proper abstraction format for pattern structures and using integrated representation for multiple queries, can be exploited as the starting point for designing their solutions.

Bibliography

[ABB$^+$03] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. Stream: The stanford stream data manager. In *SIGMOD Conference*, page 665, 2003.

[ABK$^+$06] Elke Achtert, Christian Böhm, Peer Kröger, Peter Kunath, Alexey Pryakhin, and Matthias Renz. Efficient reverse k-nearest neighbor search in arbitrary metric spaces. In *ACM SIGMOD Conference*, pages 515–526, 2006.

[ABKS99] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: Ordering points to identify the clustering structure. In *SIGMOD Conference*, pages 49–60, 1999.

[ABW06] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006.

[AF07] Fabrizio Angiulli and Fabio Fassetti. Detecting distance-based outliers in streams of data. In *CIKM*, pages 811–820, 2007.

[Agg05] Charu C. Aggarwal. On abnormality detection in spuriously populated data streams. In *SDM*, 2005.

[AGGR98] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. *Proceedings of ACM SIGMOD?8 International Conference on Management of Data, p. 94-105*, 1998.

[AHWY03] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. A framework for clustering evolving data streams. In *VLDB*, pages 81–92, 2003.

[AKK$^+$09] Elke Achtert, Hans-Peter Kriegel, Peer Kröger, Matthias Renz, and Andreas Züfle. Reverse k-nearest neighbor search in dynamic and general metric databases. In *EDBT Conference*, pages 886–897, 2009.

[AU11] Massih-Reza Amini and Nicolas Usunier. Transductive learning over automatically detected themes for multi-document summarization. In *SIGIR*, pages 1193–1194, 2011.

[AW04] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, pages 336–347, 2004.

[AW10] Charu C. Aggarwal and Haixun Wang. A survey of clustering algorithms for graph data. In *Managing and Mining Graph Data*, pages 275–301. 2010.

[BDMO03] Brian Babcock, Mayur Datar, Rajeev Motwani, and Liadan O'Callaghan. Maintaining variance and k-medians over data stream windows. In *PODS*, pages 234–243, 2003.

[BGM$^+$06] Sanghamitra Bandyopadhyay, Chris Giannella, Ujjwal Maulik, Hillol Kargupta, Kun Liu, and Souptik Datta. Clustering distributed data streams in peer-to-peer environments. *Inf. Sci.*, 176(14):1952–1985, 2006.

[BH06] Jürgen Beringer and Eyke Hüllermeier. Online clustering of parallel data streams. *Data Knowl. Eng.*, 58(2):180–204, 2006.

[BHPG11] Albert Bifet, Geoff Holmes, Bernhard Pfahringer, and Ricard Gavaldà. Mining frequent closed graphs on evolving data streams. In *KDD*, pages 591–599, 2011.

[BKNS00] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. Lof: identifying density-based local outliers. *SIGMOD Rec.*, 29(2):93–104, 2000.

[BKS93] Thomas Brinkhoff, Hans-Peter Kriegel, and Ralf Schneider. Comparison of approximations of complex objects used for approximation-based query processing in spatial database systems. In *ICDE*, pages 40–49, 1993.

[BRV11] Antonio Balzanella, Lidia Rivoli, and Rosanna Verde. Data stream summarization by on-line histograms clustering. In *EGC*, pages 317–318, 2011.

[BW96] C.L. Bentley and M.O. Ward. Animating multidimensional scaling to visualize n- dimensional data sets. *Proc. of Information Visualization '96, p. 72-3*, 1996.

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

[CDN02] Jianjun Chen, David J. DeWitt, and Jeffrey F. Naughton. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *ICDE*, pages 345–356, 2002.

[CDTW00] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaracq: A scalable continuous query system for internet databases. In *SIGMOD Conference*, pages 379–390, 2000.

[CEQZ06] Feng Cao, Martin Ester, Weining Qian, and Aoying Zhou. Density-based clustering over an evolving data stream with noise. In *SDM*, 2006.

[CKW$^+$12] Xian Chen, Yoo-Ah Kim, Bing Wang, Wei Wei, Zhijie Jerry Shi, and Yuan Song. Fault-tolerant monitor placement for out-of-band wireless sensor network monitoring. *Ad Hoc Networks*, 10(1):62–74, 2012.

[CLW12] Yue Cui, Kaihua Liu, and Junfeng Wang. Direction-of-arrival estimation for coherent gps signals based on oblique projection. *Signal Processing*, 92(1):294–299, 2012.

[CMR05] Graham Cormode, S. Muthukrishnan, and Irina Rozenbaum. Summarizing and mining inverse distributions on data streams via dynamic inverse sampling. In *VLDB*, pages 25–36, 2005.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[CT07] Yixin Chen and Li Tu. Density-based clustering for real-time stream data. In *KDD*, pages 133–142, 2007.

[CwH02] Kevin Chen-Chuan Chang and Seung won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD Conference*, pages 346–357, 2002.

[dAMFH08] José de Aguiar Moraes Filho and Theo Härder. Accurate histogram-based xml summarization. In *SAC*, pages 998–1002, 2008.

[DHYC06] Bi-Ru Dai, Jen-Wei Huang, Mi-Yen Yeh, and Ming-Syan Chen. Adaptive clustering for multiple evolving streams. *IEEE Trans. Knowl. Data Eng.*, pages 1166–1180, 2006.

[DHZS02] Chris Ding, Xiaofeng He, Hongyuan Zha, and Horst Simon. Adaptive dimension reduction for clustering high dimensional data. In *Proc. 2nd IEEE Int'l Conf. Data Mining*, pages 147–154, December 2002.

[DSJ$^+$12] Lauren Davis, Funda Samanlioglu, Xiaochun Jiang, Daniel Mota, and Paul Stanfield. A heuristic approach for allocation of data to rfid tags: A data allocation knapsack problem (dakp). *Computers & OR*, 39(1):93–104, 2012.

[EFK99] J. N. Entzminger, C. A. Fowler, and W. J Kenneally. Jointstars and gmti: Past, present and future. *IEEE Trans on Aero and Elec Sys*, 35(2):748–762, april 1999.

[EKS$^+$98] M. Ester, H. Kriegel, J. Sander, M. Wimmer, and X. Xu. Inc. clustering for mining in a data warehousing environment. In *VLDB*, pages 323–333, 1998.

[EKSX96] M. Ester, H. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231, 1996.

[GK96] Sudipto Guha and Samir Khuller. Approx. algo. for connected dominating sets. *Algorithmica*, 20:374–387, 1996.

[GKMS01] Anna C. Gilbert, Yannis Kotidis, S. Muthukrishnan, and Martin Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *VLDB*, pages 79–88, 2001.

[GKS01] Johannes Gehrke, Flip Korn, and Divesh Srivastava. On computing correlated aggregates over continual data streams. In *SIGMOD Conference*, pages 13–24, 2001.

[GM06] Marcin Gorawski and Rafal Malczok. Aec algorithm: A heuristic approach to calculating density-based clustering *ps* parameter. In *ADVIS*, pages 90–99, 2006.

[GMM$^+$03] Sudipto Guha, Adam Meyerson, Nina Mishra, Rajeev Motwani, and Liadan O'Callaghan. Clustering data streams: Theory and practice. *IEEE Trans. Knowl. Data Eng.*, 15(3):515–528, 2003.

[GMMO00] Sudipto Guha, Nina Mishra, Rajeev Motwani, and Liadan O'Callaghan. Clustering data streams. In *FOCS*, pages 359–366, 2000.

[GRDG11] Francisco Guil-Reyes and María Teresa Daza-Gonzalez. Summarizing frequent itemsets via pignistic transformation. In *EPIA*, pages 297–310, 2011.

[GRS98] S. Guha, R. Rastogi, and K. Shim. Cure: an efficient clustering algorithm for large databases. *SIGMOD Record, vol.27(2), p. 73-84*, 1998.

[GW02] Like Gao and Xiaoyang Sean Wang. Continually evaluating similarity-based pattern queries on a streaming time series. In *SIGMOD Conference*, pages 370–381, 2002.

[Han05] Jiawei Han. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[HFAE03] Moustafa A. Hammad, Michael J. Franklin, Walid G. Aref, and Ahmed K. Elmagarmid. Scheduling for shared window joins over data streams. In *VLDB*, pages 297–308, 2003.

[HMA09] Parisa Haghani, Sebastian Michel, and Karl Aberer. Evaluating top-k queries over incomplete data streams. In *CIKM*, pages 877–886, 2009.

[HW] J. A. Hartigan and M. A. Wong. A k-means clustering algorithm. *Applied Statistics*, 28(1):100–108.

[HYJS06] Yih-Ling Hedley, Muhammad Younas, Anne E. James, and Mark Sanderson. Sampling, information extraction and summarisation of hidden web databases. *Data Knowl. Eng.*, 59(2):213–230, 2006.

[IKM00] Piotr Indyk, Nick Koudas, and S. Muthukrishnan. Identifying representative trends in massive time series data sets using sketches. In *VLDB*, pages 363–372, 2000.

[INE] Inc. INETATS. Stock trade traces. http://www.inetats.com/.

[JMF99] A. Jain, M. Murty, and P. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[JOT$^+$05] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. idistance: An adaptive b+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30:364–397, June 2005.

[JWK01] Moon Jeung Joe, Kyu-Young Whang, and Sang-Wook Kim. Wavelet transformation-based management of integrated summary data for distributed query processing. *Data Knowl. Eng.*, 39(3):293–312, 2001.

[JYC$^+$08] Cheqing Jin, Ke Yi, Lei Chen, Jeffrey Xu Yu, and Xuemin Lin. Sliding-window top-k queries on uncertain streams. *PVLDB*, 1(1):301–312, 2008.

[JYC$^+$10] Cheqing Jin, Ke Yi, Lei Chen, Jeffrey Xu Yu, and Xuemin Lin. Sliding-window top-k queries on uncertain streams. *VLDB J.*, 19(3):411–435, 2010.

[KFHJ04] Sailesh Krishnamurthy, Michael J. Franklin, Joseph M. Hellerstein, and Garrett Jacobson. The case for precision sharing. In *VLDB*, pages 972–986, 2004.

[KN98a] Edwin M. Knorr and Raymond T. Ng. Algorithms for mining distance-based outliers in large datasets. In *Proc. VLDB*, pages 392–403, 1998.

[KN98b] Edwin M. Knorr and Raymond T. Ng. Algorithms for mining distance-based outliers in large datasets. In *VLDB*, pages 392–403, 1998.

[KN99] Edwin M. Knorr and Raymond T. Ng. Finding intensional knowledge of distance-based outliers. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10,* *1999, Edinburgh, Scotland, UK*, pages 211–222. Morgan Kaufmann, 1999.

[KOTZ04] Nick Koudas, Beng Chin Ooi, Kian-Lee Tan, and Rui Zhang. Approximate nn queries on streams with guaranteed error/performance bounds. In *VLDB Conference*, pages 804–815, 2004.

[KWF06] Sailesh Krishnamurthy, Chung Wu, and Michael J. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD Conference*, pages 623–634, 2006.

[LDS11] Xuan Li, Liang Du, and Yi-Dong Shen. Graph-based marginal ranking for update summarization. In *SDM*, pages 486–497, 2011.

[LLL11] Jingxuan Li, Lei Li, and Tao Li. Mssf: a multi-document summarization framework based on submodularity. In *SIGIR*, pages 1247–1248, 2011.

[LLYZ03] Xuemin Lin, Qing Liu, Yidong Yuan, and Xiaofang Zhou. Multiscale histograms: Summarizing topological relations in large spatial datasets. In *VLDB*, pages 814–825, 2003.

[LMT$^+$05] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record*, 34(1):39–44, 2005.

[LRY08] Zhen Liu, Srinivasan Parthasarathy 0002, Anand Ranganathan, and Hao Yang. Near-optimal algorithms for shared filter evaluation in data stream systems. In *SIGMOD Conference*, pages 133–146, 2008.

[LS09] Levi Lelis and Jörg Sander. Semi-supervised density-based clustering. In *ICDM*, pages 842–847, 2009.

[LT10] Kristen LeFevre and Evimaria Terzi. Grass: Graph structure summarization. In *SDM*, pages 454–465, 2010.

[LWC11] Wen-Yang Lin, You-En Wei, and Chun-Hao Chen. A generic approach for mining indirect association rules in data streams. In *IEA/AIE (1)*, pages 95–104, 2011.

[MBP06] Kyriakos Mouratidis, Spiridon Bakiras, and Dimitris Papadias. Continuous monitoring of top-k queries over sliding windows. In *SIGMOD*, pages 635–646, 2006.

[MCH11] Fei Mai, C. Q. Chang, and Y. S. Hung. A subspace approach for matching 2d shapes under affine distortions. *Pattern Recognition*, 44(2):210–221, 2011.

[MP07] Kyriakos Mouratidis and Dimitris Papadias. Continuous nearest neighbor queries over sliding windows. *IEEE Trans. Knowl. Data Eng.*, 19(6):789–803, 2007.

[MSHR02] Samuel Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *SIGMOD Conference*, pages 49–60, 2002.

[Mun00] Jame.R Munkres. *Topology*. Prentice Hall, 2000.

[MVW98] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. Wavelet-based histograms for selectivity estimation. In *SIGMOD Conference*, pages 448–459, 1998.

[NH94] R. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. *VLDB'94, p. 144-155*, 1994.

[NRB06] Michel Neuhaus, Kaspar Riesen, and Horst Bunke. H.: Fast suboptimal algorithms for the computation of graph edit distance. In *SSSPR*, pages 163–172, 2006.

[NRS08] Saket Navlakha, Rajeev Rastogi, and Nisheeth Shrivastava. Graph summarization with bounded error. In *SIGMOD Conference*, pages 419–432, 2008.

[NZTK08] Sarana Nutanong, Rui Zhang, Egemen Tanin, and Lars Kulik. The v*-diagram: a query-dependent approach to moving knn queries. *PVLDB*, 1(1):1095–1106, 2008.

[Ord96] Keith Ord. Outliers in statistical data. *International Journal of Forecasting*, 12(1):175–176, March 1996.

[Pei09] Jian Pei. Association rules. In *Encyclopedia of Database Systems*, pages 140–142. 2009.

[RLL06] Faraz Rasheed, Young-Koo Lee, and Sungyoung Lee. Towards summarized representation of time series data in pervasive computing systems. In *UIC*, pages 658–668, 2006.

[RR96] Ida Ruts and Peter J. Rousseeuw. Computing depth contours of bivariate point clouds. *Comput. Stat. Data Anal.*, 23(1):153–168, 1996.

[SIK07] Mohamed A. Soliman, Ihab F. Ilyas, and Nick Koudas. Finding skyline and top-k bargaining solutions. In *IEEE ICDE Conference*, pages 1263–1267, 2007.

[SPP$^+$06] S. Subramaniam, T. Palpanas, D. Papadopoulos, V. Kalogeraki, and D. Gunopulos. Online outlier detection in sensor data using non-parametric models. In *VLDB*, pages 187–198, 2006.

[SYRW11] Avani Shastri, Di Yang, Elke A. Rundensteiner, and Matthew O. Ward. Mtops: scalable processing of continuous top-k multi-query workloads. In *CIKM*, pages 1107–1116, 2011.

[THP08] Yuanyuan Tian, Richard A. Hankins, and Jignesh M. Patel. Efficient aggregation for graph summarization. In *SIGMOD Conference*, pages 567–580, 2008.

[TWS04] Martin Theobald, Gerhard Weikum, and Ralf Schenkel. Top-k query evaluation with probabilistic guarantees. In *VLDB*, pages 648–659, 2004.

[VNB03] Stratis Viglas, Jeffrey F. Naughton, and Josef Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, pages 285–296, 2003.

[VvLS11] Jilles Vreeken, Matthijs van Leeuwen, and Arno Siebes. Krimp: mining itemsets that compress. *Data Min. Knowl. Discov.*, 23(1):169–214, 2011.

[W09] Changliang Wang and Lei Chen 0002. Continuous subgraph pattern search over graph streams. In *ICDE*, pages 393–404, 2009.

[War94] M.O. Ward. Xmdvtool: Integrating multiple methods for visualizing multivariate data. *Proc. of Visualization '94, p. 326-33*, 1994.

[WEC09] Paul Whitney, Dave Engel, and Nick Cramer. Mining for surprise events within text streams. In *SDM*, pages 617–627, 2009.

[WFZZ10] Xiaopeng Wei, Xiaoyong Fang, Qiang Zhang, and Dongsheng Zhou. 3d point pattern matching based on spatial geometric flexibility. *Comput. Sci. Inf. Syst.*, 7(1):231–246, 2010.

[WL10] Dingding Wang and Tao Li. Many are better than one: improving multi-document summarization via weighted consensus. In *SIGIR*, pages 809–810, 2010.

[WRGB06] Song Wang, Elke A. Rundensteiner, Samrat Ganguly, and Sudeept Bhatnagar. State-slice: New paradigm of multi-query optimization of window-based stream queries. In *VLDB Conference*, pages 619–630, 2006.

[WZJS09] Xiang Wang, Kai Zhang, Xiaoming Jin, and Dou Shen. Mining common topics from multiple asynchronous text streams. In *WSDM*, pages 192–201, 2009.

[YGRW11] Di Yang, Zhenyu Guo, Elke A. Rundensteiner, and Matthew O. Ward. Clues: a unified framework supporting interactive exploration of density-based clusters in streams. In *CIKM*, pages 815–824, 2011.

[YGX$^+$10] Di Yang, Zhenyu Guo, Zaixian Xie, Elke A. Rundensteiner, and Matthew O. Ward. Interactive visual exploration of neighbor-based patterns in data streams. In *SIGMOD*, pages 1151–1154, 2010.

[YiTWM00] Kenji Yamanishi, Jun ichi Takeuchi, Graham J. Williams, and Peter Milne. On-line unsupervised outlier detection using finite mixtures with discounting learning algorithms. In *KDD*, pages 320–324, 2000.

[YLL$^+$05] Yidong Yuan, Xuemin Lin, Qing Liu, Wei Wang, Jeffrey Xu Yu, and Qing Zhang. Efficient computation of the skyline cube. In *VLDB*, pages 241–252, 2005.

[YOTJ01] Cui Yu, Beng Chin Ooi, Kian-Lee Tan, and H. V. Jagadish. Indexing the distance: An efficient method to knn processing. In *VLDB Conference*, pages 421–430, 2001.

[YRW07] Di Yang, Elke A. Rundensteiner, and Matthew O. Ward. Nugget discovery in visual exploration by query consolidation. In *CIKM*, pages 603–612, 2007.

[YRW09] Di Yang, Elke A. Rundensteiner, and Matthew O. Ward. Neighbor-based pattern detection for windows over streaming data. In *EDBT*, pages 529–540, 2009.

[YSRW11] Di Yang, Avani Shastri, Elke A. Rundensteiner, and Matthew O. Ward. An optimal strategy for monitoring top-k queries in streaming windows. In *EDBT*, pages 57–68, 2011.

[YY$^+$03] Ke Yi, Hai Yu, Jun Yang 0001, Gangqiang Xia, and Yuguo Chen. Efficient maintenance of materialized top-k views. In *ICDE*, pages 189–200, 2003.

[ZKOS05] Rui Zhang, Nick Koudas, Beng Chin Ooi, and Divesh Srivastava. Multiple aggregations over data streams. In *ACM SIGMOD Conference*, pages 299–310, 2005.

[ZMC09] Shiming Zhang, Nikos Mamoulis, and David W. Cheung. Scalable skyline computation using object-based space partitioning. In *ACM SIGMOD Conference*, pages 483–494, 2009.

[ZRL96] T. Zhang, R. Ramakrishnan, and M. Livny. Birch: an efficient data clustering method for very large databases. *SIGMOD, vol.25(2), p. 103-14*, 1996.