Approximation algorithms for mining patterns from data streams

Dang, Xuan Hong

2008

Dang, X. H. (2008). Approximation algorithms for mining patterns from data streams.
Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/13581>

<https://doi.org/10.32657/10356/13581>

Approximation Algorithms for Mining Patterns from Data Streams

A Thesis submitted to the Nanyang Technological University
in fullfilment of the requirement for the degree of
Doctor of Philosophy

by

Dang Xuan Hong

Division of Information Systems
School of Computer Engineering

2008

Abstract

Traditional data mining techniques expect all data to be managed within some form of persistent datasets. Recently, for many emerging applications, such as stock tickers, web-click streams, and telecom call records, the concept of a *data stream* is more appropriate than a stored dataset. Naturally, a data stream is generated continuously in a dynamic environment with huge volume, infinite flow, and fast changing behaviors. Furthermore, they usually arrive to a mining system in a push-based manner meanwhile system resources used in the mining process are generally restricted in advance. Consequently, there have been increasing demands for developing novel techniques that are able to discover interesting patterns from data streams while they work within system resource constraints. Moreover, the mining results returned by these techniques are often desirable to be guaranteed within some error. When such an important task is completed, it is strongly believed that the quality of making decisions can be improved significantly in streaming environments. This research aims to study and investigate various approximation algorithms in order to effectively and efficiently mine useful patterns from data streams under different system resource constraints. Two fundamental data mining tasks are explored in the streaming data context: frequent pattern discovering and cluster analysis. The contributions of this research are claimed as follows:

A novel algorithm named EStream is developed to address the problem of online mining frequent patterns from data streams with precise error guarantee. EStream is a false positive algorithm in that all frequent itemsets appearing in the stream are always reported in the mining results and it strictly guarantees that: (i) for 1-itemsets, there is no error in its frequency; (ii) for 2-itemsets, the error in the estimate is no more than ϵ ;

and (iii) for k -itemsets, the error is no more than 2ϵ , where ϵ is the preset error threshold and k is the maximal length of the frequent set in the stream. It is further shown that the EStream algorithm can be directly extended to mine frequent patterns in a forgetful mining model.

A novel problem of load shedding in mining frequent patterns from data streams with constraints in CPU capacity is formally formulated and addressed. An algorithm called LSFI (Load Shedding for mining Frequent Itemsets) is then presented to address the problem in the landmark window model. The Chernoff bound is employed in the algorithm design and such a theoretically sound tool allows LSFI to approximate closely a set of frequent patterns from data streams under various overloading situations.

An algorithm named LSSW (Load Shedding for mining frequent patterns in Sliding Window) is developed to completely address the problem of mining frequent sets with CPU constraints in one of the most challenging mining models, the time-based sliding window model. LSSW addresses and gives solutions to the following questions: How to detect overload situations? How much load needs to be shed? and How to approximate mining results under load shedding? LSSW processes data streams online; its quality of mining results is guaranteed within a preset error threshold with high probability.

A novel algorithm called SWEM, which is based on the Expectation-Maximization technique, is proposed to address the problem of clustering data streams in the sliding window model. SWEM assumes that a data stream is generated by a dynamic statistical process which consists of k multivariate Gaussian mixture models and thus, it approximates clusters from data streams by finding a set of parameters that optimally describes these mixture models. Data records arriving within each part of data streams are incrementally processed and summarized into a set of micro components. These sufficient statistics are then used in later phases to derive global clustering results when the window slides. Furthermore, in order to address the problem of distribution changes in data streams, splitting and merging components are developed in SWEM. They provide a method to move discretely micro components in the data space and thus approximate effectively new characteristics in streaming data environments.

Acknowledgments

My deepest gratitude is due to my advisor, Associate Professor Ng Wee Keong, who taught me how to do research, guided me through all important problems, helped me to improve my writing, presentation and many other things. He watched me in all my progress, and most importantly, taught me to think on my own and work on any problems that interest me. Without him, I would never have finished this dissertation.

I would like to express my deep gratitude to Dr.Ong Kok-Leong at Deakin University, Australia, who gave me many precious advices on all my work. His suggestions are great sources of ideals and improvements. With large amount of experience and knowledge, he did teach me much in doing research. Really, he is my invaluable mentor. I would also like to thank him for financially supporting me in all international conferences that we had attended together.

I am also very grateful to Professor Vincent C.S. Lee at Monash University, Australia for his valuable advice and support. He showed me how to write an appealing research proposal and continuously supported me in finding job after submitting the dissertation.

I would like to thank Associate Professor Le Hai Khoi and Ms. Tran Thi Phien at Institute for Information Technology for their constant support on me in pursuing the PhD degree. Without their help and encouragement, I would not have been like I am today.

I am thankful to Lai Chee Keong, Chua Chiew Song, and Tan Lay Choo for their assistance in providing the administration and technical support during my stay at the Center for Advanced Information Systems (CAIS). I also would like to thank all my friends Li Wenyan, Hady Wirawan Lauw, Ng Willie, Erwin Leonardi at CAIS. With

Vietnamese post-graduate students studying in NTU that I am not able to list all their names, I owe many thanks to them. We have been sharing both the joys and sadness in the life of PhD students away from home.

Last but not least, I must thank my parents for their care, support and unwavering confidence in me. Thanks to my wife and my two sons for bringing the love and strength into my life.

Contents

Abstract	i
Acknowledgments	iii
List of Figures	ix
List of Tables	xi
1 Introduction	1
1.1 Motivation	1
1.2 Objectives	6
1.3 Contributions	8
1.4 Thesis Organization	10
2 Overview of Data Stream Mining	12
2.1 Data Stream Models	12
2.2 Frequent Patterns Mining in Data Streams	15
2.2.1 Overview of Frequent Set Mining	15
2.2.2 Frequent Set Mining in Data Streams	19
2.2.3 Discussions	24
2.3 Clustering of Data Streams	25
2.3.1 Overview of Clustering Techniques	25
2.3.2 Clustering Data Streams	27
2.3.3 Discussions	32
2.4 Other Work in Data Stream Processing	33
2.4.1 Data Stream Classification	33

2.4.2	Data Stream Management Systems	37
3	Online Approximate Frequent Pattern Mining with Precise Error Guarantee	43
3.1	Motivation	43
3.2	Problem Formulation	45
3.3	Online Patterns Mining with Error Guarantee	48
3.3.1	Data Structure	48
3.3.2	Algorithm	50
3.3.3	Error Analysis	56
3.4	Performance Evaluation	58
3.4.1	Experimental Setup	58
3.4.2	Scalability Study	58
3.4.3	Time-varying Study	60
3.4.4	Comparison with Lossy Counting	62
3.5	EStream Discussions	67
3.6	Extending From Landmark model to Forgetful model with FStream . . .	69
3.7	Summary	73
4	Approximate Frequent Patterns with CPU Constraints	76
4.1	Motivation	76
4.2	Problem Formulation	78
4.3	Load Shedding Algorithm in Landmark Window Model	79
4.3.1	Probabilistic Technique to Shed Load	79
4.3.2	The Algorithm	82
4.4	Experimental Results	85
4.4.1	Experimental Setup	85
4.4.2	Measurement of Errors	86
4.4.3	LSFI Performance	87

4.4.4	Comparison with Algorithm Z	91
4.5	Discussions	95
5	CPU Constraints: From Landmark Window to Sliding Window Model	98
5.1	Motivation	98
5.2	Problem Formulation	99
5.3	Online Approximate Frequent Patterns in Sliding Window Model	100
5.3.1	Workload Estimation	100
5.3.2	The Online Algorithm	102
5.3.3	Error Analysis	108
5.4	Experimental Results	111
5.4.1	Accuracy Measurements	111
5.4.2	Adaptability Study	114
5.5	Summary	116
6	Clustering Data Streams in Sliding Window Models with EM Tech- nique	118
6.1	Motivation	118
6.2	Problem Formulation	120
6.3	The Stream Clustering Framework	122
6.4	Clustering Algorithm	126
6.4.1	Initial Phase	126
6.4.2	Incremental Phase	131
6.4.3	Expiring Phase	135
6.5	Experimental Results	139
6.5.1	Experimental Setup	139
6.5.2	Clustering Quality Evaluation	140
6.5.3	Parameter Sensitivity	144
6.5.4	Comparison with CluStream	148
6.6	Summary	154

7	Conclusions and Future Work	156
7.1	Summary	156
7.2	Future Work	159
7.3	List of Publications	162
	References	164

List of Figures

2.1	Computation model for data streams	13
2.2	An example of Landmark Window model	14
2.3	An example of Forgetful Window model	15
2.4	An example of Sliding Window model with size $\Delta = 5$	16
3.1	An example of STrieIT data structure	50
3.2	EStream Algorithm	53
3.3	Procedure for checking the frequency of all subsets of an itemset X in the landmark window model.	54
3.4	Memory usage of EStream on two representative datasets of opposite char- acteristics.	59
3.5	Execution time of EStream on two representative datasets.	61
3.6	Experiment results on a time-varying data streams.	62
3.7	Precision Comparison between EStream and Lossy Counting on T8.I4.D1000K	63
3.8	Memory Comparison between EStream and Lossy Counting on T8.I4.D1000K	65
3.9	Execution Time Comparison between EStream and Lossy Counting on T8.I4.D1000K	66
3.10	FStream Algorithm	74
3.11	Procedure for checking the frequency of all subsets of an itemset X in the forgetful window model.	75
4.1	LSFI Algorithm	83
4.2	Accuracy of LSFI on dataset T8.I4.D1000K.	88

4.3	Accuracy of LSF1 on dataset T10.I4.D1000K.	89
4.4	Varying data stream rates and σ_0 on dataset T10.I4.D1000K.	90
4.5	Varying data stream rates and domains on dataset T10.I4.D1000K.	91
4.6	Accuracy of LSF1 and algorithm Z on dataset T10.I4.D1000K at $\sigma = 0.2\%$	92
4.7	Accuracy of LSF1 and algorithm Z on dataset T10.I4.D1000K at $\sigma = 0.4\%$	93
4.8	The execution time of LSF1 and algorithm Z on dataset T10.I4.D1000K at $\sigma = 0.2\%$ and $\sigma = 0.4\%$	94
5.1	LSSW Algorithm	104
5.2	Accuracy of LSSW on dataset T8.I4.D3000K	111
5.3	Accuracy of LSSW on dataset T10.I6.D3000K	112
5.4	Accuracy vs. Itemset Length of LSSW for T10.I6.D3000K	113
5.5	Time and workload of LSSW on T5-8-10.D3000K at various support thresh- olds	114
5.6	Accuracy of LSSW on dataset T5-8-10.D3000K	116
6.1	A splitting visualization.	132
6.2	A merging visualization.	133
6.3	Average log likelihood on D4.K5.N200k	142
6.4	Accumulative processing time on D4.K5.N200k	143
6.5	Micro Components vs. Accuracy	145
6.6	Merging threshold $Avg(D_{i,j})$ vs. Accuracy	146
6.7	Fading Factor vs. Accuracy	147
6.8	Clustering quality comparison on D2.K10.N100k	149
6.9	Clustering quality comparison on D10.K4.N100k	150
6.10	Clustering quality comparison on changing distribution data streams	151
6.11	Execution time comparison on changing distribution data streams	153

List of Tables

6.1	Average log likelihood values returned by stdEM, SWEMw/oG and SWEM algorithms	141
6.2	Clustering means on $D4.K5.N100k$ and $D4.K5.N100k$ with 5% noise . .	141

Chapter 1

Introduction

1.1 Motivation

Data mining is usually defined as the process of automatically discovering useful information from large data repositories [TSK05]. Mining such datasets allows one to design algorithms that may thoroughly scan or query the data as many times as necessary. In recent years, both this definition and related approaches may no longer be appropriate due to the emergence of a new class of applications. On the one hand, many data warehouses are growing to the point where hundreds of gigabytes are considered small; multiple scanning algorithms when applied on these very large datasets turn out to pose problems due to the cost of random access. On the other hand, numerous novel systems generate data in a continuous fashion and usually in the form of multiple, rapid, unbound, and time-varying streams. Below are some of these applications.

- Web click streams [DH01, BBD⁺02]: With the rapid development of the Internet, the number of people using the Internet as a daily tool to access information and do business is increasing. Each time a user browses a web site, the server keeps track of all user actions in log files. What is captured is the “click flow” of the mouse. Thus, one can define a click-stream as the sequence of requested pages. Mining of Web usage logs aims to find interesting user behavioral patterns. This may help to

CHAPTER 1. INTRODUCTION

manage and control load-balancing as well as system caching more effectively and efficiently in a real time manner. This may also be useful in predicting the hosts or pages that will be requested in the near future [DH01].

- Financial markets [CFP⁺04, Yan03]: In financial markets, stock market transactions are often generated in a stream fashion. They consist of buy and sell orders for particular securities from individual investors. Online identification of the stocks or orders that are often sold or bought together is very critical for investors. This helps them to make decision early so that they may seize the appropriate opportunities.
- Telecommunication systems [CFPR00, LP01]: Data are often generated in streams of call transactions in the telecommunications domain. One of the many companies working in this area is Lucent Technologies company [CFPR00]. It is said that the company has to process millions of call-detail records per day that are generated by approximately 12 million customers. As a result, analyzing these huge datasets is an important task. It allows one to uncover useful patterns such as the signatures (i.e., evolving profiles) of customers, or fraud intrusions that need to be timely detected [CFP⁺04].
- Sensor networks [MF02, TcZ⁺03]: In the literature, there are a number of data stream applications in that the rate of generating data can potentially overwhelm the system's resources. Sensor networks are typical examples for such applications [MF02, TcZ⁺03]. An example could be found in astronomical applications [MF02, ZS03]. For identifying abnormal phenomenon in sky, sensors are used to detect Gamma Ray Burst. Once such a burst is detected, telescopes could then point to that portion of sky to confirm new astrophysical events. As hundreds of photons are recorded within a second, such a system often generates data in high speeds of

CHAPTER 1. INTRODUCTION

non-stopped streams. Nevertheless, mining on these applications requires abnormal phenomenons are required to be detected and alerted timely.

The nature of the above data stream applications pose many problems to data mining algorithm designers [BBD⁺02, CCP⁺04]. Compared to traditional datasets, data streams are different in almost all fundamental aspects. First, while the size of a conventional dataset is generally limited by disk based storage, the size of a data stream is very huge and possibly unbounded, making it impractical to store the entire content of the stream. Second, transactions in a data stream are only cached in main memory for a very short amount of time. Once processed, they are discarded from memory and it is expensive to access them again. In contrast, records in a conventional dataset are stored persistently in disk-based relations; they are easily retrieved and can be stored in memory as long as needed by a query or a data mining task. Third, data records in a data stream arrive in ordered sequences. There is no control over the order in which data transactions are accessed. In contrast, records in a traditional dataset could be processed in any order. Fourth, in some applications, streaming data often arrives at high rate and requires real-time response. Worse, the data arriving at the system is in a push-based manner and can vary significantly over all time-scales. In contrast, updates on conventional datasets are small and relatively infrequent.

Given these distinct properties of data streams, new issues have arisen that challenge the data mining research community. Independent of the tasks of data mining, any system developed for mining over data streams should meet the following criteria [DH01, GGR02]:

- As data streams are usually generated in high volume within a short span of time, a mining system should have the ability to process each incoming data transaction within a small amount of time. Preferably, it should deal with each transaction at

CHAPTER 1. INTRODUCTION

most once since the cost to re-visit any old data transaction is either expensive or impractical.

- The amount of memory space utilized for the mining process is generally confined. Therefore, a mining system should keep the memory independent of the total number of data instances seen so far in the data streams. Furthermore, the data arrival rate can be arbitrarily high; this sometimes overwhelms the CPU capacity. As such, the system should be able to adapt to changes in order to guarantee that the mining results are produced within a given time frame.
- Due to the continuous characteristic of data stream, users cannot wait for the mining results until the streams stop. Thus, a mining system should have the ability to report the mining results at any time instant on demand.
- Most importantly, a mining system should be capable of producing results that are equivalent or nearly identical to the ones that would be obtained by the corresponding ordinary database mining algorithm, operating without any constraint of a streaming data environment.

Due to these requirements, mining over data streams poses two fundamental challenges. The first challenge is how to design algorithms to perform data mining over data streams under the constraints of only one scan on data and yet working within given system resources. Most existing data mining algorithms assume that a dataset is not only available at any time but can also be randomly accessed multiple times. For example, the k -means, k -centroid methods [Eve93, Ber02] in clustering; the FP-growth [HPY00], Apriori [AS94] algorithms in association rule analysis; the ID3 or C4,5 decision tree [Mit97] in classification. In contrast, the new model of data streams allows one to look at each data instance at most once. The second challenge, which is even more difficult in streaming

CHAPTER 1. INTRODUCTION

environments, is how to approximate mining results at high accuracy. More importantly, it is a strong demand to guarantee them within some given error threshold. Such an important issue is not adequately addressed in the literature (as will be shown later in Chapter 2).

In addition to the above challenging tasks, stream processing models also pose more issues on data stream mining. Due to the time-varying nature of data streams and their potentially unlimited size, not all data transactions appearing in a stream are always of interest. Rather, their usefulness is application dependent. For example, some applications focus only on a fixed portion (most recently arriving) of the stream, while other applications consider all transactions seen so far; yet transaction weights (or importance) are gradually reduced over time. This gives rise to three different mining models: the Landmark window model, the Forgetful window model and the Sliding window model. The landmark window model, is initially proposed to address offline data streams where data transactions arrive in batches and all transactions are equally important. Nevertheless, this model is also widely used in various online stream applications where a set of landmarks are given in advance [GKS01]. The forgetful model is an extension of the landmark window model where a fading function is employed to reduce the effect of historical data transactions on the current mining results. Compared to the two previous models, the sliding window model is more challenging since it considers the elimination of outdated data instances. The mining results are computed based only on a portion of the stream that most recently arrives at the system. At some levels, the forgetful and sliding window models can be considered as a system approach to address the problem of concept drifts [WK96] in which the characteristics of data streams may drastically change over time [Tsy02, CWZ04].

1.2 Objectives

Given the above challenges and motivations, the objective of this thesis is to investigate various approximation algorithms that effectively and efficiently mine useful patterns from data streams under varying system resource constraints. Two fundamental data mining tasks will be explored in the streaming data context: frequent pattern discovering and cluster analysis. We focus on the following fundamental issues:

Online Mining of Frequent Patterns with Error Guarantees — With its broad applications such as iceberg query [FSGM⁺98], iceberg datacube [HPDW01], association rule mining [AS94], correlation analysis [BMS97], classification [DKK03], and clustering [WXL99], frequent pattern mining has been a central research topic in data mining for years. One of the challenges of this data mining task is the exponential explosion number of patterns that need to be explored. Given a domain of m items, the number of distinct patterns (itemsets) is 2^m . For data stream applications where incoming data transactions will not be stored and we can only scan them once, if counts are required for each itemset, an application with a moderate set of items, say $m = 1,000$, the total number of patterns is $2^m - 1 = 2^{1,000} - 1$, which is obviously intractable. Given limited memory storage and the need for real-time analysis in stream environments, we aim to investigate techniques that are able to process online transactions arriving in the stream and keep the memory space independent of its size. This means that not all patterns appearing in the stream will be maintained and counted. However, as we have indicated in the previous section, the quality of mining results is as important as the time to deliver the mining results. We target the issue of how to approximate and guarantee the mining results within a given error threshold. This is a critical issue as most stream mining algorithms proposed in the literature do not address properly.

Mining Frequent Patterns under CPU Capacity Constraints — The above stream mining of frequent patterns will be further studied under the constraint of CPU limitation. We consider the problem in the context of a high speed data stream environment where the streaming rate can be high and unpredictable. Very often, when the rate of arriving data unpredictably increases, the processing capacity of machines is overwhelmed. This critical situation usually happens in many stream environments, such as real-time stock trading, communication network traffic monitoring, or telecom fraud detection [DGR03, BDM04]. Unfortunately, while the rate of the stream is not controllable, analyzing results of these streams often requires the system to work within a given time frame in order to optimize the quality of making decision. The answers are only useful if they are timely. Consequently, guaranteeing both the quality of mining results within some given error threshold as well as the time constraint of result delivering becomes a very challenging task, yet it is very important in high speed streaming environments. We focus on investigating different one-pass and online algorithms to address this issue.

Clustering Streaming Data in Sliding Window Models — In addition to the problem of mining frequent patterns from streaming data under system resource constraints, in this thesis, we also study the problem of clustering data streams. Clustering is an unsupervised learning process which aims to partition similar data objects into the same clusters. This problem becomes much more challenging when it is considered in the context of streaming data. The need for incremental clustering is obvious given the nature of continuously arriving data streams. Also, due to this natural property, the list of clusters found so far in the stream can grow unboundedly as new data instances arrive. Consequently, there is a need to provide a compact representation for each cluster found and keep their size independent of the number of data points processed so far in the stream (due to the limitation of memory space). Furthermore, the underlying distribution of a data stream can change over time, it is necessary to update and report the

clustering results that timely reflect the current characteristics of the stream. We aim to study all of these stream clustering issues in the most challenging mining model, the sliding window model.

1.3 Contributions

As a result of this research, we have developed various approximation algorithms to mine patterns from data streams under different resource constraints. The contributions of this dissertation are summarized as follows:

- We develop a novel online approximation algorithm, named EStream, to mine frequent patterns from data streams with precise error guarantee. The algorithm is false positive oriented in that all truly frequent patterns are guaranteed to be reported in the mining results and at any time instant. Most importantly, the mining results are theoretically proven to be strictly guaranteed within a given error threshold. Our proposed EStream algorithm ensures that: (i) for 1-itemsets, there is no error in its frequency; (ii) for 2-itemsets, the error in the estimate is no more than ϵ ; and (iii) for itemsets of length $k > 2$, the error is no more than 2ϵ , where ϵ is the preset error threshold given by a user and k is the maximal length of the frequent set in the stream. We also show that the EStream algorithm can be directly extended to mine frequent patterns in a forgetful mining model.
- We formally propose and formulate a novel problem of load shedding in mining frequent patterns from data streams in order to address this data mining task in the context of CPU constraints. An algorithm named LSFI (Load Shedding for mining Frequent Itemsets) is proposed to address the problem in a landmark window model in that the streaming data rate can vary arbitrarily from time to time. The algorithm is designed based on a solid mathematical result, the Chernoff

bound, which allows LSFI to approximate closely a set of frequent patterns from data streams under various load shedding situations.

- We extend the problem of mining frequent patterns from data streams with CPU constraints to the most challenging stream mining model, the time-based sliding window and an approximate algorithm named LSSW (Load Shedding for mining frequent patterns in Sliding Window) is developed to address the problem from end to end; i.e., from overload detection to overload resolution. Both deterministic and probabilistic techniques are explored in the algorithm design in order to obtain a more accurate estimate on the mining error. Importantly, LSSW guarantees that the error in each pattern's frequency computed is no more than $\epsilon + \gamma$ with high probability of $(1 - \delta)$ where ϵ is a deterministic error given by a user, γ and δ are respectively an error bound and probability threshold guaranteed by the Chernoff bound. The contribution is significant since the mining results are theoretically proven to be guaranteed within the preset error threshold with high confidence.
- We propose a novel algorithm called SWEM, which is based on the Expectation-Maximization technique, in order to address the problem of clustering data streams in a sliding window model. By assuming that data streams are generated as a result of a dynamic statistical process that involves of k mixture models, SWEM is developed to approximate a set of parameters that reflects timely the current clustering characteristics of the stream. Data records arriving within each portion of data streams are statistically summarized into a set of micro components. These sufficient statistics are then used in later phases where the algorithm derives global clustering models in window sliding modes. In order to address the problem of stream distribution changes, splitting and merging components are developed in SWEM algorithm. These operations provide a method to discretely re-distribute

micro components in the data space and thus, they approximate effectively new characteristics in data streams. Also, SWEM is truly an incremental clustering algorithm in the sense that once data points in the stream are processed, they will be permanently eliminated from the mining model and thus, this allows the algorithm to save much memory space for data stream summaries.

1.4 Thesis Organization

This thesis is divided into seven chapters. This chapter introduces the motivations and research objectives of our study in the area of approximating patterns from data streams under various system constraints. The contributions are also summarized and briefly discussed. The rest of the thesis is organized as follows:

In the next chapter, we present an overview of the area of data stream mining and processing. The two main areas of data stream mining — frequent pattern mining and cluster analysis — are discussed in detail as they are the emphases of our research. Classification and data stream management systems are also briefly discussed as they help us to understand more about stream environments as well as complement the entire picture of data stream processing.

With this background, we begin Chapter 3 with our first contribution. In that we argue the need for various critical stream applications to satisfy both the online processing requirement and a strict error guarantee in the estimate of their results. An online algorithm to mine frequent patterns from data streams with error guarantees is presented and the results are theoretically proven. An extension of the algorithm to the forgetful window model is also discussed and presented.

In Chapter 4, we extend the frequent set mining problem to the context where CPU capacity is another limitation in streaming data environments. We formulate the problem in detail and provide the first contribution to address the problem: the algorithm of load

CHAPTER 1. INTRODUCTION

shedding in mining frequent patterns from data stream given *a priori* CPU capacity. This algorithm is designed based on a theoretically sound technique — the Chernoff bound theorem.

Chapter 5 is followed by our deeper look into the problem of CPU constraint in stream mining for frequent patterns. The problem is studied in the most challenging model, the time-based sliding window model with transaction flows being processed online. By exploring both deterministic and probabilistic techniques, the correctness of the proposed algorithm is theoretically and experimentally proven and presented.

Chapter 6 presents our work in the area of data stream clustering analysis. We present the SWEM algorithm, which is based on the Expectation-Maximization technique to address the problem in a sliding window model. The algorithm is truly an incremental method and works strictly in memory space constraint.

Finally, we give conclusions for the thesis in Chapter 7. The future work of the research is also discussed in this chapter.

Chapter 2

Overview of Data Stream Mining

In this chapter, we present an overview of existing research on data stream mining. We begin the chapter with data stream models. Then, literature review on frequent set mining and cluster analysis is presented and discussed in details. We complete the chapter with a brief discussion on data stream classification and data stream management systems.

2.1 Data Stream Models

In recent years, many query problems and mining capabilities have been explored for data stream environments, including those on aggregate query [BDM04, TZ06], statistical monitoring [ZS02, DGIM02], association rules [MM02, YCLZ04], clustering analysis [ZCY⁺07, CT07] and stream classification [AY07, AHWY04b]. Generally, a computation model for data streams, as depicted in Figure 2.1, includes the following components [TCY03]: stream processing engine and synopsis maintenance in memory; they are two principal components that generate results in a stream environment. The buffer component may be optional set for temporary storage of recent data arriving of data streams.

Generally, a data stream is usually considered a sequence of data records (or transactions) that is generated continuously in a dynamic environment with huge volume, infi-

Figure 2.1: Computation model for data streams

nite flow, and fast changing behaviors. Due to these natural characteristics, not all data records in data streams are considered equally with the same level of interest. Rather, their usefulness is dependent upon applications. For example, some applications focus only on a fixed portion (most recently arriving) of the stream, while other applications consider all the transactions seen so far; yet the transaction weight (or importance) is reduced gradually over time. This gives rise to a division of the processing engine over data streams into 3 main models:

Landmark window model: In this model (Figure 2.2), answers are computed from a set of data records in the stream identified between a specific record in the past, called landmark, and the current arriving data record. Accordingly, one endpoint (landmark) is fixed, while the other gradually increases with the arrival of new records. This model is suitable for off-line data streams where data instances usually arrive in batches [MM02], for example, with warehouse applications where new batches of records are updated at regular time intervals. However, there are also many other online streaming applications employing this model [GKS01]. For example, in a telecom transaction stream, the scope of the stream is captured on a daily basis. Accordingly, a set of landmarks is given to the

CHAPTER 2. OVERVIEW OF DATA STREAM MINING

Figure 2.2: An example of Landmark Window model

system and the answers are computed from the current point back to the immediately-preceding landmark.

Forgetful window model: Different from the landmark model, where all transactions appearing in the stream are equally important, in the forgetful window model, old transactions have less effect on current answers compared to the new arrivals in a stream. Therefore, in this model, a fading function is often used to reduce the weight (or importance) of transactions as time goes by. Figure 2.3 illustrates an example of this model.

Sliding window model: Compared to the above two models, a sliding window model further considers the elimination of transactions. When new transactions come, the oldest ones will be eliminated from the mining model. Accordingly, answers are computed only within a fixed portion of the stream which is pre-specified by a number of transactions (count-based sliding window) or by a time period (time-based sliding window). A sliding

CHAPTER 2. OVERVIEW OF DATA STREAM MINING

Figure 2.3: An example of Forgetful Window model

window model is quite suitable for various data stream applications such as stock markets, financial analysis or network monitoring where the most valuable information is one discovered from the recently arriving portion of the stream. Figure 2.4 presents an example of count-based sliding window model.

2.2 Frequent Patterns Mining in Data Streams

2.2.1 Overview of Frequent Set Mining

The problem of finding frequent itemsets in databases has been extensively studied. However, in data stream environments, they pose many new challenges for the data mining community. In this section, we first review some techniques that are often used to find frequent itemsets in a traditional large database. Then, most major work addressing this problems, both in finding frequent items and frequent itemsets in streaming data are

CHAPTER 2. OVERVIEW OF DATA STREAM MINING

Figure 2.4: An example of Sliding Window model with size $\Delta = 5$

discussed.

Traditionally, the problem of finding frequent itemsets from a large database is defined as follows. Let $I = \{a_1, a_2, \dots, a_m\}$ be a domain of m items. Let X be an itemset which is a subset of items in I : $X \subseteq I$. X is called k -itemset if it contains exactly k items. Given a database D containing N transactions $D = \{t_1, t_2, \dots, t_N\}$ where each t_i is a couple (tid, T) , we say a transaction t_i supports an itemset X if and only if $X \subseteq T$. Consequently, the *cover* (or *frequency*) of X in database D is defined by $cover(X, D) := \{tid \mid (tid, T) \in D, X \subseteq T\}$ and its support over database D is defined by the ratio $\frac{|cover(X, D)|}{N}$. The problem of finding frequent itemsets from database D aims at discovering all itemsets X such as its support $\frac{|cover(X, D)|}{N} \geq \sigma$ in that $\sigma \in (0, 1)$, called the minimum support threshold, is given by users.

It is clear that the search space in frequent pattern mining is the most challenging issue. Given a domain of $m = 1000$, the number of itemsets X can be up to $2^{1000} - 1$. For this reason, there are two main approaches to address the problem, the breadth-first search and the depth-first search strategies, which aim at pruning the search space as much as possible.

Apriori [AS94] is one of the first algorithm that approaches the problem using the breadth-first search strategy. The core idea of this algorithm is based on the downward closure property which indicates that any subset of a frequent itemset must be frequent; otherwise, the itemset is not frequent. Consequently, Apriori makes multiple passes over the database. At each pass, it generates a set of *candidate itemsets*, which have frequent subsets in the previous pass and count their occurrences in the database. More specifically, at pass 1 where the set of candidate itemsets C_1 consists of all items in I , Apriori traverses the database once and finds all frequent 1-itemsets and stores them in the set L_1 . At pass k , a set of candidate k -itemsets (which are generated based on L_{k-1}) is counted and the frequent itemsets will be kept in the set L_k . The algorithm iterates until no candidate is generated and counted. Therefore, the maximal number of passes on the database is $k + 1$ if k is the maximal length of frequent patterns found.

There are a number of algorithms that have been developed to improve Apriori's computations. For instance, Dynamic Itemset Counting [BMUT97] minimizes the number of passes over the database by dividing it into a number of intervals. Candidate 1-itemsets are then counted in the first intervals, those that are frequent are then used to generate candidate 2-itemsets that are then counted, with 1-itemsets in the second interval of the database. In general, after each interval, candidate itemsets are generated and counted. The algorithm stops when no candidates are generated and all candidates are counted in the complete database. With the same idea of partitioning database, Savasere *et al.* proposed the Partition algorithm in [SON95]. Each partition is fitted into the memory and all itemsets that are found to be locally frequent within this partition are maintained. Once all partitions have been processed, those locally frequent itemsets are then combined to form a set of candidate itemsets with which the algorithm needs only one pass on the entire database to check their frequency. It is easy to prove (by contradiction) that if an itemset is globally frequent, then it must be locally frequent in at least one partition.

CHAPTER 2. OVERVIEW OF DATA STREAM MINING

Eclat [ZPOL97, Zak00] is an algorithm that addresses the problem of finding frequent patterns from database using a depth-first search strategy. In order to count the cover of an itemset, it uses the vertical database layout and the intersection operation. In this format, each record consists of an item a_i and a set of transaction identifiers tid in which the item appears, which means the $cover(a_i, D)$. To find the cover of an itemset X , it is simple to intersect the cover sets of each item $a_i \in X$. The depth-first search is employed in such a way that each frequent pattern X found in previous pass k is added by an item a_{k+1} at the next pass $k + 1$. Then, the cover of each candidate itemset $(X \cup a_{k+1})$ is found by intersecting the $cover(X, D)$ and the $cover(a_{k+1}, D)$. It is noted that by reordering $\{a_1, a_2, \dots, a_m\}$ in ascending order of $|cover(a_i, D)|$ and each a_i is considered in that sequence, Eclat essentially reduces the number of candidate itemsets, and hence, reduces the number of intersections that need to be computed as well.

FP-growth is another well-known algorithm addressing the frequent set mining problem from the depth-first approach. Instead of storing the cover of each item as in Eclat, FP-growth stores the actual transactions from the database in a novel data structure named FP-tree. Each node in the tree has a label which is an item and a counter to register the number of transactions sharing the branch from root to the node. In addition to this data structure, FP-growth uses another header table where each entry is an item in the database and a link connecting to the same item in the FP-tree via a chain of node-links. Therefore, we can realize that FP-growth has integrated both the vertical layout (represented by node-links starting from the header table) and the horizontal layout (represented by branches in FP-tree) of the database. Also, items in each transaction used to build the FP-tree are ordered in descending frequency. Consequently, the FP-tree is small since the more frequently occurring items are arranged closer to the root and thus they are more likely to be shared. Once the FP-tree is built, the cover of each candidate can be found by following the node-links starting from the header table.

2.2.2 Frequent Set Mining in Data Streams

When extending the problem of mining frequent sets from a large database to a data stream, it is challenging to adapt the above algorithms in this environment. The difficulty lies in the fact that while most traditional algorithms introduced above are multiple passes over a dataset, it is allowed to use at most a linear single scan on a data stream. The problem is more challenging given the fact that both the memory space and the CPU capacity are also limited. Consequently, most algorithms introduced in stream environments can only deliver approximate results. We review the most significant algorithms below.

Lossy Counting [MM02] is one of the first algorithms to find all frequent sets from an entire transactional data stream. It is a deterministic approach since the mining results are guaranteed within some error bounds. Given an error threshold ϵ and a minimum support threshold σ , Lossy Counting guarantees that all itemsets whose true occurrence (or frequency) exceeds σN are included in the mining results (where N is the number of transactions seen so far in the stream). Furthermore, the estimated frequency for each output itemset is guaranteed to be no more than its true frequency by an amount of ϵN . In Lossy Counting, the stream is divided into buckets, each has the same size of $w = 1/\epsilon$ transactions and is labelled with bucket ids, starting from 1. At transaction N , the bucket id is $b = \lceil N/w \rceil$. During the mining process, Lossy Counting maintains a synopsis S which is a set of entries (X, f, Δ) , where X is an itemset, f is its estimated frequency, and Δ is the maximal possible error in f . For each X found in the stream, its frequency will be incremented by 1 if it is found in S ; otherwise, a new entry is created with the value $(X, f, b - 1)$. At each bucket boundary (i.e., $N \equiv 0 \pmod{w}$), S is pruned by deleting all itemsets that have $f + \Delta \leq b$. The frequent itemsets are those having $f \geq (\sigma - \epsilon)N$.

CHAPTER 2. OVERVIEW OF DATA STREAM MINING

To see how Lossy Counting approximates frequency errors, it is important to note that the stream is divided into equally sized buckets, and the occurrence of each itemset X is tracked only if it appears frequently enough; i.e., on average, at least once on every bucket sized $1/\epsilon$. Accordingly, the deletion rule ensures that whenever $f \leq b - \Delta$, X is deleted. Note that $(b - \Delta)$ is the number of buckets since the last time X occurs frequently enough to be counted. This value is always no more than N/w which is the number of buckets seen so far in the stream. Consequently, the pruning condition always makes sure that $f \leq b - \Delta \leq N/w$ or $f \leq \epsilon N$. This is also the maximal loss on counting frequency of every itemset.

FDPM-Frequent Data Stream Pattern Mining: Based on a statistical tool, the Hoeffding bound [Wat00], Jeffrey *et al.* [YCLZ04] develop an algorithm named Frequent Data stream Pattern Mining (FDPM). Similar to Lossy Counting, a data stream is divided into segments and each segment is mined separately to find frequent patterns. All segments have the same size of $k \times n_0$ transactions, where n_0 is identified by the Hoeffding bound; k is a parameter used to control the number of transactions in each segment. If k is large, FDPM requires more memory space. In contrast, if k is small, the algorithm prunes potential infrequent patterns more frequently and thus requires more CPU time but less memory space. A data structure P is used to maintain potential frequent patterns for each segment; and F is a data structure to keep all these patterns in the data stream seen so far. A pattern X in P is pruned if its support is smaller than $(s - \epsilon_n) \times n$, where n increases from 0 to $k \times n_0$, and ϵ_n is computed from the Hoeffding bound. Its value approaches to 0 when more transactions are processed in the data stream. For this reason, ϵ_n is a running value instead of a parameter specified by users.

After pruning all infrequent patterns in P , the remaining ones are used to merge with patterns in F . If a pattern $X \in P$ is found in F then its counting value is incremented by the count of X in P . Otherwise, a new entry is created in the form of $\{X, c\}$ where c

CHAPTER 2. OVERVIEW OF DATA STREAM MINING

is the count value of X in P . When the size of $|F| > c_u \times n_0$, all potentially infrequent patterns from F are deleted. Here, we note that the value of c_u is identified through experiments. When a user requests the result, all patterns in F whose counting value greater than $s \times N$ are output. The drawback of this algorithm is that the upper bound of F is identified through experiments. The authors call this value the *empirical upper bound*. For example, if two datasets T10.I4.D1000K and T15.I6.D1000K are used by the algorithm, c_u is then determined from T15.I6.D1000K as it is more representative in comparison with T10.I4.D1000K. As indicated by the authors, it is still an open problem to study the issues related to identifying representative data streams.

FP-Streaming [GHP⁺03] is a typical algorithm addressing the problem of mining frequent sets from data streams with the forgetful window model. In this work, the stream is physically divided into batches and frequent itemsets are discovered separately from each other. Their frequencies are reduced gradually by using a fading factor $\Phi < 1$. For instance, let X be an itemset that has been found frequent in a set of consecutive batches $B_i, \dots, B_j$, where i, j are batch indices and $i < j$, then its frequency will be faded gradually by the formula $\sum_{k=i}^j \Phi^{j-k} f_X(B_k)$, where $f_X(B_k)$ is the number of X 's occurrences (frequency) in batch B_k . Note that since Φ is always smaller than 1, the older the B_k , the smaller the Φ^{j-k} value. Furthermore, in FP-Streaming, the frequency of an itemset will be maintained for every batch as long as it is still frequent; meanwhile the number of batches is increased with time. Therefore, in order to address the memory space constraint, a logarithm tilted-time window is utilized for each itemset in FP-Stream. It stores the itemset's frequency with fine granularity for recent batches and coarser granularity for older ones. Consequently, this approach not only requires less memory space but also allows one to find changes or trends in the stream (by comparing itemsets' frequencies between different periods of time frame).

FTP-DS (Frequent Temporal Patterns of Data Streams) [TCY03] is a typical algorithm to mine frequent sets from a time-based sliding window. In this approach, the

CHAPTER 2. OVERVIEW OF DATA STREAM MINING

stream is divided into several disjoint sub-streams where each one represents the transaction flow of one customer. Accordingly, an itemset's frequency is defined as the ratio between the number of customers that subscribe to it and the total number of customers in the sliding window. When the window slides, the frequency of each itemset is then tracked for each time period. In order to address the issue of memory space constraint, the least mean square (LMS) method is applied. Therefore, a linear function $f = a + b \times t$ is used as a summary form to represent each itemset's frequency history. Given time period t , the itemset's frequency in this period can be derived. The interesting point of this approach is that both coefficients a and b can be updated incrementally. For example, given $\{(t_s, f_s), \dots, (t_c, f_c)\}$ to be a set of time period and frequency count pairs for an itemset X from t_s to t_c , according to the LMS method, the sum of squared error (SSE) needs to be minimized:

$$SSE = \sum e_i^2 = \sum (f_i - a - b \times t_i)^2 \quad (\text{where } i = s, \dots, c) \quad (2.1)$$

By differentiating SSE with respect to a and b , the formulas to compute two coefficients are:

$$b = (\sum f_i t_i - q^{-1} \sum f_i \sum t_i) \times (\sum t_i^2 - q^{-1} (\sum t_i)^2)^{-1} \quad (2.2)$$

and

$$a = q^{-1} (\sum f_i - b \times \sum t_i) \quad (2.3)$$

where $q = t_c - t_s + 1$.

Note that two summations $\sum f_i t_i$ and $\sum f_i$ are easily updated incrementally at each time period t_i ; meanwhile $\sum t_i$ and $(\sum t_i)^2$ can be simply interpolated from t_s and t_c . Therefore, only 3 numbers $(t_s, \sum f_i t_i, \sum f_i)$ need to be maintained for each itemset, making the memory space independent from the data stream size. However, it is still an open

CHAPTER 2. OVERVIEW OF DATA STREAM MINING

problem to provide an error bound on the mining results when the LMS method is used in this algorithm.

Time Sensitive Sliding window [CLC05] is another algorithm proposed to mine frequent sets from a time-based sliding window. The window is defined as a set of time intervals during which the rate of arrival transaction flow can change over time. Accordingly, the number of transactions arriving in each time interval (thus the sliding window) may not be fixed. Potentially frequent itemsets are then uncovered from each time interval. Their local frequencies are stored in a counting frequency table. Unlike the methods presented above (e.g., Lossy Counting, FDPM) where the memory space is affected by the error threshold, this method directly fixes the memory space for the counting frequency table. Consequently, whenever this table is full, there is a need to adjust it in order to get space for newly generated itemsets. The adjustment can be achieved by merging two adjacent entries, which have frequency counting found in the same time interval, into one entry. This new entry may select either the smaller or the larger value (between two original entries) for its frequency counting. Due to this procedure, the mining results can only be either false-positive oriented or false-negative oriented. Furthermore, in this algorithm, the error in frequency counting of each output itemset cannot be precisely estimated and bounded.

estWin: Another algorithm addressing mining time change in a data stream is based on the sliding window [CL03a, CL04]. The authors suppose that users may be interested in frequent patterns in recent times. Stale patterns may not contribute or contribute slightly in current results. This algorithm, named *estWin*, mines all frequent itemsets in most w recent transactions, where w is the size of the window. During processing a data stream, the algorithm maintains itemsets in main memory by a monitoring lattice. The algorithm is divided into 5 phases: Phase 1 is transaction appending. When a new transaction arrives, it will be appended into a structure called *current transaction list*,

the CTL. In Phase 2, the counting updating, the counting value of itemsets that appear in the new transaction is updated in the monitoring lattice. In Phase 3, a new frequent itemset in the current transaction will be inserted into the lattice. In Phase 4, the oldest transaction is deleted. Therefore, the counting value of all itemsets containing in this transaction is decreased by 1 in the monitoring lattice. When a user requests results, Phase 5 is performed to produce all frequent itemset.

2.2.3 Discussions

We have presented and discussed most typical work addressing the problem of mining frequent sets from data streams in this section. Due to space limitation, many other algorithms [JA03, CL03b, JG06], including those in mining frequent items over data streams [DGIM02, DLOM02, CM05, KSP03], are not covered. We refer the readers to our survey paper [DNOL] as a good reference for the problem.

From the above discussion, it can be seen that most algorithms reported so far mostly focus on the problem of maintaining frequent itemsets within limited memory space. Some of them can further provide error guarantees on the mining results. Yet, a data stream must be processed in batches. This approach may be acceptable if applications do not require timely results. However, in some applications, it is desirable for transaction flows in data streams to be processed in a real-time manner and the results are available at any time instant. Another observation is that, most algorithms simply assume that mining systems are fast enough to manage all incoming transactions without delay or need to drop any. In other words, they all ignore the fact that CPU is also a bounded resource. In practice, for many stream applications such as stock monitoring or telecom fraud detection, the data arrival rate is rapid and mostly unpredictable (due to the dynamic changing behavior of data sources). Meanwhile, the mining results of these streams usually require delivery under time constraints to enable optimal decisions. Therefore, it

is necessary to properly address the problem of controlling CPU workload in the context of finding frequent sets from high speed streaming environments.

2.3 Clustering of Data Streams

2.3.1 Overview of Clustering Techniques

Cluster analysis is the process of grouping data objects into classes or clusters so that objects within a cluster have high similarity in comparison to one another, but are very dissimilar to objects in other clusters [DHS01]. In this section, we briefly review some fundamental algorithms used in cluster analysis. These techniques are then the background for data stream clustering analysis.

Partitioning methods A partitioning clustering algorithm tries to organize data objects into a pre-specified k partitions where each partition represents a cluster. Based on greedy heuristics, objects are relocated into k clusters in order to minimize an objective function. This process is often performed several times until some criteria are satisfied [Ber02, JMF99].

k -means: k -means [HW79] is one of the well-known partitioning methods. Given n data objects and the number of clusters k , a k -means algorithm tries to cluster n data objects into k clusters. With each cluster, the mean value is computed as the average value of data objects assigned in the cluster. At each iteration, data objects are re-assigned to the cluster to which they are the most similar based on the mean value of the clusters. Then, the mean value of each cluster is re-calculated. This process is performed iteratively until an objective function is satisfied. Typically, the square-error function is used $f = \sum_{i=1}^k \sum_{x \in C_i} |x - m_i|^2$ where x is a data object, m_i is the mean of cluster C_i .

k -medoids: Algorithms based on the k -means method are quite sensitive to noise. Especially, when there are some data objects with extremely large values appeared in

CHAPTER 2. OVERVIEW OF DATA STREAM MINING

the dataset, the mean of the cluster that these data objects belong to will be strongly affected. In such a case, the clustering quality is often poor. To relax this limitation, k -medoids algorithms, such as PAM and CLARA [KR90], are introduced. Basically, the idea of k -medoids algorithm is similar to the k -means algorithm. However, instead of choosing the mean value representing for each cluster, it chooses a specific data object as a representative, named *medoid*, for the cluster. Other data objects are then assigned to the nearest cluster by comparing its similarity with the medoid of each cluster. The objective function often used is the sum of dissimilarities of all the objects to their nearest medoid. This process is performed repeatedly by replacing a medoid with a new one if the result returned by the objective function is improved.

Hierarchical methods A hierarchical clustering method works by grouping data objects into a tree of clusters. There are two main approaches in hierarchical clustering methods: the agglomerative (bottom-up) and the divisive (top-down) approaches.

In the agglomerative clustering approach, initially each data object is a cluster. Then, at each iteration, these clusters are merged together in order to form larger ones. This process is repeated until a single cluster or a certain termination condition is satisfied. In contrast, in the divisive clustering approach, starting from all data objects in one cluster, it subdivides the cluster into smaller ones until each data object forms its own clusters or a certain termination condition is satisfied.

The result of hierarchical methods is often a dendrogram, this is a tree that represents how clusters are related. Most hierarchical clustering methods belong to the agglomerative hierarchical clustering, they differ only by the definition of the similarity (distance) between clusters [Eve93]. Some representative algorithms are CURE [GRS98] and Chameleon [KHK99].

Density based methods Algorithms approaching the clustering problem based on density usually aim at finding clusters of arbitrary shapes. They consider clusters as dense regions of objects. DBSCAN [EKSX96] is one such algorithm. By using two parameters to define what constitutes the neighborhood of a data object and whether its neighborhood is dense enough, DBSCAN grows regions with sufficiently high density into clusters. A cluster is identified as a maximal set of density-connected points and thus, it may be of any arbitrary shape. DENCLUE is another algorithm addressing the clustering problem from density-based approach. Different from DBSCAN, in DENCLUE the influence of each data object to its neighborhood is modelled by a mathematical function (such as multivariate normal function, square wave, etc.) named influence function. The overall density of the data space is then defined as the sum of the influence functions of all objects. Based on this, clusters are found by locating the local maxima, known as density-attractors, of the overall density function.

Model based methods Self-Organizing Map (SOM) introduced by Kohonen [Koh90] is one of the most well-known model-based technique. In this method, clustering is performed by having several units compete for a given data object which is represented by a numerical vector of d -dimension. The unit whose weight vector (also in d -dimension) is closest to the given data object becomes the winning unit. The weight vector of this winning unit, as well as those of its nearest neighbors, is then adjusted to move even closer to the input object. SOM assumes that there is some topology among the input data objects, and that its units will eventually take on this structure in space. The organization of SOM is said to form a feature map.

2.3.2 Clustering Data Streams

Similar to the problem of mining data streams for frequent patterns, clustering streaming data also has to deal with the problem of system resource constraints and the one-pass

CHAPTER 2. OVERVIEW OF DATA STREAM MINING

scanning data flow challenge. Furthermore, the problem of concept drift [Tsy02], which illustrates those situations where the change in data stream characteristics results in the change in clustering models [WK96], is often mentioned and addressed in stream clustering analysis.

CluStream [AHWY03] is one of the pioneering algorithms designed to address the issue of mining evolving data streams. Two new concepts, *micro-clusters* and *pyramidal time frame*, are proposed in this approach. A micro cluster, which is in fact an extension of Cluster Feature Vector in BIRCH [ZRL96], is a kind of synopsis used to maintain statistical information for a set of data objects (assigned to the micro cluster). It has a compact form of $(2d + 3)$ tuple $(\overline{CF2^x}, \overline{CF1^x}, CF2^t, CF1^t, n)$, where d is the dimension of data points. $\overline{CF2^x}$ and $\overline{CF1^x}$ are respectively the sum and squared sum of the data values for each dimension in the micro cluster. Both are d -dimensional vectors. $CF2^t$ and $CF1^t$ are the sum and squared sum of the time stamps respectively; n is the number of data points assigned to the micro cluster. With these statistical information, the centroid of each micro cluster can be easily computed.

In operation, CluStream always maintains a set of m -micro clusters in the data stream. Initially, the algorithm applies the k -means method on the first portion of the stream in order to get m micro clusters. For later phases when more and more data points arrive, CluStream tries to update these data points into this micro cluster set. Those that are too far from the centroid of each micro cluster will be either an outlier or a new micro cluster. They will be kept for a certain time in order to identify if they are truly an evolving micro cluster. Also, those micro clusters that are not updated for long time will be eliminated from the set of micro clusters. A snapshot is periodically taken in order to capture the status of micro clusters in the data stream. It is stored at level i of a pyramidal time table if the interval time taking the snapshot is exactly divisible for

CHAPTER 2. OVERVIEW OF DATA STREAM MINING

α^i and not divisible for α^{i+1} (where $\alpha > 1$ is an integer). For this reason, only a small number of snapshots need to be maintained at each level and snapshots captured far from the current time will be stored in higher levels and thus, with coarse granularity. Also, given a time horizon h and the current time t_c , the final k clusters in this time horizon can be derived from the set of q micro clusters found by subtracting the snapshot at time $t_c - h$ from the snapshot at time t_c .

With this approach, CluStream can provide the status of a data stream at different time horizons while the memory space (used to store snapshots) is only logarithmically proportional to the size of the data stream processed so far. As such, it can answer various queries regarding the evolving of the data stream (concept drift) by comparing the difference between maintained snapshots. However, its drawback is that, while taking k -means as the core algorithm for clustering data points, CluStream only guarantees its convergence in the initial phase when it derives the first m micro clusters. For the rest of the stream, there is no guarantee on the convergence since data points are permanently and immediately assigned to a micro cluster as they arrive on the stream. For this reason, the quality of cluster analysis can be reduced with time. The framework in CluStream is also later extended in HPStream [AHWY04a] where the authors address the problem of high dimensional data streams in a forgetful model.

Since CluStream employs k -means method in its design, clusters are often of spherical shapes. Motivated by this observation, various algorithms have been recently introduced approaching the stream clustering problem from density-based approaches. DenStream [CEQZ06] is such an algorithm. It extends the DBSCAN into stream environments by using a set of micro clusters. Each micro cluster has a center and a radius which defines its scope. Given a center c and radius r , a data point p is said to belong to the micro cluster if its Euclidean distance $dis(p, c) \leq r$. The number of data points falling into a micro cluster identifies its density level. Consequently, micro cluster can be

CHAPTER 2. OVERVIEW OF DATA STREAM MINING

divided into 3 types: the core one, the potential core one, and the outlier one corresponding to their density levels. A fading function is also employed in DenStream in order to reduce the density of each micro cluster with time. For clustering process, DenStream is quite similar to CluStream in that data points arriving in the stream are absorbed by the set of micro clusters. To derive the set of final clusters, DBSCAN is applied on these micro clusters to form density regions if micro clusters are dense enough and close to each other.

D-Stream With the same motivation, Yixin et al [CT07] recently proposed a method named D-Stream to cluster data streams in a forgetful window model. D-Stream is a grid-based method in that the data space is divided into a number of cells. Each dimension in the data space is divided into p partitions (or intervals). Therefore, given a data stream in d -dimensional, the number of cells can be up to p^d . Data points arriving in the stream can be updated online into a set of these cells since the time to locate a cell is generally linear given p and d fixed. The weight (or density) of each cell is defined by the number of data points falling into it. Furthermore, a fading function is also employed in D-Stream in order to reduce the weight of each cell with time. It is interesting to observe the relevance between this approach with the problem of finding frequent singleton items from data streams. We may view each cell as an item and the number of data points in the cell as the occurrence (or frequency) of that item. Consequently, reducing the weight of each cell is as simple as reducing the frequency of each frequent item (this problem will be shown clearly in Chapter 3). Based on the set of dense cells, clustering results can be simply derived by checking their connection in the grid. The positive side of this grid-based approach is that the algorithm is simple and effective. The time and space complexity of defining the grid, assigning each object to a cell and computing its density is only $O(m)$ where m is the number of data objects in the stream. However,

CHAPTER 2. OVERVIEW OF DATA STREAM MINING

in the negative side, it may suffer from the problem of “curse of dimensions”; i.e., when dimensionality increases, the number of potential grid cells increases exponentially. It can easily happen that most grid cells contain only a single object. Furthermore, it is hard to define the density threshold since if it is set too high, the clusters may be lost. In contrast, if the density threshold is set too low, two clusters that should be separated may be joined.

k -median clustering data stream In a more theoretical aspect, approaches based on the k -median in a metric space are proposed to cluster data streams [GMMO00, GMM⁺03, BDMO03, COP03, CMM02]. In k -median problems, the sum of the distance from each data point to its closest assigned median is expected to be minimized. Formally, let $\Omega = (Q, dist)$ be a metric space, where Q is a ground set; and $dist: Q \times Q \longrightarrow \mathbb{R}^+ \cup \{0\}$ be a distance function. Let M be a median set, and Y be the dataset that we want to cluster into k clusters, they are both the subsets of N . Then for all data points $x \in Y$, we denote $ASG(x, M) = \min_{m \in M} dist(x, m)$ as the closest member of M to x , and denote $AD(x, M) = dist(x, ASG(x, M))$ as the assignment distance of x , under the set M . Using these notations, the cost function of k -median problem can be defined as:

$$\Phi^Y(M) = \sum_{x \in Y} AD(x, M)$$

Thus, the target of the k -median problem is to find a set M that minimizes the cost function $\Phi^Y(M)$. The k -median problem can be further divided into two sub-problems: continuous k -median and discrete k -median. We call continuous k -median if the set M is chosen from space Q , which means it is not limited in Y . Otherwise, we call it discrete k -median, i.e., M is chosen from Y . To cluster stream data, a bicriterion (a, b) approximation algorithm is often used. In this approach, the produced set M has at most $a \times k$ medians and the cost function for that M is no more than b times the cost of the

optimal solution. In [GMMO00, GMM⁺03], an algorithm, named STREAM, for this k -median problem is introduced. It is a divide-and-conquer algorithm that builds clusters from data streams incrementally and hierarchically using a landmark window model. Later, this approach is extended in [COP03, CMM02] in which both the approximation factor b and the storage memory are improved. It is also extended to work in a sliding window model [BDMO03]; however, it also contains many drawbacks that we shall discuss in the following section.

2.3.3 Discussions

The characteristic of data streams often invokes three basic requirements in clustering algorithms: Compactness of representation; incremental processing of new data objects; and clear identification of outliers. Most algorithms reviewed in this section have tried to address these challenges at certain levels. However, most of them address the problem either in a landmark window model [GMMO00, GMM⁺03] or in a forgetful window model [CEQZ06, CT07] which take into account all data points seen so far in the stream. As has been shown in Section 2.1, the sliding window model is a more general mining model and thus, it is also more challenging since it concerns the elimination of outdated data points. Decisions are made based on only the most recently arriving data points in the stream (rather than based on the analysis of the entire data stream together). In [BDMO03], the authors have proposed a k -medians method to address the clustering problem. The window is divided into buckets which follow an exponential histogram (EH) [DGIM02]. Then each bucket of the stream is mined separately in order to derive a set of k local clusters. The local clusters in two consecutive buckets can be clustered again to derive another k local clusters in later phases whenever their two corresponding buckets are merged together due to relying on the exponential histogram technique. The final clustering results are then derived based on the set of local clusters summarized in

each bucket of the sliding window. There are some drawbacks with this method. First, it is not designed to directly address an evolving data stream in that its distribution can change over time. Knowledge obtained from previous buckets of the stream is not used in later phases and thus, each bucket is mined separately and independently regardless of how similar (or different) their distributions are. Second, both k -means and k -medians techniques are special cases of the EM technique [DLR77] in that each data point is assigned to only one cluster [HK01]. Such an approach often yields poorer results compared to a method based on an EM technique where each data point is assigned to all clusters yet with different level of membership. Finally, there are also no experimental results reported about how this method is working [BDMO03]. These observations are the main motivation for our work presented in Chapter 6 in that we present a truly incremental clustering algorithm in order to address the problem in a *time-based* sliding window model.

2.4 Other Work in Data Stream Processing

In this section, we briefly summarize some other prominent work in the area of data stream classification and data stream management systems. They are not directly related to our contributions. Nevertheless, these work help us to understand more about data stream environments and thus complete the entire picture in data stream mining and processing.

2.4.1 Data Stream Classification

The ultimate target of classification systems is to build a model that can learn and predict the class (or missing attribute value) of future data objects whose class (or value) may not be known. Generally, classification process includes two steps. In the first step, a model is built based on a set of training samples where the class label of each sample is

CHAPTER 2. OVERVIEW OF DATA STREAM MINING

known in advance. This step is often called the learning process. In the second step, the constructed model is used to categorize future data objects [Mit97]. Typically, learning models are represented in the form of classification rules [Kan03], decision trees [Mur98], or mathematical formulae [DH73, Dun03]. In the following section, we briefly review some typical approaches to classification in the context of mining stream data.

VFDT-Very Fast Decision Tree: VFDT [DH00] incrementally builds a decision tree from data streams by recursively replacing leaf nodes by decision nodes if there are sufficient statistics collected to support the decision. Specifically, at each leaf node, a count n_{ijk} representing the number of data points of class label k reaching the leaf node when the attribute j has the value i . When there are enough number of data points arriving at the leaf node, some heuristic evaluation function such as information gain (in C4.5) or Gini index (in CART) is used to evaluate whether this leaf node can be changed to a decision node and which attribute should be the best for the testing (e.g., the attribute having smallest entropy in case the information gain used). The new decision node takes the chosen attribute as a test condition and it has as many decision leaf nodes as the number of its possible values. The mathematical foundation of VFDT is based on the Hoeffding bound which allows one to estimate how many data points are necessary to observe before taking the splitting test at each leaf node. Consider a real-value random variable r whose range is R . Suppose n independent occurrences of this variable are observed, and their computed mean is r' . Based on the Hoeffding bound, the true mean of the variable is at least $r' - \epsilon$ with probability $1 - \delta$, where:

$$\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}}$$

VFDT can treat the difference between two (highest) heuristic measures $\Delta \bar{G} = \bar{G}(a) - \bar{G}(b) \geq 0$ of two attributes a and b at a node as a random variable. When n data points

have been seen at a node and $\Delta\overline{G}$ is greater than ϵ , it can be said that the true difference ΔG between two attributes a and b is truly larger than 0 with probability of $1 - \delta$. Hence, a should be chosen as the test attribute with confidence $1 - \delta$. VFDT is further developed in [GRM03] in order to address the problem of data stream classification with continuous attributes. It is also extended to mine continuously-changing stream data [HSD01] in a sliding window model.

Ensemble classifiers: A general framework for mining concept-drifting data streams using weighted ensemble classifiers is introduced in [WFYH03]. This work has argued that utilizing a single model to learn concepts does not give a good classifying result. Instead, getting average classifying results learned by many classifiers is often better. A set of classifiers may be in any form of decision trees, Naive networks, or Ripper, and so on. The goal of ensemble classifiers is to build the top k -classifiers that can learn new concepts most correctly. Whenever a new chunk of data points arrives in the stream, a new classifier is created for that portion. This new data chunk is also used to test and update the weights of the old classifiers in the ensemble. The weight of each classifier is updated based on the expected error of that classifier in classifying new data chunk. It is often inversely proportional to this error value. Then, top k classifiers that give the most correct results are chosen to be included in the classifier ensemble. This approach has an advantage that it can produce better classifying results when compared to those returned by single classifier models (such as VFDT [DH00]). However, the learner model is quite complicated as it has to maintain a set of k classifiers. Moreover, classifying results are quite sensitive to the size of data chunks as well as the number of classifiers in the ensemble. These parameters are often not easy to identify, practically.

StreamMiner: StreamMiner is a classifier system that can detect and learn changes in data streams automatically [Fan04, FaHWY04]. The system relies on the decision tree

CHAPTER 2. OVERVIEW OF DATA STREAM MINING

model. There are two types of changes that the StreamMiner can learn: concept drift and distribution drift. Different from other learning systems, StreamMiner does not need the labelled data in new data stream immediately. It only needs them when some change is detected. To identify changes, it is based on statistically estimated loss measures. In detail, firstly, the model based on the decision tree is built from a training dataset. When processing new stream data, the system compares the statistical value at each leaf in the tree to detect the change. Let dt be a decision tree created from training dataset D ; and S be a data stream. Assume that n_ℓ is the number of data points classified by leaf ℓ ; and N is the size of the data stream seen so far. The statistics at leaf ℓ is defined as $P(\ell) = n_\ell/N$.

Clearly, the sum of $P(\ell)$ over all leaf nodes in the tree dt is equal to 1. $P(\ell)$ describes how the instance space of the stream S is distributed among the leaf nodes solely based on attribute test results of the given decision tree dt . It does not consider either the true class labels or attributes that are not tested by dt . Now the statistical difference between the training dataset D and the stream data S can be identified via $P(\ell)$ as follows:

$$PS = \frac{\sum_{\ell \in dt} |P_S(\ell) - P_D(\ell)|}{2} \times 100\%$$

where $P_D(\ell)$ is statistics at leaf ℓ on the training dataset D ; and $P_S(\ell)$ is statistics at leaf ℓ on the data stream S .

This PS is called *error rate* or *guessed loss* of the model on the new stream data. As such, StreamMiner does not need the labelled data in the data stream for identifying changes. Instead, it compares statistical difference at each leaf node in the model. If this PS value is much higher than an application-specific tolerable maximum value, a small number of data records in the new data stream is chosen to investigate their *true class* labels. With these true class labels, the *true loss* of current decision tree is statistically estimated. If this value is also higher than the tolerable maximum value, the decision

tree should be reconstructed by using the same true class labels sampled in the previous step.

2.4.2 Data Stream Management Systems

Generally, a traditional database management system (DBMS) stores data in a set of relations, then operations such as insertions, updates, deletions and queries are executed over the relations. When processing data in stream environments, it is not feasible to simply load the arriving data into a traditional DBMS and operate on it there. Traditional DBMS's are not designed for rapid and continuous loading of individual data instance. It has the requirement to build a new class of data management systems, the data stream management system (DSMS), in order to manage and process streaming data [BBD⁺02, GO03]. In this section, we highlight the main points that make a DSMS different from a conventional DBMS. Then, we briefly describe some fundamental techniques in summarizing data streams.

We distinguish a traditional database management system from a data stream management system by the following properties [MWA⁺03]:

- **Data storage:** In a traditional database management, data are often managed in persistent relations or tables. In a data stream management, data appear in transient streams and may not be stored in any repository. Generally, the streaming data are often stored and processed in main memory only.
- **Query accuracy:** A traditional database management provides exact answers to specific queries and is blind to real-time deadlines. In contrast, a data stream management must return results in a timely manner. Therefore, it provides only approximate results to queries.

CHAPTER 2. OVERVIEW OF DATA STREAM MINING

- **Data access:** In a traditional database management, queries are often executed in one-time manner with a random access over datasets. As opposed, in a data stream management, queries are executed in a continuous manner with a sequential access over streaming data.
- **Granularity:** In a traditional database management, data can be stored and summarized in any granularity. In contrast, streaming data often arrive at a fine granularity, thus, there is no summary in advance in a data stream management.
- **Query planning:** In a traditional database management, access plans are often determined by a query processor and they can be designed beforehand based on the physical database. In a data stream management, as opposed, the arrival of streaming data is unpredictable and variable. Therefore, query plans are often hard to design in advance.

Due to these differences, processing data streams requires building new techniques to summarize data into different types of synopses [GM99, GO03, Mut03]. Then, a variety of queries (selections/projections, joins, aggregations) can be performed over these synopses. Commonly used techniques to produce synopses are histograms, wavelets, sketches, and sampling.

Histograms Histograms are one of the most popular data reduction techniques [BDF⁺97, BBD⁺02]. They use binning methods for approximating data distributions to minimize the information stored. A histogram technique often partitions a data distribution into a certain number of β disjoint buckets (or intervals) $B_j; 1 \leq j \leq \beta$. Given a data distribution T , it is often represented by $T = \{(v_1, f_1), (v_2, f_2), \dots, (v_d, f_d)\}$, where v_i denotes for a distinct value in T and f_i denotes for its corresponding frequency.

CHAPTER 2. OVERVIEW OF DATA STREAM MINING

In histogram techniques, the distribution of values in a bucket usually takes the uniform spread assumption, that is, the values are assumed to locate at equal distance from each other. Consequently, one may need to store only the minimum and maximum values together with the number of values in the buckets. The other values can be approximately derived according to this assumption. We present some of the most common histograms that are often used in data stream processing [Ioa03, IP95]:

- *Equi-width histograms*: In these histograms, the data distribution is partitioned into buckets such that the number of v_i values falling into each bucket is uniform across all buckets.
- *Equi-height histograms*: These histograms partition the data distribution into buckets based on the summation of the frequencies. Generally, this value is the same for every bucket.
- *Maxdiff (Max difference) histograms*: In this type of histograms, the data distribution is partitioned in such a way that the differences in frequencies between adjacent bucket boundaries are maximized.
- *V-optimal histograms*: If n_j denotes the number of entries in the bucket j^{th} ; $\overline{f_j}$ the average frequency of that bucket; $f_{j,k}$ the frequency of k^{th} entry in j^{th} bucket, then these histograms partition data distribution T such that the value $\sum_{j=1}^{\beta} \sum_{k=1}^{n_j} (\overline{f_j} - f_{j,k})^2$ is minimized. In other words, we know that $\sum_{k=1}^{n_j} (\overline{f_j} - f_{j,k})^2$ is the variance of j^{th} bucket, hence, a V-optimal histogram is the one with the least (smallest) variance of all histograms with the same number (β) of buckets.
- *Compressed histograms*: In these histograms, most n ($n < \beta$) frequent values are stored separately in n singleton buckets. The remaining values are partitioned into $(\beta - n)$ buckets using Equi-width or Equi-height histograms.

Wavelet transformation The *discrete wavelet transformation* (DWT) is a linear signal processing technique that is well suited for data reduction [BDF⁺97, GGR02]. From a numerical dataset, by applying DWT, one can transform it into a numerically different dataset of wavelet coefficients with the same length. However, in the new wavelet transformed dataset, only coefficients that are higher than a user-specified threshold are retained. They are selected as a compact representation of the original dataset. Also, from these compressed wavelet coefficients, it is easy to re-construct the original dataset without losing much information. In data stream management systems, wavelet based methods are often used for approximating queries [GKMS03, MS04] whereas in data stream mining, they are often used for burst detection [ZS03].

Sampling techniques The intuitive idea behind sampling techniques is that a set of random samples can be used as a summary structure with the expectation that it can capture the essential characteristics of the original dataset. Sampling techniques are often used extensively in processing large databases [MTV94, ZPLO97]. In stream environments, one major challenge for these approaches is that most sampling techniques are applied for finite datasets whereas the size of stream data is potentially unbounded. Vitter [Vit85] has introduced various algorithms to sample m records (without replacement) from a pool of M records, where the value of M is unknown a priori. These methods have been adapted for sampling in stream processing with replacement [POSG04], and later it is extended to sample in a sliding window model [BDM02].

To summarize this section, we introduce some typical projects on DSMSs. Some of them have also been deployed in real-life applications [ACc⁺03, TcZ⁺03], the others are still being developed and studied.

- **Aurora** (Brown/MIT) is a DSMS built for stream monitoring applications. In Aurora, data streams are collected from sensors, satellites or stock feeds and they

CHAPTER 2. OVERVIEW OF DATA STREAM MINING

are analyzed by using various techniques including aggregation, reduction and correlation analysis in order to detect different abnormal activities [CBB⁺03].

<http://www.cs.brown.edu/research/aurora>.

- **STREAM** (Stanford) is a general-purpose DSMS focusing on querying data streams and memory management [MWA⁺03]. Besides DSMS building, this project is also extended to a number of stream mining tasks such as clustering [GMM⁺03], frequent patterns mining [MM02].

<http://www-db.stanford.edu/stream>.

- **Cougar** (Cornell) is a project developed for querying data streams from sensor networks. A query layer is built to process declarative queries. A declarative query is a kind of query that a user may issue without knowing how the results are generated, processed and returned [YG03]. <http://www.cs.cornell.edu/database/cougar>.

- **Hancock** (AT&T) is a project aimed at building a domain-specific language, named Hancock, to compute signatures (i.e., evolving profiles of customers) over massive data streams such as telecommunication streams, stock market transaction streams, or credit card transaction streams [CFPR00, CFP⁺04].

- **NiagaraCQ** (OGI/Wisconsin) is a project focused on continuous queries over internet XML stream data. This project is motivated by the observation that many Web queries often share similar structures, thus, grouping queries can reduce computation and I/O cost significantly [CDTW00]. <http://cs.wisc.edu/niagara>.

- **TelegraphCQ** (Berkeley) is a DSMS developed for querying in sensor networks. This DSMS mainly focuses on building continuously adaptive queries over stream sensor network data [Cha03]. <http://telegraph.cs.berkeley.edu>.

CHAPTER 2. OVERVIEW OF DATA STREAM MINING

- **Traderbot** is a Web-based financial search engine that evaluates (both one-time and continuous) queries over real-time streaming financial data such as stock tickers or news feeds [BBD⁺02]. <http://www.traderbot.com>.

Chapter 3

Online Approximate Frequent Pattern Mining with Precise Error Guarantee

3.1 Motivation

As we have presented in Chapter 1 and 2, the volume of stream data is generally unbounded meanwhile the need to analyze it in real-time over limited computing resource is important in a number of data stream applications. Consequently, it is imperative to design algorithms that are capable of computing answers over data streams in an online fashion with only one scan on the data whilst operating under resource limitations. As such, it is not possible to compute an exact answer for complex queries often found in data analysis. Fortunately, it has been found that approximate answers are usually sufficient for these applications. However, while approximates are desirable in the context of data stream applications, it is important to remember that the quality of the approximation is equally important as the time of delivering answers. Let us consider the case of processing financial data streams. Clearly, if a poor approximation were to exist in the approximate answers, there would be serious financial consequences despite delivering the results within the given timeframe.

From the work reviewed in Chapter 2 (e.g., [MM02, GHP⁺03, YCLZ04]), we have

CHAPTER 3. ONLINE APPROXIMATE FREQUENT PATTERN MINING WITH PRECISE ERROR GUARANTEE

realized that the quality of the approximate answer is often governed by an error parameter. Although this defines the allowable error margin in the approximate answer, we observed that this is only the case when the algorithms analyze the data stream in batch mode. In other words, algorithms that are capable of online processing do not conform strictly to a precise error guarantee. Since online processing is essential to these applications *and* so is the precision of the error guarantee, it is essential that algorithms for data streams meet both criteria. Yet, this is not the case especially on the discovery of frequent sets in data streams.

We recall that the goal of discovering frequent sets is to find the set of collections of items (or any objects), whose occurrence count is at least greater than a certain threshold based on a fraction of the stream seen so far. Given a domain of m distinct items, there are 2^m distinct collections (or itemsets) that may appear in the data stream. In practice, given limited space and the need for real-time analysis, it is impossible to enumerate and count all itemsets for each of the incoming transactions in the stream. Hence, the downward closure property [AS94] is usually exploited and longer itemsets are generated and counted only after all their subsets are found frequent in the stream [CL03a, CL03b, TCY03]. This delay in generating longer candidates implies that there are larger error margins on itemsets of longer sizes. Consequently, it is impossible to guarantee the same error threshold for frequent itemsets of different sizes.

In this chapter, we present EStream, a proposed method to approximate frequent sets along with their estimated supports and error guarantees. Underpinning the design of our EStream algorithm is the ability to give a precise error estimation of the support of frequent sets based on their length *during online processing*. Specifically, our proposed EStream algorithm ensures that (i) there is no error in counting 1-itemsets; (ii) for 2-itemsets, the error in the estimate is no more than ϵ ; and (iii) for itemsets of length $k > 2$, the error is no more than 2ϵ , where ϵ is the preset error threshold and k is

CHAPTER 3. ONLINE APPROXIMATE FREQUENT PATTERN MINING WITH PRECISE ERROR GUARANTEE

the maximal length of the frequent set in the stream. Furthermore, to improve the efficiency of EStream, we propose a trie-like data structure called *STrieIT* (Significant Trie Itemset). This data structure is not only used to maintain potentially frequent sets, it also enables our algorithm to count the frequency of each itemset separately in each portion of the stream, which in turn helps to minimize the generation of redundant itemsets.

This chapter is organized as follows. In the next section, we introduce the preliminaries of discovering frequent sets in the context of data streams together with the problem formulation. Section 3.3 describes our approach. The data structure is introduced first, then the EStream algorithm is represented, followed by the error analysis. Section 3.4 reports our experimental results. The discussion on EStream method is then presented in Section 3.5. In Section 3.6, we discuss the problem of extending EStream into a forgetful mining model and the FStream algorithm is introduced. Finally, our conclusions on the chapter are presented in Section 3.7.

3.2 Problem Formulation

We address the problem of mining frequent sets from landmark window models. As introduced in Chapter 2, in this model frequent sets will be discovered from a set of data transactions in the stream identified between a specific transaction in the past and the current transaction. In [GKS01], Gehrke *et al.* have shown that there are many high speed and online data stream applications using this model in their stream computation. A telecom transaction data stream is such an example where the scope of the stream is captured on a daily basic. Consequently, a set of time points (landmarks) is given to the mining system and the results are computed from the current moment back to the immediately-preceding time point.

CHAPTER 3. ONLINE APPROXIMATE FREQUENT PATTERN MINING WITH PRECISE ERROR GUARANTEE

Therefore, with this landmark mining model, we define the following concepts for the problem of discovering frequent sets from streaming data:

Let $I = \{a_1, a_2, \dots, a_m\}$ be a set of literals called *items* (or objects). Let $\mathcal{DS}$ be a transactional data stream, which is a sequence of continuously incoming transactions $\mathcal{DS} = \{t_1, t_2, \dots, t_N, \dots\}$, where each transaction t_i contains a set of items such that $t_i \subseteq I$, and N is the current length (or timestamp) of the data stream. We define the following concepts.

Definition 3.1 *An itemset X is a set of items such that $X \in (2^I - \{\emptyset\})$. An ℓ -itemset is an itemset that contains exactly ℓ items. For a given itemset $X \subseteq I$ and a transaction t_i , we say that t_i supports (or contains) X if and only if $X \subseteq t_i$.*

Definition 3.2 *Given an itemset X of size ℓ (i.e., an ℓ -itemset), a set of all its subsets, denoted by the power set $\mathcal{P}(X)$, is composed of all possible itemsets that can be generated by one or more items of the itemset X ; i.e., $\mathcal{P}(X) = \{Y | Y \in (2^X - \{X\}) \wedge Y \neq \emptyset\}$. An immediate subset Y of X is an itemset such that $Y \in \mathcal{P}(X)$ and $|Y| = (\ell - 1)$.¹*

Definition 3.3 *The cover of an itemset X in the data stream $\mathcal{DS}$ is defined as the set of transactions seen so far in the stream that support X :*

$$\text{cover}(X, \mathcal{DS}) = \{t_i | t_i \in \mathcal{DS}, X \subseteq t_i\}$$

Definition 3.4 *Let N be the number of transactions seen so far in $\mathcal{DS}$. Then the frequency of an itemset X , denoted by $\text{freq}(X)$, is defined as the number of transactions in $\mathcal{DS}$ that contain X :*

$$\text{freq}(X) = |\text{cover}(X, \mathcal{DS})|$$

¹We use the notation $|Y|$ to denote the number of items in Y .

CHAPTER 3. ONLINE APPROXIMATE FREQUENT PATTERN MINING WITH PRECISE ERROR GUARANTEE

The support of itemset X , denoted by $\text{supp}(X)$, is defined as the ratio of the frequency of X to the number of transactions seen so far:

$$\text{supp}(X) = \frac{|\text{cover}(X, \mathcal{DS})|}{|\mathcal{DS}|} = \frac{\text{freq}(X)}{N}$$

Let σ be a given threshold called *minimum support*: $\sigma \in (0, 1]$. Let ϵ be a given error parameter such that $\epsilon \in (0, 0.5\sigma]$. We define three types of itemsets as follows:

Definition 3.5 *An itemset X is called a frequent pattern at the point of output timestamp N if $\text{supp}(X)$ is no less than σ ; it is called an infrequent pattern if $\text{supp}(X)$ is no more than 2ϵ ; otherwise, it is called a sub-frequent pattern. Both frequent and sub-frequent patterns are called significant (or potentially frequent) patterns.*

It is important to note that, both the size of the data stream and the frequency of an itemset X are defined as absolute values. Thus, they increase when more transactions arrive on the stream. Meanwhile, the support of an itemset X and the minimum support threshold are defined as relative values which are proportional to the size of the data stream.

We define our problem as follows: Given three user-defined parameters σ , ϵ , k (the maximal length of expected patterns, $k > 2$), our goal is to develop an online algorithm that, at any time instant, discovers all frequent sets of size up to k from a transactional data stream and guarantees:

- (1) no error in the support of 1-itemsets;
- (2) an error of no more than ϵ on the support of 2-itemsets;
- (3) an error of no more than 2ϵ on the support of k -itemsets.

3.3 Online Patterns Mining with Error Guarantee

In this section, we present our proposed EStream algorithm in order to address the problem formulated in Section 3.2. We first describe our data structure in Section 3.3.1 that will be used as a synopsis to capture potentially frequent sets in our algorithm. Then, in Section 3.3.2, we present our algorithm for processing data streams where the support errors in output frequent sets are guaranteed within the given threshold. Finally, the correctness of this proposed algorithm is theoretically proved in Section 3.3.3.

3.3.1 Data Structure

To capture potentially frequent sets during the online mining process, we use a trie-like data structure. A *trie* is a tree structure whose organization is based on key-space decomposition. In key space decomposition, the key range is equally subdivided and the splitting position within the key range for each node is predefined. An *alphabet trie* is a trie used to store a dictionary of words. In EStream, we adopt a variation of the alphabet trie.

Let the set of items $I = \{a_1, a_2, \dots, a_m\}$ be ordered so that for any two items $a_i \in I, a_j \in I$ ($1 \leq i, j \leq m$), $a_i \prec a_j$ if and only if $i < j$. Likewise, every transaction $t_i \subseteq I$ is also ordered with respect to the ordering in I . We now define the following:

Definition 3.6 A **TrieIT** (a.k.a. *Trie Itemset*) is a set of tree nodes where each node v is a 2-tuple (v_ℓ, v_c) such that $v_\ell \in I$ is the node's label and v_c is the frequency. Since each node corresponds to an item $a_i \in I$, we also use v_i (for brevity) to refer to a node that corresponds to $a_i \in I$. Then, the following conditions hold:

- (i) Let $C(v_i)$ be the ordered set of children nodes of v_i . If $C(v_i) \neq \emptyset$, then $C(v_i) \subseteq \{v_{i+1}, v_{i+2}, \dots, v_m\}$.

CHAPTER 3. ONLINE APPROXIMATE FREQUENT PATTERN MINING WITH PRECISE ERROR GUARANTEE

- (ii) Given a node v_i , let $v_\ell, v_{\ell+1}, \dots, v_{i-1}$ ($1 \leq \ell \leq i-1$) be the set of nodes on the path from the root to the parent of v_i , then v_c is the frequency of the itemset $\{a_\ell, a_{\ell+1}, \dots, a_i\}$.

Each TrieIT W_i corresponds to some $a_i \in I$ such that the root node has label a_i .

Based on the definition of a TrieIT, we define a STrieIT as follows:

Definition 3.7 A **STrieIT** (a.k.a. *Significant TrieIT*) is a set of TrieITs that maintains the set of significant itemsets found in the stream. It has a special root node that references the root node of each TrieIT. Except for the root node, each node in the STrieIT is a 4-tuple $\langle v_i, f_c, f_p, w \rangle$ where $v_i \in I$ is the node label; f_c is the frequency of the itemset (formed by the set of node labels from root to v_i) in the current window;² f_p is its accumulated frequency before the current window; and w indexes the window at which the itemset is inserted into the STrieIT.

An example of STrieIT is shown in Figure 3.1.

It is important to note that STrieIT is different from the FP-tree data structure proposed in [HPY00]. It is designed to support the online processing of transactions arriving in the stream. Each transaction in the stream and each itemset maintaining in the tree are all ordered with respect to the order in I . This helps to easily identify if an itemset in the tree is also appearing in a mining transaction. On the other hand, as presented in [HPY00], it is necessary to scan a dataset twice in order to build an FP-tree. The first scan is used to separately identify the frequency of each individual item in the dataset. All items are then ordered in descending frequency. Accordingly, items in each transaction are re-arranged with respect to this descending order. This task helps to build the FP-tree in the second scan of the dataset with more frequently occurring items

²The concept of ‘window’ will become clear when we present our EStream algorithm in the next section.

Figure 3.1: An example of STrieIT data structure

being arranged closer to the root of the FP-tree. Due to this two-phase approach, it is very hard to build an FP-tree online with only single scan on the data stream.

3.3.2 Algorithm

Given the above concepts and the data structure, in this section, we describe the EStream algorithm that efficiently discovers all frequent sets over a stream of transactions and guarantees that the error on the support of each frequent pattern is within some pre-specified error threshold. Its pseudo code and related procedure are shown in Figures 3.2 and 3.3. EStream requires three parameters: minimum support threshold σ , error parameter ϵ , and maximal length k of expected frequent patterns. The algorithm also utilizes a STrieIT data structure $\mathcal{S}$ as a synopsis in order to maintain all potentially frequent itemsets.

EStream processes each transaction on the fly and discards it immediately from memory after processing. As each transaction arrives, the algorithm inserts each potentially frequent itemset discovered from the transaction into $\mathcal{S}$. After processing each Δ transactions, the algorithm scans $\mathcal{S}$ to remove those itemsets that now become infrequent.

CHAPTER 3. ONLINE APPROXIMATE FREQUENT PATTERN MINING WITH PRECISE ERROR GUARANTEE

In order to guarantee the error stated in Section 3.2, Δ is determined based on the ϵ error and the maximal length k of expected frequent patterns $\Delta = \lceil 2^{k-2} \times \epsilon^{-1} \rceil$. It also defines the size of the *conceptual* window. Since transactions are processed one by one, each window provides a conceptual grouping of the transactions and a logical time for scanning $\mathcal{S}$.³

For itemsets of length ℓ ($1 \leq \ell \leq k$), we identify a corresponding minimal frequency threshold. This threshold is used to specify whether an ℓ -itemset can be generated as a potentially frequent itemset in a (conceptual) window. More specifically, an ℓ -itemset is generated and inserted into $\mathcal{S}$ *only after* the frequencies of all its immediate subsets in the window are above this threshold. Also due the reason of error guarantee, these thresholds are defined based on the length ℓ of each itemset. Specifically, we use an array *arr* of k elements to store them. The value stored in the j^{th} element of the array is defined by the following equation:

$$arr[j] = 2^{k-2} \sum_{i=2}^j (1/2^{i-2})$$

for $2 \leq j \leq k$ and $arr[1]$ is set to 0 for 1-itemsets since they have no subset except the trivial one $\{\emptyset\}$.

The following example illustrates how Δ and array *arr* are computed based on the given the values of ϵ and k , and the condition for which an itemset of size j is generated at each window.

Example 3.1 Suppose $k = 4$ and $\epsilon = 25\%$. Then, by definition, $\Delta = \lceil 2^{k-2} \times \epsilon^{-1} \rceil = 16$ and $arr[1] = 0$, $arr[2] = 4$, $arr[3] = 6$, and $arr[4] = 7$. This means that a 2-itemset found in a transaction will be inserted into $\mathcal{S}$ if all its 1-itemsets have a frequency of at

³Consequently, it can be understood that the data stream is divided into a set of conceptual windows (or windows for short), each having size of Δ transactions. Also, this window should not be mistaken with the landmark window.

CHAPTER 3. ONLINE APPROXIMATE FREQUENT PATTERN MINING WITH PRECISE ERROR GUARANTEE

least 4 in the window. Similarly, for a 3-itemset to be inserted into $\mathcal{S}$, the frequencies of all its immediate subsets must be at least 6. And for a 4-itemset, the frequencies of all its immediate subsets must be at least 7.

From the above description, we now present the detail steps of EStream. Let us denote the index of the current window by $w_c = \lceil n/\Delta \rceil$. Whenever a new transaction t_n arrives, the algorithm processes it as follows:

Increment: If an itemset X appearing in t_N is also maintained in $\mathcal{S}$, then its frequency in f_c is increased by 1.

Insert: For each $X \subseteq t_N$ not in $\mathcal{S}$, insert X into $\mathcal{S}$ with initial values of $\langle X, 1, 0, w_c \rangle$ if X is a singleton⁴; otherwise, let Y be any immediate subset of X , then X is inserted into $\mathcal{S}$ if the three conditions below hold:

- Y is in $\mathcal{S}$;
- If $f_p(Y) \neq 0$ then $f_c(Y) > arr[|X|]$; i.e., for those subsets Y being inserted into $\mathcal{S}$ from previous windows, their frequency in the current window must be sufficiently significant;
- If $f_p(Y) = 0$ then $f_c(Y) > (arr[|X|] - arr[|Y|])$; i.e., for those Y just being inserted into $\mathcal{S}$ in the current window, their frequency must be at least $(arr[|X|] - arr[|Y|])$.

Here we assume that, when processing current transaction t_N , the frequency of all X 's immediate subsets has already been increased once before the algorithm checks above conditions for X insertion.

⁴Recall that the immediate subsets of a 1-itemset is $\{\emptyset\}$ which appears in every transaction, all 1-itemsets are therefore inserted into $\mathcal{S}$ without conditions. For the same reason, they are also not pruned from $\mathcal{S}$.

CHAPTER 3. ONLINE APPROXIMATE FREQUENT PATTERN MINING WITH PRECISE ERROR
GUARANTEE

Algorithm EStream: Find frequent sets from transactions sighted in $\mathcal{DS}$.

Input

$\mathcal{DS}$: data stream of transactions;
 $\sigma \in (0, 1]$: minimum support threshold;
 $\epsilon \in (0, 0.5\sigma]$: maximum error allowable in σ ;
 k : the longest frequent patterns to find;

Output

At anytime on demand, return all estimated frequent sets
seen so far in the stream

Method

```

1:  $\Delta = \lceil 2^{(k-2)} \times \epsilon^{-1} \rceil$ ;  $w_c = 1$ ;  $arr[1] = 0$ ;
2: for  $\{j = 2, j \leq k, j++\}$  do  $arr[j] = 2^{k-2} \sum_{i=2}^j (1/2^{i-2})$ ;
3: for each  $t_N \in \mathcal{DS}$  do
4:   for each 1-itemset  $X \in t_N$  do
5:     if  $X \in \mathcal{S}$  then
6:        $f_c(X)++$ ;
7:     else
8:       Insert  $(X, 1, 0, w_c)$  to  $\mathcal{S}$ ;
9:     end if
10:  end for
11:  for each itemset  $X \subseteq t_N$  do
12:    if  $X \in \mathcal{S}$  then
13:       $f_c(X)++$ ;
14:    else
15:      if Check_Subset( $X$ ) then
16:        Insert  $(X, 1, 0, w_c)$  to  $\mathcal{S}$ ;
17:      end if
18:    end if
19:    if  $X$  cannot be inserted into  $\mathcal{S}$  then
20:       $t_N = t_N - \{\mathcal{X} \subseteq t_N | \mathcal{X} \text{ is a superset of } X\}$ 
21:    end if
22:  end for
23:  if  $N \equiv 0 \pmod{\Delta}$  then
24:    for each itemset  $X \subseteq t_N$  do
25:      if  $(|X| \geq 2)$  and  $(f_c(X) + f_p(X) \leq (w_c - w(X)) \times arr[|X|])$  then
26:        Remove  $X$ 
27:      else
28:         $f_p(X) = f_p(X) + f_c(X)$ ;
29:         $f_c(X) = 0$ ;
30:      end if
31:    end for
32:  end if
33:  if mining results are requested then
34:    return  $\{X | f_c(X) + f_p(X) + w(X) \times arr[|X|] \geq \sigma \times N\}$ ;
35:  end if
36: end for
```

Figure 3.2: EStream Algorithm

CHAPTER 3. ONLINE APPROXIMATE FREQUENT PATTERN MINING WITH PRECISE ERROR GUARANTEE

```

Procedure Check_Subset( $X$ : an itemset)
1: for (all  $Y \subset X$ ) do
2:   if  $Y \notin \mathcal{S}$  then
3:     return FALSE; //  $Y$  is not in  $\mathcal{S}$ 
4:   end if
5:   if ( $f_p(Y) > 0$  and  $f_c(Y) \leq \text{arr}[|X|]$ ) then
6:     return FALSE; // frequency of  $Y$  is not significant
7:   end if
8:   if ( $f_p(Y) = 0$  and  $f_c(Y) \leq \text{arr}[|X|] - \text{arr}[|Y|]$ ) then
9:     return FALSE; //  $Y$  appeared in previous windows but its frequency is not significant enough
10:  end if
11: end for
12: return TRUE;

```

Figure 3.3: Procedure for checking the frequency of all subsets of an itemset X in the landmark window model.

Example 3.2 We continue with Example 3.1 to illustrate the above. Suppose after processing the previous window, we have the following patterns in $\mathcal{S}$: $\{a\}, \{b\}, \{c\}$, and $\{b, c\}$. Furthermore, assume that some transactions have been processed in the current window and now t_N contains $\{a, b, c\}$. To insert this itemset, we check the frequencies of its immediate subsets; i.e., $\{a, b\}, \{a, c\}$, and $\{b, c\}$. Since $\{a, b\}$ and $\{a, c\}$ are not maintained in previous windows, their frequencies in the current window must be at least 2 (i.e., $\text{arr}[3] - \text{arr}[2]$). Of course, before that all their 1-itemsets $\{a\}, \{b\}$ and $\{c\}$ must have attained sufficient frequency of 4 (in the current window) for $\{a, b\}$ and $\{a, c\}$ to be in $\mathcal{S}$. On the other hand, as $\{b, c\}$ has been maintained from previous windows, its frequency in the current window must be at least 6 (i.e., $\text{arr}[3]$).

In cases where X is not inserted into $\mathcal{S}$, all its supersets in t_N need not be further checked.

Prune: This step is invoked each time $N \equiv 0 \bmod \Delta$ and after t_N is processed. The algorithm prunes $\mathcal{S}$ by removing all but 1-itemsets that satisfy $f_c(X) + f_p(X) + w(X) \times \text{arr}[|X|] \leq w_c \times \text{arr}[|X|]$. Consequently, if an itemset is removed, all its supersets are

CHAPTER 3. ONLINE APPROXIMATE FREQUENT PATTERN MINING WITH PRECISE ERROR GUARANTEE

also removed. An exception are those itemsets recently inserted into $\mathcal{S}$ in the current window. These itemsets are generated after their immediate subsets became sufficiently frequent. Also in this step, for each remaining itemset X , its accumulated frequency is updated by $f_p(X) = f_p(X) + f_c(X)$ and then reset $f_c(X) = 0$.

At any instant upon the request of the analyst, the algorithm scans $\mathcal{S}$ to produce all 1-itemsets satisfying $f_c(X) + f_p(X) \geq \sigma \times N$ and those ℓ -itemsets satisfying $f_c(X) + f_p(X) + w(X) \times arr[X] \geq \sigma \times N$.

It is important to note that, to generate a new candidate X , EStream does not rely on the frequencies (counting so far in the stream) of all its subsets. Instead, only the frequencies of these subsets counting in the current window are considered. This can be understood by the fact that frequencies of itemsets are not uniformly distributed in all parts of the stream. For some itemsets whose occurrences in some parts of the stream may not be sufficiently frequent, they are still always globally frequent in the stream since their occurrences in the other parts of the stream are very high. If we always treat them as frequent ones, it is possible that we may need to generate their longer supersets according to the downward closure property.

Yet, the frequencies of such supersets are very close to the threshold used to eliminate infrequent itemsets. Therefore, they are likely to be deleted again due to time-variation in many data streams. In view of this, EStream records the variations on the frequency of each itemset in different parts of the stream by separating its frequency in the current window (f_c) and the accumulated one in previous windows (f_p). This helps identify frequent itemsets in the entire stream so far but may not be locally frequent in the current window. As a result, EStream can efficiently reduce the generation of redundant supersets and therefore, improves its performance.

3.3.3 Error Analysis

In this section, we discuss the accuracy of frequency estimates produced by EStream. In the process, we prove the error guarantees presented in Section 3.2 of this chapter. In this case, 1-itemsets will not be discussed since their frequencies have been counted precisely and thus, there is no error.

For each itemset X , we denote its *true frequency* by $freq_T(X)$ and *estimated frequency* by $freq_E(X) = f_p(X) + f_c(X)$ using synopsis $\mathcal{S}$. Respectively, $supp_T(X)$ and $supp_E(X)$ denote its *true* and *estimated supports*.

Lemma 3.1 *If an itemset X is deleted in the current window w_c , then its true frequency $freq_T(X)$ seen so far in the stream is no more than $w_c \times arr[|X|]$.*

Proof:

We prove by induction. In the base case when $w_c = 1$, nothing is deleted at the end of this window since an itemset is inserted into $\mathcal{S}$ only if its immediate subsets are sufficiently significant. Therefore, any itemset X not inserted into $\mathcal{S}$ in the first window will not have their true frequency greater than $arr[|X|]$; i.e., $freq_T(X) \leq arr[|X|]$.

It is important to note that the maximal error in counting frequency of an inserted itemset X (at this first window) is also equal to $arr[|X|]$. This is because X is inserted into $\mathcal{S}$ *only after* all X 's subsets gain enough $arr[|X|]$ on their frequency.

When $w_c = 2$, only itemsets inserted in the first window might be removed since now they can become infrequent in this second window. Let X be such an itemset, its error is at most $arr[|X|]$. On the other hand, $freq_E(X)$ is its frequency count since X was inserted in the first window. Thus, $freq_T(X) \leq freq_E(X) + arr[|X|]$. When combined with the condition for deleting an itemset; i.e., $freq_E(X) + arr[|X|] \leq w_c \times arr[|X|]$, we have $freq_T(X) \leq w_c \times arr[|X|]$.

CHAPTER 3. ONLINE APPROXIMATE FREQUENT PATTERN MINING WITH PRECISE ERROR GUARANTEE

Induction case: Suppose X is deleted at $w_c > 2$. Then, X must be inserted at some window $w_i + 1$ before w_c ; i.e., $w(X) = w_i + 1$. In the worst case, X could possibly be deleted in the previous window w_i . By induction, X 's true frequency was no more than $w_i \times arr[|X|]$ when it was deleted in window w_i . Since $arr[|X|]$ is the maximum error at window $w_i + 1$ and $freq_E(X)$ is the frequency count since insertion in window $w_i + 1$, it follows that $freq_T(X)$ is at most $freq_E(X) + w_i \times arr[|X|] + arr[|X|] = freq_E(X) + (w_i + 1) \times arr[|X|]$; or $freq_T(X) \leq freq_E(X) + w(X) \times arr[|X|]$. Again, combined with the deletion rule that $freq_E(X) + w(X) \times arr[|X|] \leq w_c \times arr[|X|]$, we have $freq_T(X) \leq w_c \times arr[|X|]$.

Lemma 3.2 *The true frequency of any itemset X in $\mathcal{S}$ is bounded as follows: $freq_E(X) \leq freq_T(X) \leq freq_E(X) + arr[|X|] \times w(X)$.*

Proof:

As $freq_E(X)$ is the frequency count of X since it was inserted into $\mathcal{S}$, it is the lower bound on the true frequency of X . Thus, the lower limit of the inequality is always true. For the upper limit of the inequality, we prove as follows. If X is inserted in the first window, the maximum error in its frequency is $arr[|X|]$. Therefore, $freq_T(X) \leq freq_E(X) + arr[|X|]$; otherwise, X is possibly deleted some time earlier in the first w_i windows and then inserted into $\mathcal{S}$ at window $w_i + 1$. By Lemma 3.1, $freq_T(X)$ is at most $w_i \times arr[|X|]$ when such a deletion took place. Hence, $freq_T(X) \leq freq_E(X) + arr[|X|] + w_i \times arr[|X|] = freq_E(X) + w(X) \times arr[|X|]$.

Theorem 3.1 *If N is the number of transactions seen so far and X is a 2-itemset, then $supp_E(X) \leq supp_T(X) \leq supp_E(X) + \epsilon$.*

Proof:

By definition, we have $arr[2] = 2^{k-2} \sum_{i=2}^2 (1/2^{i-2}) = \Delta\epsilon$ when $|X| = 2$. Since $\Delta \times w(X) \leq N$, we have $arr[|X|] \times w(X) \leq N\epsilon$. Dividing the inequality in Lemma 3.2 by N , we have $supp_E(X) \leq supp_T(X) \leq supp_E(X) + \epsilon$.

Theorem 3.2 *If N is the number of transactions seen so far and X is a k -itemset, then $\text{supp}_E(X) \leq \text{supp}_T(X) < \text{supp}_E(X) + 2\epsilon$.*

Proof:

By definition, we know $\text{arr}[k] = 2^{k-2} \sum_{i=2}^k (1/2^{i-2}) \leq 2\Delta\epsilon - 1 < 2\Delta\epsilon$ when $|X| = k$. Again, since $\Delta \times w(X) \leq N$ and $\text{arr}[|X|] \times w(X) < 2N\epsilon$, dividing the inequality in Lemma 3.2 by N , we obtain the above result.

With the above analysis, we conclude that the algorithm guarantees: (i) no error in the support of 1-itemsets; (ii) the maximum error in the support of 2-itemsets is no more than ϵ ; and (iii) the maximum error in the support of k -itemsets is no more than 2ϵ .

3.4 Performance Evaluation

3.4.1 Experimental Setup

Our EStream algorithm is written in C++ and compiled using Microsoft Visual C++ version 6.0. All experiments are performed on a Pentium machine with a CPU clock rate of 1.9GHz, 1GB of main memory and running on the Windows XP platform. The method used for generating datasets is similar to the one described by Agrawal *et al.* [AS94]. Specifically, datasets are generated by the IBM synthetic market-basket data generator.⁵ To describe a dataset, we use the notation $Tw.Ix.Dy$ where w is the average size of transactions, x is the average size of maximal potentially frequent itemsets and y is the size of the dataset.

3.4.2 Scalability Study

To evaluate the scalability of the proposed EStream algorithm, both the environments of long transactions with fewer items and short transactions with more items are considered.

⁵available at “<http://www.almaden.ibm.com/software/quest/>”

CHAPTER 3. ONLINE APPROXIMATE FREQUENT PATTERN MINING WITH PRECISE ERROR GUARANTEE

3.4.a: T5.I3.D1000K

3.4.b: T8.I4.D1000K

Figure 3.4: Memory usage of EStream on two representative datasets of opposite characteristics.

Therefore, two data streams of size 1 million transactions are generated for study. One has an average transaction size of 8 items with an average size of maximal potentially frequent itemset of 4. This dataset is generated by using 5,000 distinct items. The other has an average transaction size of 5 with average size of maximal potentially frequent itemset of 3. It is generated using 10,000 distinct items. Using the above notation, we denote the two datasets by T8.I4.D1000K and T5.I3.D1000K respectively. We measured two parameters that are important for stream applications: (i) execution time and (ii) memory consumption.

To simulate an online data stream, transactions are processed individually and discarded immediately. This leaves the available memory for synopsis $\mathcal{S}$. Figure 3.4 shows the memory usage on the two datasets using EStream. The usage levels indicated in the figure represent the maximum number of itemsets generated during processing. This is equivalent to the maximum number of nodes allocated in $\mathcal{S}$ before pruning. We measure the usage levels over a number of minimum supports, with σ ranging from 0.1% to 1%, after 200,000 transactions. Here we note that since Δ is determined by $\lceil 2^{k-2} \times \epsilon^{-1} \rceil$ and

CHAPTER 3. ONLINE APPROXIMATE FREQUENT PATTERN MINING WITH PRECISE ERROR GUARANTEE

σ is varied from 0.1% to 1% (with $\epsilon = 0.1\sigma$), Δ is therefore changed accordingly. In these experiments, we took measurements at the window around this value, which is 200K transactions. As expected, a drop in the minimum support increases the memory usage. For example, when σ decreases from 1% to 0.1%, the number of nodes maintained in $\mathcal{S}$ increases from 7K to 56K with T5.I3.D1000K, and respectively from 5K to 230K with T8.I4.D1000K. This is due to the fact that, at small minimum support thresholds, we can find more frequent itemsets with larger sizes. Nevertheless, the interesting point here is that the memory usage remains constant at any given support threshold throughout the lifetime of processing the stream. This happens because EStream is designed to maintain only potentially frequent sets in $\mathcal{S}$; and by the pruning step, all insignificant candidates are removed periodically after each window.

With the same range of support thresholds, Figure 3.5 shows the execution time of the algorithm on the two selected datasets. As we can see, the cumulative execution time of EStream grows linearly with the number of transactions processed in the streams. At support threshold $\sigma = 0.1\%$, for instance, the time to process every 200K transactions of T5.I3.D1000K is approximately 26s, and T8.I4.D1000K is approximately 95s. This indicates the uniform processing time over each window; or uniform processing time for each transaction arriving from the stream. This stable execution time is also important in a data stream environment where the size of the stream is potentially unbounded.

3.4.3 Time-varying Study

In the previous experiments, we have excluded a time-variation parameter that has an effect on memory consumption and execution time. In this section, we evaluate the impact and show how EStream addresses this issue effectively in its design.

We have mentioned that the data stream distribution may change over time. In EStream, we separate the counting of itemsets in the current window from the previous

CHAPTER 3. ONLINE APPROXIMATE FREQUENT PATTERN MINING WITH PRECISE ERROR GUARANTEE

3.5.a: T5.I3.D1000K

3.5.b: T8.I4.D1000K

Figure 3.5: Execution time of EStream on two representative datasets.

ones. New itemsets are generated only based on the presence of significant subsets in the current window. Consequently, a large number of redundant candidates are reduced. While we have discussed this at the end of Section 3.3.2, we confirm the practicality of this approach in the experiments below.

Using T8.I4.D1000K, the effects of time-variation is simulated on the dataset using the method proposed by Giannella *et al.* [GHP⁺03], where an item mapping table is utilized. Here, 20% of items are randomly chosen for itemset generation with low frequencies and periodically after every 50,000 transactions, random permutations among all items are applied to the table. To test the difference, we have a variation of EStream that generates candidates using all windows since starting the algorithm; i.e., relying on f_p only. We denote this variant as EStream(w/o f_c). With the same range of support thresholds, Figure 3.6 shows the comparison in terms of execution and memory usage of EStream and its variant that uses only the cumulative counts; i.e., EStream(w/o f_c). Although a decrease in support threshold causes both to increase their number of nodes used in $\mathcal{S}$, we see that the memory utilization grows sharply in the case of EStream(w/o f_c). This

CHAPTER 3. ONLINE APPROXIMATE FREQUENT PATTERN MINING WITH PRECISE ERROR GUARANTEE

3.6.a: Execution time

3.6.b: Memory consumption

Figure 3.6: Experiment results on a time-varying data streams.

happens as 1-itemsets are often significant if considering their frequency in the entire data stream but may not be sufficiently frequent in some windows due to time-variations.

In the case of EStream, a distinction is made through the use of f_p (the cumulative frequencies of previous windows) and f_c (the frequencies of the current window). By distinguishing the frequencies of itemsets in the current window, our algorithm is able to detect the presence of such spurious items and therefore avoid unnecessary processing on transactions in which those items appear. In contrast, EStream (w/o f_c) would tend to view them as frequent which in turn leads to the generation of larger itemsets causing an increase in memory and processing demand.

3.4.4 Comparison with Lossy Counting

In this section, we compare the performance of EStream with that of the Lossy Counting algorithm [MM02]. Two algorithms will be investigated and compared in terms of accuracy, memory space and execution time. We continue using the dataset T8.I4.D1000K for these experiments. Also, ϵ is still set to be equal to 0.1σ .

CHAPTER 3. ONLINE APPROXIMATE FREQUENT PATTERN MINING WITH PRECISE ERROR GUARANTEE

Figure 3.7: Precision Comparison between EStream and Lossy Counting on T8.I4.D1000K

As described in Chapter 2, Lossy Counting processes data streams in batches. Therefore, in order to make the comparison fair, we set the batch size (in terms of number of transactions) in Lossy Counting equal to the size of a conceptual window defined in EStream. It is also important to note that, different from EStream where thresholds for identifying infrequent itemsets are based on itemsets' lengths, Lossy Counting identifies an itemset as infrequent if its frequency is no more than $\epsilon \times N$ (where N is the current stream's size) and it is independent from the length of the itemset.

Figure 3.7 reports the accuracy of two algorithms at various levels of minimum support thresholds. One can observe that when the minimum support threshold is set high, the precision of EStream is slightly better than that of Lossy Counting. For example, at $\sigma = 0.8\%$, the precision of EStream is 100% while that of Lossy Counting is 98.5%. This is because at this high level of minimum support, only 1-itemsets are found frequent in both algorithms. In EStream, as presented in Section 3.3.2, it counts precisely all 1-itemsets appearing in the Stream and thus, there is no error in their frequency count.

CHAPTER 3. ONLINE APPROXIMATE FREQUENT PATTERN MINING WITH PRECISE ERROR GUARANTEE

On the other hand, Lossy Counting only tracks an itemset if its frequency is estimated to be no less than $\epsilon \times N$ (regardless of the itemset's length) and it outputs all itemsets having frequency at least $(\sigma - \epsilon) \times N$ as frequent one. Accordingly, some 1-itemsets that are not truly frequent but have frequency between $(\sigma - \epsilon) \times N$ and $\sigma \times N$ are also outputted in the results of Lossy Counting. This makes the precision of Lossy Counting slightly smaller than that of EStream.

However, when the minimum support threshold is set smaller, the precision of Lossy Counting is a little bit better than that of EStream. This happens as Lossy Counting is a batch mode algorithm. Transactions arriving in the stream are first buffered into the main memory. Then such a batch of transactions are mined together (with multiple-scans on the data) for finding potentially frequent itemsets. For this reason, Lossy Counting is able to guarantee the same ϵ on the support's error of every frequent itemset (independent from their length). On the contrary, EStream is an online algorithm and thus, it guarantees the support's error of each itemset based on its length. The error margin is larger on a longer itemset and at most 2ϵ on the maximal frequent itemsets found. As such, its precision is compromised and is slightly lower than that of Lossy Counting. However, at the lowest support threshold $\sigma = 0.1\%$, EStream is still able to obtain 93.7% in its precision (as observed in the Figure 3.7).

Although Lossy Counting can achieve a higher precision than that of EStream, we make the observation that the memory space it uses in the mining process is much larger than that of EStream. As shown in Figure 3.8, when the minimum support threshold reduces, the memory space used by both algorithms increases. However, at all levels of the support threshold, Lossy Counting always needs more nodes for maintaining potentially frequent itemsets than EStream. For example, at $\sigma = 0.2\%$, Lossy Counting consumes 145,017 nodes for its synopses whereas EStream maintains only 88,376 nodes in its synopsis $\mathcal{S}$. To explain for this, notice that while EStream uses only a single synopsis

CHAPTER 3. ONLINE APPROXIMATE FREQUENT PATTERN MINING WITH PRECISE ERROR GUARANTEE

Figure 3.8: Memory Comparison between EStream and Lossy Counting on T8.I4.D1000K

$\mathcal{S}$ to maintain significant itemsets found in the stream, Lossy Counting has to use two prefix trees to perform the same task. One prefix tree is used to maintain the global set of potentially frequent itemsets. The other is used to find significant itemsets within each batch of transactions. Consequently, the total number of nodes maintained in two data structures are much larger than that of EStream. Also from Figure 3.8, one can observe that the difference in the memory usage of both algorithms looks more clearly when the support threshold is set smaller. At these small thresholds, there are more potentially frequent itemsets found by both algorithms and they need to be maintained in the algorithms' synopses.

With the same range of support thresholds, we report the execution time of EStream and Lossy Counting algorithms on the dataset T8.I4.D1000K in Figure 3.9. It can be seen that the processing time of Lossy Counting is also higher than that of EStream. For instance, at the support threshold $\sigma = 0.4\%$, Lossy Counting needs 305s to process 1 millions transactions in the dataset T8.I4.D1000K whilst EStream needs only 248s to do the same task. Similarly, at $\sigma = 0.2$, Lossy Counting needs 460s whereas EStream

CHAPTER 3. ONLINE APPROXIMATE FREQUENT PATTERN MINING WITH PRECISE ERROR GUARANTEE

Figure 3.9: Execution Time Comparison between EStream and Lossy Counting on T8.I4.D1000K

requires only 392s.

In order to explain for the difference in the execution time of two algorithms, it is worth noting that, in addition to the time for finding significant itemsets in each batch of transactions and keeping them in a local prefix tree, Lossy Counting also needs to periodically update the global prefix tree with this local tree. For each significant itemset found in the local prefix tree, Lossy Counting needs to find it in the global prefix tree for updating frequency. In case the itemset is not found, a new node in the global tree will be created. As such, this process usually requires a considerable amount of time at the end of each batch of transactions. EStream, on the other hand, processes the data stream online and it directly updates potentially frequent itemsets found in each transaction into the synopsis $\mathcal{S}$. As such, it can save a substantial amount of time by maintaining only one synopsis for significant itemsets. Furthermore, due to mining the stream online, EStream is able to output the set of frequent itemsets at any time by simply scanning the synopsis $\mathcal{S}$ once. Meanwhile, the mining results of Lossy Counting is available only

after each batch of transactions.

In the above experiments, notice that we do not evaluate the recall measure of Lossy Counting and EStream. This is because both algorithms are deterministic techniques, meaning that they are able to guarantee a prior error threshold in their mining results. In Lossy Counting, it guarantees that the estimated frequency of an itemset is less than its true frequency by an amount of at most ϵN . As such, by returning all itemsets having frequency no less than $(\sigma - \epsilon)N$, Lossy Counting ensures all truly frequent itemsets in the stream being reported in its mining results. Hence, its recall measure is always equal to 100%.

Similarly, EStream is also able to guarantee a pre-defined error threshold in each itemset's frequency. However, due to processing the stream online, the error is guaranteed based on the length of each itemset. As formally proved in Lemma 3.2, EStream guarantees that the estimated frequency of an itemset X is less than its true frequency by an amount of at most $arr[|X|] \times w(X)$. Consequently, it returns all itemsets having frequency no less than $(\sigma N - arr[|X|] \times w(X))$ as frequent patterns. Therefore, EStream also ensures that all truly frequent patterns are reported in the mining results. In other words, the recall measure is also equal to 100%.

3.5 EStream Discussions

Problems related to frequency counting that have been studied in the context of data streams can be classified into two groups: finding frequent items, and finding frequent sets.

The motivation for finding frequent items over streaming data arises from a variety of applications such as traffic engineering, network system analysis, sensor networks [CFPR00, CCFC04, MSD05] where each generated data instance in data streams

CHAPTER 3. ONLINE APPROXIMATE FREQUENT PATTERN MINING WITH PRECISE ERROR GUARANTEE

can be viewed as a single item. In this category, solutions have online processing capabilities and precise error guarantees while working within bounded memory. Examples of such algorithms include Lossy-Counting [MM02], Sticky-Sampling [MM02], FDPM-1 [YCLZ04], Group-Test [CM05].

On the other hand, with the emergence of applications that generate data in streams of transactions, such as web click streams, retail transaction flows, stock market transaction flows [DH01, BBD⁺02], the problem of discovering frequent sets, instead of frequent singletons, has been motivated. However, this important problem has not been well-addressed in terms of satisfying both online processing and precise error guarantees. The main difficulty is due to the exponential explosion of itemsets, which caused most existing algorithms to approach the constraints in separation.

One of the first online processing algorithm called CARMA is introduced by Hidber [Hid99]. In CARMA, the error in an itemset's frequency is based on the highest frequency error among its subsets. This estimation is often too conservative as itemsets are frequently deleted and inserted throughout the runtime of the algorithm. More importantly, a loose estimate on smaller itemsets often causes a much higher loose estimate in their supersets. This is why a vast majority ($> 95\%$) of the itemsets generated in CARMA turns out to be false positives. Chang *et al.* [CL03b] recently introduced an extension of CARMA in a forgetful model. In Chang's extension, the weights on older transactions are gradually decreased as new ones arrive in the stream. Despite this non-uniform threshold, the maximum error of a new itemset is derived by choosing the highest frequency error from its subsets. Consequently, the error varies for each pattern. More importantly, Chang's method cannot guarantee this error within any given threshold although it can approximate it based on subsets' frequency of each pattern. This issue is shown clearly in their later version [CL06].

In contrast, our EStream's strategy is not based on the highest error among subsets. Rather, a set of frequency thresholds is specified in advance for every itemset of specific

CHAPTER 3. ONLINE APPROXIMATE FREQUENT PATTERN MINING WITH PRECISE ERROR GUARANTEE

lengths. These thresholds are then used to identify the candidates within each window. If an itemset is not generated, its frequency in the window is guaranteed to be no more than the corresponding threshold. Since the thresholds and the window size are pre-specified, the support of each itemset can be precisely estimated and guaranteed within the same error.

As has been reviewed in Chapter 2, Lossy Counting [MM02] by Manku and Motwani is the first attempt to guarantee errors in the patterns discovered. Our EStream algorithm is similar to theirs by virtue that both find all frequent sets and guarantees error in each pattern to be within user-specified limits. In order to achieve this, however, their approach processes transactions in batches and thus, results cannot be produced online due to buffering. Another work that provides error guarantees on mining results is FDPM [YCLZ04]. Like the Lossy-Counting approach, Jeffery *et al.*'s work is also a batch-processing algorithm. Furthermore, the lossy nature of FDPM shown in Chapter 2 causes truly frequent patterns to be missed in the final result. On the other hand, our EStream algorithm guarantees that all patterns (if truly frequent) are discovered while operating in online-processing mode. In the following section, we discuss the extension of EStream in forgetful mining models.

3.6 Extending From Landmark model to Forgetful model with FStream

In Chapter 2, we have presented that a sliding window model and a forgetful window model are two other popular mining models in streaming environments, besides the landmark window model. In this section, we discuss how to adapt EStream algorithm to mine frequent patterns from data streams with forgetful window models. We name the algorithm FStream.

CHAPTER 3. ONLINE APPROXIMATE FREQUENT PATTERN MINING WITH PRECISE ERROR GUARANTEE

In a forgetful mining model, a fading factor is often used to reduce the effect of past events over time. Let γ be a fading factor; this factor is often chosen to be smaller than 1 and greater than 0. The closer to 0 the γ is, the more we want to forget the past transactions. Also, it is easy to observe that if $\gamma = 1$, the forgetful model becomes the landmark window model. On first look, the implementation of the forgetful window model is as easy as the landmark window model, as the two models only differ in the definition of the frequency. However, in the forgetful model, the weight of each occurrence of any itemset X is dependent on the index value of the current transaction. For example, if X appeared in transactions t_k, t_ℓ, t_m , and the current transaction is t_N , then the frequency of X at current time is $\gamma^{N-k} + \gamma^{N-\ell} + \gamma^{N-m}$. Thus, it seems that the algorithm has to maintain all weights at all times regardless of whether X appeared in the current transaction or not. Fortunately, it is not the case as we can compute the frequency of each itemset by the incremental method described below:

Let t_j be the last transaction that X appeared. Let $freq_j(X)$ be the frequency value of X at that time. At any transaction t_i , where $i > j$, the frequency of X at the transaction t_i is computed by the formula:

$$freq_i(X) = Appr(X) + freq_j(X) \times \gamma^{i-j} \quad (3.1)$$

where

$$Appr(X) = \begin{cases} 1 & \text{if } X \subseteq t_i \text{ (i.e., } X \text{ appears in } t_i) \\ 0 & \text{otherwise} \end{cases}$$

It is obvious that we only have to update the frequency of X when X appears in the current transaction or when we want to output the mining results.

Example 3.3 Suppose we have the following transactions: $\{ab\}, \{b\}, \{ac\}$, a fading factor is chosen as $\gamma = \frac{1}{2}$. The target is to count the frequency of item a in the forgetful

CHAPTER 3. ONLINE APPROXIMATE FREQUENT PATTERN MINING WITH PRECISE ERROR GUARANTEE

model. When the first transaction $\{ab\}$ is processed, the frequency of a is 1, as a is contained in this transaction. When the second transaction $\{b\}$ is processed, a does not appear; thus, its frequency is not updated. When processing the third transaction $\{ac\}$, due to the appearance of a in this transaction, its frequency is updated by Formula 3.1 as follows:

$$freq_3(a) = 1 + freq_1(a) \times \gamma^{3-1} = 1 + 1 \times \left(\frac{1}{2}\right)^2 = 1 + \frac{1}{4}$$

In order to adapt algorithm EStream in the forgetful model, we make the following modifications:

- We denote t_i as the current transaction on the stream. At each node of the data structure $\mathcal{S}$, we add another field, Tid , to store the index of the last transaction in which the corresponding itemset appeared. This field may replace the field w_c as we can easily derive the index of each window from the index of each transaction. Then, at lines 6 and 13, we apply Equation 3.1 to compute the frequency value in the current window, and we update Tid by the index value of the current transaction; i.e., $Tid := i$.
- At lines 8 and 16 which are the insertions, we initialize the value i for the field Tid .
- At the checking condition at line 15, which refers to procedure Check.SubsetE, we denote j as the number of transactions seen so far in the current window. If Y is an immediate subset of X and Y appeared in previous windows, then in order for X to be inserted into $\mathcal{S}$, Y must satisfy the condition:

$$f_c(Y) > \gamma^{j-arr[|X|]} \times \sum_{m=0}^{arr[|X|]-1} \gamma^m$$

In the worst case, if all the first transactions of the current window contain X , the lower bound of its missing fading frequency is $\sum_{m=0}^{arr[|X|]-1} \gamma^m$. The value $\gamma^{j-arr[|X|]}$

CHAPTER 3. ONLINE APPROXIMATE FREQUENT PATTERN MINING WITH PRECISE ERROR GUARANTEE

only indicates the fading frequency that we have to reduce when j increases. Similarly reasoning in the case that Y has just been inserted in the current window, its fading frequency must satisfy this condition:

$$f_c(Y) > \gamma^{j-arr[|X|]} \times \sum_{m=0}^{arr[|X|]-1} \gamma^m - \gamma^{j-arr[|X|-1]} \times \sum_{m=0}^{arr[|X|-1]-1} \gamma^m$$

- Note that the value in Tid may change when the algorithm continues processing the data stream in the current window. However, the accumulated frequency of X is not updated during this process. Fortunately, we can fade the accumulated frequency value in advance by modifying line 28 as:

$$f_p(X) = (f_p(X) + f_c(X)) \times \gamma^{i+\Delta-X.Tid}$$

- Now we discuss the pruning condition, which is the line 25 of the algorithm. For easy discussion, we re-write the second inequality in that line:

$$f_c(X) + f_p(X) \leq (w_c - w(X)) \times arr[|X|]$$

On the left side of this inequality, the value of $f_c(X)$ is kept originally as it was updated during the mining process of the current window. The value of $f_p(X)$ is also not changed as its fading value has been updated in advance by line 28. As the estimated error frequency is reduced over time, the right side of this inequality is:

$$\begin{aligned} (w_c - w(X)) \times arr[|X|] &= \left(\sum_{m=1}^{w_c} \gamma^m - \sum_{m=w(X)}^{w_c} \gamma^m \right) \times arr[|X|] \\ &= arr[|X|] \times \sum_{m=1}^{w_c-w(X)} \gamma^m \end{aligned}$$

Therefore, the prune condition in line 25 becomes:

$$f_c(X) + f_p(X) \leq arr[|X|] \times \sum_{m=1}^{w_c-w(X)} \gamma^m$$

To calculate the support, the size of the data stream should also be diminished. Let N be the size of the current data stream, then each time when a new transaction comes, N is updated: $N = N \times \gamma + 1$. Note that if we do not update N each time a new transaction t_N is processed, we can derive this value directly from N through the geometric series: $N = (\gamma^i - 1)/(\gamma - 1)$.

The complete pseudo code of this adapting algorithm, which is named *FStream*, is given in Figures 3.10 and 3.11.

We note that the error in the support of each output pattern as discussed in Section 3.3.3 is still guaranteed in *FStream* since we have diminished the weight of both itemset frequencies and the number of transactions seen so far simultaneously.

3.7 Summary

In this chapter, we have addressed the problem of mining frequent patterns from data streams in that we argue the need for critical stream applications to both satisfy online processing requirements and strict error guarantees in the estimate of their mining results. These two important problems are often addressed separately by the class of algorithms in existence. We present *EStream* and its extension *FStream* to address these two constraints simultaneously in a single solution. We believe that this will be an important step towards effective data stream applications and our work in this chapter contributes in part to this goal.

Nevertheless, in this chapter, we have addressed the problem of frequent patterns mining with the assumption that the speed of a data stream is always stable or it can be

CHAPTER 3. ONLINE APPROXIMATE FREQUENT PATTERN MINING WITH PRECISE ERROR GUARANTEE

Algorithm FStream: Find frequent sets from $\mathcal{DS}$ in a forgetful model.

Input

$\mathcal{DS}$: data stream of transactions;
 $\sigma \in (0, 1]$: minimum support threshold;
 $\epsilon \in (0, 0.5\sigma]$: maximum error allowable in σ ;
 $\gamma \in (0, 1)$: fading factor;
 k : the longest frequent patterns to find;

Output

At anytime on demand, return all frequent sets seen so far in the stream

Method

```

1:  $\Delta = \lceil 2^{(k-2)} \times \epsilon^{-1} \rceil$ ;  $w_c = 1$ ;  $N = 1$ ;  $arr[1] = 0$ ;
2: for  $\{j = 2, j \leq k, j++\}$  do  $arr[j] = 2^{k-2} \sum_{i=2}^j (1/2^{i-2})$ ;
3: for each  $t_i \in \mathcal{DS}$  do
4:   for each 1-itemset  $X \in t_i$  do
5:     if  $X \in \mathcal{S}$  then
6:        $f_c(X) = 1 + f_c(X) \times \gamma^{i-Tid(X)}$  ;
7:        $Tid(X) = i$  ;
8:     else
9:       Insert  $(X, 1, 0, w_c, i)$  to  $\mathcal{S}$ ;
10:    end if
11:  end for
12:  for each itemset  $X \subseteq t_i$  do
13:    if  $X \in \mathcal{S}$  then
14:       $f_c(X) = 1 + f_c(X) \times \gamma^{i-Tid(X)}$  ;
15:       $Tid(X) = i$  ;
16:    else
17:      if Check_SubsetF( $X$ ) then
18:        Insert  $(X, 1, 0, w_c, i)$  to  $\mathcal{S}$ ;
19:      end if
20:    end if
21:    if  $X$  cannot be inserted into  $\mathcal{S}$  then
22:       $t_i = t_i - \{X \subseteq t_i | X \text{ is a superset of } X\}$ 
23:    end if
24:  end for
25:  if  $i \equiv 0 \pmod{\Delta}$  then
26:    for each itemset  $X \subseteq t_i$  do
27:      if  $(|X| \geq 2)$  and  $f_c(X) + f_p(X) \leq arr[|X|] \times \sum_{m=1}^{w_c-w(X)} \gamma^m$  then
28:        Remove  $X$ 
29:      else
30:         $f_p(X) = f_p(X) + f_c(X) \times \gamma^{i+\Delta-Tid(X)}$ ;
31:         $f_c(X) = 0$ ;
32:      end if
33:    end for
34:  end if
35:  if mining results are requested then
36:    return  $\{X | f_c(X) + f_p(X) + arr[|X|] \times \sum_{m=w(X)}^{w_c} \gamma^m \geq \sigma \times N\}$ ;
37:  end if
38:   $N = N \times \gamma + 1$  ;
39: end for

```

Figure 3.10: FStream Algorithm

CHAPTER 3. ONLINE APPROXIMATE FREQUENT PATTERN MINING WITH PRECISE ERROR
GUARANTEE

Procedure Check_SubsetF(X : an itemset)

```

1: for (all  $Y \subset X$ ) do
2:   if  $Y \notin S$  then
3:     return FALSE;
4:   end if
5:   if ( $f_p(Y) > 0$  and  $f_c(Y) \leq \gamma^{j-arr[|X|]} \times \sum_{m=0}^{arr[|X|]-1} \gamma^m$ ) then
6:     return FALSE;
7:   end if
8:   if ( $f_p(Y) = 0$  and  $f_c(Y) \leq \gamma^{j-arr[|X|]} \times \sum_{m=0}^{arr[|X|]-1} \gamma^m - \gamma^{j-arr[|X|-1]} \times \sum_{m=0}^{arr[|X|-1]-1} \gamma^m$ ) then
9:     return FALSE;
10:  end if
11: end for
12: return TRUE;

```

Figure 3.11: Procedure for checking the frequency of all subsets of an itemset X in the forgetful window model.

varied within the processing system capability. In other words, the CPU capacity is not a limited resource and it is always powerful enough to process all transactions arriving in a data stream regardless of how fast the data flow is. Such an assumption is often violated by various high speed data stream applications [TcZ⁺03, TLPY06, TcZ07] in that their arriving rate is unpredictable and can significantly increase at any time scale. This problem of CPU constraints in data stream processing has been extensively studied in the context of data stream querying. A number of load shedding algorithms have been proposed and developed [TcZ⁺03, BDM04, GWYL05, TZ06, TLPY06]. Unfortunately, in the context of data stream mining and especially with the frequent mining problem, the CPU constraint has not been well addressed [CSWR05]. This observation gives us the motivation to continue addressing the frequent pattern mining problem and extend it to the context in that the CPU capacity is strictly a limited resource. We shall address this issue in the next two chapters in that both the landmark window and the sliding window models are explored.

Chapter 4

Approximate Frequent Patterns with CPU Constraints

4.1 Motivation

For many stream applications such as stock monitors, telecom fraud detection or sensor network surveillance, the data arrival rate is rapid and mostly unpredictable (due to the dynamic changing behavior of data sources) [DGR03]. Meanwhile, analyzing results of these streams usually requires delivery under time constraints in order to enable optimal decisions. Unfortunately, when the rate of the data stream significantly increases and exceeds the system capacity, overloading situations happen and the system may not be able to deliver the results within the given timeframe. Consequently, the quality of service can be degraded without bound. To cope with such situations, one may add more resources to handle the increased load or distribute the computation to multiple nodes. However, such an approach is expensive and generally infeasible in practice [TcZ⁺03, TZ06]. Therefore, it has been recognized that dropping unprocessed data (thus, only approximate results can be obtained) to reduce workload and maintain the timeliness of output results is generally preferred.

In the literature, the problem of dropping some data in order to cope with high speed data streams is often known as *load shedding*. Currently, this problem has been intensively studied in data stream querying [TcZ⁺03, BDM04, GWYL05, TLPY06, TcZ07].

However, it has not been well-addressed in the context of data stream mining [CSWR05], especially for the problem of finding frequent sets from transactional data streams. For this fundamental mining task, as we have presented in Chapter 2, most algorithms reported so far focused on the issue of managing limited memory space [CLC05, GHP⁺03, CL03a, TCY03, YCLZ04]. They ignore the fact that CPU is also a bounded resource. It is important to note that the main difficulty in frequent set mining is the large number of itemsets whose frequencies need to be tracked. Given a transaction of size m , the number of itemsets is exponentially proportional to its size; i.e., for a transaction of size $m = 10$, the number of itemsets may be up to $2^{10} - 1$. This clearly makes the frequent set mining problem susceptible to overloading.

In this chapter, we extend our problem of mining frequent patterns from data streams under the assumption that the CPU resource is confined (in addition to the limited memory space). When the CPU capacity is overloaded, the mining system will not be able to keep up with new data, so *load shedding* — discarding some fraction of unprocessed data — becomes necessary. In this chapter, we address this problem in a landmark window model with the assumption that overload situations can be properly detected and the amount of load needed to be shedded is known in advance.¹ We propose and present the LSFI (Load Shedding for mining Frequent Itemsets) algorithm based on a solid mathematical result, the Chernoff bound [Che53], to approximate a set of frequent patterns from streaming data.

The rest of the chapter is organized as follows. In the following section, we formally formulate the problem of load shedding in mining frequent patterns from data streams. The proposed approach to the problem is described in Section 4.3 where a load shedding scheme based on sampling transactions is represented. In Section 4.4, we report our experimental results and in Section 4.5, we present our discussions on the proposed algorithm.

¹These assumptions will be addressed when we continue extending the problem in the next chapter.

4.2 Problem Formulation

In this chapter, we continue adopting the notations and concepts presented in Chapter 3. Accordingly, we formulate the load shedding problem for finding frequent itemsets over a landmark window model in data stream environments as follows.

We are given a processing capacity (CPU) C of a mining system and a data stream $\mathcal{DS}$ with arbitrary high arrival rates. Let $Load(\mathcal{DS})$ indicate the workload of the system. Then a load shedding is invoked when $Load(\mathcal{DS}) > C$ (i.e., the system is overloaded). The objective is to construct an adaptive algorithm that can detect and drop a fraction of transactions to guarantee $Load(\mathcal{DS}) \leq C$ and yet discover a set of frequent itemsets that closely approximates to the set of actual frequent itemsets from the data stream. Clearly, in order to solve this problem, we must further address the following fundamental issues:

- (i) **When to shed load.** We need a method to identify overload situations. Thus, the mining process should be continuously evaluated. If there is overload, it should be detected quickly.
- (ii) **How much load to shed.** When the mining system is identified as being overloaded, we must decide how much load to shed. This may involve a decision on the percentage or number of transactions to be discarded.
- (iii) **How to approximate mining results under overload situations.** As mining results depend on the number of transactions processed and the content of transactions, dropping transactions directly affects the quality of the results. We need a method that not only drops transactions to guarantee the system workload but also yields the approximate results close to the exact ones.

In this chapter, we ignore the above first two questions. They will be addressed in details when we extend our problem to the time-based sliding window model which is presented in the next Chapter 5.

4.3 Load Shedding Algorithm in Landmark Window Model

4.3.1 Probabilistic Technique to Shed Load

When the system is overloaded, an immediate approach is to drop a fraction of the stream to reduce workload. We approach the problem by utilizing a technique from probability, the Chernoff bound [Che53]. Such a theoretically sound tool allows us to obtain a more accurate estimate on the mining error. To apply the Chernoff bound in our frequent set approximation, we clarify the following concepts.

Let $P \in (0, 1]$ be a parameter adjusted adaptively to make the total workload of the mining system no more than C (the processing capacity of the mining system). It expresses the fraction of transactions that should be discarded. Each coming transaction is then chosen with probability P . For a set of N transactions arriving in the stream, n transactions are chosen randomly. Given an itemset X , we want to approximate how close is its computed frequency in n sampling transactions, as compared to its actual frequency p in N transactions.

Note that the appearance of event X in a transaction can be seen as a Bernoulli trial and thus, we denote random variable $A_i = 1$ if X appears in the i th transaction and $A_i = 0$ otherwise. Obviously, $\Pr(A_i = 1) = p$ and $\Pr(A_i = 0) = 1 - p$.² Hence, n randomly drawn transactions are viewed as n independent Bernoulli trials. Let r be the number of times that $A_i = 1$ occurs in these n transactions; r is called a *binomial* random variable and thus, its expectation is np . Then, the Chernoff bound states that given an error bound ϵ , $0 < \epsilon < 1$:

$$\Pr\{|r - np| \geq n\epsilon\} \leq 2e^{-np\epsilon^2/2} \quad (4.1)$$

²We use $\Pr(\cdot)$ to denote the probability of a condition being met.

CHAPTER 4. APPROXIMATE FREQUENT PATTERNS WITH CPU CONSTRAINTS

Let us call $\text{supp}_E(X) = r/n$ the estimated support of X computed from n sampling transactions. This equation gives us the probability that the true support $\text{supp}_T(X)$ deviates from its estimated support $\text{supp}_E(X)$ by an amount $\pm\epsilon$. If we want this probability to be no more than δ , then the required number of sampling transactions is at least (by setting $\delta = 2e^{-np\epsilon^2/2}$)

$$n_0 = \frac{2p \ln(2/\delta)}{\epsilon^2} \quad (4.2)$$

For example, consider $p = 0.002$, $\delta = 0.05$ and $\epsilon = 0.001$, then $n_0 \approx 15,000$. This means that for itemset X , if we sample 15,000 transactions from a partition of the stream, its true support $\text{supp}_T(X)$ in this partition is beyond the range of $[\text{supp}_E(X) - 0.001, \text{supp}_E(X) + 0.001]$ with probability 0.05. In other words, $\text{supp}_T(X)$ is within $\pm\epsilon$ of $\text{supp}_E(X)$ with a high confidence of 95%.

In the load shedding problem, it is obvious that if the data stream is uniformly distributed, then n_0 transactions carry the same statistical information about the entire data stream. Therefore, processing these n_0 transactions gives us a set of frequent itemsets that closely approximates (within (ϵ, δ)) the set of actual frequent itemsets over the entire data stream. Unfortunately, the assumption of uniform distribution is often unrealistic in stream environments. Rather, both the arrival rate and the underlying distribution that generate data streams often change over time. It is also reasonable to believe that when the arrival rate of the stream significantly changes, the underlying distribution generating it changes as well.

When the workload of the system changes, we must quickly detect the sampling rate P . Then, each incoming transaction is chosen with probability P until we sample enough n_0 transactions. Each of these n_0 transactions is called a *sample batch*. All frequent itemsets in this sample batch are then discovered. We refer to them as *local* frequent itemsets as they are found from the recent portion of stream data. This procedure is

CHAPTER 4. APPROXIMATE FREQUENT PATTERNS WITH CPU CONSTRAINTS

repeated until the system workload changes to another level. By the Chernoff bound, we are guaranteed that the true support of each local frequent itemset is close to its estimated support computed in these n_0 transactions within (ϵ, δ) approximation scheme. In practice, there may be the case that the workload changes to another level before the system samples enough n_0 transactions. In this case, if the number of sampling transactions is quite close to n_0 , we treat these sampling transactions as a normal sample batch; otherwise, we can ignore if this number is too small.

Now we address the problem of how to report the global frequent itemsets in the entire data stream. For ease of explanation, we ignore the error produced by applying the Chernoff bound (i.e., assuming $\epsilon = \delta = 0$). It has been proven that a global frequent itemset must be locally frequent in at least one part of the data stream [SON95]. Hence, we can safely report all local frequent itemsets as an approximate set of all global frequent ones in the data stream. However, due to the non-uniform distribution of data streams, this approximation may result in many false global frequent itemsets that are locally frequent only. To address this issue, in the following we introduce another support threshold which is used to control the maximum support error for each itemset appearing in the data stream.

It is believed that the time-varying and unbounded characteristics of data streams always change the frequency of each itemset from time to time. In addition, due to the limitation of memory space, we are not allowed to maintain frequencies of all itemsets. Therefore, one way is to maintain only itemsets that are potentially frequent in the near future and quickly drop those that become infrequent. Nevertheless, it is not able to assure that an itemset previously deleted cannot become frequent any more. This means that the deletion leads to some error in estimating the support of itemsets. Thus, we need to control this error within some threshold. To do so, we use another parameter called the *significant support* σ_0 ($\ll \sigma$) and further divide itemsets into three categories:

frequent, subfrequent, and infrequent itemsets. An itemset X is called a *frequent itemset* if $\text{supp}(X)$ is no less than σ . It is called a *infrequent itemset* if $\text{supp}(X)$ is no more than σ_0 . Otherwise, it is a *subfrequent itemset*. Both frequent and subfrequent itemsets are called *significant or potential* itemsets.

With the introduction of a significant support threshold, we need to re-visit the problem of identifying the value of n_0 . So far, n_0 is computed from Equation 4.2. However, it is clear that the value of n_0 also cannot be set too small since it is dependent on σ_0 – the threshold to find significant itemsets in each sampling batch of n_0 transactions. An itemset is considered significant if its frequency is no less than $\sigma_0 \times n_0$. It is obvious that when $\sigma_0 = 0.01\%$ and if n_0 is chosen only 10,000 transactions, then $\sigma_0 \times n_0 = 1$. This makes all itemsets appearing in the sample batch significant. Certainly, the number of significant itemsets is extremely huge due to the exponential number of itemsets in each transaction. Therefore, in order to avoid this case, n_0 must be chosen larger and it is proposed to be β/σ_0 where β is an integer at least larger than 2. For example, $\beta = 4$ and given the same $\sigma_0 = 0.01\%$, the sampling batch's size n_0 will be $\beta/\sigma_0 = 40,000$. This means that only itemsets having frequency count at least 4 in the sampling batch are considered as significant ones. In view of this, n_0 is chosen as the larger value between $\{\frac{\beta}{\sigma_0}; \frac{1}{2\epsilon^2} \ln \frac{2}{\delta}\}$ (where $\beta \geq 2$).

4.3.2 The Algorithm

With the above analysis, we present in this section our algorithm which we call Load Shedding for mining Frequent Itemsets (LSFI). The pseudo code of LSFI is shown in Figure 4.1. We use a prefix tree *mathcal{S}* as a synopsis to maintain significant itemsets. Initially, *mathcal{S}* is empty. Each node in *mathcal{S}* corresponds to an itemset X and has the following three fields:

- *Item*: The last item of itemset X . Thus, X is represented by the set of items on the path from the root to the node.

CHAPTER 4. APPROXIMATE FREQUENT PATTERNS WITH CPU CONSTRAINTS

- *Acnt*: The accumulated frequency of X seen so far in the data stream.
- *Bid*: The index of the sample batch at which X is inserted into the prefix tree.

Algorithm LSFI: Find frequent itemsets from an entire data stream with CPU constraint.

Input

$\mathcal{DS}$: data stream of transactions;
 $\sigma \in (0, 1]$: minimum support threshold;
 $\sigma_0 \in (0, \sigma)$: a significant threshold;
 C : a processing capacity (CPU) of the mining system;

Output

At anytime on demand, return an approximate set of frequent itemsets seen so far in the stream

Method

```

1: Identify  $n_0$ ;
2: Periodically identify sampling rate  $P$ ;
3:  $b_{curr} = 0$ ;
4: for each  $t_N \in \mathcal{DS}$  do
5:   Sampling it with probability of  $P$ 
6:   for each time sampling enough  $n_0$  transactions do
7:      $b_{curr}++$ ;
8:     Mine all itemsets  $X$  whose current frequency  $Ccnt(X) > \lceil \sigma_0 \times n_0 \rceil$ 
9:     if  $X \in \mathcal{S}$  then
10:      Update its accumulated frequency:  $Acnt(X) = Acnt(X) + \lceil \frac{Ccnt(X)}{P} \rceil$ ;
11:     else
12:      Insert  $(X, \lceil \frac{Ccnt(X)}{P} \rceil, b_{curr})$  to  $\mathcal{S}$ ;
13:     end if
14:   Scan  $\mathcal{S}$  to prune all nodes  $\{X \mid Acnt(X) \leq (b_{curr} - X.Bid) \times \lceil \sigma_0 \times n_0 \rceil\}$ ;
15:    $Acnt(X) = Acnt(X) + Ccnt(X)$ ;
16:    $Ccnt(X) = 0$ ;
17:    $b_{curr} = \lceil \frac{N}{n_0} \rceil$ ;
18:   end for
19:   if mining results are requested then
20:     return  $\{X \mid Acnt(X) + X.Bid \times \lceil \sigma_0 \times n_0 \rceil \geq \sigma \times N\}$ ;
21:   end if
22: end for
  
```

Figure 4.1: LSFI Algorithm

The algorithm receives the following parameters: processing capacity (CPU) C of the system; transactional data stream $\mathcal{DS}$; minimum support threshold σ and significant sup-

CHAPTER 4. APPROXIMATE FREQUENT PATTERNS WITH CPU CONSTRAINTS

port threshold $\sigma_0 \in (0, \sigma]$. Load shedding is invoked when the system workload exceeds the given processing capacity C . On demand, the algorithm returns an approximate set of frequent itemsets seen so far in the data stream. LSFI includes the following steps:

- (i) Before processing the data stream, the algorithm initializes the sample batch index $b_{curr} = 0$ and computes the sampling size $n_0 = \text{Max}\{\frac{\beta}{\sigma_0}; \frac{1}{2\epsilon^2} \ln \frac{2}{\delta}\}$.
- (ii) Periodically, LSFI estimates the workload of the system and identifies an appropriate sampling rate P for the current status of the stream. If the workload is no more than C , set P to 1.
- (iii) Each time when a new transaction t_N arrives, the algorithm samples it with probability P .
- (iv) When n_0 transactions have been sampled, LSFI performs the following steps:
 - It firstly increases the index of sample batch by 1.
 - Then, all significant itemsets from this sample batch are mined. Any itemset X whose frequency, denoted by $Ccnt(X)$, is greater than $\lceil \sigma_0 \times n_0 \rceil$ will be viewed as a significant one.
 - For each such itemset X :
 - If X is already maintained in *mathcal{S}*, then the algorithm updates its frequency $Acnt(X)$ by adding an amount of $\lceil 1/P \times Ccnt(X) \rceil$. Note that to compensate for the dropping transactions caused by the introduction of sampling rate P , the frequency of each itemset computed within this sample batch must be scaled up appropriately by $1/P$ to approximate its true frequency in the current sampling part of the stream.
 - Otherwise, if X is not in *mathcal{S}*, create a new node for X in the prefix tree with $Acnt(X) = \lceil 1/P \times Ccnt(X) \rceil$ and $X.Bid = b_{curr}$.

- After that, for saving memory space, the algorithm traverses the prefix tree *mathcal{S}* to prune all infrequent itemsets X whose frequency seen so far is no more than $(b_{err} - X.Bid) \times \lceil \sigma_0 \times n_0 \rceil$. To be clear with this delete condition, we need to clarify some points. First, the minimum frequency needed to make an itemset X significant in each sampling batch n_0 is at least more than $\lceil \sigma_0 \times n_0 \rceil$. Since X was inserted at the sample batch $X.Bid$, the total minimum frequency it must accumulate to continue staying significant until the current sample batch must be more than $(b_{err} - X.Bid) \times \lceil \sigma_0 \times n_0 \rceil$. On the other hand, $Acnt(X)$ is its true frequency since it was inserted at $X.Bid$. Therefore, if this value is no more than $(b_{err} - X.Bid) \times \lceil \sigma_0 \times n_0 \rceil$, X is no longer significant. In case X is removed, all its supersets are also removed.
 - For the next sample batch index, the algorithm updates it by setting $b_{err} = \lceil N/n_0 \rceil$.
- (v) At any time instant when a user requests for the results, the algorithm scans $\mathcal{S}$ and produces all itemsets X whose $Acnt(X) + X.Bid \times \lceil \sigma_0 \times n_0 \rceil \geq \sigma \times N$. It is worth noting that $X.Bid \times \lceil \sigma_0 \times n_0 \rceil$ is the maximum frequency lost of itemset X caused by the pruning step described above.

4.4 Experimental Results

4.4.1 Experimental Setup

LSFI algorithm is written in C++ and compiled using Microsoft Visual C++ version 6.0. All experiments are conducted on a Pentium machine with a CPU clock rate of 1.9GHz, 1GB of main memory and running on Windows XP platform. The objective of our experiments is to evaluate the accuracy of LSFI under various high speeds of the stream. We use two synthetic datasets of size 1 million transactions which are generated

by using the IBM test data generator. The first dataset has an average transaction size of 8 items and the second one has average transaction size 10. Both have the same average size of maximal potentially frequent itemsets of 4 and are generated by using 5,000 unique items. Following the conventions in [AS94], we denote the two datasets by T8.I4.D1000K and T10.I4.D1000K respectively.

Since our LSFI algorithm is probabilistic, it can compute only an approximate set of frequent itemsets. Therefore, throughout the experiments, we evaluate the effectiveness of our algorithm by measuring its errors on recalling truly frequent itemsets. The false positive percentage and false negative percentage, which are introduced in the following section, will be used for this purpose. Also, due to the probabilistic approach, we repeat each experiment 10 times for each parameter combination and report the results using their average values.

4.4.2 Measurement of Errors

In order to measure the accuracy of our algorithm under different workload scenarios, we define two error measures as follows.

Let FIT be the set of truly frequent itemsets that the algorithm needs to discover. FIT is found by applying a classical method, such as Apriori [AS94] or FP-growth [HPY00] (in our experiments, Apriori is employed), on the dataset without any constraints. Let FIE be the set of itemsets that our LSFI algorithm reports as frequent. An itemset X is called *false positive* if X is in FIE but not in FIT . It is called *false negative* if X is in FIT but not in FIE . The collection of false positives is denoted by

$$FP = \{X | (X \in FIE) \wedge (X \notin FIT)\}$$

and the collection of false negatives is denoted by:

$$FN = \{X | (X \in FIT) \wedge (X \notin FIE)\}$$

Consequently, we define the false positive percentage and the false negative percentage as:

$$PFP = \frac{|FP|}{|FIT|} \quad (4.3)$$

and

$$PFN = \frac{|FN|}{|FIT|} \quad (4.4)$$

4.4.3 LSFI Performance

In our experiments, we fix $\epsilon = 0.001$, $\delta = 0.01$ and $p = 0.002$. Accordingly, the number of transactions for each sampling batch as determined by the Chernoff bound is $n_0 \approx 20K$. Since $p = 0.002$, the minimum support threshold σ will be chosen to be no less than this value. We also fix $\beta = 4$ and therefore, the last value for n_0 is identified by $\text{Max}\{\frac{4}{\sigma_0}; 20K\}$. For each experiment, the CPU capacity is fixed but the system workload (i.e. data stream rate) is varied from 2 to 10. A system workload is expressed as a multiple of CPU capacity. For example, a system workload of 2 corresponds to an input stream rate that is twice as high as the CPU capacity when no load shedding is needed.

Figures 4.2 and 4.3 show our experimental results on the two selected datasets where the minimum support threshold is varied from 0.2% to 0.8%. For both datasets, only frequent 1-itemsets were found at supports of more than 1%. Therefore, only minimum support thresholds less than those were considered. The figures show that at lower levels of system workload, LSFI generates a higher number of true frequent itemsets, as indicated by the small value of PFN , and a lower number of false frequent itemsets

CHAPTER 4. APPROXIMATE FREQUENT PATTERNS WITH CPU CONSTRAINTS

Figure 4.2: Accuracy of LSFI on dataset T8.I4.D1000K.

indicated by smaller value of PFN . It is also interesting to note that at all levels of minimum support thresholds, the algorithm still finds more than 92% of all the true frequent itemsets and retains the percentage of false frequent itemsets below 9% even when the system workload is 10 times higher than the CPU capacity. This is because the estimate on the true frequency of each itemset (which affects the percentage of false negatives) is guaranteed by the Chernoff bound and generally dependent on the number of sampling transactions. On the other hand, the number of false positives is guaranteed by the threshold σ_0 which is essentially used to estimate the maximal error in frequency count of each itemset. In this experiment, this threshold is set to be equal to 0.1σ and LSFI outputs all itemsets having frequency no less than $(\sigma - \sigma_0) \times N$ as frequent ones. As such, the algorithm is able to obtain a low level on the percentage of false positives (at all levels of minimum support).

In the above experiment, we have set $\sigma_0 = 0.1\sigma$, the threshold to prune infrequent itemsets (and thus, causing errors on true frequency counts of itemsets as analyzed in Section 4.3.1). In the following experiment, besides varying the system workload, we

CHAPTER 4. APPROXIMATE FREQUENT PATTERNS WITH CPU CONSTRAINTS

Figure 4.3: Accuracy of LSFI on dataset T10.I4.D1000K.

also vary the value of σ_0 and see how both these factors affect the accuracy of the LSFI algorithm. The dataset T10.I4.D1000K will be used for this experiment.

We still fix the CPU capacity and vary the stream rate from 2 to 10, σ_0 from 0.1σ to 0.5σ . Figure 4.4 reports our experimental results. As we can observe, the percentage of false negatives is almost not changed when the value of σ_0 is increased. It is reduced only by increasing system workload. However, the percentage of false frequent itemsets found is increased when σ_0 is enlarged. This happens since in our algorithm framework, σ_0 is used to prune infrequent itemsets. It is also the approximated support error of each frequent itemsets (under no load shedding). As a result, when σ_0 is expanded, the number of itemsets wrongly identified as frequent is also increased. Nevertheless, as we observe from the figure, at all levels of system workload, this value is still being kept below than 8%.

The last experiment which we test with our LSFI algorithm is with the dense and sparse datasets. We vary the domain sizes to generate dataset T10.I4.D1000K from 1000 to 10,000 distinct items. With 1000 unique items, we have a dense dataset and with 10,000 items, we have a sparse one. In this experiment, the minimum support threshold

CHAPTER 4. APPROXIMATE FREQUENT PATTERNS WITH CPU CONSTRAINTS

Figure 4.4: Varying data stream rates and σ_0 on dataset T10.I4.D1000K.

CHAPTER 4. APPROXIMATE FREQUENT PATTERNS WITH CPU CONSTRAINTS

Figure 4.5: Varying data stream rates and domains on dataset T10.I4.D1000K.

is set to $\sigma = 0.2\%$ and $\sigma_0 = 0.5\sigma$. Figure 4.5 plots our experimental results when we vary the stream rate from 2 to 10 times higher than the CPU capacity. From the figures we make the observation that the LSFI algorithm is not sensitive to the number of unique itemsets at all levels of system workload. There is a slightly high number of false frequent patterns when $n = 1000$ unique items. This is because with this dense dataset, when the minimum support is set too small, say $\sigma = 0.2\%$, there are more patterns having support between σ and $(\sigma - \sigma_0)$, the range in which our algorithm approximates as frequent ones. However, as we can see from the figures that such a number of false positives are always no more than 5% and they are reduced when the dataset is sparser. Therefore, we may conclude that our LSFI algorithm works well in both dense and sparse datasets and at all levels of shedding load.

4.4.4 Comparison with Algorithm Z

In [Vit85], Vitter has proposed 3 sampling algorithms for selecting a random sample of n records (or transactions) without replacement from a dataset of N records, where the value of N is unknown beforehand. The proposed methods ensure that the sample (or

CHAPTER 4. APPROXIMATE FREQUENT PATTERNS WITH CPU CONSTRAINTS

Figure 4.6: Accuracy of LSFI and algorithm Z on dataset T10.I4.D1000K at $\sigma = 0.2\%$

reservoir) is updated in such a way that all transactions having equal probability of being in the reservoir at any time. Amongst three algorithms presented in [Vit85], algorithm Z has the best time performance and it is chosen for comparison with LSFI algorithm. The dataset T10.I4.D1000K is also selected for conducting the experiments and we run algorithm Z at two sample sizes of 100K transactions and 200K transactions. They are denoted in figures respectively by *algorithm Z(100k)* and *algorithm Z(200k)*.

Figures 4.6 and 4.7 report the accuracy of LSFI and algorithm Z at the support thresholds $\sigma = 0.2\%$ and $\sigma = 0.4\%$. For LSFI algorithm, the threshold of σ_0 is set to 0.2σ and we measure its accuracy at load factors of 2 and 4. From the figures, it can be seen that at the beginning of the stream, both the percentages of false positives and false negatives of algorithm Z are better than those of LSFI. This is because at this period, the reservoir is not full and transactions are accumulated to it. There are no transactions sampled at the stream size of 100k with *algorithm Z(100k)*, and 200k with *algorithm Z(200k)*. Therefore, both PFP and PFN of algorithm Z in this period are 0%.

However, when more and more transactions come, the percentages of false positives

CHAPTER 4. APPROXIMATE FREQUENT PATTERNS WITH CPU CONSTRAINTS

Figure 4.7: Accuracy of LSFI and algorithm Z on dataset T10.I4.D1000K at $\sigma = 0.4\%$

and false negatives in algorithm Z are reduced and become worse than those of LSFI. For example, at the minimum support threshold $\sigma = 0.4\%$ of T10.I4.D1000K, the PFP of *algorithm Z*(100k) increases from 3% to 6.1% when the size of the dataset increases from 200k to 1000k whereas that value of LSFI (at load factor 4) is always around 3.8% in the entire dataset. The similar thing can be observed with the PFN measure. To explain for this, notice that algorithm Z is a reservoir method and regardless of how large the stream is, it always samples and maintains a pre-defined number of sampling transactions. As such, the representative capability of such a fixed size of sample is reduced when the size of the stream increases, leading to the numbers of false positives and false negatives found more.

On the other hand, LSFI also ensures that transactions are randomly selected from the stream (the sampling rate is based on the load factor). However, the more transactions come, the more they are sampled and updated into the synopsis $\mathcal{S}$. This makes the number of sampling transactions proportional to the size of stream (yet note that LSFI does not accumulate successfully sampled transactions since once being processed, they

CHAPTER 4. APPROXIMATE FREQUENT PATTERNS WITH CPU CONSTRAINTS

Figure 4.8: The execution time of LSFIs and algorithm Z on dataset T10.I4.D1000K at $\sigma = 0.2\%$ and $\sigma = 0.4\%$

will be deleted immediately from the stream). Consequently, LSFIs are able to maintain the same level of accuracy throughout the stream. As we observe from both Figures 4.6 and 4.7, the percentages of false positives and false negatives of LSFIs almost remain unchanged when the size of the stream increases. They only reduce when the load factor increases.

With the same sets of parameters setting, we report the cumulative execution time of two algorithms in Figure 4.8. For LSFIs algorithm, we update the mining results after each $n_0 \approx 20K$ transactions (computed from $\text{Max}\{\frac{4}{0.2\sigma}; 20K\}$). For *algorithm Z*(100k) and *algorithm Z*(200k), we update the mining results after each 100K transactions. From the figures we make the observation that when more transactions come, the cumulative processing time of both algorithms Z and LSFIs linearly increases. This implies that both algorithms are stable with the size of the stream. However, it also can be observed that the execution time of algorithm Z is much higher than that of LSFIs algorithm. For example, at the minimum support threshold $\sigma = 0.4\%$, the processing time of *algorithm Z*(100k)

is 344s and that of *algorithm Z*(200k) is 659s; meanwhile, at the same support threshold, the processing time of LSFI algorithm is only 244s at the load factor 2 and 115s at the load factor 4.

To explain for this, it is important to note that while LSFI only has to mine frequent itemsets in the sampling batch and update them to the global set of frequent patterns, algorithm Z has to mine the entire reservoir after each period of time (i.e., each 100K records) since transactions in the reservoir are randomly replaced by the new ones sampled from the stream. This makes the reservoir's content changed after each period of time. It is also worth noting that although algorithm Z has the most effective time in the sampling process, the most consuming time is the time to find patterns from the sampled transactions, rather than the time of sampling transactions. Although we let the algorithm Z update its mining results at every 100k (instead of every 20K transactions as in LSFI), its execution time is still more than that of LSFI.

Another drawback of a reservoir algorithm is that it only provides a technique to randomly select a pre-defined number of transactions from the data stream. It does not directly provide any theoretical tool to approximate errors in the mining results. On the other hand, LSFI is able to do so since it is designed based on a solid mathematical result, the Chernoff bound. As we shall present in the next chapter where the problem is further extended to a time-based sliding window, with a certain level of confidence, it is able to guarantee an error bound on the mining results.

4.5 Discussions

Load shedding is the problem of dropping a fraction of datasets when systems are overloaded. It has been studied intensively in real-time databases [WA94]. Recently, this problem has been extended for data stream querying [TcZ⁺03, BDM04, TLPY06, TZ06]. In the context of querying stream data, the problem of load shedding is defined as the

CHAPTER 4. APPROXIMATE FREQUENT PATTERNS WITH CPU CONSTRAINTS

process of finding an optimal plan for inserting drop-operators along existing arcs of a query network (i.e., a query plan). One of the first stream querying projects that focused on this problem is the Aurora system [ACc⁺03, TcZ⁺03] (and later the Borealis distributed system [TcZ07]). This project addresses the load shedding problem by utilizing different Quality of Service (QoS) graphs that represent various important levels of querying objects. Based on QoS graphs, when the system is overloaded, transactions will be dropped progressively starting from those that contain information about the lowest important objects. Addressing the same problem is the STREAM project [BDM04]. In this project, a load shedding scheme based on sampling has been proposed for sliding window aggregate queries. This approach modifies the query network by inserting load shedder operators together with their sampling rates in such a way that the total sampling rate eliminates sufficient amount of dropping data. This work is similar to our work in this chapter in that the random sampling is used as a means of load shedding. However, it addresses the problem of querying objects with optimized sampling rates in a query network. In contrast, our work addresses the issue in the context of mining frequent patterns. In data stream mining, Loadstar [CSWR05] recently is reported as the first work focusing on the load shedding problem for classification. It argues that the most time consuming task in stream classification is data analysis and preparation. Thus, when the streaming rate significantly increases and overwhelms the CPU capacity, it is beneficial if the mining system could eliminate transactions early from this process. To do so, this work introduces Quality of Decision (QoD) metrics to measure the level of uncertainty in classification when exact values of the data are not available because of load shedding. A Markov model is utilized to predict the distribution of feature values and then classification decisions are made based on predicted values and QoD metrics.

On the other hand, with the class of frequent pattern mining algorithms reviewed in Chapter 2, we have realized that most of them have ignored the resource constraint on

CHAPTER 4. APPROXIMATE FREQUENT PATTERNS WITH CPU CONSTRAINTS

CPU capacity by simply assuming that the arriving rate of a data stream is relatively stable during the entire course of computation and it never overwhelms the system capacity. Such an assumption is generally impractical due to the dynamic changing behavior of data sources generating data streams. Although a reservoir sampling algorithm [Vit85] (such as the algorithm Z that we have conducted the experiments in the previous section) can help to reduce the system's workload by maintaining only a pre-defined number of transactions randomly sampled from a data stream, it is important to note that it does not provide any technique to approximate errors in the mining results.

Therefore, our work in this chapter, also motivated by work in data stream querying management presented above, has addressed this problem of CPU constraint in the context of frequent set mining. Such an important problem should be properly handled in order to guarantee the quality of service in various data stream applications. In the following chapter, we continue addressing this problem by providing solutions for how to detect overloading situations, how to identify workload that needs to be adaptively shedded, and how to approximate and guarantee the mining results. The problem will be addressed in the most challenging mining model of data streams, the time-based sliding window model.

Chapter 5

CPU Constraints: From Landmark Window to Sliding Window Model

5.1 Motivation

In this chapter, we continue extending the problem of approximating frequent sets from data streams under the constraint of CPU capacity. We consider the most general mining model: the *time-based sliding window* model. An approximate algorithm to compute a set of frequent patterns online will be presented. Specifically, we fully address the three questions introduced in previous Chapter 4: (1) How to determine overload situations? (2) How much load need to be shedded? (3) How to approximate frequent sets under load shedding? We adopt an adaptive and self-regulating approach to these questions in that the transaction flows are processed online and they will be adaptively dropped whenever overload situations are detected.

It is also important to note that, although frequent sets are computed in a sliding data window, no revisiting of expiring transactions is needed in our solution. The data stream is processed online and transactions are shedded, if necessary, in order to guarantee the timeliness of the mining results. More importantly, we theoretically prove that the error in the mining results will be kept within a preset threshold by exploiting both the Chernoff bound property [Che53] and the deterministic technique.

Recall that, for various data stream applications, not only approximating, guaranteeing error in mining results is always one of the most desirable and important characteristics requested by users. Unfortunately, much work reported so far in the literature cannot fulfill this property (as have been shown in the Chapter 2). Most of them can only empirically report that as many patterns can be found as possible [TCY03, CL03a, CL06]. Meanwhile, those satisfying this property [MM02, YCLZ04] have to process data streams in batch models and thus, they have limited data stream applications [MM02]. Our work in this chapter directly addresses the online, time-varying data stream whilst both the error bound on the mining results and the system resource constraints are simultaneously guaranteed.

This chapter is organized as follows. In the next section, we formally formulate our frequent set mining problem with CPU constraints in the *time-based sliding window* model. Section 5.3 describes our approach in that we present the workload estimation followed by the algorithm design and its accuracy analysis. Section 5.4 reports our experimental results. Finally, we present our conclusions on the chapter in Section 5.5.

5.2 Problem Formulation

To begin, we formulate the problem of approximating frequent sets from a time-based window sliding on data streams as follows.

We consider a high speed data stream arriving as a time ordered series of transactions $\mathcal{DS} = \{t_1, t_2, \dots, t_N, \dots\}$. Each transaction t_i contains a set of items such that $t_i \subseteq I$, where $I = \{a_1, a_2, \dots, a_m\}$ is a set of literals called items (or objects) and t_N is called the current transaction arriving on the stream.

We focus on *time-based* sliding window. In this model, let $TS_0, TS_1, \dots, TS_{i-k+1}, \dots, TS_i$ denote the time periods elapsed so far in the stream. Each time period (or time slot) contains multiple transactions arriving in that interval and thus, they form a

partition of transactions in the stream. Given an integer k , the time-based sliding window is defined as the set of transactions arriving in the last k time periods and denoted by $\mathcal{SW} = \{TS_{i-k+1}, \dots, TS_{i-1}, TS_i\}$. TS_i is called the latest time slot and TS_{i-k} is called the expiring one in the sliding window. When the time shifts to the new slot TS_i , the effect of all transactions in TS_{i-k} will be eliminated from the mining model.

Let n_w be the number of transactions arriving between TS_{i-k+1} and TS_i , the cover of an itemset X in the sliding window, denoted by $cover(X, \mathcal{SW})$, is defined as the number of transactions between TS_{i-k+1} and TS_i that contain X ; i.e., $cover(X, \mathcal{SW}) = \{t_i | t_i \in \mathcal{SW}, X \subseteq t_i\}$. Its frequency in the sliding window is denoted by $freq(X) = |cover(X, \mathcal{SW})|$ and its support, $supp(X)$, is defined as the ratio between $freq(X)$ and n_w ; i.e., $supp(X) = freq(X) \times n_w^{-1}$. Given $\sigma \in (0, 1]$ as a threshold for minimum support, X is called a frequent itemset in the sliding window if $supp(X) \geq \sigma$; it is called a maximal frequent itemset (MFI) if none of its immediate supersets are frequent.

For this time-based sliding window model, the problem is defined as follows: We are given a processing capacity (CPU) C of the mining system and a data stream $\mathcal{DS}$ with arbitrarily high arrival rates (the characteristics of $\mathcal{DS}$ also may change with time). Let $Load(\mathcal{DS})$ indicate the workload of the system, then a load shedding is invoked whenever $Load(\mathcal{DS}) > C$. The objective is to find all frequent patterns together with their estimated frequencies in the sliding window while guaranteeing that $Load(\mathcal{DS}) \leq C$.

5.3 Online Approximate Frequent Patterns in Sliding Window Model

5.3.1 Workload Estimation

Since the behavior of data streams often changes over time, detecting overload situations is an important step in our algorithm. For the frequent set mining problem, the system workload may not be simply estimated by regularly checking the rate of transactions

CHAPTER 5. CPU CONSTRAINTS: FROM LANDMARK WINDOW TO SLIDING WINDOW MODEL

arriving in one time unit. Rather, it is essentially dependent on the number of itemsets included in each transaction whose frequencies must be updated. Certainly, an accurate method to evaluate the system workload is to have an exact count of this number in each transaction. Unfortunately, this task is generally impossible since the system may not be able to fully process all incoming transactions under overload situations. Therefore, it is necessary to find a technique that is able to approximately estimate this number meanwhile it is efficient to compute. We propose such an estimate based on a small set of maximal frequent itemsets (MFIs). The key intuition behind using MFIs for this task is that the number of MFIs is exponentially smaller than the number of frequent itemsets, as shown by [Yan04]. Meanwhile, such a compact set completely captures the entire set of frequent itemsets. To give an explicit formula for this estimate, let us denote the number of MFIs in a transaction by m and let X_i be an MFI, $1 \leq i \leq m$. We assume the following estimated time (or load coefficient) to process one transaction:

$$L = \sum_{i=1}^m 2^{|X_i|} - \sum_{i,j=1}^m 2^{|X_i \cap X_j|} \quad (5.1)$$

In this equation, the first summation estimates the number of frequent itemsets within each MFI. The second one estimates the number of itemsets overlapping between MFIs. Apparently, it is supposed that computing L is expensive and eventually defeats the purpose of quickly detecting CPU overload. Nevertheless, we do not explicitly compute L by finding all MFIs and the overlapping subsets among them. Instead, the set of MFIs is maintained in a prefix tree and the equation is computed by matching transactions to this structure to derive the number of distinct frequent sets. As we shall see later from the empirical results reported in Section 5.4, this approach yields very good approximation while the computational time is negligible.

Given this computed statistics for each transaction, we measure it for n transactions over one time unit and let r be the current rate of the data stream. The following

inequality imposes a constraint on load shedding decisions:

$$P \times r \times \frac{\sum_{i=1}^n L_i}{n} \leq C \quad (5.2)$$

In this inequality, $P \in (0, 1]$ is a parameter adjusted adaptively to make the inequality hold. It also expresses the fraction of transactions that should be discarded. The expression $r \times \frac{\sum_{i=1}^n L_i}{n}$ gives the estimated system workload to process transactions arriving in one time unit. L_i is described in Equation 5.1; C , as formulated above, is the processing capacity of the mining system.

Recall that, when the system is overloaded, an immediate approach is to drop a fraction of data streams in order to reduce workload. Certainly, when all incoming data is not entirely processed and dropping transactions are lost forever, one can expect some errors in the mining outputs. We design our algorithm in such a way that it is able to approximate this error in a precise manner. In other words, the error is guaranteed within some specific value. In order to achieve such an important task, the Chernoff bound [Che53] is used. We present our algorithm in the following section.

5.3.2 The Online Algorithm

In this section, we present our algorithm which we name *Load Shedding* for mining frequent sets from *Sliding Windows* (LSSW).

In our sliding window framework, frequent patterns are discovered from a time-based sliding window where the window of interest is the set of transactions arriving in the last k time slots. We further divide the data stream into *conceptual windows* (condows) having the same size of Δ transactions and assume that each time slot consists of multiple condows. Accordingly, these condows form the basic units of each time slot (rather than individual transactions) in our model. The value of Δ is chosen to be equal to n_0 as computed in Equation 4.2.

CHAPTER 5. CPU CONSTRAINTS: FROM LANDMARK WINDOW TO SLIDING WINDOW MODEL

When the window slides to a new time slot TS_i , the effect of all condows in the expired one, i.e., TS_{i-k} will be eliminated and thus, the current window now consists of condows arriving only in time slots $\mathcal{SW} = \{TS_{i-k+1}, \dots, TS_i\}$. It is also important to note that since the rate of the data stream can change over time, the number of condows appearing in each time slot may not be the same.

In essence, not all itemsets appearing in the stream should be recorded and counted due to system resource limitations. By conceptually dividing the stream into condows, we can employ a deterministic threshold to filter itemsets whose frequencies are insignificant. In our LSSW algorithm, the threshold $\gamma (< \sigma)$ is used for this task. If a pattern has a support of no more than γ , it is unlikely to be frequent in the near future and therefore, should be removed early from the results. Furthermore, our goal is not only to produce an approximate set of frequent patterns but also wants to deliver it within a precise error limit. This guarantee is a challenging task since our algorithm is designed to scan transactions online to find frequent sets; a potential candidate is generated only after all its subsets are found to be significantly frequent. In other words, longer patterns will suffer from a larger margin of error and as a result, it is not able to guarantee the same error for every pattern of different lengths. However, when partitioning the data stream into equal-sized condows, we can approximate the patterns' error based on their lengths and more importantly, it is able to tighten their upper error bound and the condition to generate potential candidates.

In our design, an array storing minimum frequency thresholds is utilized for this purpose. If m is the maximal size of frequent sets that the user wants to explore from the stream, then the only condition this array needs to satisfy is:

$$a[i] < a[j] \quad \text{for} \quad 1 \leq i < j \leq m, \quad \text{and} \quad a[m] \leq \gamma \times \Delta. \quad (5.3)$$

CHAPTER 5. CPU CONSTRAINTS: FROM LANDMARK WINDOW TO SLIDING WINDOW MODEL

Algorithm LSSW. Find frequent sets from a time-based sliding window.

Input:

- (1) A processing capacity (CPU) C of the mining system;
- (2) A window sized of k time slots sliding on data stream $\mathcal{DS}$;
- (3) A minimum support threshold $\sigma \in (0, 1]$;
- (4) A significant threshold $\gamma \in (0, \sigma)$ and condow size Δ ;

Output: At anytime on demand, return all estimated frequent sets computed in the sliding window

```

1:  $TS = 0$ ;  $w_o = 0$ ;  $w_c = 1$ ;  $W[\text{mod}(TS, k)] = w_c$ ;
2: Periodically identify sampling rate  $P$ ;
3: for each  $t_N \in \mathcal{DS}$ , sampling it with probability of  $P$  and if  $t_N$  is chosen do
4:   for all  $X \subseteq t_N$  and  $X \in \mathcal{S}$  do  $Ccnt(X)++$ 
5:   for all  $X \subseteq t_N$  and  $X \notin \mathcal{S}$  do
6:     if  $X$  is 1-itemset then
7:       Insert  $X$  to  $\mathcal{S}$  with  $X.Wid = w_c$ ;  $Ccnt(X) = 1$ 
8:     else
9:       if (for all  $Y \subset X$ ,  $\nexists Y \in \mathcal{S}$  such that
10:         $(Y.Wid < w_c$  and  $Ccnt(Y) \leq arr[X])$  or
11:         $(Y.Wid = w_c$  and  $Ccnt(Y) \leq arr[X] - arr[Y])$ ) then
12:         Insert  $X$  to  $\mathcal{S}$  with  $X.Wid = w_c$ ;  $Ccnt(X) = 1$ ;
13:       end if
14:     end if
15:   if  $X$  cannot be inserted into  $\mathcal{S}$  then
16:      $t_N = t_N - \{Z \subseteq t_N \mid Z \text{ is a superset of } X\}$ 
17:   end if
18: end for
19: for every  $\Delta$  transactions sampled, scan  $\mathcal{S}$  and do
20:    $X.Freq[w_c] = Ccnt(X) \times P^{-1}$ 
21:    $S = S - \{X \mid X.Wid < w_o \text{ and } \sum X.Freq[i] \leq (w_c - w_o + 1) \times a[X]\}$ 
22:    $S = S - \{X \mid X.Wid \geq w_o \text{ and } \sum X.Freq[i] \leq (w_c - X.Wid) \times a[X]\}$ 
23:    $w_c = \lceil N/\Delta \rceil + 1$ 
24: end for
25: if A new time slot arrives and  $TS \geq k$  then
26:    $TS++$ ;  $w_o = W[\text{mod}(TS, k)]$ ;  $W[\text{mod}(TS, k)] = w_c$ 
27:   Remove  $X.Freq[i]$  such that  $i \leq w_o$ 
28: end if
29: end for
30: if mining results are requested then
31:   return  $\{X \mid X.Wid < w_o \text{ and } \sum X.Freq[i] \geq (w_c - w_o + 1) \times \Delta \times \sigma\}$ 
32:   return  $\{X \mid X.Wid \geq w_o \text{ and } \sum X.Freq[i] + (X.Wid - w_o + 1) \times a[X] \geq (w_c - w_o + 1) \times \Delta \times \sigma\}$ 
33: end if

```

Figure 5.1: LSSW Algorithm

CHAPTER 5. CPU CONSTRAINTS: FROM LANDMARK WINDOW TO SLIDING WINDOW MODEL

Therefore, an itemset of size j will be generated if its immediate subset frequencies in the current condow are above the threshold specified in $a[j]$; i.e., its subsets are significant in the condow. On the other hand, by choosing the size of each condow to be equal to Δ transactions, we can approximate (in terms of probability) the true support of patterns within each time slot to be no more than ϵ when load shedding is involved. As will be shown in Section 5.3.3, the error in frequent sets is guaranteed to be within the preset error thresholds γ and ϵ .

Given the above analysis, we describe each step of our LSSW algorithm as follows. Its pseudo code is outlined in Algorithm LSSW. LSSW uses a prefix tree $\mathcal{S}$ to maintain frequent sets discovered from the sliding window. Each node in $\mathcal{S}$ corresponds to an itemset X having the following fields:

- *Item*: The last item of itemset X . Thus, X is represented by the set of items on the path from the root to the node.
- *Wid*: The index of the condow at which X is inserted into $\mathcal{S}$.
- *Freq*[1..*max*]: A circular queue storing frequencies counted at each condow in the sliding window.

We denote the latest (or current) condow in the sliding window by w_c , and the oldest one with w_o . To efficiently eliminate expired time periods, the indexes of the first condows in each time slot are tracked globally and stored in a circular array W of k elements corresponding to the last k time slots. Accordingly, the value of w_o is always cleared each time the window slides. The time slot index TS is initialized to 0 when the algorithm starts and is incremented at each new time period. Let t_N be the new arriving transaction. LSSW consists of the following steps:

CHAPTER 5. CPU CONSTRAINTS: FROM LANDMARK WINDOW TO SLIDING WINDOW MODEL

- (i) The system workload is periodically estimated to identify overloading. If such a situation occurs, the maximal value of P is identified via Equation 5.2. Otherwise, P is set to 1.
- (ii) For each incoming transaction t_N , it is sampled with probability P . If t_N is chosen, the following steps are performed:
 - Increment: If an itemset X appearing in t_N is also currently maintained in $\mathcal{S}$, then its frequency in the current condow, denoted by $Ccnt(X)$, is increased by 1.
 - Insert: For each $X \subseteq t_N$ and $X \notin \mathcal{S}$, insert X into $\mathcal{S}$ with $X.Wid = w_c$ and $Ccnt(X) = 1$ if X is a singleton¹; otherwise, let Y be any immediate subset of X , then X is inserted into $\mathcal{S}$ if the following three conditions hold:
 - All immediate subsets of X are in $\mathcal{S}$;
 - $\nexists Y$ such that $Y.Wid < w_c$ and $Ccnt(Y) \leq arr[|X|]$; i.e., there is no Y being inserted into $\mathcal{S}$ from previous condows whose frequency in the current condow is insufficiently significant;
 - $\nexists Y$ such that $Y.Wid = w_c$ and $Ccnt(Y) \leq (arr[|X|] - arr[|Y|])$; i.e., there is no Y that has just been inserted into $\mathcal{S}$ in the current condow after which its frequency is no more than $(arr[|X|] - arr[|Y|])$.

In cases where X is not inserted into $\mathcal{S}$, all of its supersets in t_N need not be further checked.

- Update frequency: After each Δ transactions have been sampled, the algorithm updates $Freq[w_c] = Ccnt(X) \times P^{-1}$. This compensates for dropping

¹Note that the immediate subsets of a 1-itemset is $\{\emptyset\}$ which appears in every transaction, all 1-itemsets are therefore inserted into $\mathcal{S}$ without conditions. For the same reason, they are also not pruned from $\mathcal{S}$.

CHAPTER 5. CPU CONSTRAINTS: FROM LANDMARK WINDOW TO SLIDING WINDOW MODEL

transactions using sampling rate P and X 's frequency is appropriately scaled up by P^{-1} to approximate its true frequency in the sampling part of the stream. At this step, LSSW also scans $\mathcal{S}$ to remove all but 1-itemsets that satisfy either of the two conditions:

- If $X.Wid < w_o$ and $\sum X.Freq[i] \leq (w_c - w_o + 1) \times a[|X|]$
- If $X.Wid \geq w_o$ and $\sum X.Freq[i] \leq (w_c - X.Wid) \times a[|X|]$

Certainly if an itemset is removed, all its supersets are also removed. Those itemsets recently inserted into $\mathcal{S}$ in the current condow will not be removed since they are generated after their immediate subsets became sufficiently frequent. After that, LSSW updates the index for the next condow by $w_c = \lceil N/\Delta \rceil + 1$.

- Remove expiring time slot: In case the window slides to a new time slot, TS will be incremented by 1 and if $TS \geq k$, the algorithm further removes frequency counts in queue $X.Freq$ where indexes are smaller than value w_o stored in $W[mod(TS, k)]$. Then LSSW updates $W[mod(TS, k)] = w_c$ to register the first condow index of the new time slot.

(iii) At any instant upon the user's request, the algorithm scans $\mathcal{S}$ to output all itemsets satisfying either of the conditions below:

- If $X.Wid < w_o$ and $\sum X.Freq[i] \geq (w_c - w_o + 1) \times \Delta \times \sigma$
- If $X.Wid \geq w_o$ and $\sum X.Freq[i] + (X.Wid - w_o + 1) \times a[|X|] \geq (w_c - w_o + 1) \times \Delta \times \sigma$

For the second condition, it is noted that $(X.Wid - w_o + 1) \times a[|X|]$ is X 's maximal frequency error from w_o to $X.Wid$ (as will be proven in the following section).

5.3.3 Error Analysis

We prove the correctness of our LSSW algorithm in this section. For simplicity, we shall omit the notation X and re-denote its *true* frequency between two condows w_α and w_β (where $\alpha \leq \beta$) by $f_T(w_\alpha, w_\beta)$ and its *estimated* one by $f_E(w_\alpha, w_\beta) = \sum_{i=\alpha}^{\beta} X.Freq[i]$. Respectively, $s_T(w_\alpha, w_\beta)$ and $s_E(w_\alpha, w_\beta)$ denote its *true* and *estimated* supports in this period. In the proof of Lemmas 5.3 and 5.4, we ignore 1-itemsets and itemsets those have $X.Wid < w_o$ since their frequencies have been precisely counted in the sliding window.

Lemma 5.3 *Under no load shedding, if an itemset X is deleted at condow w_d , then*

- (i) $f_T(w_s, w_d) \leq (w_d - w_s + 1) \times a[|X|]$; i.e., the true frequency of X between w_d and w_s – any condow at which X was previously inserted – is no more than $(w_d - w_s + 1) \times a[|X|]$; and
- (ii) $f_T(w_d^p + 1, w_d) \leq (w_d - w_d^p) \times a[|X|]$, where w_d^p is any condow (previous w_d) at which X was also deleted.

Proof. Let us denote condow indexes of the two latest periods at which X is inserted and deleted by $w_s^{\ell-1}$, $w_d^{\ell-1}$ and w_s^ℓ , w_d^ℓ respectively.

When X is inserted at w_s^ℓ , its maximum error at this condow is $a[|X|]$, and $f_E(w_s^\ell, w_d^\ell)$ is its frequency count since then to w_d^ℓ . Therefore $f_T(w_s^\ell, w_d^\ell) \leq f_E(w_s^\ell, w_d^\ell) + a[|X|]$. Together with the deletion rule at w_d^ℓ , we have $f_T(w_s^\ell, w_d^\ell) \leq (w_d^\ell - w_s^\ell + 1) \times a[|X|]$. On the other hand, X is not inserted anytime between $w_d^{\ell-1} + 1$ and $w_s^\ell - 1$, its true frequency in each of these condows is no more than $a[|X|]$. This means that $f_T(w_d^{\ell-1} + 1, w_s^\ell - 1) \leq (w_s^\ell - 1 - w_d^{\ell-1}) \times a[|X|]$. Summing up these 2 periods, $f_T(w_d^{\ell-1}, w_d^\ell) \leq (w_d^\ell - w_d^{\ell-1}) \times a[|X|]$ is derived.

Similar to above when X was inserted at $w_s^{\ell-1}$ and deleted at $w_d^{\ell-1}$, we have $f_T(w_s^{\ell-1}, w_d^{\ell-1}) \leq (w_d^{\ell-1} - w_s^{\ell-1} + 1) \times a[|X|]$. Together with the above result, $f_T(w_s^{\ell-1}, w_d^\ell) \leq (w_d^\ell - w_s^{\ell-1} +$

CHAPTER 5. CPU CONSTRAINTS: FROM LANDMARK WINDOW TO SLIDING WINDOW MODEL

$1) \times a[|X|]$ is satisfied. By repeating the same calculation for every previous insertion and deletion periods, the lemma follows. $\square$

Lemma 5.4 *Under no load shedding, if an itemset X is deleted at condow w_d , then its true frequency between w_d and any previous window w_i is no more than $(w_d - w_i + 1) \times a[|X|]$; i.e., $f_T(w_i, w_d) \leq (w_d - w_i + 1) \times a[|X|]$.*

Proof. We distinguish two cases.

For the first case, where X 's frequency is not counted at w_i . Then, from w_i to its next insertion at w_s^i (but not include w_s^i itself), its true frequency at each condow is no more than $a[|X|]$. Thus, $f_T(w_i, w_s^i - 1) \leq (w_s^i - w_i) \times a[|X|]$. On the other hand, by Lemma 5.3, its true frequency since w_s^i to w_d is bounded by $f_T(w_s^i, w_d) \leq (w_d - w_s^i + 1) \times a[|X|]$. Therefore, $f_T(w_i, w_d) \leq (w_d - w_i + 1) \times a[|X|]$.

For the second case where X 's frequency is counted at w_i , we prove by contradiction. Assume that $f_T(w_i, w_d) > (w_d - w_i + 1) \times a[|X|]$. Let us denote the closest condows of w_i , at which X is inserted and deleted by w_s^i and w_d^i respectively (i.e., $w_s^i \leq w_i \leq w_d^i$). By Lemma 5.3, $f_T(w_d^i + 1, w_d) \leq (w_d - w_d^i) \times a[|X|]$. Meanwhile, we have assumed that $f_T(w_i, w_d) > (w_d - w_i + 1) \times a[|X|]$. Accordingly, its true frequency between w_i and w_d^i must be greater than $(w_d^i - w_i + 1) \times a[|X|]$. Together with the fact that X has been counting since w_s^i , we have $f_E(w_s^i, w_i - 1) > (w_i - 1 - w_s^i) \times a[|X|]$. Consequently $f_E(w_s^i, w_d^i) > (w_d^i - w_s^i) \times a[|X|]$, making X not be deleted at w_d^i . This contradicts the fact that X was deleted at w_d^i . The lemma is proved. $\square$

Lemma 5.4 is important since it allows to approximate the maximal error of every itemset discovering within the sliding window regardless of the new index value of w_o . Note that the value of w_o is arbitrary since the number of condows within each time period is variable.

CHAPTER 5. CPU CONSTRAINTS: FROM LANDMARK WINDOW TO SLIDING WINDOW MODEL

Theorem 5.3 *Let w_o and w_c be the oldest and current condows of the sliding window respectively. Under no load shedding, $s_T(w_o, w_c) \leq s_E(w_o, w_c) + \gamma$.*

Proof. If X is inserted at w_o , the first condow of the current sliding window, its maximum error at this condow is at most $a[|X|]$. Therefore, $f_T(w_o, w_c) \leq f_E(w_o, w_c) + a[|X|]$. Otherwise, X is possibly deleted some time earlier, as late as, at condow $w_i - 1$, and then inserted into $\mathcal{S}$ at condow w_i . By Lemma 5.4, $f_T(w_o, w_i - 1)$ is at most $(w_i - w_o) \times a[|X|]$ when such a deletion took place. Since $a[|X|]$ is the maximal error when X is inserted at w_i and $f_E(w_i, w_c)$ is its frequency count since then, it follows that $f_T(w_o, w_c) \leq f_E(w_i, w_c) + (w_i - w_o + 1) \times a[|X|]$.

On the other hand, let n_w be the number of transactions in the sliding window. Certainly, $\Delta \times (w_i - w_o + 1) \leq n_w$ (since $w_i \leq w_c$), we have $(w_i - w_o + 1) \times a[|X|] \leq \gamma \times n_w$. Therefore, dividing the inequality above for n_w , we derive $s_T(w_o, w_c) \leq s_E(w_o, w_c) + \gamma$. $\square$

Theorem 5.4 *Under load shedding, $s_T(w_o, w_c) \leq s_E(w_o, w_c) + \gamma + \epsilon$ with a probability of at least $1 - \delta$.*

Proof. This theorem can be directly derived from the Chernoff bound and Theorem 5.3 above. At time slots where the load shedding happens, the true support of X is guaranteed within $\pm\epsilon$ of the counting support with probability $1 - \delta$ when the Chernoff bound is applied for sampling. Meanwhile, by Theorem 5.3, this counting support is limited by $s_T(w_o, w_c) \leq s_E(w_o, w_c) + \gamma$. Therefore, getting the upper bound (i.e., ϵ) in the Chernoff bound, we derive the true support of X to be no more than its estimated support by $\gamma + \epsilon$ with a confidence of $1 - \delta$. $\square$

Figure 5.2: Accuracy of LSSW on dataset T8.I4.D3000K

5.4 Experimental Results

We implemented our LSSW algorithm in C++ and performed experiments on a 1.9GHz Pentium machine with 1GB of memory running Windows XP. Three synthetic data streams are generated with a size of 3 million transactions each by using the IBM data generator [AS94]. We obtained dataset T10.I6.D3000K by setting the average size of transactions $T=10$, the average size of maximal potentially frequent itemset $I=6$. With similar setting, we obtained T8.I4.D3000K and T5.I3.D3000K. Since our approach is probabilistic, the precision and recall measures will be used to evaluate the effectiveness of our algorithm. For the same reason, each experiment is repeated 10 times for each parameter combination and the average results are reported.

5.4.1 Accuracy Measurements

In our experiments, we fix $\epsilon = 0.001$, $\delta = 0.01$ and $p = 0.002$. Accordingly, the number of transactions for each condow as determined by the Chernoff bound is $\Delta \approx 20K$. Since $p = 0.002$, the minimum support threshold σ will be chosen to be no less than this

CHAPTER 5. CPU CONSTRAINTS: FROM LANDMARK WINDOW TO SLIDING WINDOW MODEL

Figure 5.3: Accuracy of LSSW on dataset T10.I6.D3000K

value. Specifically, σ will be varied between 0.002 and 0.01. We also fix $\gamma = 0.1\sigma$ (except $\gamma = 0.25\sigma$ for $\sigma = 0.002$) and the maximal length of frequent patterns $m = 10$ (interval between $a[i]$ is $\gamma\Delta/m$). The sliding window consists of 10 time slots and each time slot receives 200K transactions (i.e., 10 condows). For simplicity, instead of varying loads, we fix the load and change the number of condows that the system can handle at each time period. For example, if only 2 of 10 condows can be processed in each time slot, this translates to an input stream rate that is 5 times higher than the CPU capacity and thus, the percentage of load shedding is 80%.

Figures 5.2 and 5.3 show our experiment results on two datasets T8.I4.D3000K and T10.I6.D3000K respectively where the minimum support threshold is varied from 0.2% to 1% and the load shed is increased from 20% to 90%. We observed that at lower levels of system workload, the algorithm returns a higher number of true frequent itemsets as indicated by the higher value of recall, and a smaller number of false frequent itemsets from the precision measure. A larger gap in accuracy occurs when the load shed increases from 60% to 80% and 90%. At these levels, only 2 and 1 condows are processed (respectively) at each time slot. Nevertheless, LSSW continues to find more than 92% of

CHAPTER 5. CPU CONSTRAINTS: FROM LANDMARK WINDOW TO SLIDING WINDOW MODEL

Figure 5.4: Accuracy vs. Itemset Length of LSSW for T10.I6.D3000K

all true frequent itemsets (while keeping the false frequent itemsets at below 10%) even though the system workload has increased 10 times than that of the CPU capacity, i.e., a load shed of 90%. And this is maintained across all levels of support thresholds.

More details are shown in Figure 5.4 when we plot the precision and recall for each itemset length. We report the results on T10.I6.D3000K with $\sigma = 0.2\%$ since at this threshold, it is possible to find frequent sets that has a length of up to 10. It can be observed that the precision decreases as the length of a frequent set increases. This happens because longer patterns are generated *only after* all its subsets are found to be significantly frequent. Hence, the precision on longer itemsets is expected to be lower as a consequence of the larger margin error. Fortunately, distinguishing the error by its pattern length allows LSSW algorithm to closely approximate the maximum lost for each pattern. As seen in Figure 5.4, the precision on 10-itemsets is maintained above 80%. Note that the recall is not affected by this approximation as the true frequency of itemsets is guaranteed by the Chernoff bound, which is generally dependent on the number of sampling transactions. We observed that other than very large load shedding

CHAPTER 5. CPU CONSTRAINTS: FROM LANDMARK WINDOW TO SLIDING WINDOW MODEL

Figure 5.5: Time and workload of LSSW on T5-8-10.D3000K at various support thresholds

(more than 80%), the recall is always found to be higher than 92% for every itemset length.

5.4.2 Adaptability Study

To test the adaptability of our LSSW algorithm, we use T5-8-10.D3000K where its first, second and third parts are made up of each 1000K transactions from T5.I3.D3000K, T8.I4.D3000K, and T10.I6.D3000K. To evaluate the workload using Equation 5.2, we first set the CPU capacity to be infinite and let the algorithm process the entire dataset. Figures 5.5(a),(b) and (c) show the relationship between the processing time and the

CHAPTER 5. CPU CONSTRAINTS: FROM LANDMARK WINDOW TO SLIDING WINDOW MODEL

statistics measured within each condow, where the range of support threshold varies between 0.2% and 0.6%. In the figures, σT and $\sigma T(\text{w/o MFI})$ respectively indicates the time to process a condow when we enable and disable the function detecting workload using MFI tree. σS refers to the statistics computed in Equation 5.2 for each condow. As seen from the figure, σT and $\sigma T(\text{w/o MFI})$ are very close to each other, indicating that the time to compute the statistics is very small compared to the time of finding all frequent sets from the stream. When using MFI tree to evaluate workload, it only incurs more CPU usage when the algorithm updates this structure. However, this operation is performed only when a significant change in workload is detected in the stream or it is periodically performed for a set of condows. In the experiments presented, the MFI tree is updated at every time slot (or every 10 condows).

Also from Figure 5.5 we observe that the statistics computed is almost linearly proportional to the processing time across all the support thresholds experimented. For instance, at $\sigma = 0.2\%$ (Figure 5.5(a)), the average statistics computed for one condow for transactions sized T5 is 242.1K, T8 is 890K and T10 is 1275K; and the average time to process them are respectively 5.05s, 18.5s and 26.6s. Similarly, at $\sigma = 0.4\%$ (Figure 5.5(b)), the average statistics computed for one condow of transactions sized T5 is 160.7K, T8 is 514.3K and T10 is 792.5K; and the average processing time are respectively 4.1s, 13.0s and 19.8s. What these statistics mean is that (e.g., for the case $\sigma = 0.4\%$) if we limit the CPU capacity and the data stream is initially sent to the system at a rate just below its CPU capacity, then LSSW will have to shed load by sampling at a rate of $1/3$ ($\approx \frac{160.7}{514.3} \approx \frac{4.1}{13}$) and $1/5$ ($\approx \frac{160.7}{792.5} \approx \frac{4.1}{19.8}$) in the second and third parts of the stream. In other words, it is effective to use the computed statistics to identify the appropriate amount of data for shedding.

In Figure 5.6, we report the accuracy of our algorithm for this experiment. As we can observe, the recall at all minimum support thresholds is still very high ($\geq 94\%$) and

Figure 5.6: Accuracy of LSSW on dataset T5-8-10.D3000K

is likely the same. Nevertheless, we find that the precision is slightly lower than that in the pure dataset T10.I6.D3000K. This happens since we note that the last sliding window also includes 1000K transactions sized T8 and when the algorithm processes T10.I6.D1000K, more frequent itemsets are found in this part of the stream. According to our approximation (when estimating $(X.Wid - w_o) \times a[|X|]$ to be X 's maximal frequency lost), a small fraction of those itemsets discovered in T10.I6.D1000K have over-estimated frequencies. This means that they were locally frequent in the third part of the stream, but not in the second part. However, it is interesting to note that at all support thresholds examined, the percentage of false frequent itemsets is consistently within 10%.

5.5 Summary

Together with previous chapter, in this chapter, we have visited the issue of applications with data streams that have a variable arrival rate and data characteristics that could inadvertently push the workload above the system's capacity. To avoid such situations, it is important to design algorithms that can adapt to the underlying changes in the data characteristics to guarantee the timely delivery of results. Finding frequent patterns

CHAPTER 5. CPU CONSTRAINTS: FROM LANDMARK WINDOW TO SLIDING WINDOW MODEL

is an example of a difficult problem in such a situation, and our work in these two chapters presents the solution to this problem. Two algorithms, respectively addressing the landmark window and the sliding window models, are proposed and presented to approximate the set of frequent patterns from the data stream while taking the limitation of CPU capacity. It has been shown that not only do our proposed algorithms perform well in overloaded situations, the results they deliver are well guaranteed to be within error bounds. We believe that such contributions are solid and valuable in term of effective data stream mining.

Chapter 6

Clustering Data Streams in Sliding Window Models with EM Technique

6.1 Motivation

Besides frequent pattern mining, clustering is also one of the most important tasks that widely helps to analyze and uncover structures in data. Under some definition of similarity, clustering aims to partition data into groups (or clusters) so that data points in one group are similar to each other, and are different from those in other ones. Research in data stream clustering reported so far has usually focused on two mining models, the landmark window [GMMO00, AHWY03, ZCY⁺07] and the forgetful window models [CEQZ06, CT07]. In the former model, clustering results are computed based on entire data points generated so far in the stream. In the latter model, they are also discovered from entire data stream; however, the weight (or importance) of each data records is decreased with time by using some fading function. While these two mining models are useful in some data stream applications, there is a strong demand to devise novel techniques that are able to cluster streaming data in a sliding window model. It has been shown [ZS02] that the sliding window model is the most general and also the most challenging mining model since it concerns the elimination of outdated data. Unlike two previous mining models where entire data streams are concerned, in sliding window

model, only the most recent portion of the stream is taken into account for clustering computation. As such, when more data points come, the oldest ones in the window need to be removed from the clustering results. This characteristic brings more challenges to the mining process given the fact that the stream is processed in a single linear scan fashion and the window's size can be arbitrarily large.

In this chapter, we study the problem of approximating a set of clusters in a sliding window model. With the assumption that a data stream is generated by a statistical process which consists of k mixture models, we propose an algorithm namely SWEM (clustering data streams in Sliding Window model with EM) based on the Expectation Maximization technique [DLR77]. Compared to other data stream clustering algorithms relying on the k -means method [GMMO00, AHWY03], our EM-based algorithm is a softer clustering method and thus it has some properties that are desirable in stream environments. For example, it is robust to noise which commonly happens in stream environments. An EM-based algorithm also can handle missing data. Furthermore, it has a strong statistical basis and theoretical guarantees about the optimality convergence [DLR77].

It is important to note that, the memory space used in data stream analysis is always limited even when the sliding window model is employed. This happens in both centralized and distributed stream applications [ZCY⁺07]. For example, with distributed sensor networks where each sensor has to process an incoming data stream before sending the results back to the central station, the storage of each sensor itself is very confined; this leaves the space sufficient to maintain only certain summarized information. Inspired by the micro and macro clustering framework proposed in [AHWY03], our algorithm SWEM utilizes a set of micro components, each characterized by a triple of component's size, center and spread, to statistically summarize the data stream. This information is then used in later phases where SWEM removes expiring time slots in the sliding

window. Our algorithm consists of two stages. The first one is used to summarize raw data records into triplets of statistics. The second stage is then to use these statistics to update the set of global clusters. These two stages are iterated for each time slot that comes and leaves the mining model. To address the issue of local maxima problem arisen when characteristics of the stream change over time (i.e., drifting concepts), we propose a method to informatively split and merge components which takes into account both their positions and shapes. Furthermore, when the window slides and information in an outdated time period is required to be eliminated from the clustering results, we propose two approaches to update the global clustering models. The first approach updates the global models by applying the EM technique on the summarized statistics that are still active in the sliding window. The second approach directly updates the global clustering results by gradually removing the statistics summarized in the expiring time slot. Therefore, this approach is independent of the size (i.e., number of time slots) of the sliding window.

The rest of this chapter is organized as follows. In the following section, we formally formulate the problem of clustering data streams from a sliding window given the constraints on memory space and one-pass processing manner. Section 6.3 describes our overall SWEM algorithm. Then its detailed steps are presented in Section 6.4. Section 6.5 reports our experimental results. Finally, our conclusions are presented in Section 6.6.

6.2 Problem Formulation

We consider a data stream as a time ordered series of data points $\mathcal{DS} = \{x_1, x_2, \dots, x_n, \dots\}$ where each x_i has the form of $x_i = \{x_i^1, x_i^2, \dots, x_i^d\}$ in a d -dimensional space.

We continue focusing on the *time-based* sliding window. In this model, let $TS_0, TS_1, \dots, TS_{i-b+1}, \dots, TS_i$ denote the time periods elapsed so far in the stream. Each time period contains multiple data points arriving in that interval. Given an integer b , the time-based

CHAPTER 6. CLUSTERING DATA STREAMS IN SLIDING WINDOW MODELS WITH EM TECHNIQUE

sliding window is defined as the set of data records arriving in the last b time periods and denoted by $\mathcal{SW} = \{TS_{i-b+1}, \dots, TS_{i-1}, TS_i\}$. TS_i is called the latest time slot and TS_{i-b} is called the expiring one in the sliding window. When the time shifts to the new slot TS_i , the effect of all data points in TS_{i-b} will be eliminated from the mining model.

In our sliding window model, once data points in a time slot are processed, they cannot be retrieved for further computation at a later time (unless their summarized statistics are explicitly stored in memory). The amount of memory available is assumed to be limited, in particular, sub-linear in the size of the sliding window. Therefore, approaches that require storing the entire data points in the sliding window are not acceptable within this model.

We also assume that the data streams are evolving with time and they are generated as a result of a dynamic statistical process which consists of k mixture models (or components). Each model corresponds to a cluster which follows a multivariate normal distribution. Consequently, any cluster $C_h, 1 \leq h \leq k$, is characterized by a parameter: $\phi_h = \{\alpha_h, \mu_h, \Sigma_h\}$ in that α_h is the cluster weight, μ_h is its vector mean of d -dimension (determining the cluster center), and Σ_h is its $d \times d$ covariance matrix (determining the cluster shape).

It is important to note that when each cluster is assumed to follow a multiple normal distribution, projecting it to any subspace or dimension, we also obtain a multiple normal distribution. This can be seen by the following simple linear algebra transformations.

For simplifying discussion, let $p(x) \sim \mathcal{N}(\mu, \Sigma)$ (a multivariate normal distribution) in d dimensions of cluster C , A be a d -by- q matrix, and $y = A^t x$ is a q -component vector. Then it can be seen that $p(y) \sim \mathcal{N}(\mu', \Sigma') = \mathcal{N}(A^t \mu, A^t \Sigma A)$. This is because:

$$\mu' = E(y) = E(A^t x) = A^t E(x)$$

and

$$\begin{aligned}
\Sigma' &= E[(x - \mu)(x - \mu)^t] \\
&= E[(A^t x - A^t \mu)(A^t x - A^t \mu)^t] \\
&= E[A^t(x - \mu)[A^t(x - \mu)]^t] \\
&= E[A^t(x - \mu)(x - \mu)^t A] \\
&= A^t \Sigma A
\end{aligned}$$

In the special case where $q = 1$ and matrix A is a unit-length vector a , then $y = a^t x$ is a scalar that represents the projection of x onto a line in the direction of a . In this case, $a^t \Sigma a$ is the variance of the projection of x onto a . Therefore, we can say that the knowledge of the covariance matrix allows us to calculate the dispersion of the data points in cluster C in any direction. This reveals the shape of the cluster.

Accordingly, we define the data stream clustering problem in a sliding window model as follows: Given the number of time slots b , our goal is to incrementally cluster the data stream by identifying a set of parameters $\Phi_G = \{\phi_1, \dots, \phi_k\}$ that optimally fit the current set of data points arriving in the last b time periods of the data stream.

6.3 The Stream Clustering Framework

It is often convenient and effective to assume that a data stream is generated as a result of a statistical process and to describe the data by finding the statistical models, which consist of k different probability distributions, that best fit the data. The k approximating mixture models are described in terms of distributions characterized by a set of parameters $\Phi_G = \{\phi_1, \dots, \phi_k\}$. Each distribution corresponds to a cluster $C_h, 1 \leq h \leq k$, and the parameters of each distribution provide a description of the corresponding cluster, as we have formulated in Section 6.2, typically in terms of its center, spread and weight.

It is important to note that, the probability distributions can be any type; however, we use the form of multivariate normal since this type of distribution is well understood, mathematically easy to compute and has been shown to produce good results in many applications [OO05, ZCY⁺07]. When this assumption has been made, each model $\phi_h \in \Phi_G$ presented by a triple $\{\alpha_h, \mu_h, \Sigma_h\}$ that completely characterizes for a Gaussian density function. In our approach, we employ the Expectation-Maximization (EM) technique to approximate this set of parameters Φ_G .

Conceptually, given an initial guess for the parameter values, the EM technique computes the probability that each data point belongs to each distribution and then uses these probabilities to find a new estimate for the parameters such that they maximize the likelihood of the data (this approach is known in statistics as maximum likelihood principle). These two steps are iterated until the estimates of the parameters do not change or the change is smaller than a given threshold ϵ . When such a situation happens, the process converges to an optimal and stable solution and the set of parameters is known to optimally describe these distributions (i.e., clusters).

When applying the EM technique to cluster data streams in sliding window models, a trivial approach is to run this method each time the window is updated with data points in new set of time intervals. However, such an approach is prohibitively expensive and impractical given the assumption that data points arrive and leave the sliding window continuously. Furthermore, the memory space is also not adequate to maintain all raw data points arriving in the window. Therefore, our approach to this problem is from an incremental fashion and data points in the stream are statistically summarized in a form that they are sufficient to approximate the mixture models at any time given. Our SWEM algorithm is divided into two stages. In the first one, raw data points arriving in each time slot are summarized into sufficient statistics that each approximates and represents for a set of data records. These statistics are also called micro components

CHAPTER 6. CLUSTERING DATA STREAMS IN SLIDING WINDOW MODELS WITH EM TECHNIQUE

(or micro clusters) and each is presented by a triple $T_\ell = \{N_\ell, \Gamma_\ell, \theta_\ell\}$ (we shall explain each of these entries latter). In the second stage, these sufficient statistics are then used to update the set of global models (clusters). The update includes both adding new statistics in the new time slot and removing ones in the oldest time interval. Since this second stage requires only the summary statistics as input, the computation turns out to be very effective and efficient.

Certainly, when the stream is summarized into micro components, the number of such patterns provides a trade-off between the storage requirements and the ability to approximate the k global Gaussian distributions. At one extreme where each micro component is used to summarize for a data point, the approximation process becomes the trivial case where the standard EM is applied in the entire raw data records. There is no information lost but the memory usage is maximal. At the other extreme, each micro component is used to summarize a global cluster. The memory space is minimal; however, we may loose much information about the data details and thus lead to a poor quality of approximation in the incremental mining process, especially for the case when the characteristics of the stream dramatically change. Therefore, it is desirable to choose an approach that compromises these two extremes. In our SWEM algorithm design, given that k is the number of clusters in questions, we choose the number of micro components to be $m = a \times k$, where a is a small integer that we will examine in our experiments. Generally, we have $k < m \ll n$, with n is the number of data points arriving in a time slot.

Throughout the computation on data streams, our SWEM algorithm maintains a set of m micro components (MCs) for each time slot and a global set of k clusters which are the overall mixture models for the entire sliding window. At the very beginning time slot of the data stream, a standard EM algorithm is utilized in the first stage of the algorithm to find a mixture model of m micro components. The parameters of this

process is then kept when the algorithm converges into an optimal solution. Then, for each micro component, the algorithm summarizes its characteristics into a triple in which the main features of each micro can be statistically approximated. The second stage of the algorithm is then applied on these sufficient statistics to derive the k global components (clusters). These two stages are incrementally applied when the data stream is advanced with new data points arriving to the system. Nonetheless, for each next time slot, there are several techniques to improve our EM-based clustering process. Consider the case where the characteristics of the stream do not change (or slightly change), it is obvious that the converged parameters of previous time slot is a good initial parameters for the EM algorithm applying in the new time slot. For the other case where significant changes appear in the stream, certainly utilizing previous converged results for the new time slot is not a proper selection. In order to address these evolving stream situations, our SWEM algorithm re-distributes models by informatively splitting and merging micro components. Such discrete moves can avoid the local maximal situation (where components are not well located in the data space).

When the window slides and statistics in expiring time slots need to be removed from the globally clustering results, there are two approaches in our SWEM algorithm. In the first one, which is also a natural way, our algorithm can adapt the EM technique to cluster on entire micro components maintained so far in the sliding window. Such an approach requires the number of input micro components of $m \times b$ and it linearly increases when the window includes more time slots. In SWEM, a second approach is designed to directly update the k global clusters by exploring the characteristics of the m micro components in the expiring time slot. Consequently, the number of components needed is only $m + k$ and it is independent of the size of the sliding window. In the following sections, we present each step of our SWEM algorithm in details.

6.4 Clustering Algorithm

6.4.1 Initial Phase

As mentioned above, we compute m distributions which is also called micro component density functions modelling the data within each time slot of the sliding window. These components are assumed to sufficiently summarize the characteristics of the stream within that time period. Let Φ_L be the set of parameters of these local components, i.e., $\Phi_L = \{\phi_1, \dots, \phi_m\}$, where each micro component MC_ℓ , $1 \leq \ell \leq m$, is also assumed to follow a multivariate Gaussian distribution characterized by three parameters $\phi_\ell = \{\alpha_\ell, \mu_\ell, \Sigma_\ell\}$. For the initial phase where $SW = \{TS_0\}$, we have no information about these parameters. Thus, their initial values will be randomly chosen and the EM algorithm is then used to find the optimal values that best fit the data points arriving in the first stage of the algorithm. When the EM algorithm converges, these micro components are summarized into sufficient statistics forms which then are applied in the second stage of the algorithm to derive the k global clusters of the data stream. We clarify the following concepts and assumptions.

We denote $p_\ell(x|\phi_\ell)$ the probability of the data point x in the ℓ^{th} distribution. Therefore, the probability density function evaluated for a given data point x is computed by the following equation:

$$p(x|\Phi_L) = \sum_{\ell=1}^m \alpha_\ell \times p_\ell(x|\phi_\ell) \quad (6.1)$$

We assume that each data point belongs to all components yet with different probabilities of membership. Given a data point x , its probability membership (or weight) in a micro component ℓ^{th} is computed by:

$$p(\phi_\ell|x) = \frac{\alpha_\ell \times p_\ell(x|\phi_\ell)}{p(x)} = \frac{\alpha_\ell \times p_\ell(x|\phi_\ell)}{\sum_{i=1}^m \alpha_i \times p_i(x|\phi_i)} \quad (6.2)$$

CHAPTER 6. CLUSTERING DATA STREAMS IN SLIDING WINDOW MODELS WITH EM TECHNIQUE

We further make the assumption that d attributes of the stream are independent over all the data points within a given component. Consequently, the density function of a micro component MC_ℓ can be decomposed as a product of density functions over each attribute:

$$p_\ell(x|\phi_\ell) = \prod_{j=1}^d p_{\ell,j}(x^j|\phi_\ell) \quad (6.3)$$

In term of vector and matrix notations, we re-write the above equation as below:

$$p_\ell(x|\phi_\ell) = \frac{1}{(2\pi)^{d/2}|\Sigma_\ell|^{1/2}} \exp \left[-\frac{1}{2}(x - \mu_\ell)^T \Sigma_\ell^{-1} (x - \mu_\ell) \right] \quad (6.4)$$

where μ_ℓ is the d -dimensional mean vector and Σ_ℓ is the covariance matrix of size $d \times d$. $|\Sigma_\ell|$ and $(\Sigma_\ell)^{-1}$ are respectively the determinant and inver matrix of Σ_h . The superscript T denotes the transpose of a vector. It is worth noting that since we assume that each attribute is independent of one another, the covariance matrix Σ_ℓ is a diagonal variance matrix (i.e., zeroes off the diagonal). Therefore, its determinant and inverse matrix can be simply computed. Specifically,

$$|\Sigma_\ell| = \prod_{i=1}^d (\sigma_\ell^i)^2$$

and

$$\Sigma_\ell^{-1} = \begin{bmatrix} 1/(\sigma_\ell^1)^2 & 0 & \dots & 0 \\ 0 & 1/(\sigma_\ell^2)^2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1/(\sigma_\ell^d)^2 \end{bmatrix}$$

where σ_ℓ^2 is the variance of dimension i of micro component MC_ℓ .

In our framework, we assume that data points arriving in the data stream are generated independently and thus the probability of a set of n data points arriving in the

CHAPTER 6. CLUSTERING DATA STREAMS IN SLIDING WINDOW MODELS WITH EM TECHNIQUE

first time slot TS_0 is computed by the product of the probabilities of each individual x_i in this time slot:

$$p(TS_0|\Phi_L) = \prod_{x \in TS_0} p(x_i|\Phi_L) = \prod_{i=1}^n \sum_{\ell=1}^m \alpha_\ell \times p_\ell(x_i|\phi_\ell)$$

Since this probability is small, our algorithm typically works with its log likelihood form. Therefore, the quantity to measure how well the set of micro components approximating the data stream within this time period is computed by the following measure which is called the average log likelihood of the data:

$$\begin{aligned} Q(\Phi_L) &= \frac{1}{|TS_0|} \log \prod_{x \in TS_0} \sum_{h=1}^m \alpha_\ell \times p_\ell(x_i|\phi_\ell) \\ &= \frac{1}{|TS_0|} \sum_{x \in TS_0} \log \sum_{\ell=1}^m \alpha_\ell \times p_\ell(x_i|\phi_\ell) \end{aligned}$$

We apply the EM technique to maximize this quantity. In other words, when this quantity is maximized, we obtain an optimal set of micro components that can be used to summarize the data points arriving in this time interval. Our SWEM algorithm begins with an initial estimation of Φ_L and iteratively updates it until $|Q(\Phi_L^{t+1}) - Q(\Phi_L^t)| \leq \epsilon$. Specifically, given Φ_L^t at iteration t , the algorithm updates it at the iteration $t + 1$ as follows:

In the E-step, for each data point x in TS_0 , SWEM computes its membership probability (i.e., the posterior probability) in each micro component $\ell = 1, \dots, m$:

$$p(\phi_\ell|x) = \frac{\alpha_\ell^{(t)} \times p_\ell(x|\mu_\ell^{(t)}, \Sigma_\ell^{(t)})}{\sum_i \alpha_i^{(t)} \times p_i(x|\mu_i^{(t)}, \Sigma_i^{(t)})} \quad (6.5)$$

subject to

$$p(\phi_\ell|x) > 0, \quad \sum_{\ell=1}^m p(\phi_\ell|x) = 1$$

CHAPTER 6. CLUSTERING DATA STREAMS IN SLIDING WINDOW MODELS WITH EM TECHNIQUE

$$\alpha_\ell^{(t)} > 0, \quad \sum_{\ell=1}^m \alpha_\ell^{(t)} = 1$$

In the M-step, the weight, mean and covariance matrix of each component are updated by the following order:

$$\alpha_\ell^{(t+1)} = \frac{1}{n} \sum_{x \in TS_0} p(\phi_\ell | x) \quad (6.6)$$

$$\mu_\ell^{(t+1)} = \sum_{x \in TS_0} \frac{p(\phi_\ell | x)}{n_\ell} \times x \quad (6.7)$$

$$\Sigma_\ell^{(t+1)} = \sum_{x \in TS_0} \frac{p(\phi_\ell | x)}{n_\ell} \times (x - \mu_\ell^{(t+1)})(x - \mu_\ell^{(t+1)})^T \quad (6.8)$$

where $n_\ell = \sum_{x \in TS_0} p(\phi_\ell | x)$.

If $|Q(\Phi_L^{t+1}) - Q(\Phi_L^t)| \leq \epsilon$ then the iterative process is stopped.

When the algorithm converges, these set of micro components are approximated by only keeping a triple $T_\ell = \{N_\ell, \theta_\ell, \Gamma_\ell\}$ for each micro component MC_ℓ . A hard assignment is applied on the data points. Essentially, let S_ℓ be the set of data points assigned to MC_ℓ (to which they have the highest probability), then:

- $N_\ell = |S_\ell|$, which is the cardinality of the set S_ℓ ;
- $\theta_\ell = \sum_{x_i \in S_\ell} x_i$, which is the linear summation of the data points in the set S_ℓ ; and
- $\Gamma_\ell = \sum_{x_i \in S_\ell} x_i x_i^T$ which is the squared summation of these points.

The important property of a triple T_ℓ is that they are sufficient to compute the vector mean and the covariance matrix of the MC_ℓ as shown by the following equations:

$$\mu_\ell = \frac{1}{N_\ell} \theta_\ell \quad (6.9)$$

CHAPTER 6. CLUSTERING DATA STREAMS IN SLIDING WINDOW MODELS WITH EM TECHNIQUE

$$\Sigma_\ell = \frac{1}{N_\ell} \Gamma_\ell - \left(\frac{1}{N_\ell} \right)^2 \theta_\ell \times \theta_\ell^T \quad (6.10)$$

Furthermore, its additive property guarantees that the mean and covariance matrix of a merged component can also be computed from the triples of each member component: $T_\ell = \{N_{\ell_1} + N_{\ell_2}, \theta_{\ell_1} + \theta_{\ell_2}, \Gamma_{\ell_1} + \Gamma_{\ell_2}\}$, given two member components $T_{\ell_1} = \{N_{\ell_1}, \theta_{\ell_1}, \Gamma_{\ell_1}\}$ and $T_{\ell_2} = \{N_{\ell_2}, \theta_{\ell_2}, \Gamma_{\ell_2}\}$.

Therefore, in our algorithm, these triples are treated as the sufficient statistics summarizing the information regarding the data points within the first time slot of the data stream. These statistics are then used in the second stage of the algorithm to derive the k global clusters in the sliding window. We note that, in this stage, each triple is viewed as a big data point weighted by the number of raw data points assigned to it. Given a global model $\phi_h \in \Phi_G$, $1 \leq h \leq k$, at iteration t^{th} , SWEM updates its parameters at iteration $(t+1)^{th}$ as follows:

E-step: Calculate the posterior probability $p(\phi_h|T_\ell)$:

$$p(\phi_h|T_\ell) = \frac{\alpha_h^{(t)} \times p_h(\frac{1}{N_\ell} \theta_\ell | \mu_h^{(t)}, \Sigma_h^{(t)})}{\sum_{i=1}^k \alpha_i^{(t)} \times p_i(\frac{1}{N_\ell} \theta_\ell | \mu_i^{(t)}, \Sigma_i^{(t)})} \quad (6.11)$$

M-step: Update global parameters by:

$$\alpha_h^{(t+1)} = \frac{1}{n} \sum_{\ell=1}^m N_\ell \times p(\phi_h|T_\ell) \quad (6.12)$$

$$\mu_h^{(t+1)} = \frac{1}{n_h} \sum_{\ell=1}^m p(\phi_h|T_\ell) \times \theta_\ell \quad (6.13)$$

$$\Sigma_h^{(t+1)} = \frac{1}{n_h} \left[\sum_{\ell=1}^m p(\phi_h|T_\ell) \Gamma_\ell - \frac{1}{n_h} \sum_{\ell=1}^m (p(\phi_h|T_\ell) \theta_\ell) (p(\phi_h|T_\ell) \theta_\ell)^T \right] \quad (6.14)$$

where $n_h = \sum_{\ell=1}^m N_\ell \times p(\phi_h|T_\ell)$.

It is worth noting that Equation 6.14 provides a method to compute a global covariance matrix based on the weighted covariance matrix of each micro component (which

summarizes spread of raw data points). As such, the information details regarding raw data are well approximated.

6.4.2 Incremental Phase

This phase is incrementally applied when data in a new time slot arrive to the system. Different from the initial phase where our SWEM algorithm has to randomly choose the initial values for micro components, in this phase, SWEM utilizes the converged parameters in previous time slot as the initial values for the parameters of the mixture models. The rationale for this approach is that we may expect the characteristics of the stream not to change or slightly change between two consecutive time slots and thus inheriting the converged parameters of previous step can minimize the number of iterations in the new time slot.

In essence, after the first iteration on the data points arriving in the new time slot, our algorithm compares the quantity $Q(\Phi_L)$ with the converged one in the previous time slot. If the two values are not much different, then it is safe to say that the distribution in the new time interval is similar to the one in the previous time interval. Otherwise, it may be learnt that such a set of initial parameters is not a proper choice for the current changing distribution. It is important to note that the local maximal problem may arise if the initial parameters of the mixture models are not properly chosen. Therefore, in order to avoid this issue, two split and merge operations are proposed in SWEM to discretely re-distribute micro components in the data space.

Figure 6.1 illustrates the idea of SWEM's split operation. A micro component will be chosen for dividing if it is big enough and has the highest variance sum (its data are mostly spread). The dimension chosen for the split is the one with the maximum variance. A component MC_ℓ is identified big if its weight $\alpha_\ell > \frac{2}{m} \sum_i \alpha_i$. Assume that, MC_ℓ is the most spread component and dimension e is chosen for the split. Let $g_{\ell,e}(x|\phi_\ell) =$

CHAPTER 6. CLUSTERING DATA STREAMS IN SLIDING WINDOW MODELS WITH EM TECHNIQUE

6.1.a: Before Splitting

6.1.b: After Splitting

Figure 6.1: A splitting visualization.

$2p_{\ell,e}(x|\phi_\ell)$, then the new means and variances for that dimension in two new micro components MC_{ℓ_1} and MC_{ℓ_2} are approximated as follows:

$$\mu_{\ell_1}^e = \int_{\mu_\ell^e - 3\sigma_\ell^e}^{\mu_\ell^e} x \times g_{\ell,e}(x|\phi_\ell) dx \quad (6.15)$$

$$\mu_{\ell_2}^e = \int_{\mu_\ell^e}^{\mu_\ell^e + 3\sigma_\ell^e} x \times g_{\ell,e}(x|\phi_\ell) dx \quad (6.16)$$

and

$$(\sigma_{\ell_1}^e)^2 = \int_{\mu_\ell^e - 3\sigma_\ell^e}^{\mu_\ell^e} x^2 \times g_{\ell,e}(x|\phi_\ell) dx - (\mu_{\ell_1}^e)^2 \quad (6.17)$$

$$(\sigma_{\ell_2}^e)^2 = \int_{\mu_\ell^e}^{\mu_\ell^e + 3\sigma_\ell^e} x^2 \times g_{\ell,e}(x|\phi_\ell) dx - (\mu_{\ell_2}^e)^2 \quad (6.18)$$

For other dimensions, their mean and variance values are kept unchanged. The above computations are derived given the assumption that each dimension is to follow a Gaussian distribution in which 99.7% of data within $\mu \pm 3\sigma$. Therefore, the integrals with lower and upper bounds chosen in this range can approximately cover all component's data points projecting on that dimension of the micro component.

CHAPTER 6. CLUSTERING DATA STREAMS IN SLIDING WINDOW MODELS WITH EM TECHNIQUE

6.2.a: Before Merging

6.2.b: After Merging

Figure 6.2: A merging visualization.

It is also noted that in the above formulae, $g_{\ell,e}(x|\phi_{\ell})$ is set to be equal to $2p_{\ell,e}(x|\phi_{\ell})$ since the range of each integral covers only a half of the range of the original density distribution function $p_{\ell,e}(x|\phi_{\ell})$ (which being assumed $\int_{\mu_{\ell}^e-3\sigma_{\ell}^e}^{\mu_{\ell}^e+3\sigma_{\ell}^e} p_{\ell,e}(x|\phi_{\ell})dx = 1$), meanwhile we also want to use $g_{\ell,e}(x|\phi_{\ell})$ to approximate the density distribution function of each new component (w.r.t dimension e).

For the merging operation, Figure 6.2 illustrates our approach. Two components are merged if they are small and close enough. A micro component MC_{ℓ} is identified small if its weight $\alpha_{\ell} < \frac{1}{2m} \sum_i \alpha_i$. The closeness between two components is measured based on the Mahalanobis distance. Compared to the Euclidean distance, the Mahalanobis one works more effectively in our SWEM framework since it takes into account the covariance matrix which essentially describes the data spread of the component. In SWEM, we define the average squared Mahalanobis distance between two components as follows:

$$Avg(D_{i,j}) = \frac{1}{2}[(\mu_i - \mu_j)^T \Sigma_j^{-1} (\mu_i - \mu_j) + (\mu_j - \mu_i)^T \Sigma_i^{-1} (\mu_j - \mu_i)] \quad (6.19)$$

where μ_i and μ_j are the mean vectors of two micro components. Correspondingly, Σ_i and Σ_j are their covariance matrices. Given two MC_{ℓ_1} and MC_{ℓ_2} having smallest average

CHAPTER 6. CLUSTERING DATA STREAMS IN SLIDING WINDOW MODELS WITH EM TECHNIQUE

squared Mahalabonis distance are merged into MC_ℓ , the parameters for the merged component MC_ℓ is computed as follows (based on the additive property presented in Section 6.4.1):

$$\alpha_\ell = \alpha_{\ell_1} + \alpha_{\ell_2} \quad (6.20)$$

$$\mu_\ell = \frac{\alpha_{\ell_1}}{\alpha_{\ell_1} + \alpha_{\ell_2}} \times \mu_{\ell_1} + \frac{\alpha_{\ell_2}}{\alpha_{\ell_1} + \alpha_{\ell_2}} \times \mu_{\ell_2} \quad (6.21)$$

$$\Sigma_\ell = \frac{\Gamma_\ell}{n(\alpha_{\ell_1} + \alpha_{\ell_2})} - \frac{\theta_\ell \times \theta_\ell^T}{(n(\alpha_{\ell_1} + \alpha_{\ell_2}))^2} \quad (6.22)$$

in that,

$$\theta_\ell = n \times (\alpha_{\ell_1} \times \mu_{\ell_1} + \alpha_{\ell_2} \times \mu_{\ell_2})$$

and

$$\Gamma_\ell = n[\alpha_{\ell_1}(\Sigma_{\ell_1} + \mu_{\ell_1}\mu_{\ell_1}^T) + \alpha_{\ell_2}(\Sigma_{\ell_2} + \mu_{\ell_2}\mu_{\ell_2}^T)]$$

where n is the number of data points in the current time slot.

Two above fundamental operations will be applied in our SWEM algorithm to re-distribute micro components if it is detected that the data distribution in the new time slot is significantly different from the one in previous time interval. When the algorithm converges into a new optimal solution, again, m micro components are then summarized into m triples similarly to the initial phase. These patterns are treated as the sufficient statistics reflecting the data stream distribution in the current time slot. Also, the k global clusters are updated with these statistics. This second stage of the algorithm is similar to the second stage in the initial phase when SWEM approximates k global clusters from m micro components (formulae 6.11-6.14). However, in this incremental phase, the algorithm derives the k global models from $(m + k)$ components.

6.4.3 Expiring Phase

This phase of the SWEM algorithm is applied when the window slides and the oldest time slot is expiring. It is necessary to remove the effect of the expiring time slot from the final k global clusters. There are two approaches to address the problem. In the first one which is also the natural approach, we may re-cluster entire micro components that have been summarized and maintained for each time slot so far in the sliding window. However, this approach is not effective in terms of processing time since it has to compute on $(b \times m)$ components. The problem is worse when the number of time slots in the sliding window is extended. In SWEM, we propose a second approach that directly updates the current k global clusters by using only the information summarized in the expiring time slot. The number of components needed to manipulate is thus fixed by $(k + m)$. Therefore, this approach is independent of the size of the sliding window.

For ease of presentation, in this section we imply that the index h will run on the set of the global models, i.e., $1 \leq h \leq k$; and the index ℓ runs on the set of local micro components in the expiring time slot, i.e., $1 \leq \ell \leq m$. Given the k global models characterized by $\Phi_G = \{\phi_1, \dots, \phi_k\}$, we need to update them with the subtraction of the sufficient statistics summarized in the m local components $\Phi_L = \{\phi_1, \phi_2, \dots, \phi_m\}$ in the expiring time slot. Recall that each ϕ_ℓ is presented by a triple $T_\ell = \{N_\ell, \theta_\ell, \Gamma_\ell\}$.

In essence, the SWEM algorithm is an iterative process that converges to statistical models that optimally describe the data. Therefore, in order to remove the effect of micro components appearing in the expiring time slot from the global models, our algorithm removes their sufficient statistics gradually. We use a fading factor $\lambda (< 1)$ to control this process. The closer to 1 the λ is, the smaller the amount of reducing weights is eliminated at each iteration. Therefore, this factor provides a method to prevent the effect of each expiring component from reducing too fast or too slow, which would cause the problem

CHAPTER 6. CLUSTERING DATA STREAMS IN SLIDING WINDOW MODELS WITH EM TECHNIQUE

of local optimal situation convergence in SWEM algorithm. At each iteration t^{th} of the algorithm, we reduce the weight of each expiring model MC_ℓ by the following equation:

$$N_\ell^{(t)} = \lambda^{(t)} N_\ell \quad (6.23)$$

which means the reducing amount, denoted by $r_\ell^{(t)}$, is

$$r_\ell^{(t)} = (1 - \lambda)\lambda^{(t-1)} N_\ell \quad (6.24)$$

Theoretically, the following theorem guarantees that the number of iterations t can be arbitrarily large meanwhile the total number of reducing weights on each expiring component will approach to (and never exceed) its original value.

Theorem 6.5 *Let t be an arbitrary number of iterations used by the SWEM algorithm. Then, for each expiring micro component MC_ℓ :*

$$\lim_{t \rightarrow \infty} \sum_t r_\ell^{(t)} = N_\ell$$

Proof. At the first iteration, $N_\ell^{(1)} = \lambda N_\ell$, and thus the reducing amount is $r_\ell^{(1)} = N_\ell - \lambda N_\ell = (1 - \lambda)N_\ell$.

At the second iteration, $N_\ell^{(2)} = \lambda N_\ell^{(1)} = \lambda^2 N_\ell$ and $r_\ell^{(2)} = N_\ell^{(1)} - \lambda N_\ell^{(1)} = (1 - \lambda)\lambda N_\ell$.

By induction, at the iteration t^{th} , the reducing weight is $r_\ell^{(t)} = (1 - \lambda)\lambda^{(t-1)} N_\ell$.

Therefore, the total reducing amount so far is:

$$\begin{aligned} \sum_t r_\ell^{(t)} &= (1 - \lambda)N_\ell + (1 - \lambda)\lambda N_\ell + \dots + (1 - \lambda)\lambda^{(t-1)} N_\ell \\ &= (1 - \lambda)N_\ell [1 + \lambda + \dots + \lambda^{t-1}] \end{aligned}$$

It is clear that:

$$\begin{aligned}
 \lim_{t \rightarrow \infty} (1 - \lambda)[1 + \lambda + \dots + \lambda^{t-1}] &= \lim_{t \rightarrow \infty} (1 + \lambda + \dots + \lambda^{t-1} - \lambda - \lambda^2 - \dots - \lambda^t) \\
 &= \lim_{t \rightarrow \infty} (1 - \lambda^t) \\
 &= 1
 \end{aligned}$$

since $\lambda < 1$. □

Given the set parameters of k global models at iteration t , the E-step computes the posterior probability of each micro component $T_\ell = \{N_\ell, \theta_\ell, \Gamma_\ell\}$ in the expiring time slot with respect to each of k global models at iteration $t + 1$ by the following equation:

$$p(\phi_h | T_\ell) = \frac{\alpha_h^{(t)} \times p_h(\frac{1}{N_\ell} \theta_\ell | \mu_h^{(t)}, \Sigma_h^{(t)})}{\sum_{i=1}^k \alpha_i^{(t)} \times p_i(\frac{1}{N_\ell} \theta_\ell | \mu_i^{(t)}, \Sigma_i^{(t)})} \quad (6.25)$$

At the M-step of iteration t , these posterior probabilities will be subtracted from the global models. Specifically,

First, the number of data points in the sliding window which we denote by n_G is updated by:

$$n_G^{(t+1)} = n_G^{(t)} - \sum_{\ell=1}^m r_\ell^{(t+1)} \quad (6.26)$$

In this equation, the second factor on the RHS is the total number of data points removed when applying equation 6.24 for every expiring micro component at iteration $t + 1$. Given this new value, the weight of each global model ϕ_h is simply updated by:

$$\alpha_h^{(t+1)} = \frac{n_h^{(t+1)}}{n_G^{(t+1)}} \quad (6.27)$$

where $n_h^{(t+1)}$ is computed by:

$$n_h^{(t+1)} = \alpha_h^{(t)} \times n_G^{(t)} - \sum_{\ell=1}^m p(\phi_h | T_\ell) \times r_\ell^{(t+1)}$$

CHAPTER 6. CLUSTERING DATA STREAMS IN SLIDING WINDOW MODELS WITH EM TECHNIQUE

The first factor on the RHS is the estimated number of data points belonging to the global model ϕ_h in previous iteration t while the second factor is its total number of data points (weighed by the posterior probability) now expiring in iteration $t + 1$.

Before updating two other parameters, the vector mean μ_h and the covariance matrix Σ_h , of each global model ϕ_h , we compute the following value:

$$\theta_h^{(t+1)} = \theta_h^{(t)} - \sum_{\ell=1}^m p(\phi_h | T_\ell) \times r_\ell^{(t+1)} \times \frac{\theta_\ell}{N_\ell}$$

Note that, in the RHS's second factor, the value θ_ℓ/N_ℓ , in fact is the vector mean of the expiring micro component (which does not change during this process).

Now $\mu_h^{(t+1)}$ and $\Sigma_h^{(t+1)}$ are updated by:

$$\mu_h^{(t+1)} = \frac{\theta_h^{(t+1)}}{n_h^{(t+1)}} \quad (6.28)$$

$$\Sigma_h^{(t+1)} = \frac{1}{n_h^{(t+1)}} \left[\Gamma_h^{(t+1)} - \frac{1}{n_h^{(t+1)}} \theta_h^{(t+1)} \theta_h^{(t+1)T} \right] \quad (6.29)$$

where

$$\Gamma_h^{(t+1)} = \Gamma_h^{(t)} - \sum_{\ell=1}^m \left[p(\phi_h | T_\ell) \times r_\ell^{(t+1)} \times \frac{\theta_\ell}{N_\ell} \right] \left[p(\phi_h | T_\ell) \times r_\ell^{(t+1)} \times \frac{\theta_\ell}{N_\ell} \right]^T$$

Note that

$$\Gamma_h^{(t)} = n_h^{(t)} \left[\Sigma_h^{(t)} + \mu_h^{(t)} \mu_h^{(t)T} \right]$$

During this iterative process, whenever any global model having weight smaller than a threshold, it is safe to remove it from the mining results. This happens if a global model was formed by only data points arriving in the expiring time slot and thus, when this time interval is beyond the sliding window, that global model is eliminated as well. When a global model is deleted, another one which has the highest summation on variances should

be split in order to keep k , the number of global clusters, unchanged. The computation for a global model splitting is similar to the case with splitting micro components presented in Section 6.4.2.

Finally, the memory space of our SWEM algorithm is guaranteed by the following theorem.

Theorem 6.6 *The memory space consumption of SWEM is $O(n + (d^2 + d + 1)(mb + k))$, where n is the maximal number of data points arriving in a time slot and b is the number of time slots within a sliding window.*

Proof. The memory consumption includes two parts. The first part is the data buffer storing the new data points. We assume that n is the maximal number of data points that can arrive in one time slot. The second part is the space for recording the parameters of the mixture Gaussian models at each each time slot as well as the entire sliding window. For a Gaussian mixture model of k global components, the memory consumption includes k weights α_h , k d -dimensional mean vectors μ_h , and k $d \times d$ covariance matrices. At each time slot of the sliding window, the memory space needed to maintain m micro components, each characterized by a weight N_ℓ , a d -dimensional vector sum θ_ℓ and a $d \times d$ matrix of squared sum Γ_ℓ . There are b time slots in the sliding window and therefore, in total, the memory consumption of the algorithm is $O(n + (d^2 + d + 1)(mb + k))$.

□

6.5 Experimental Results

6.5.1 Experimental Setup

All of our experiments are conducted on a 1.9GHz PentiumIV PC with 1GB memory space running on Windows XP platform. We implement three algorithms, the standard Expectation Maximization technique denoted by stdEM, our algorithm SWEM and its

variation SWEM(w/oG) in C++ and compile them using Microsoft Visual C++ version 6.0. Recall that SWEM(w/oG) is different from SWEM in the way it updates the global clustering model in the expiring step. While SWEM directly removes expiring statistics from the global clustering model, SWEM(w/oG) re-clusters entire micro components to derive it (i.e., without using global clustering models).

In the following experiments, to describe a dataset we use D to denote its dimensions, K to denote its number of clusters, and N to denote its size in terms the number of data records.

6.5.2 Clustering Quality Evaluation

To test the clustering quality results, we generate three datasets, each with 100,000 data records, in that both the number of dimensions and clusters are varied from 2 to 10. The data points of each dataset will follow a series of Gaussian distributions. With the above notations, three datasets are respectively denoted $D2.K10.N100k$, $D4.K5.N100k$ and $D10.K4.N100k$. Unless otherwise mentioned, we use the following parameter setting in our experiments: the number of micro components $m = 6K$ (where K is the number of global clusters), the error difference between average log likelihoods $\epsilon = 0.001$, the merging threshold based on Mahalanobis distance $Avg(D_{i,j}) = 1$ and the fading factor $\lambda = 0.8$. The quality of clustering results is evaluated by using the average log likelihood measure (the closer to zero the value, the better the quality of clustering analysis).

Table 6.1 shows the average log likelihood results returned by three algorithms where datasets are divided into 10 time intervals, each with 10k of data points. The sliding window consists of 5 time slots and we report the results after each of the 2 time slots. It can be seen that, in all cases, our SWEM and SWEMw/oG algorithms almost obtain the accuracy close to that of the stdEM algorithm working without any stream mining constraints. It is important to note that, these data streams are processed incrementally

CHAPTER 6. CLUSTERING DATA STREAMS IN SLIDING WINDOW MODELS WITH EM TECHNIQUE

TS	D2.K10.N100k			D4.K5.N100k			D10.K4.N100k		
	stdEM	w/oG	SWEM	stdEM	w/oG	SWEM	stdEM	w/oG	SWEM
2	-10.436	-10.512	-10.512	-19.252	-19.276	-19.276	-47.846	-47.869	-47.869
4	-10.427	-10.446	-10.446	-19.192	-19.215	-19.215	-47.933	-48.010	-48.010
6	-10.451	-10.604	-10.716	-19.164	-19.220	-19.326	-47.702	-47.712	-47.726
8	-10.444	-10.700	-10.735	-19.188	-19.226	-19.245	-47.859	-47.884	-47.886
10	-10.439	-10.523	-10.579	-19.202	-19.247	-19.258	-47.759	-47.820	-47.873

Table 6.1: Average log likelihood values returned by stdEM, SWEMw/oG and SWEM algorithms

in our algorithms whilst it is assumed that stdEM can observe and process entire data points appearing in the sliding window together and at the same time.

		D4.K5.N100k				D4.K5.N100k with noise		
		TM	stdEM	w/oG	SWEM	stdEM	w/oG	SWEM
C1	Dim 1	-165	-165.861	-165.618	-165.930	-163.054	-161.716	-162.050
	Dim 2	281	282.269	282.597	282.797	280.152	279.594	276.758
	Dim 3	-114	-114.070	-113.65	-113.800	-112.744	-110.741	-109.878
	Dim 4	175	175.863	176.609	176.471	172.770	172.969	172.368
C2	Dim 1	124	122.275	122.365	121.860	123.955	123.915	123.539
	Dim 2	-127	-125.064	-125.412	-125.454	-115.209	-116.539	-122.902
	Dim 3	188	188.376	188.3	188.527	179.520	177.993	186.276
	Dim 4	92	91.753	91.9252	91.523	89.571	91.093	97.919
C3	Dim 1	-3	-1.918	-1.90395	-1.745	-3.732	-2.686	-2.377
	Dim 2	224	223.657	223.699	223.446	222.349	222.635	222.477
	Dim 3	-53	-52.288	-52.2454	-52.113	-52.760	-51.036	-50.682
	Dim 4	-176	-175.382	-175.045	-175.102	-175.299	-174.635	-174.607
C4	Dim 1	295	297.043	297.839	297.536	295.111	294.120	296.555
	Dim 2	155	155.647	155.704	156.406	154.611	153.671	154.623
	Dim 3	276	275.964	275.681	275.236	275.875	274.569	274.624
	Dim 4	-73	-72.912	-73.3848	-73.182	-73.159	-75.363	-77.620
C5	Dim 1	245	246.302	246.922	246.851	245.685	245.970	243.827
	Dim 2	11	10.4482	8.990	9.525	11.182	14.011	9.430
	Dim 3	-154	-152.044	-152.077	-152.012	-155.230	-153.562	-152.924
	Dim 4	153	152.852	153.947	153.555	153.428	153.834	152.462

Table 6.2: Clustering means on $D4.K5.N100k$ and $D4.K5.N100k$ with 5% noise

Table 6.2 provides more details on these algorithms where we list the true means

Figure 6.3: Average log likelihood on D4.K5.N200k

(TM column) and approximate means of all clusters returned by each algorithm on the dataset $D4.K5.N100k$ (at its last time slot). Also with this dataset, we further add 5% of random noise and the last three columns in the Table 6.2 report these results. It is interesting to observe that the approximate means returned by our algorithms are still nearly the same to those of the true clusters despite the noise appearance.

In the above experiments, we have kept the distribution generating each data stream stable. In the following experiments, in order to provide more insights into our algorithms, we generate a data stream with changing distributions. We choose the number of dimensions $D = 4$ and the number of clusters $K = 5$ and randomly generate two different distributions, each with 100k data points. Consequently, the dataset $D4.K5.N200k$ with changing characteristics is formed by merging these two distributions. The size of each time slot is set to 20k data points and the sliding window consists of 5 time intervals. Figures 6.3 and 6.4 show the clustering accuracy and the accumulative processing time of three algorithms. In all cases, the average log likelihood of stdEM is the largest. This happens since stdEM is allowed to access all data points in the sliding window whilst

CHAPTER 6. CLUSTERING DATA STREAMS IN SLIDING WINDOW MODELS WITH EM TECHNIQUE

Figure 6.4: Accumulative processing time on D4.K5.N200k

SWEM and SWEMw/oG are allowed to work only on their statistics summarized in previous time slots. They are only able to access raw data points in the current time slot. However, in all time slots reported, we can see that SWEM and SWEMw/oG's average log likelihood measures are still very close to those of stdEM.

The accumulative processing time of SWEM and stdEM is reported in Figure 6.4. In this figure, it can be observed that at the beginning of the stream, stdEM outperforms SWEM in terms of accumulative processing time. This is because while stdEM directly updates K global clusters, SWEM has to derive a sufficient amount of summary information consisting of m micro components ($m = 6K$) in order to maintain the clustering quality. Such a process requires a considerable amount of time. However, when more and more time slots arrive in the stream, the accumulative processing time of stdEM is higher than that of SWEM. This happens since with stdEM, it has to cluster all data points in the entire sliding window in order to obtain the global clustering results. On the contrary, SWEM is an incremental method and it has only to cluster data points in each new time slot and update these statistics into the current maintaining global mod-

els. Furthermore, SWEM also benefits from its design when the distribution of the data stream does not dramatically change. As can be clearly seen from Figure 6.4, the most time consuming is at the 1st and 6th time slots where the algorithm detects significant changes in the distribution. For other time slots, SWEM quickly converges after several iterations since it utilizes the converged parameters of the previous time slot as its initial ones for a new time slot arriving in the stream.¹

In the above experiments, we do not show the time to process SWEMw/oG since its time is almost identical to that of SWEM due to the small number of time slots of the sliding window (5 time slots). Recall that SWEM and SWEMw/oG are only different in the way they use the sufficient statistics (i.e., micro components) to update the global clustering model in the expiring phase. While SWEMw/oG re-clusters all micro components summarized in each time slot of the sliding window to generate global clusters, SWEM updates the global clustering results by gradually removing sufficient statistics summarized in the expiring time slot. It is trivial to realize that in the expiring phase, SWEMw/oG's time increases linearly with the size of the sliding window (since the number of micro components is incremented) whilst the time of its counterpart SWEM is independent of the sliding window's size. However, as observed from previous experiments, the clustering results of both algorithms are almost the same and very close to those of the stdEM algorithm.

6.5.3 Parameter Sensitivity

By using the above changing distribution dataset *D4.K5.N200k*, in this section we test the sensitivity of various parameter settings on the accuracy of SWEM algorithm.

¹In these experiments, if the difference between the first iteration's average log likelihood of a new time slot and the converged one of the previous time slot is more than 10ϵ then the stream distribution is considered significantly changed.

Figure 6.5: Micro Components vs. Accuracy

Micro Components Varying In Section 6.3, we indicated that the number of micro components should be larger than the number of natural clusters in order to obtain a clustering of good quality. Nevertheless, a very large number of micro components is inefficient in terms of running time and memory storage since it is clear that the complexity of the first stage of the algorithm increases linearly with the number of micro components. To provide more insights into our algorithm regarding this parameter, we run the SWEM with $D4.K5.N200k$ where the ratio between the number of micro components and the natural clusters is varied from 2 to 10. Figure 6.5 reports the experimental results where we compute the average accuracy on all time slots. We can see that if the number of micro components is close to the number of global clusters, the clustering quality is poor. This is because a very small number of micro components is harder to approximate the characteristics of data streams, especially in the situation where the distribution changes. A poor approximation often causes worse in later phases where the SWEM needs to remove expiring information. When the number of micro components increases, the average log likelihood also increases. However, it becomes

Figure 6.6: Merging threshold $Avg(D_{i,j})$ vs. Accuracy

relatively stable when the number of micro components is set around $m = 6K$. This indicates that we do not need to set the number of micro components too large, yet still can achieve a high clustering accuracy.

Merging Threshold Varying Recall that the merging and splitting operations are only invoked to re-distribute micro components when SWEM detects that there is a significant change in data stream distributions. Furthermore, the algorithm always maintains a set of m micro components at each time slot. When two components are merged, there should be another one with biggest summation in variances being split. Therefore, in this section we only report the clustering results where the merging threshold $Avg(D_{i,j})$, which is the squared Mahalanobis distance, is being varied. Figure 6.6 reports our results at the time slot 6 at which the algorithm detects the significant change in the data distribution. From the figure, we make the observation that when the merging threshold is either being set to be too small or too large, the average log likelihood results are poor. This happens since when $Avg(D_{i,j})$ is set too small, there are almost no micro compo-

Figure 6.7: Fading Factor vs. Accuracy

nents being merged although they may be close to each other. In the other extreme, when $Avg(D_{i,j})$ is set too large, many micro components are frequently merged and split which causes the poor results on clustering analysis. The highest average log likelihood is achieved when this merging threshold is set to be round 1.

Fading Factor Varying The last parameter that may impact the clustering quality of SWEM is the fading factor λ . Figure 6.7 shows the relationship between the fading factor λ , the number of iterations used in expiring phases, and the average log likelihood quality. The results are reported at the time slot 10 where the last interval of the first distribution is eliminated from the sliding window. As expected, when λ is set closer to 1, the algorithm reduces the weight of each expiring micro component slowly and needs more iterations. Accordingly, the quality of clustering results is better indicating by the larger average log likelihood value. In order to obtain high quality of clustering results, the best value of λ should be set between 0.8 and 0.9. Note that, since this expiring step is executed in the second stage of the algorithm where SWEM works only with micro components and global clusters which in fact are only a small number of weighted

data points; therefore the large number of iterations utilizing in this stage is generally acceptable.

6.5.4 Comparison with CluStream

CluStream [AHWY03] is an algorithm proposed to cluster an evolving data stream in a landmark window model. Nevertheless, in this section, we make a very simple modification to CluStream in order to make it work in a sliding window model. As such, it is able to compare its performance with that of SWEM algorithm.

As presented in Chapter 2, the clustering technique used in CluStream is the k -means method. During its clustering process, CluStream updates incoming data points into a set of micro clusters that it has maintained so far in the stream. Periodically, a snapshot is taken on this set of micro clusters. When adapting CluStream to a sliding window model, we do not store the captured snapshots at different levels of granularity (as presented in [AHWY03]). Instead, each snapshot is maintained precisely at every time slot and those that are expired from the sliding window will be deleted immediately. The time horizon in CluStream is also chosen equal to the size of the sliding window. Furthermore, as indicated in [AHWY03], the factor 2 of the root mean square (RMS) deviation (used to define the maximal boundary of a micro cluster) usually produces the best clustering results. Therefore, we will run CluStream's experiments with this value.

It is also important to note that, in CluStream, only data points arriving in the first time slot are clustered until the k -means method converges (i.e., with multiple iterations on the data). For the rest of the stream, each arriving data point is clustered online and it is either absorbed by one micro cluster or forms itself as a new micro cluster. Hence, the convergence of the algorithm is not guaranteed after the first time slot. However, we have realized that such a process is only effective when the stream distribution is relatively stable over time. When the stream distribution significantly changes, the clustering

Figure 6.8: Clustering quality comparison on D2.K10.N100k

results of CluStream also reduce remarkably. Therefore, in the following experiments, we let CluStream perform multiple iterations on data points of each time slot until the k -means method converges (the error tolerance is set to 0.001). This may compromise the processing time but improve the clustering quality of CluStream. Also, it is fair to compare its results with those of SWEM (which is also allowed to iterate until converged).

The number of micro components used in SWEM and the number of micro clusters used in CluStream are set the same. They are chosen equal to $6K$ (where K is the true number of clusters) and are maintained precisely at each time slot of the sliding window. Therefore, both algorithms use fixed memory space. We compare SWEM and CluStream on their clustering quality and execution time.

To evaluate the clustering quality of two algorithms, we use the sum of squared distance (SSQ) measure which is computed based on the L_2 (standard Euclidean) distance of each data point to its closest cluster's center.² For this reason, we shall convert the

²This measure has also been used in CluStream [AHWY03] as well as in other clustering methods [CT07, CMM02].

Figure 6.9: Clustering quality comparison on D10.K4.N100k

clustering results obtained from SWEM by assigning each data point to exactly one cluster to which it has the highest probability.

In Figures 6.8 and 6.9 (with 5% of random noise), we report the SSQ measures of two algorithms on datasets D2.K10.N100k and D10.K4.N100k. Each dataset is divided into 10 time slots (each with 10k of transactions) and the sliding window consists of 5 time slots. As observed from the figures, the SSQ value of SWEM at various times is always less than that of CluStream. For example, at the time slot 6 of D10.K4.N100k, the SSQ value of CluStream is 462,679,966 whereas that of SWEM is only 348,915,129. This implies that data points in each of the clusters obtained by SWEM are more similar than those obtained by CluStream. To explain for this, note that CluStream computes micro clusters' centers based on the assignment of each data point to exactly one micro cluster to which it has the closest distance. The set of micro clusters derived from such a hard assignment are often highly affected by the noise and outlier data points since a small number of noise data can substantially influence the computation of the micro clusters' means [HK01, TSK05]. Meanwhile SWEM, which is based on the EM technique, allows

Figure 6.10: Clustering quality comparison on changing distribution data streams

each data point belonging to all micro components yet with different probability. Such a soft assignment is usually less affected by the noise and missing data and can produce a better quality on approximating a set of micro components. Consequently, SWEM's global clusters are also derived more accurately.

To provide more insights, it is also interesting to observe the clustering quality of two algorithms on a changing distribution data stream. We use the dataset D4.K5.N100k which consists of 10 time slots (each having 10k data points) for this purpose. Data points in the first five time slots are randomly generated from one distribution and those in the last five time slots are generated by using another distribution. Figure 6.10 shows the clustering quality of two algorithms on this changing distribution dataset in which we set the window's size to 4.³

As observed from Figure 6.10, the SSQ values of both SWEM and CluStream are linearly increased in the first four time slots of the dataset. This is because the sliding window is in its initial phase where data points are accumulated to the window. When

³This allows the window slide once within each distribution.

CHAPTER 6. CLUSTERING DATA STREAMS IN SLIDING WINDOW MODELS WITH EM TECHNIQUE

the window slides to the time slot 5, the SSQ values of both algorithms slightly change since data in the first 5 time slots are generated by using the same distribution. However, it can be observed that the clustering quality of CluStream is remarkably worse than that of SWEM at the time slot 6 (at which the stream distribution significantly changes) and the subsequent ones. This can be explained by that, in the design of CluStream, when a new data point is determined far from the set of micro clusters and cannot be absorbed by any one, CluStream simply creates a new micro cluster for that point and merges the other micro clusters (in order to keep the number of micro clusters unchanged). As such, some new micro clusters have only one or a few points whereas the others have many. This causes the weights of micro clusters very imbalance and usually leads to a poor result on approximating new changing characteristics of the stream.

On the contrary, in SWEM, when the stream distribution is detected significantly changed, the set of micro components will be re-distributed by applying both merging and splitting operations on the set of micro components. A large micro component having the highest variance sum will be split, and two small micro components will be merged if they are close enough. Such operations not only re-distribute micro components discretely in the data space, it also manages to make the weight of each micro component not much different from one another. Consequently, SWEM is very effective in approximating new characteristics in the stream. As can be seen from the Figure 6.10, the SSQ value at the time slot 10 of SWEM is only 180,526,392 whereas that of CluStream is 233,362,289.

In order to evaluate the execution time of two algorithms, we continue using the changing distribution dataset D4.K5.N100k. The maximal number of iterations (until converge) of both algorithms is set to 50 and the execution time is measured in terms of the number of iterations. Figure 6.11 reports the execution time of both algorithms at each time slot of the data stream. As can be seen from this figure, at the first time slot, SWEM uses a slightly more number of iterations than that of Clustream. At the next

Figure 6.11: Execution time comparison on changing distribution data streams

four time slots, the number of iterations of both algorithms reduces considerably since the distribution of the dataset is relatively stable. At the critical time slot 6 where the stream distribution is significantly changed, the number of iterations of two algorithms reaches the maximum value. However, it is interesting to observe the time in the subsequent time slots. The number of iterations of SWEM reduces sharply whereas that of CluStream remains high. For example, at time slot 8, the number of iterations of SWEM is only 5 while that of CluStream is 28.

To explain for this, note that SWEM computes a set of micro components based on the data points arriving in each time slot separately. Nevertheless, it always uses the converged parameters of the previous time slot as the initial parameters for the set of micro components in the next time slot. Consequently, the clustering process can quickly converge if the stream distribution does not change much between two consecutive time slots. On the other hand, CluStream always tries to update new data points arriving in each new time slot into the set of micro clusters that it has maintained so far in the

stream, regardless of how much the current distribution is different from the previous one in the past. Consequently, it needs more time to make the algorithm converge in the case where the distribution is changing remarkably. As observed from Figure 6.11, the difference in processing time of two algorithms is clearly reflected in the last four time slots of the figure.

6.6 Summary

Cluster analysis has been known to play an important role in data understanding. When such an important data mining task is extended to the context of data streams, it becomes more challenging since streaming data arrive to a mining system sequentially and the data are almost unbound meanwhile the memory space of the system is generally confined. The problem is even more difficult when the clustering task is considered in the sliding window model in which the elimination of outdated data must be properly dealt with. In this chapter, we have formulated the problem of clustering data streams in a time-based sliding window and the SWEM algorithm is then proposed to deal with it. Data records in a stream are incrementally processed and only their statistics, being presented in very compact forms of micro components, are summarized and maintained in the main memory. Based on these sufficient statistics, SWEM can update and derive a set of global clustering models in window sliding modes. The problem of distribution changing over time, which usually appears in evolving data streams, is also well addressed. We have proposed and developed the splitting and merging operations for re-distributing micro components discretely in order to approximate new characteristics of data streams. The SWEM algorithm works effectively and efficiently in the sliding window model indicating by its clustering models close to those returned by algorithms working without any constraints of data stream environments. Furthermore, SWEM has

CHAPTER 6. CLUSTERING DATA STREAMS IN SLIDING WINDOW MODELS WITH EM TECHNIQUE

a solid mathematical background since it is developed based on the Expectation Maximization technique. This theoretically sound tool has been shown stable and effective in many domains even though the mixture models it employs are assumed to follow multivariate Gaussian distributions.

Chapter 7

Conclusions and Future Work

7.1 Summary

Data mining is a study of automatic methods for efficiently extracting interesting knowledge, including rules and patterns, from a large collection of data stored in databases. For the last decades, data mining is extensively studied in theory and practice, and applied to various aspects of real-life including those in business, industry, and natural science. In recent years, with the emergence of many applications, such as web click streams, financial stock monitoring, telecommunication operations, various data mining capabilities have been extended to a new kind of streaming data. For stream applications, data are usually generated in a continuous fashion and exhibit properties of rapid, possibly unbound and fast changing behaviors. Furthermore, they arrive to a mining system in a push-based manner meanwhile system resources used in the mining process are usually confined in advance. Consequently, there is a strong demand for developing novel mining techniques to discover useful patterns from data streams whilst working within system resource constraints. Furthermore, it is also very important to approximate and guarantee the mining results within some error threshold. Motivated by these challenges, in this thesis, we have investigated and proposed various approximation algorithms to effectively and efficiently mine interesting patterns from data streams under different system resource constraints. Two fundamental data mining tasks have been explored

CHAPTER 7. CONCLUSIONS AND FUTURE WORK

in the streaming data context: frequent pattern discovering and cluster analysis. We summarize our contributions as follows:

- We develop the EStream algorithm in order to address the problem of online mining frequent patterns from data streams with precise error guarantee. This research is motivated by the fact that, while the limitation of memory space in mining stream mining does not allow one to store and count exactly the occurrence of each pattern (or itemset) appearing in the stream and approximate mining results are usually sufficient in many online applications, it is still a strong demand to guarantee the mining results to be computed within some given error threshold. Such an important task is not adequately addressed in the literature where most algorithms are not able to guarantee the mining results within any precise threshold if the stream is processed online. Our EStream algorithm is proposed to address this important challenge. It is a false positive algorithm in that all frequent itemsets appearing in the stream are always reported in the mining results and it guarantees that: (i) for 1-itemsets, there is no error on its frequency; (ii) for 2-itemsets, the error in the estimate is no more than ϵ ; and (iii) for itemsets of length $k > 2$, the error is no more than 2ϵ , where ϵ is the preset error threshold given by a user and k is the maximal length of the frequent set in the stream. We then further show that the EStream algorithm can be directly extended to mine frequent patterns in a forgetful window model with the proposal of FStream algorithm.
- For many data stream applications, data usually arrive to a mining system in a push-based manner meanwhile their arrival rate can be high and unpredictable. During its computation, the rate of a data stream can significantly increase and exceeds the system capacity. Such overloading situations are usually unseen and if no corrective action is taken, queues will form and the delivery of mining results is

CHAPTER 7. CONCLUSIONS AND FUTURE WORK

no longer guaranteed. Motivated by this observation, we continue developing the frequent set mining problem to the context in which the CPU capacity is strictly another constraint in stream environments. A novel problem of load shedding in mining frequent patterns is formally formulated and presented. The LSF algorithm is then proposed in order to address the problem in a landmark mining model. The Chernoff bound is employed in our algorithm design and such a theoretically sound tool allows us to approximate closely a set of frequent patterns from data streams under various overloading situations.

- We further study the problem of mining frequent patterns from high speed data streams with CPU constraints in the time-based sliding window model. Such a mining model has been known to be one of the most challenging model since it requires the elimination of outdated data transactions. We present the LSSW algorithm to address the problem from overload detection to overload resolution. LSSW processes data streams online with arbitrarily large size of a sliding window. Most importantly, its quality of mining results is probabilistically guaranteed within a preset error threshold. By integrating both the Chernoff bound and the deterministic technique in its design, our LSSW algorithm assures that the support error on each frequent pattern computed is no more than $\epsilon + \gamma$ with high probability of $(1 - \delta)$ where ϵ is a deterministic error given by the data stream analyst, γ and δ are respectively the error bound and the probability threshold guaranteed by the Chernoff bound. It is strongly believed that when the mining results are guaranteed within some given error threshold governing by a user, the quality of making decisions can improve significantly in streaming environments.
- We present the SWEM algorithm, which is based on the Expectation Maximization technique, in order to address the problem of clustering data streams in a sliding

window model. We assume that a data stream is generated as a result of a dynamic statistical process that consists of k mixture Gaussian distributions and the SWEM is designed to find a set of parameters that optimally approximate these mixture models. Each portion of the data stream arriving in the time-based sliding window is summarized into a set of compact micro components and these sufficient statistics are used to derive the global clustering models. In order to address the problem of distribution changes which usually occurs in evolving data streams, we develop two techniques, the spitting and merging operations, in order to re-distribute the set of micro components in the data space and such operations can approximate effectively new characteristics in streaming data environments.

7.2 Future Work

In this section, we highlight some of potential directions that the proposed techniques in this thesis can be further broaden and explored in the near future.

In Chapter 2, EStream is proposed to rely on the downward closure property for finding frequent patterns. In term of memory space constraints, such an approach still suffers from the exponential growth of the increasing size of itemsets; this means that in order to identify the frequency of a long itemset, the algorithm also has to compute the frequency of all its subsets. Consequently, the computation efficiency could be affected when data streams are dense and the length of maximal frequent patterns is large. For instance, in order to update the frequency of a frequent itemset that has a length of 20, we must update the frequency of $2^{20} - 1 \approx 10^6$ its subsets that are also frequent. In mining data streams, this situation is not desirable and in some certain cases not feasible due to the time constraint. Furthermore, not all patterns satisfying the definition of being frequent (i.e., higher than a support threshold) are interesting to a data stream analyst. Hence, for the future work, we would like to broaden our research into two areas: mining compact

CHAPTER 7. CONCLUSIONS AND FUTURE WORK

sets of frequent patterns, including closed frequent itemsets and maximal frequent itemsets; and mining patterns with various constraints [WHH00]. For the problem of mining maximal frequent sets, many algorithms such as Pincer-search [LK02], MAFFIA [BCG01], Depth First Travelling [RJB98, AAP00, GZ01] and Tree Projection [HPY00] have been proposed in mining conventional datasets. Nevertheless, adapting them to data stream environments is very challenging but also interesting. Certainly, the error on mining results of these special patterns is still the emphasis of our broadening directions.

In temporal pattern mining problems, there are four main kinds of interesting patterns: sequential patterns [AS95], episode patterns [MTV97], inter-transaction association rules [LHF98], and causality rules [CHLC02]. The main difference among these temporal patterns lies in the ordering of event (i.e., item) occurrences in patterns. In the literature, the authors in [TCY03] have focused on the inter-transaction association rules in the data stream context. However, such a kind of patterns are still simple as there is no constraint on the order of events. Therefore, our counting techniques with error guarantee can be extended to this direction in that other interesting patterns, such as sequential patterns and episode patterns, will be the target to be computed. Mining such patterns are much more challenging since they are defined based on the strict order of events. This study can also be considered in all three mining models: the landmark window, the forgetful window and the sliding window models.

Load Shedding in data stream querying has been extensively studied in the literature [TLPY06, GWYL05, TcZ07, TcZ⁺03]. Various techniques relying on sampling [BDM04] and quality of service maximization [TcZ⁺03, TZ06] have been addressed and proposed. Since the frequent set mining problem is quite similar to an iceberg querying (which performs an aggregate function over a set of attributes of a relation and eliminates those aggregate values that are below some user-specified threshold) [FSGM⁺98], our research in this thesis has addressed this problem from the sampling approach (in the context of

CHAPTER 7. CONCLUSIONS AND FUTURE WORK

CPU constraints). However, it is interesting and challenging to extend it into the second approach of load shedding with maximizing quality of service. The important questions that need to be answered in this direction are how to define the importance (or quality) of patterns; and how to approximate them in order to maximize the overall quality of mining results under that definition. Again, the error in the results is also desirable to be proven and guaranteed in this extension.

Cluster analysis is an important task for data stream understanding and our SWEM algorithm studied in this thesis is one of the first efforts addressing this problem in the context of time-based sliding window model. Although SWEM is truly an incremental method and it is designed to address various resource constraints in streaming environments, it still needs to traverse each part of the stream multiple times due to relying on the Expectation Maximization technique. It is therefore necessary to study this clustering problem (in the window sliding model) with other mining techniques in order to improve the processing time. One of the most potential directions is based on grid models. When a data space is divided into a set of cells, locating each cell given the value of a data point generally takes only a constant time and thus, it is possible to online update data into the grid cells. In Chapter 5, we have proposed the LSSW algorithm that are able to count and approximate itemsets' frequency within a given error threshold (see Theorem 5.3 Section 5.3.3, under no load shedding situation). It is interesting to observe that if we consider each cell in the grid-based model as an itemset in the frequent set mining problem, then we can directly extend the counting technique proposed in Chapter 5 to the grid-based clustering approach. In this case, the density of each cell (defined as the number of data points falling into it) can be closely estimated and guaranteed. Consequently, when such a density-based approach and the frequency counting technique are integrated together, the clustering results can be well approximated and their error is theoretically guaranteed.

7.3 List of Publications

Book chapters and International Journal papers

- **X. H. Dang**, W. K. Ng, K.-L. Ong. “Online Mining of Frequent Sets in Data Streams with Error Guarantee”, International Journal of Knowledge and Information Systems, Online First, October, 2007.
- **X. H. Dang**, W. K. Ng, K.-L. Ong, V.C.S Lee, “Error Guarantee with Load Shedding in Mining Frequent Patterns from Data Streams”, under revision.
- **X. H. Dang**, W. K. Ng, K.-L. Ong, V.C.S Lee “Frequent Sets Mining in Data Stream Environments”, Encyclopedia of Data Warehousing and Mining (2nd Ed.), IGI Publishing, 2008.
- Wan Li, W. K. Ng, **X. H. Dang**, Zhang Kuan, Philip S. Yu, “Density-Based Clustering of Data Streams at Multiple Resolutions”, submitted to IEEE Transactions on Knowledge and Data Engineering.

International Conference papers

- **X. H. Dang**, W. K. Ng, K. L. Ong, “Adaptive Load Shedding for Mining Frequent Patterns from Data Streams,” 8th International Conference on Data Warehousing and Knowledge Discovery (DaWaK’06), 2006.
- **X. H. Dang**, W. K. Ng, K. L. Ong, “EStream: Online Mining of Frequent Sets with Precise Error Guarantee,” 8th International Conference on Data Warehousing and Knowledge Discovery (DaWak’06), 2006.
- **X. H. Dang**, W. K. Ng, K. L. Ong, V.C.S Lee, “Discovering Frequent Sets from Data Streams with CPU Constraint”. Australasian Data Mining Conference (AusDM’07), 2007.

CHAPTER 7. CONCLUSIONS AND FUTURE WORK

- **X. H. Dang**, W. K. Ng, K. L. Ong, V.C.S Lee, “An EM-based algorithm for Clustering Data Streams in Sliding Windows”, under review.

References

- [AAP00] Ramesh C. Agarwal, Charu C. Aggarwal, and V. V. V. Prasad. Depth first generation of long patterns. In *KDD '00: Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 108–118, New York, NY, USA, 2000. ACM Press.
- [ACc⁺03] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2):120–139, 2003.
- [AHWY03] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. A framework for clustering evolving data streams. In *VLDB Conference*, pages 81–92, 2003.
- [AHWY04a] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. A framework for projected clustering of high dimensional data streams. In *VLDB*, pages 852–863, 2004.
- [AHWY04b] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. On demand classification of data streams. In *ACM SIGKDD Conference*, pages 503–508, New York, NY, USA, 2004. ACM Press.
- [AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB Conference*, pages 487–499. Morgan Kaufmann, 1994.

REFERENCES

- [AS95] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In Philip S. Yu and Arbee S. P. Chen, editors, *Eleventh International Conference on Data Engineering*, pages 3–14, Taipei, Taiwan, 1995. IEEE Computer Society Press.
- [AY07] Charu C. Aggarwal and Philip S. Yu. On string classification in data streams. In *SIGKDD*, pages 36–45, 2007.
- [BBD⁺02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.
- [BCG01] Douglas Burdick, Manuel Calimlim, and Johannes Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. In *Proceedings of the 17th International Conference on Data Engineering*, pages 443–452, Washington, DC, USA, 2001. IEEE Computer Society.
- [BDF⁺97] Daniel Barbará, William DuMouchel, Christos Faloutsos, Peter J. Haas, Joseph M. Hellerstein, Yannis E. Ioannidis, H. V. Jagadish, Theodore Johnson, Raymond T. Ng, Viswanath Poosala, Kenneth A. Ross, and Kenneth C. Sevcik. The new jersey data reduction report. *IEEE Data Eng. Bull.*, 20(4):3–45, 1997.
- [BDM02] Brian Babcock, Mayur Datar, and Rajeev Motwani. Sampling from a moving window over streaming data. In *SODA*, pages 633–634, 2002.
- [BDM04] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load shedding for aggregation queries over data streams. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, Washington, DC, USA, 2004. IEEE Computer Society.
- [BDMO03] Brian Babcock, Mayur Datar, Rajeev Motwani, and Liadan O’Callaghan. Maintaining variance and k-medians over data stream windows. In *Proceedings of the 22nd Symposium on Principles of Database Systems(PODS 2003)*, San Diego, CA, June 9-12 2003.

REFERENCES

- [Ber02] Pavel Berkhin. Survey of clustering data mining techniques. *Technical Report*, 2002.
- [BMS97] Sergey Brin, Rajeev Motwani, and Craig Silverstein. Beyond market baskets: Generalizing association rules to correlations. In Joan Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 265–276. ACM Press, 1997.
- [BMUT97] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, and Shalom Tsur. Dynamic itemset counting and implication rules for market basket data. In *SIGMOD Conference*, pages 255–264, 1997.
- [CBB⁺03] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Çetintemel, Ying Xing, and Stanley B. Zdonik. Scalable distributed stream processing. In *CIDR*, 2003.
- [CCFC04] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theor. Comput. Sci.*, 312(1):3–15, 2004.
- [CCP⁺04] Y. Dora Cai, David Clutter, Greg Pape, Jiawei Han, Michael Welge, and Loretta Auvil. Maids: mining alarming incidents from data streams. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 919–920, New York, NY, USA, 2004. ACM Press.
- [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.
- [CFP⁺04] Corinna Cortes, Kathleen Fisher, Daryl Pregibon, Anne Rogers, and Frederick Smith. Hancock: A language for analyzing transactional data streams. *ACM Trans. Program. Lang. Syst.*, 26(2):301–338, 2004.

REFERENCES

- [CFPR00] Corinna Cortes, Kathleen Fisher, Daryl Pregibon, and Anne Rogers. Hancock: a language for extracting signatures from data streams. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 9–17, New York, NY, USA, 2000. ACM Press.
- [Cha03] S. Chandrasekaran. Telegraphcq: Continuous dataflow processing for an uncertain world, 2003.
- [Che53] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Ann. Math. Statist.*, 23:493–507, 1953.
- [CHLC02] P. S. Yu C.-H. Lee and M.-S. Chen. Causality rules: Exploring the relationship between triggering and consequential events in a database of short transactions. In *Proceedings of the 2nd SIAM International Conference on Data Mining*, pages 403–419, 2002.
- [CL03a] Joong Hyuk Chang and Won Suk Lee. estwin: adaptively monitoring the recent change of frequent itemsets over online data streams. In *CIKM '03: Proceedings of the twelfth international conference on Information and knowledge management*, pages 536–539, New York, NY, USA, 2003. ACM Press.
- [CL03b] Joong Hyuk Chang and Won Suk Lee. Finding recent frequent itemsets adaptively over online data streams. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 487–492, New York, NY, USA, 2003. ACM Press.
- [CL04] Joong Hyuk Chang and Won Suk Lee. A sliding window method for finding recently frequent itemsets over online data streams. *J. Inf. Sci. Eng.*, 20(4):753–762, 2004.
- [CL06] Joong Hyuk Chang and Won Suk Lee. Finding frequent itemsets over online data streams. *Information & Software Technology*, 48(7):606–618, 2006.

REFERENCES

- [CLC05] Y.H. Wu C.H. Lin, D.Y. Chiu and A.L.P. Chen. Mining frequent itemsets from data streams with a time-sensitive sliding window. In *Proc. SIAM International Conference on Data Mining*, 2005.
- [CM05] Graham Cormode and S. Muthukrishnan. What's hot and what's not: tracking most frequent items dynamically. *ACM Trans. Database Syst.*, 30(1):249–278, 2005.
- [CMM02] Streaming-data algorithms for high-quality clustering. In *ICDE '02: Proceedings of the 18th International Conference on Data Engineering*, page 685, Washington, DC, USA, 2002. IEEE Computer Society.
- [COP03] Moses Charikar, Liadan O'Callaghan, and Rina Panigrahy. Better streaming algorithms for clustering problems. In *STOC '03: Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 30–39, New York, NY, USA, 2003. ACM Press.
- [CSWR05] Yun Chi, Philip S.Yu, Haixun Wang, and Richard R.Muntz. Loadstar: A load shedding scheme for classifying data streams. In *Proc. SIAM International Conference on Data Mining*, 2005.
- [CT07] Yixin Chen and Li Tu. Density-based clustering for real-time stream data. In *SIGKDD*, pages 133–142, 2007.
- [CWZ04] Fang Chu, Yizhou Wang, and Carlo Zaniolo. An adaptive learning approach for noisy data streams. In *ICDM*, pages 351–354, 2004.
- [DGIM02] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. In *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 635–644, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.
- [DGR03] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. Approximate join processing over data streams. In *SIGMOD '03: Proceedings of the 2003*

REFERENCES

- ACM SIGMOD international conference on Management of data*, pages 40–51, New York, NY, USA, 2003. ACM Press.
- [DH73] Richard O. Duda and Peter E. Hart. *Pattern Classification and Scene Analysis*. John Wiley & Sons Press, New York, 1973.
- [DH00] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *KDD '00: Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 71–80, New York, NY, USA, 2000. ACM Press.
- [DH01] Pedro Domingos and Geoff Hulten. Catching up with the data: Research issues in mining data streams. In *DMKD*, 2001.
- [DHS01] H. Mannila D. Hand and P. Smyth. *Principles of Data Mining*. MIT Press, 2001.
- [DKK03] Mukund Deshpande, Michihiro Kuramochi, and George Karypis. Frequent sub-structure-based approaches for classifying chemical compounds. In *ICDM*, pages 35–42, 2003.
- [DLOM02] Erik D. Demaine, Alejandro Lopez-Ortiz, and J. Ian Munro. Frequency estimation of internet packet streams with limited space. In *ESA '02: Proceedings of the 10th Annual European Symposium on Algorithms*, pages 348–360, London, UK, 2002. Springer-Verlag.
- [DLR77] A. Dempster, N. Laird, and D. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Royal statistical Society B*, 39:1–38, 1977.
- [DNOL] X. H. Dang, W. K. Ng, K.-L. Ong, and V.C.S Lee. *Frequent Sets Mining in Data Stream Environments*. Encyclopaedia of Data Warehousing and Mining (2nd Ed.), IGI Publishing.
- [Dun03] Margaret H. Dunham. *Data Mining: Introductory and Advanced Topics*. Prentical Hall, 2003.

REFERENCES

- [EKSX96] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *SIGKDD*, pages 226–231, 1996.
- [Eve93] B. S. Everitt. *Cluster Analysis*. Edward Arnold and Halsted Press, 1993.
- [FaHWY04] Wei Fan, Yi an Huang, Haixun Wang, and Philip S. Yu. Active mining of data streams. In *SDM*, 2004.
- [Fan04] Wei Fan. Streamminer: A classifier ensemble-based engine to mine concept-drifting data streams. In *VLDB*, pages 1257–1260, 2004.
- [FSGM⁺98] Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, and Jeffrey D. Ullman. Computing iceberg queries efficiently. In Ashish Gupta, Oded Shmueli, and Jennifer Widom, editors, *VLDB Conference*, pages 299–310. Morgan Kaufmann, 1998.
- [GGR02] Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Querying and mining data streams: you only get one look a tutorial. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 635–635, New York, NY, USA, 2002. ACM Press.
- [GHP⁺03] C. Giannella, J. Han, J. Pei, X. Yan, and P.S. Yu. *Mining Frequent Patterns in Data Streams at Multiple Time Granularities*. Next Generation Data Mining (H. Kargupta, A. Joshi, K. Sivakumar, and Y. Yesha, eds.). 2003.
- [GKMS03] Anna C. Gilbert, Yannis Kotidis, S. Muthukrishnan, and Martin Strauss. One-pass wavelet decompositions of data streams. *IEEE Trans. Knowl. Data Eng.*, 15(3):541–554, 2003.
- [GKS01] Johannes Gehrke, Flip Korn, and Divesh Srivastava. On computing correlated aggregates over continual data streams. In *SIGMOD Conference*, pages 13–24, 2001.

REFERENCES

- [GM99] Phillip B. Gibbons and Yossi Matias. Synopsis data structures for massive data sets. In *SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 909–910, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.
- [GMM⁺03] S. Guha, A. Meyerson, N. Mishra, R. Motwani, and L. O’Callaghan. Clustering Data Streams: Theory and Practice. *IEEE TKDE*, 15, 2003.
- [GMMO00] S. Guha, N. Mishra, R. Motwani, and L. O’Callaghan. Clustering Data Streams. In *Proc. Symp. on Foundations of Computer Science*, November 2000.
- [GO03] Lukasz Golab and M. Tamer Ozsu. Issues in data stream management. *SIGMOD Rec.*, 32(2):5–14, 2003.
- [GRM03] Joo Gama, Ricardo Rocha, and Pedro Medas. Accurate decision trees for mining high-speed data streams. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 523–528, New York, NY, USA, 2003. ACM Press.
- [GRS98] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. CURE: an efficient clustering algorithm for large databases. pages 73–84, 1998.
- [GWYL05] Bugra Gedik, Kun-Lung Wu, Philip S. Yu, and Ling Liu. Adaptive load shedding for windowed stream joins. In *CIKM*, pages 171–178, 2005.
- [GZ01] Karam Gouda and Mohammed Javeed Zaki. Efficiently mining maximal frequent itemsets. In *ICDM '01: Proceedings of the 2001 IEEE International Conference on Data Mining*, pages 163–170, Washington, DC, USA, 2001. IEEE Computer Society.
- [Hid99] Christian Hidber. Online association rule mining. In *SIGMOD Conference*, pages 145–156, 1999.
- [HK01] Jiawei Han and Micheline Kamber. *Data mining: concepts and techniques*. Morgan Kaufmann, 2001.

REFERENCES

- [HPDW01] Jiawei Han, Jian Pei, Guozhu Dong, and Ke Wang. Efficient computation of iceberg cubes with complex measures. In *SIGMOD Conference*, pages 1–12, 2001.
- [HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. *SIGMOD Rec.*, 29(2):1–12, 2000.
- [HSD01] G. Hulten, L. Spencer, and P. Domingos. Mining Time-Changing Data Streams. In *Proc. ACM SIGKDD*, pages 97–106, San Francisco, CA, 2001.
- [HW79] J. A. Hartigan and M. A. Wong. A K-means clustering algorithm. *Applied Statistics*, 28:100–108, 1979.
- [Ioa03] Yannis E. Ioannidis. The history of histograms (abridged). In *VLDB*, pages 19–30, 2003.
- [IP95] Yannis E. Ioannidis and Viswanath Poosala. Balancing histogram optimality and practicality for query result size estimation. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 233–244, New York, NY, USA, 1995. ACM Press.
- [JA03] R. Jin and G. Agrawal. An algorithm for in-core frequent itemset mining on streaming data, 2003.
- [JG06] Nan Jiang and Le Gruenwald. Cfi-stream: mining closed frequent itemsets in data streams. In *KDD*, pages 592–597, 2006.
- [JMF99] Anil K. Jain, M. Narasimha Murty, and Patrick J. Flynn. Data clustering: A review. *ACM Comput. Surv.*, 31(3):264–323, 1999.
- [Kan03] M. Kantardzic. *Data Mining: Concepts Models, and Algorithms*. IEEE Press and John Wiley, 2003.
- [KHK99] George Karypis, Eui-Hong Han, and Vipin Kumar. Chameleon: Hierarchical clustering using dynamic modeling. *IEEE Computer*, 32(8):68–75, 1999.

REFERENCES

- [Koh90] T Kohonen. The self-organizing map. In *Proceedings of the IEEE*, pages 1464–1479, 1990.
- [KR90] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley, 1990.
- [KSP03] Richard M. Karp, Scott Shenker, and Christos H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, 28(1):51–55, 2003.
- [LHF98] H. Lu, J. Han, and L. Feng. Stock movement prediction and n-dimensional inter-transaction association rules. In *Proc. ACM SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery.*, 1998.
- [LK02] Dao-I Lin and Zvi M. Kedem. Pincer-search: An efficient algorithm for discovering the maximum frequent set. *IEEE Trans. Knowl. Data Eng.*, 14(3):553–566, 2002.
- [LP01] Diane Lambert and Jos C. Pinheiro. Mining a stream of transactions for customer patterns. In *KDD '01: Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 305–310, New York, NY, USA, 2001. ACM Press.
- [MF02] Samuel Madden and Michael J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE*, pages 555–566, 2002.
- [Mit97] Tom M. Mitchell. *Machine Learning*. McGraw Hill, 1997.
- [MM02] G. S. Manku and R. Motwani. Approximate Frequency Counts over Streaming Data. In *Proc. VLDB*, Hong Kong, China, August 2002.
- [MS04] S. Muthukrishnan and Martin J. Strauss. Approximate histogram and wavelet summaries of streaming data. *Technical Report*, 2004.

REFERENCES

- [MSDO05] Amit Manjhi, Vladislav Shkapenyuk, Kedar Dhamdhere, and Christopher Olston. Finding (recently) frequent items in distributed data streams. In *ICDE*, pages 767–778, 2005.
- [MTV94] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Efficient algorithms for discovering association rules. In *SIGKDD Workshop*, pages 181–192, 1994.
- [MTV97] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data Min. Knowl. Discov.*, 1(3):259–289, 1997.
- [Mur98] Sreerama K. Murthy. Automatic construction of decision trees from data: A multi-disciplinary survey. *Data Min. Knowl. Discov.*, 2(4):345–389, 1998.
- [Mut03] S. Muthukrishnan. Data streams: algorithms and applications. In *SODA*, pages 413–413, 2003.
- [MWA⁺03] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Singh Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.
- [OO05] Carlos Ordonez and Edward Omiecinski. Accelerating em clustering to find high-quality solutions. *Knowl. Inf. Syst.*, 7(2):135–157, 2005.
- [POSG04] Byung-Hoon Park, George Ostroouchov, Nagiza F. Samatova, and Al Geist. Reservoir-based random sampling with replacement from data stream. In *SDM*, 2004.
- [RJB98] Jr. Roberto J. Bayardo. Efficiently mining long patterns from databases. *SIGMOD Rec.*, 27(2):85–93, 1998.
- [SON95] Ashok Savasere, Edward Omiecinski, and Shamkant B. Navathe. An efficient algorithm for mining association rules in large databases. In *VLDB*, pages 432–444, 1995.

REFERENCES

- [TCY03] Wei-Guang Teng, Ming-Syan Chen, and Philip S. Yu. A regression-based temporal pattern mining scheme for data streams. In *VLDB*, pages 93–104, 2003.
- [TcZ⁺03] Nesime Tatbul, Ugur Çetintemel, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *VLDB*, pages 309–320, 2003.
- [TcZ07] Nesime Tatbul, Ugur Çetintemel, and Stanley B. Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. In *VLDB*, pages 159–170, 2007.
- [TLPY06] Yi-Cheng Tu, Song Liu, Sunil Prabhakar, and Bin Yao. Load shedding in stream databases: A control-based approach. In *VLDB*, pages 787–798, 2006.
- [TSK05] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.
- [Tsy02] Alexey Tsymbal. The problem of concept drift: Definitions and related work. *Technical Report*, 2002.
- [TZ06] Nesime Tatbul and Stanley B. Zdonik. Window-aware load shedding for aggregation queries over data streams. In *VLDB*, pages 799–810, 2006.
- [Vit85] Jeffrey Scott Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.
- [WA94] Halang Wolfgang and Stoyenko Alexander. *Real time computing*. Springer-Verlag, 1994.
- [Wat00] O. Watanabe. Simple sampling techniques for discovery science. *IEICE Trans.*, pages 19–26, 2000.
- [WFYH03] H. Wang, W. Fan, P. S. Yu, and J. Han. Mining Concept-Drifting Data Streams using Ensemble Classifiers. Washington DC, August 2003.

REFERENCES

- [WHH00] Ke Wang, Yu He, and Jiawei Han. Mining frequent itemsets using support constraints. In *VLDB*, pages 43–52, 2000.
- [WK96] Gerhard Widmer and Miroslav Kubat. Learning in the presence of concept drift and hidden contexts. *Machine Learning*, 23(1):69–101, 1996.
- [WXL99] Ke Wang, Chu Xu, and Bing Liu. Clustering transactions using large items. In *CIKM*, pages 483–490, 1999.
- [Yan03] Jiong Yang. Dynamic clustering of evolving streams with a single pass. In *ICDE*, pages 695–697, 2003.
- [Yan04] Guizhen Yang. The complexity of mining maximal frequent itemsets and maximal frequent patterns. In *SIGKDD Conference*, pages 344–353, 2004.
- [YCLZ04] Jeffrey Xu Yu, Zhihong Chong, Hongjun Lu, and Aoying Zhou. False positive or false negative: Mining frequent itemsets from high speed transactional data streams. In *VLDB*, pages 204–215, 2004.
- [YG03] Y. Yao and J. Gehrke. Query processing in sensor networks. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research. Asilomar, California.*, 2003.
- [Zak00] Mohammed Javeed Zaki. Scalable algorithms for association mining. *IEEE Trans. Knowl. Data Eng.*, 12(3):372–390, 2000.
- [ZCY⁺07] Aoying Zhou, Feng Cao, Ying Yan, Chaofeng Sha, and Xiaofeng He. Distributed data stream clustering: A fast em-based approach. In *ICDE*, pages 736–745, 2007.
- [ZPLO97] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Wei Li, and Mitsunori Ogihara. Evaluation of sampling for data mining of association rules. In *RIDE*, pages 0–, 1997.
- [ZPOL97] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. New algorithms for fast discovery of association rules. In *SIGKDD*, pages 283–286, 1997.

REFERENCES

- [ZRL96] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: An efficient data clustering method for very large databases. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 103–114. ACM Press, 1996.
- [ZS02] Yunyue Zhu and Dennis Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *VLDB Conference*, 2002.
- [ZS03] Yunyue Zhu and Dennis Shasha. Efficient elastic burst detection in data streams. In *SIGKDD*, pages 336–345, 2003.