Discovery of Frequent Patterns in Transactional Data Streams

Willie Ng

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Doctor of Philosophy

2010

For Mom, Dad, and Celeste

Acknowledgements

I would like to express my gratitude to the following people and organizations for their support and contributions, without which this dissertation would not have been possible.

Dr. Manoranjan Dash, my supervisor, who provides much-needed guidance, kind support, and constructive evaluation of my work, and whose streams of suggestions and ideas make this research as much his as mine.

Dr. Sourav Bhowmick and **Dr. Ng Wee Keong**, current and former Directors of Centre for Advanced Information Systems (CAIS), for generally ensuring a conducive research environment in the centre.

Prof. Lim Ee-Peng and **Dr. Sun Aixin** for giving me the opportunity to work in their Semantic Social Network Research Group (SSNet) during my PhD studies.

Dr. Liu Ying for pursuing joint research, co-authoring papers, and giving advice on my work.

Dr. Alper Cabuk for giving advice on my work and helping me to print the thesis.

Professors, Researchers, and Fellow Students at CAIS for various constructive discussions and debates, seminars, exchanges of ideas, and their kind friendship.

Lai Chee Keong, **Chua Chiew Song**, and **Tan Lay Choo**, current and former support staff at CAIS, for the technical and administrative support in the course of my research at the lab.

Koh Soh Hong, Elaine, Secretary to Head of Division (Information Systems) for her generous help with any administrative need I encountered over the course of my research.

School of Computer Engineering of Nanyang Technological University for the resources and financial support provided for the duration of the course.

Also many thanks to my parents and family for their unconditional support and encouragement during the many years of my studies at the University. Finally, I would like to thank my wife Celeste for putting up with my weekends in the office, for listening to my sagas from work, and for giving me the motivation to finish this thesis. Thanks to all of you for making these five years into a positive and stimulating experience.

Contents

List of Figures

List of Tables

Abstract

We investigate the problem of finding frequent patterns in a continuous stream of transactions. In the literature, two prominent approaches are often used: (a) perform approximate counting (e.g., lossy counting algorithm (LCA) of Manku and Motwani, VLDB 2002) by using a lower support threshold than the one given by the user, or (b) maintain a running sample (e.g., reservoir sampling (Algo-Z) of Vitter, TOMS 1985) and generate frequent patterns from the sample on demand. Although both are known to be practically useful, to the best of our knowledge, there has been no comparison carried out between them.

In LCA, the proper quantification of the error parameter, ϵ, is non-trival. Our error analysis shows that when mining for frequent patterns, one does not need to use a fixed ϵ for all size of itemsets. Based on this finding, we propose a *Customized Lossy Counting* algorithm (CLCA). Interestingly, CLCA outperforms LCA in mining for the frequent patterns of user's choice. In addition, for mining of frequent patterns over data streams with a landmark window, we propose a distance based sampling algorithm (*DSS*). An empirical comparison study on the algorithms is performed using synthetic and benchmark data sets. Results show that *DSS* is consistently more accurate than LCA and Algo-Z, whereas LCA performs better than Algo-Z.

In this dissertation, we extended the use of *DSS* to sample transactions for utility mining. Traditional frequent pattern mining identifies frequent patterns from data set and generates patterns by considering each item in equal value. However, items are actually different in many aspects in a number of real applications, such as retail

marketing, network log, etc. The difference between items makes a strong impact on the decision making in these applications. Therefore, frequent pattern mining cannot meet the demands arising from these applications. By considering the different values of individual items as utilities, utility mining focuses on identifying the patterns with high utilities.

In addition, data streams can change their behavior over time and, when significant change occurs, much harm is done to the mining result if it is not properly handled. In the past, there have been many studies mainly on adapting to changes in data streams. We contend that adapting to changes is simply not enough. The ability to detect and characterize change is also essential in many applications, for example intrusion detection, network traffic analysis, data streams from intensive care units etc. In this dissertation, we explore an algorithm for change detection in utility mining.

Chapter 1

Introduction

Data Mining

The ubiquity of computers in business and commerce has led to the daily generation of data. With advanced technologies, it is now practicable to pile up, store, and retrieve huge collections of data. We are overwhelmed by data. In fact, virtually every company or organization stores large amounts of data. It has become a common belief that the stored data will eventually have some value, either for intention that initially motivated its collection or for the intentions not yet envisioned. Many assume that these mountains of data present the potential for us to discover useful information and knowledge that we could not see before. Indeed, it is of vital interest to any company to be able to analyze the data and to find actionable knowledge and information from them. The trick is to extract the hidden information from the surrounding mass of uninteresting data, but how?

Unfortunately, we are limited in our ability to manually process large amounts of data to discover useful information and knowledge. Overcoming this limitation demands the use of some automatic tools. As data sets continue to grow in size and complexity, there has been a shift away from direct hands-on data analysis towards indirect, automatic data analysis using more complex and sophisticated tools. This challenge is the main motivation for data mining. Data mining, which is also referred to as knowledge discovery in databases (KDD), has been recognized as the *non-trivial*

process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data [FPSS96].

The field of data mining grew out of the limitations of traditional data analysis techniques in handling the challenges presented by huge data sets. In fact, data mining is a multidisciplinary field, drawing from areas that have long histories such as machine learning, pattern recognition, statistics, database technology, etc. In general, the type of available data and the nature of a data mining problem typically determine which data mining methodology is appropriate. Some of the popular data mining tasks include association rule mining, classification and prediction, cluster analysis, outlier detection and sequential pattern mining [HMS01, HK06a, TSK06]. In particular, association rule mining [AIS93] is more recent while the rest had their origins in other fields. These tasks assist us in finding regularities and interesting patterns from the data as well as to summarize the data in novel ways that are understandable and beneficial to the data owners.

Data Stream

Moving one step further, in recent years, the database and data mining communities have developed strong interest on a new model of data processing where data arrives in the form of continuous streams. This new model is often referred to as streaming data or data stream. Such strong interest can be attributed to the fact that in many real world applications, data is not static but arrives in streams [GGR02, BBD+02]. Here, a data stream is a massive unbounded sequence of data elements continuously generated at rapid rate. Examples of such streaming data include telecommunications, sensor networks, Web record and click-streams in Web applications, stock market, astronomical, and news organizations data.

In telecommunications, call records are generated as a stream, and it is required that most processing be done by examining a call record once, after which records

are archived and seldom examined again. Cortes et al. reported the work done with AT&T long distance call records [CFPR00] which consist of 300 million records per day for 100 million customers. In a single day, Wal-Mart records 20 million sales transactions [DH01]. In networking, it is required to make important management decisions (e.g., dynamic bandwidth allocation, troubleshooting, etc.) online. Similarly, traffic data management requires online analysis and querying in order to provide novel services (e.g., provide driving directions given traffic conditions). In addition, the development of sensor technology has resulted in the possibility of monitoring many events in real time. Likewise, scientific data collection (e.g., by astronomical observatories or earth sensing satellites) routinely produces gigabytes of data per day. In many such applications, the data stream is normally accumulated and archived in data warehouses.

However, access to these archived data is often prohibitively expensive. Here, it should be noted that while data mining has become a fairly well established field, the data stream problem poses a number of unique challenges which are not easily solved by conventional data mining methods. One reason can be attributed to the fact that traditional OLAP and data mining methods typically require multiple scans of the data and are thus infeasible for stream data applications. For effective processing of stream data, new data structures, techniques, and algorithms need to be considered.

Data stream mining is currently an emerging research area where interest is strongly motivated by the huge amount of raw computerized data that many organizations now have. Faced with fierce business competition, every single piece of new information is considered valuable to an organization. With such large volumes of routine data being collected, many organizations are increasingly turning to extraction of useful information from data streams. In fact, the topic of data streams is a very recent one. The first research papers on this topic appeared slightly under a decade ago, and since then this field has grown rapidly. The work is also of great

interest to practitioners in the field who have to mine actionable insights within large volumes of continuously growing data. The goal is to discover significant patterns or interesting rules from data streams for real-time database analysis, decision support, prediction and so on. Data mining can be employed to exploit data streams because these streams contain a large volume of simple records, any of which by itself is rather uninformative. However, the aggregate can depict a picture of evolving patterns, in effect, uncovering the "signature" of certain entities. Such high-level inference processes may also provide information on customer buying patterns, shelving criterion in supermarkets, stock trends, etc. Further, in large corporations, the ability to make fast decisions and infer interesting pattern on-line is crucial for many mission-critical tasks that can have significant monetary value (e.g., telecom fraud detection). Therefore, massive transactional streams present a number of opportunities for data mining techniques. The development of highly efficient algorithms for data streams has a top priority.

1.1 Research Challenges in Mining Data Stream

Unfortunately, mining data stream is non-trivial. Unlike persistent data sets, streaming data arrives constantly at varying speeds and is potentially infinite in size. The sheer volume of a stream over its lifetime could be massive and it is impossible to store and process the entire stream in the main memory. Furthermore, current technology has limited ability to backtrack over previously-arrived data elements (only one sequential pass over the data is permitted), and there is a lack of system control over the order in which the data arrive. Even simply preserving these data for future use can be a problem when they need to be sent to secondary storage. Multiple scans on the data can be very expensive. In addition, they are easily lost or corrupted, or become unusable when the relevant contextual information is no longer available. The problem is exacerbated when the underlying process that generates the data streams

evolves over time. Past data may become outdated and be of little use when compared to the most recent one. All these unique characteristics make the handling of data streams extremely challenging.

In order to process a data stream, a mining algorithm can only access the input via linear scans without random access and only a single or few such scans over the data is permitted. Moreover, as the volume of data far exceeds the space available in the main memory, it is not possible for the algorithm to save too much of the data scanned in the past. Thus, once a new data element arrives, it must be processed quickly. In general, the period for a data element staying in the main memory should be short. Once a data element is removed from the main memory, it is not available to be accessed again. In other words, we can only have one look at the data. In contrast to the persistent data sets that are typically processed offline by storing it in secondary storage, streaming data requires fast processing. For some applications, a month worth of data can be easily piled up to billions of records. Thus conventional techniques for mining complex model will not be able to cope with even a small fraction of this data in useful time. The space and time constraints necessitate the design of novel algorithms that capture only a synopsis of the past data, leaving sufficient memory for future incoming data. One needs to ensure that data does not accumulate much faster than it can be mined.

As technology advances and ubiquitous computing becomes a reality, we can expect that such data volumes will eventually become the rule rather than the exception in future. In this respect, overcoming this state of affairs demands a shift in our frame of mind from mining static databases to mining data streams.

1.2 Subject of the Thesis

In this dissertation, we investigate the problem of mining for frequent patterns (or itemsets) in data streams. Frequent pattern mining (FPM) on data streams is an active research topic in data mining. It has been well recognized to be fundamental to several prominent data mining tasks such as associations [AIS93], sequences [AS95], correlations [BMS97] and even classificatons [LHM98]. Here, the goal of FPM is to uncover a set of itemsets (or any objects), whose occurrence count is at least greater than a pre-defined support threshold based on a fraction of the stream processed so far.

In reality, with limited space and the need for real-time analysis, it is practically impossible to enumerate and count all the itemsets for each of the incoming transactions in the stream. To get a taste of the challenges posed by the data stream model, let us consider two typical queries on a stream of values: average and median. Without much effort, the average can be easily generated online over the data stream by keeping track of only two numbers: the sum of all the values observed in the stream and the number of values in the stream. On the other hand, computing the exact median of n values in a stream is not possible without using $\Omega(n)$ storage [AMS96]. Answering many other useful queries such as counting distinct items, frequent itemset, top-k, etc. also demand large amounts of memory in the data stream setting. For FPM, with a domain of x unique items, we can generate up to 2^x distinct collections (or itemsets) in the data stream!

1.2.1 Approximate Counting

To ensure the completeness of frequent itemsets for data streams, it is necessary to capture not only the information related to frequent itemsets, but also the information related to infrequent ones. If the information about the currently infrequent itemsets were not stored, such information would be lost. If these itemsets become frequent

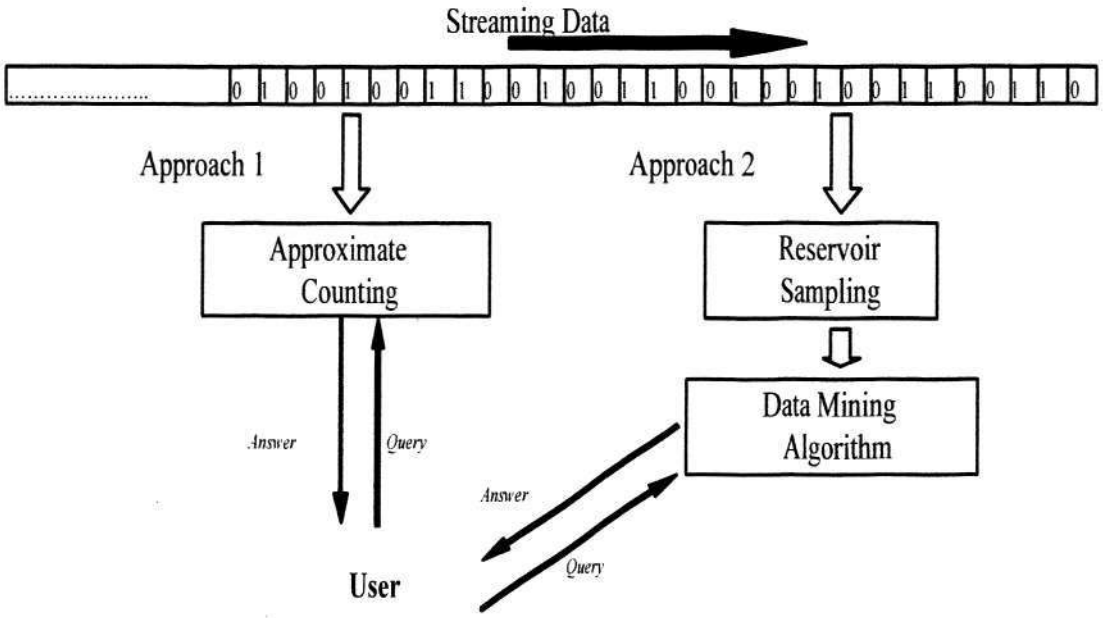


Figure 1.1: An overview of the two approaches.

later on, it would be impossible to figure out their correct support and their relationship with other itemsets. The data stream mining algorithms have to sacrifice the exactness of the analytical results by allowing some tolerable support errors since it is unrealistic to store all the streaming data into the limited main memory.

When mining data stream, one typically cannot obtain the exact frequencies of all itemsets, but has to make an estimation. In general, approximate solutions in most cases may already be satisfactory to the need of users. Indeed, when faced with an infinite data set to analyze, many existing works explicitly trade off accuracy for efficiency where the quality of the final approximate counts are governed by an error parameter, ϵ [MM02]. Thus, the work related to the online mining for frequent patterns then boils down to the problem of finding the right form of data structure and related construction algorithms so that the required frequency counts can be obtained with a bounded error for unbounded input data and limited memory. However, before such online algorithm can be successfully implemented, we note that defining a proper value of ϵ is non-trivial. Usually, we end up facing a dilemma. That is, by setting a small error bound, we achieve good accuracy but suffer in terms of efficiency. On

the other hand, a bigger error bound improves the efficiency but seriously degrades the mining accuracy. In this dissertation , we will empirically show that even setting ϵ to one-tenth (so called "rule of thumb") of the support threshold as suggested by [MM02], it does not necessarily yield an appealing result. To address this drawback in approximate counting, we aim to design a much flexible error bound.

1.2.2 Sampling

Yet another feasible approach to the problem of online FPM is to exploit the fact that approximate answers often suffice, and execute conventional mining algorithms over a synopsis (sampling, wavelet, histogram etc.), that is, over a lossy compressed representation of the data. With this strategy, it is possible to output useful approximate answers in a fraction of the time required to compute the exact solution and thus speeding up the mining process by orders of magnitude [GM99]. Using a sample of the data set as the synopsis is an attractive technique which can scale very well as the data swells in size. The problem of how to maintain a sample of specified size for dynamic data arriving online has been discussed [JMR05]. In practice, one may choose either approximate counting or sampling approach based on a qualitative assessment of the trades-offs (see Figure 1.1). In this dissertation, we put the selection on a firmer basis: We make a comparative study between approximate counting (direct mining) and sampling (indirect mining) for frequent pattern mining on data streams. To the best of our knowledge, this comparison, although very important, has never been reported in literature. We strongly believe that the comparison study will benefit researchers of this topic.

In particular, we advocate and study the use of reservoir sampling for mining transactional data streams. One such sampling technique that we are investigating is the Algo-Z by J.S. Vitter [Vit85]. Reservoir sampling algorithms can be used to dynamically maintain a fixed-size sample of n records from a stream, so that at any given instant, the n records in the sample constitute a true random sample of all of

the records that have been produced by the stream. However, we note that using simple random sampling in Algo-Z alone is not an ideal method because the quality of the sample may not be good due to random fluctuation or noisy data set. We claim that judicious modifications to simple random sampling can make a viable means for attaining both acceptably accurate results and high performance.

1.2.3 Utility Mining and Change Detection

Going one step further, we extend our research to utility mining. We observe that traditional FPM algorithms only consider if an item is absent or present in a transaction. The quantity of the items present in a transaction has been tacitly ignored. Moreover, additional information such as the profit or cost of the item are not considered as well. Usually, a large number of highly frequent patterns are generated. They do not necessarily provide adequate answers for what the high utilities patterns are. For example, a customer who bought a brand new car also bought an air refresher. Obviously we do not buy car very frequent but we cannot deny the fact that there is a relationship between the new car and the air refresher. Here, we can use the cost price to compute the utility value for this pattern. Utility mining permits the users to quantify their preferences concerning the usefulness of pattern using utility values.

In addition, data streams can change their behavior over time and, when significant change occurs, much harm is done to the mining result if it is not properly handled. In the past, there have been many studies on adapting to changes in data streams. We contend that simply adapting to changes is not enough. The ability to detect and characterize change is also essential in many applications, such as intrusion detection, network traffic analysis and data streams from intensive care units. In this dissertation, we explore an online algorithm for change detection in utility mining. In order to provide a mechanism for making quantitative description of the detected change, we adopt the statistical test. We believe there is opportunity for an immensely rewarding synergy between data mining and statistics.

1.3 Contributions of the Dissertation

Concretely, the major contributions of this dissertation are summarized as follows.

- We make modifications to the *Lossy Counting* algorithm (LCA) and derive some analytical bounds on the error [MM02]. We then propose a *Customized Lossy Counting* algorithm (CLCA). Our error analysis shows that when mining for frequent patterns, one does not need to use a fixed ϵ for all size of itemsets. Instead, CLCA allows us to set different values of error bounds, depending on the cardinality of the itemset. The experiments demonstrate that this is useful, resulting in improved precision for the targeted itemsets.

- We propose a distance based sampling algorithm (*DSS*) that maintains the sample on-the-fly in a streaming fashion. If the sampling ratio is small, we show that it is advisable to adopt deterministic sampling than random sampling although deterministic sampling is more time consuming. An empirical study using both real and synthetic databases supports our claims of efficiency and accuracy. Random sampling produces sample of poor quality when the sampling ratio is small. Results show that *DSS* is significantly and consistently more accurate than the popular Algo-Z algorithm by Vitter [Vit85]

- We compare the strengths and weaknesses of the approximate counting with sampling. Surprisingly, even though both approaches yield approximate answers, a comparison has never been done before. For the comparison, we implemented three algorithms LCA, Algo-Z and *DSS*. Moreover, we address the merits and the limitations and present an overall analysis of these algorithms.

- An online algorithm *ACD* for change detection in utility mining (UM) is proposed. Experiments have been conducted to evaluate the performance of ACD using different statistical tests and our study shows that Chi-square test is the most suitable for UM.

1.4 Organization of the Dissertation

This dissertation is organized as follows.

- Chapter 2 gives a formal definition of the problems we study in this dissertation and discusses on the literature related to mining data streams. Some of the sampling techniques are also described.

- Chapter 3 highlights the issues pertaining to the definition of the error bound ϵ and presents a modified version of LCA.

- Chapter 4 covers reservoir sampling. In the same chapter, we shall elaborate on our approach, namely, DSS. This chapter also provides a empirical comparison study for the three algorithms LCA, Algo-Z and DSS.

- Chapter 5 covers online utility mining where the aim is to identify the itemsets with high utilities in the stream. This chapter also discusses how to engineer a change detector, ACD, to trigger the alarm when changes in the stream occur.

- Finally, in Chapter 6, we conclude this dissertation by summarizing the contributions of this research and highlighting promising directions for future work.

Chapter 2

Frequent Pattern Mining Overview

In this chapter, we shall discuss prior works in topics related to our research in frequent pattern mining. Frequent patterns are patterns that occur in a data set frequently. For example, a set of items, such as bread and butter, that occur frequently together in a transactional data set is a frequent pattern. We start off by giving a formal definition of the frequent pattern mining problem. With the introduction of the well known Apriori property, we show how this property leads to many other new algorithms for mining conventional data sets in Section 2.2. Next, owing to the advanced database system technologies and rapid progress of data collection, and the WWW, the growth of data in complex forms (e.g, data streams) has been explosive. We discuss some of the popular data stream models and present a number of the state-of-the-art algorithms on mining frequent patterns over data streams. Lastly, this chapter also reviews recent literature on techniques for obtaining samples from transactional databases.

2.1 Frequent Pattern Mining

Frequent pattern mining (FPM) is a core data mining operation that has been extensively studied over the last two decades. It plays a significant role in many data mining tasks that attempt to discover interesting patterns from databases, such as association rules, classifiers, clusters, correlations, sequences, episodes and many more.

In particular, the mining of association rules is one of the most popular problems. The original motivation for mining frequent patterns or association rules came from the need to analyze the market basket data set in order to provide the retailers with information to understand the purchasing behavior of a customer. Here, frequent patterns describe how often items are purchased together. For example, an association rule " bread⇒ butter (70%)" implies that seven out of ten customers who purchase bread also purchase butter [AIS93, AS94]. Such information will enable the retailer to understand the customer's needs and rearrange the store's layout accordingly, develop cross-promotional programs, or even capture new customers

Since the introduction of the Apriori algorithm in 1994 by Argawal et al., FPM has received a great deal of attention [AS94, SON95, ZPOL97, HPY00, PHM00, Zak00, BCG01, GZ01]. Within a few years, numerous research papers have been published introducing novel algorithms or improvements on existing algorithms to address the problem of mining frequent pattern more efficiently. For a start, this section provides the common notations that can be found in FPM. For ease of exposition, they will be used throughout the dissertation.

Definition 2.1 [**Frequent Pattern**[1]] *Let* $\mathcal{I} = \{i_1, i_2, \ldots, i_n\}$ *be a set of distinct literals, called items.*

- *A transaction has a unique identifier (tid) and contains a set of items. A transactional database* D *over* $\mathcal{I}$ *is a set of transactions over* $\mathcal{I}$*. We omit* $\mathcal{I}$ *whenever it is clear from the context.*

- *An itemset* X *is a set of items such that* $X \in (2^{|\mathcal{I}|} - \{\emptyset\})$ *where* $2^{|\mathcal{I}|}$ *is the power set of* $\mathcal{I}$*.*

[1]In some literature, the terms *large* or *covering* have been used for *frequent*, and the term *itemset* for *pattern*. We will use the terms *frequent pattern* and *frequent itemset* interchangeably.

- *Let j define the length of an itemset. Thus, an itemset with j items is called a j-itemset. Alternatively, we may use the notation $|X|$ to denote the number of items in X. We write "ABC" for the itemset $\{A, B, C\}$ when no ambiguity arises. We assume that items in each transaction are kept sorted in their lexicographic order.*

- *The frequency of an itemset X, denoted by $freq(X)$, is the number of transactions in D that contain X.*

- *The support of an itemset X, denoted by $\sigma(X)$, is the ratio of the frequency of X to the number of transactions processed so far, i.e., $\sigma(X) = freq(X)/|D|$.*

- *Given a pre-defined support threshold σ_{min}, an itemset X is considered a* ***frequent pattern*** *(FP) if its frequency, $freq(X)$, is more than or equal to $\sigma_{min} \times |D|$. Let AFP denote the set of all frequent patterns.*

□

Example 2.1 [**Frequent Pattern**] In the domain of market basket analysis, items refer to the products in the stock (e.g., milk, butter, bread, jam, eggs). Consider the following transactional database D over the set of items $\{A, B, C, D, E, F\}$ (see Table 2.1). A row in the database then corresponds to the contents of a shopping basket (or transactions). If σ_{min} is equal to 0.5, an important query is to find all frequent patterns.

The answer to this frequent pattern problem is $AFP = \{A,\ B,\ D,\ E,\ AB,\ DE\}$.

□

Table 2.1: An example of a Database D over the set $\mathcal{I} = \{A, B, C, D, E, F\}$.

Transaction ID	*Items*
T_1	A, D, E
T_2	A, B
T_3	A, F
T_4	A, B, D, E
T_5	A, B, C, D, E
T_6	C, D, E
T_7	B, F
T_8	A, B, F

Although the above example is seemingly straightforward, the problem itself is far from trivial. A naive brute-force solution would generate all possible combination of items, count their frequency by scanning the database, and output those meeting minimum support criterion. However,the product range of a supermarket, for example, may consist of several thousand different products, which in turn give rise billions or even trillions of possible patterns. With a domain of n unique items, it is not difficult to see that such approach exhibits complexity exponential in n, and is very impractical. We can generate up to 2^n distinct collections (or itemsets) in D! The power set of the set of items cannot be maintained in today's memory. Furthermore, it was proven by Gunopulos et al. that given a database D and a support threshold σ_{min}, the decision problem asking whether AFP contains an itemset I with $|I| \geq k$ for a given k is **NP**-complete [GMS97].

2.2 Mining Large Data Sets

One of the most prominent observations in FPM is the Apriori (or downward closure) property [AS94]:

> Let $X \subseteq Y$ be two itemsets. In every transactional database D, the frequency of Y will be at most as high as the frequency of X.

This belongs to a special category of properties called anti-monotone in the sense that if a set cannot pass a test, all of its supersets will fail the same test as well. For this, it implies that (a) an itemset can only satisfy a downward closure constraint if all its subsets satisfy the constraint and that (b) if an itemset is found to satisfy a downward closure constraint all its subsets need no inspection because they must also satisfy the constraint. These important facts are used by mining algorithms to narrow down the search space which is often referred to as pruning or finding a border in the lattice representation of the search space. For example, this property is exploited as much as possible by the Apriori algorithm [AS94]. The essential idea is to iteratively generate the set of candidate patterns of length $(i+1)$ from the set of frequent patterns of length i (for $i \geq 1$), and check their corresponding occurrence frequencies in the database. A pass over the data is made to identify which candidate patterns are actually frequent. Those patterns that are found to be frequent will be expanded by combining with a new item to form new candidates. This process is repeated till there are no more frequent sets. Therefore, such level-wise algorithm would required $i + 1$ passes if there are candidates of size $i + 1$.

In general, there are two possible layouts of the database for FPM. One is the horizontal layout which consists of a list of transactions, where each transaction has an identifier followed by a list of items (see Table 2.1). Approaches based on the horizontal format include the popular Apriori algorithm and its variants. In this dissertation, we will use this format for our algorithms. The other is the vertical layout which consists of a list of items where each item contains a list of transactions that transacted on that particular item. The vertical format has the advantage that the support for candidate i-itemset can be computed by simple tid-list intersections. Eclat by Zaki uses this data format to find all frequent patterns [ZPOL97]. The tid-lists cluster relevant transactions, and avoid scanning the whole databases to compute support, and the larger the itemset, the shorter the tid-lists, resulting in faster intersections.

The Partition algorithm is one that seek to reduce the database activity: it discovers all frequent patterns in just two passes over the database [SON95]. Contrast this with the previous algorithms, where the database is not only scanned multiple times but the number of scans cannot even be determined in advance. Like the Apriori algorithm, the Partition algorithm works in the level-wise manner, but the notion is to partition the database into sections small enough to be handled in the main memory. That is, a partition is read once from the disk, and level-wise generation and evaluation of candidates for that part are performed in the main memory without further database activity. The first database scan consists of identifying in each part the collection of all locally frequent patterns. For the second scan, the union of the collections of locally frequent patterns is used as the candidate set. The first scan is guaranteed to locate a superset of the collection of frequent patterns; the second scan is needed to merely compute the frequencies of the patterns.

Yet, there is an invention by Han et al. that breaks the norm [HPY00]. They proposed FP-growth in 2000 which was the first effort in mining frequent patterns without candidate generation. The algorithm does not subscribe to the iterative generate and test paradigm of Apriori. The key idea behind FP-growth is to use a compact data structure called FP-tree to obtain a representation of the original transactions so that they can fit into the main memory. Once an FP-tree is generated from the input transaction database, the algorithm mines frequent patterns from the FP-tree. The algorithm generates itemsets from shorter ones to longer ones adding items one by one to those itemsets already generated. Since all the required information is already kept in the FP-tree, any subsequent operations that are required to find frequent patterns can be performed quickly from it, without having to access the database again. The FP-growth algorithm achieves that by performing just two passes over the transactions. The data structure adopted in FP-growth has now been extended to many other algorithms. For example, the LPMiner algorithm by Seno and Karypis

tries to reduce supports tendency to favor smaller itemsets by proposing a minimum support which decreases as a function of itemset length [SK01]. Since this invalidates the downward closure of support, the authors develop a property called smallest valid extension, which can be exploited for pruning the search space.

One of the major problems with these algorithms is that they may generate an exponentially large number of patterns when the support threshold is low. To address this problem, two classes of techniques/problem-formulations have been developed. The first emphasizes on mining maximal/closed itemset. Many recent studies (Charm, Closet, Mafia, etc.) have demonstrated that this can lead to more compact result sets and better efficiency in finding frequent long patterns from large data sets [PHM00, Zak00, BCG01, GZ01]. The second class attempts to lower the number of potentially uninteresting patterns by incorporating various anti-monotone, monotone, or convertible constraints within the constant-support-based frequent pattern mining framework [JAG99, LHM99a, LHM99b, PH00, SK02].

On static data sets, many algorithms for frequent pattern mining have been proposed [Goe03]. This research has led to further efforts in various directions, for instance, mining rules with or without a variable minimum support threshold [LHM99a], alternative interest measures (besides confidence) [BMUT97], constraint-based mining [PHL01], mining sequential data [SK02], using association rules for classification [LHM98], quantitative or causal rules [SBMU00], concise representations of frequent itemsets (closed, maximal, etc.) [GZ05], correlations [BMS97], utility mining [WSTZ05], outlier detection [DN10], sampling [Par02] and clustering [Toi96], etc. In the next section, we will discuss how FPM performs on streaming data which is much more challenging than static data.

2.3 Mining Data Streams

As discussed in the previous section, many efficient frequent pattern algorithms have been developed in the last decade. These algorithms typically require data sets to be stored in persistent storage and involve two or more passes over the data set. However, for streaming data, the advancement in research has not been so spectacular. In a streaming environment, a mining algorithm must take only a single pass over the data. Most of the time the algorithms can only guarantee an approximate result.

Contrasted with other stream processing tasks, the unique challenges in discovering frequent patterns are apparent. Firstly, FPM needs to search a space with an exponential number of patterns. The cardinality of the answer set itself which contains all frequent patterns can be significant. In particular, more space is required to generate an approximate answer set for frequent patterns in a streaming environment. In addition, exact mining requires keeping track of all itemsets in the database and their actual frequency, because any infrequent pattern may become frequent later in the stream. Unfortunately, maintaining all patterns requires $O(2^{|\mathcal{I}|})$, making exact mining computationally unrealistic, in terms of both memory and CPU. In this respect, the mining algorithm needs to be very memory-efficient.

Secondly, in order to generate frequent patterns, FPM depends on the anti-monotone property to prune infrequent patterns. This process (even without the streaming constraint) is very computation intensive. As a result, keeping pace with high speed data streams can be very difficult for a FPM task. With these challenges, a more important and practical issue is the quality of the approximate mining results. For more accurate results, we will usually require more memory space and computations. What should be the acceptable mining results to a data miner? To deal with this problem, a mining algorithm needs to provide users the flexibility to control the accuracy of the final mining results. Although for several years many researchers have been proposing algorithms on frequent pattern mining over streaming data (first prominent paper

appeared in VLDB 2002 [MM02] [2]), even the recent papers on the topic show that finding frequent patterns in streaming data is not trivial [CDG07]. Manku and Motwani nicely described this problem in their seminal work - Lossy Counting Algorithm (LCA) [MM02]. Their work led to many other similar papers that use approximate counting [CKN08, CKN06, CL04, GHP+03, YCLZ04]. We will describe this algorithm in greater detail in the next chapter.

2.4 Data Stream Models and Approximate Mining Algorithms

A transaction data stream is a sequence of incoming transactions, $Ds = (t_1, t_2, ..., t_i, ...)$, where t_i is the i-th transaction. Here, a portion of the stream is called a window. A window, W, is a subsequence of the stream between the ith and the jth point where $W = (T_i, T_{i+1}, ..., T_j)$ and $i < j$. The point, T, can be either time-based or count-based. For time-base, W consists of a sequence of fixed length time units, where a variable number of transactions may arrive within each time unit. As for count-based, W is composed of a sequence of batches, where each batch consists of an equal number of transactions. To process and mine the stream, different window models are often employed. In this dissertation, we identify four types of data stream mining models: *landmark window*, *sliding window*, *damped window* and *tilted-time window* [Sha02, JG06, Agg07].

2.4.1 Landmark Window

A model is considered a *landmark window* if $W = (T_i, T_{i+1}, ..., T_\tau)$, where τ is the current time point. In this model, we are mining for frequent patterns starting from

[2] Prior to this work, back in 1982, Misra and Gries [MG82] proposed the earliest deterministic algorithm for ϵ-approximate frequency counts. The same algorithm has been rediscovered recently by Demaine et al [DLOM02] and Karp et al [KSP03].

the time point called landmark, i, to the most recent time point. Usually, i is set to 1 and thus we are trying to discover frequent patterns over the entire history of the data stream.

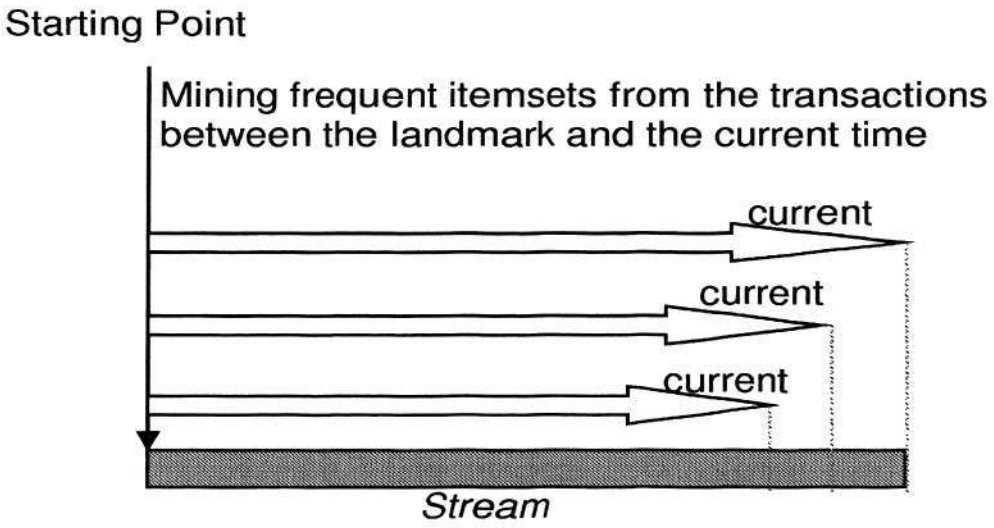


Figure 2.1: Landmark model.

Manku and Motwani [MM02] proposed a method for mining all frequent itemsets over the life of the stream. LCA stores itemsets in a tree structure. With each itemset stored, they record an under-estimate of the frequency and an error term. Upon the arrival of a new batch of transactions, the tree is updated; some itemsets may be dropped and others may be added. The dropping and adding conditions ensure that: (1) the estimated frequency of all itemsets in the tree are less than the true frequency by no more than ϵN (N is the length of the stream and ϵ is a user-defined error bound) and (2) no frequent itemset is omitted from the tree. By scanning the tree and returning all patterns whose estimated frequency is greater than $(\sigma_{min} - \epsilon) \times N$, an approximation to the frequent patterns can be made. For single itemset mining, LCA requires at most $\frac{1}{\epsilon} log(\epsilon N)$ of memory space.

Similarly, Li et al. proposed the online incremental projected, single-pass algorithms, called DSM-FI [LLS04] and DSM-MFI [LLS05] to mine the set of all frequent itemsets and maximal frequent itemsets over the entire history of data streams. Extended prefix-tree based data structures are developed in the proposed algorithms to

store items and their support values, window ids, and nodes links pointing to the root or a certain node. The experiments show that when data set is dense, DSM-FI is more efficient than LCA.

Observe that in a landmark window, we consider all the time points from the landmark till the most recent point as equally important. There is no distinction made between past and present data. However, since data streams may contain time-varying data elements, patterns which are frequent may evolve as well. Often these changes make the mining result built on the old transactions inconsistent with the new ones. To emphasize the recency of data, we can adopt the *sliding window*, *damped window* or *tilted-time window*.

2.4.2 Sliding Window

As a simple solution for time changing environment, it is possible to consider the *sliding window* model [BDM02]. A *sliding window* keeps a window of size w. Here, we are interested to mine the frequent patterns in the window $W = (T_{\tau-w+1}, ..., T_\tau)$ where T_i is a batch or time unit. When a new transaction arrives, the oldest resident in the window is considered obsolete and deleted to make room for the new one. The mining result is totally dependent on the range of the window. All the transactions in the window need to be maintained. We need to remove their effects on the current mining result when they become obsolete. For example, when $T_{\tau+1}$ arrives, we need to remove the effect of $T_{\tau-w+1}$.

Based on the estimation mechanism of LCA, Chang and Lee introduced a sliding window method for finding recent frequent patterns in a data stream when the minimum support threshold, error parameter and sliding window size are provided [CL04]. A recent frequent pattern is one whose support in the transactions within the current sliding window is greater than or equal to the support threshold. By restricting the target of a mining process as a fixed number of recently generated transactions

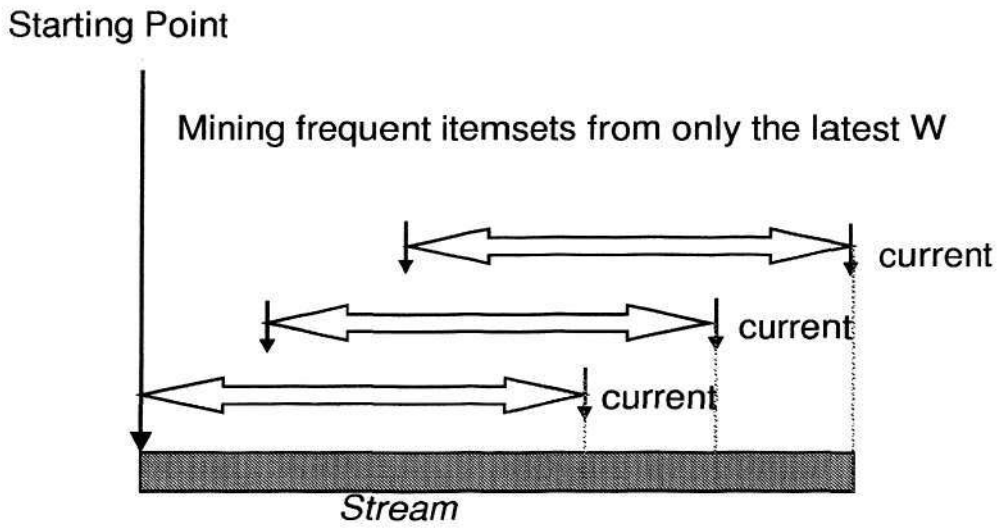


Figure 2.2: Sliding-window model.

in a data stream, the recent change of the data stream can be efficiently analyzed. Consequently, the proposed method can catch the recent change of a data stream as well as analyze the past variations of knowledge embedded in a data stream. In addition, Chang and Lee also proposed another similar algorithm (*estWin*) to maintain frequent patterns over a sliding window [CL03a].

Chi et al. proposed a new mining algorithm, called MOMENT to monitor transactions in the sliding window so that we can output the current closed frequent itemsets at any time [CWYM04]. Key to this algorithm is a compact data structure, the closed enumeration tree (CET), used to maintain a dynamically selected set of patterns over a sliding-window. The selected patterns consist of a boundary between closed frequent itemsets and the rest of the itemsets. Concept drifts in a data stream are reflected by boundary movements in the CET. Since the boundary is relatively stable, the cost of mining closed frequent itemsets over a sliding window is dramatically reduced to that of mining transactions that can possibly cause boundary movements in the CET.

In most algorithms that adopt the *sliding window* model, the biggest challenge will be quantifying w. In most cases, it is externally (user-)defined. However, if w is too large and there is concept drift, it is possible to contain outdated information, which

will reduce the learned model accuracy. On the other hand, if w is too small, it may have insufficient data and the learned model will likely incur a large variance. Here, the open direction of research is to design a window that can dynamically decide its width based on the rate of the underlying change phenomenon [CDG07].

In addition, the use of a sliding window to mine for frequent patterns from the immediately preceding points may represent another extreme and rather unstable solution. This is because one may not wish to completely lose the entire history of the past data. Distant historical behavior and trend may also be queried periodically. In such cases, we can adopt the *damped window* which does not erase old data completely but rather slowly "forgets" them.

2.4.3 Damped window

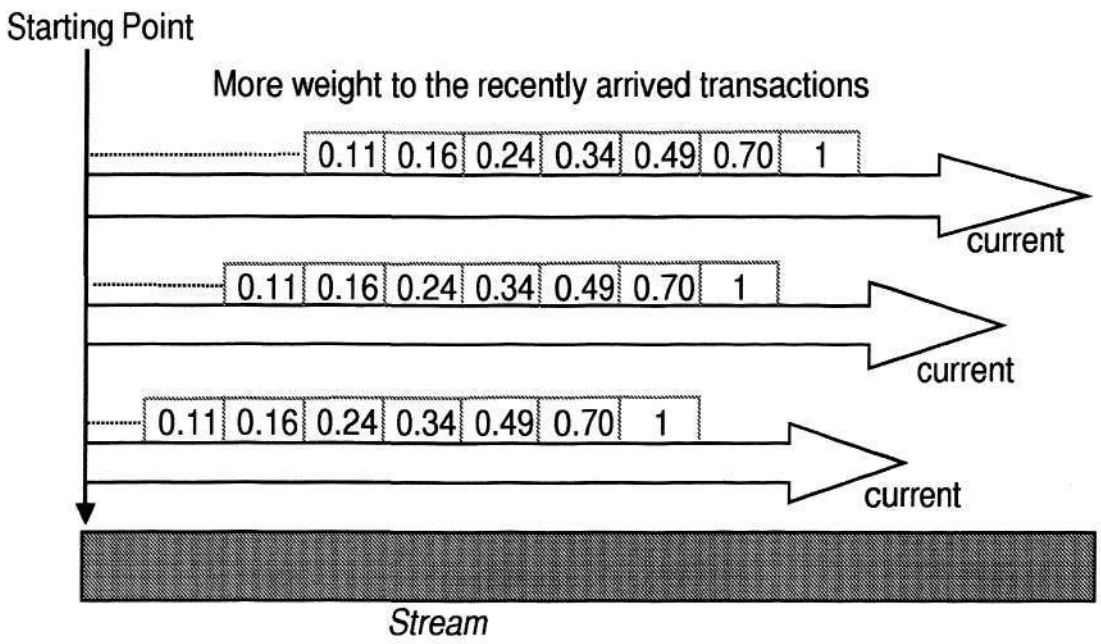


Figure 2.3: Time-fading model.

The *damped window* (also called *time-fading window*) assigns different weights to transactions such that new transactions have higher weights than old ones [CL03b]. The use of the *damped window* diminishes the effect of the old and obsolete information

of a data stream on the mining result. At every moment, based on a fixed decay rate [CS03] or forgetting factor [YSJ+00], a transaction processed d time steps ago is assigned a weight m^d, where $m < 1$. The weight decreases as time passes by (see Figure 2.3). In general, the closer to 1 the decay, the more the history is taken into account. Correspondingly, the count of an itemset is also defined based on the weight of each transaction.

Chang and Lee proposed an *estDec* method for mining recent frequent patterns adaptively over an online data stream [CL03b]. *estDec* examines each transaction in a data stream one-by-one without candidate generation. The algorithm only needs to update the counts for the itemsets which are the subsets of the newly arrived transaction. The occurrence count of a significant pattern that appears in each transaction is maintained by a prefix-tree lattice structure in the main memory. The effect of old transactions on the current mining result is diminished by decaying the old occurrence count of each pattern as time goes by. In addition, the rate of decay of old information is flexibly defined as needed. The total number of significant patterns in the main memory is minimized by delayed-insertions and pruning operations on an itemset. As a result, its processing time is flexibly governed while sacrificing its accuracy.

As highlighted in [CKN08], estimating the frequency of a pattern from the frequency of its subsets can generate a large error. The error may propagate all the way from 2-subsets to n-supersets. Therefore, it is difficult to formulate an error bound on the computed frequency of the resulting itemsets and a large number of the false positive itemsets will be produced, since the computed frequency of an itemset can be larger than its actual frequency. Moreover, *damped* model suffers from similar problems as in *sliding windows* model. The challenge of determining a suitable w in sliding windows model is now translated to that of determining a suitable m.

2.4.4 Tilted-time window

In data stream analysis, some people are more concerned about recent changes at a fine scale but in long term changes at a coarse scale. We can register time at different levels of granularity. The most recent time is registered at the finest granularity while the earlier time is registered at a coarser granularity. Such time dimension model is often called a tilted-time frame [CDH+02]. In this model, we are interested in frequent patterns over a set of windows. Here, each window corresponds to different time granularity based on their recency. There are several ways to design a titled-time frame. For example, Figure 2.4 shows such a *tilted-time window*: the most recent 4 quarters of an hour, then the last 24 hours, and 31 days. Based on this model, one can compute frequent patterns in the last hour with the precision of quarter of an hour, the last day with the precision of hour, and so on, until the whole month. This model registers only $4 + 24 + 31 = 59$ units of time, with an acceptable trade-off of lower granularity at a distance time.

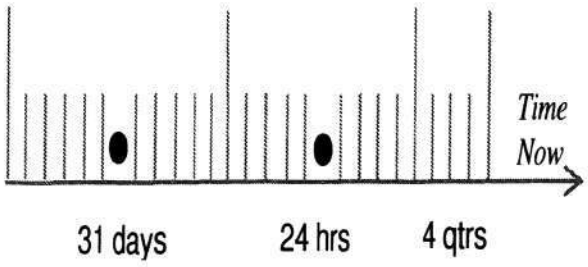


Figure 2.4: Tilted-time model.

With this model, we can compute frequent patterns with different precisions. Giannella et.al. make use of the characteristics of the FP-Growth [HPY00] and a logarithmic *tilted-time window* to mine frequent patterns in data stream. Their algorithm, FP-Stream, is able to count itemsets and save the frequent patterns in an effective FP-Tree-based model [GHP+03]. FP-Stream can be seen as a combination of the different models. For an itemset, frequencies are computed for the most recent windows of lengths w, $2w$, $4w$, $8w$, etc. So, the more recent part of the stream is covered more

thoroughly. The combination of these frequencies permit efficient query answering over the history of the stream.

Although this method adopts the *tilted-time window* to solve the aforesaid problem in different granularity and to update frequent patterns with the incoming data stream, it considers all the transactions as the same. In the long run, FP-Stream will inevitably become very large over time and updating and scanning such a large structure may degrade its performance. In this respect, we need to design a better data structure to mine frequent patterns efficiently.

2.4.5 Concluding remarks on Approximate Counting

We have covered four different types of mining models. All these four models have been considered in current research on data stream mining. Choice of a model largely depends on the application needs. In this work, we consider the *landmark window*, finding frequent patterns over the entire data stream, as the most challenging and fundamentally important one. Often, it can serve as the basis to solve the latter three. For example, it is not difficult to see that an algorithm based on the *landmark window* can be converted to that using the *damped window* by adding a decay function on the upcoming data streams. It can also be converted to that using *sliding window* by keeping track of and processing data within a specified *sliding window*.

We note that all the previously described algorithms adopt a false-positive approach [3] (we will describe the false negative approach[4] in the next section). Such approach uses a relaxed minimum support threshold, ϵ, to reduce the number of false positives and to obtain a more accurate frequency of the result itemsets. However, in reality, it is difficult to define ϵ. Setting a wrong value for ϵ may either trigger a massive generation of false positives thus degrading the mining result, or slow down

[3] A false positive approach is one that returns a set of patterns that contains all frequent patterns and also some infrequent patterns.

[4] A false negative approach is one that returns a set of patterns that does not contain any infrequent patterns but may miss some frequent patterns.

the processing speed due to over-intensive updating and scanning of the large data structure. This drawback will be demonstrated in the next chapter.

In addition, we observe that current data stream mining algorithms require the user to define one or more parameters before execution. Unfortunately, most of them do not provide any clue as to how we can adjust these parameters online while they are running. Some proposed methods let users adjust only certain parameters online, but these parameters may not be the key ones to the mining algorithms, and thus are not enough for a user friendly mining environment. From a practical point of view, it is also not feasible for user to wait for a mining algorithm stops before he can reset the parameters. This is because it may take a long time (or never) for the algorithm to finish due to the continuous arrivals and huge amount of data streams. For further improvement, we may consider either to allow the user to adjust online or to let the mining algorithm auto-adjust most of the key parameters in FPM, such as support, ϵ, window size and decay rate.

2.5 Sampling

FPM algorithms require multiple passes over the whole database, and consequently the database size is by far the most influential factor of the execution time for very large databases. This is prohibitively expensive. For most mining tasks, exact counts are not necessarily required and an approximate representation is often more appropriate. This motivates an approach called data reduction (or synopsis representation) [BDF+97]. The problem of synopsis maintenance has been studied in great detail because of its applications to problems such as query estimation in data streams [AY06]. Several synopsis approaches such as sampling, histogram, sketches and wavelets are designed for use with specific applications. We do not attempt to survey all data reduction methods here. In this dissertation, we focus our research on sampling.

Sampling has great appeal because of its simplicity and general purpose reduction method which applies to a wide range of applications. Moreover, the benefits of sampling over other data reduction methods are more pronounced with increasing dimensionality: the larger the dimensionality, the more compact sampling becomes compared to, for example , multi-dimensional histograms or wavelet decompositions [BDF+97]. Also, sampling retains the relations and correlations between the dimensions, which may be lost by histograms or other reduction techniques. This latter point is important for data mining and analysis. Thus, the generated sample can be easily fed to arbitrary data mining applications with little changes to the underlying method and algorithms. The survey by Olken and Rotem offers a good overview of sampling algorithms in databases [OR95]. Interested readers can also refer to [KK06] for a recent survey on association rule mining where some additional techniques on the use of sampling are given.

In recent years, a lot of algorithms based on sampling in mining frequent patterns in large databases have been proposed. In [CHS02], a method called *FAST* is introduced. It is a two-phase sampling based algorithm for discovering association rules in large databases. In Phase I, a large initial large sample of transactions is selected. It is used to quickly and almost accurately estimate the support of each individual item in the database. In Phase II, these estimated supports are used to either trim outlier transactions or select representative transactions from the initial sample, thereby forming a small final sample that more accurately reflects the statistical characteristics (i.e., itemset supports) of the entire database. The expensive operation of discovering all association rules is then performed on the final sample. In [HK06b], a new association mining algorithm is proposed that uses two phase sampling that addresses a limitation of *FAST*. *FAST* has the limitation that it only considers the frequent 1-itemsets in trimming/growing, thus, it did not have ways of considering mulit-itemsets including 2-itemsets. The new algorithm reflects the multi-itemsets in

sampling transactions. It improves the mining results by adjusting the counts of both missing itemsets and false itemsets.

Lee et al. studied the usefulness of sampling techniques in data mining [LCK98]. By applying sampling techniques, they devised a new algorithm DELI for finding an approximate upper bound on the size of the difference between association rules of a database before and after it is updated. The algorithm uses sampling and statistical methods to give a reliable upper bound. If the bound is low, then the changes in association rules are small. So, the old association rules can be taken as a good approximation of the new ones. If the bound is high, the necessity of updating the association rules in the database is signaled. Experiments show that DELI is not only efficient with high reliability, but also scalable. It is effective in saving machine resources in the maintenance of association rules.

2.5.1 ϵ-approximation

The concept of ϵ-approximation has been widely used in machine learning. A seminal result of Vapnik and Chervonenkis shows that a random sample of size $O(\frac{d}{\epsilon^2}\log\frac{1}{\epsilon})$, where d is the "VC dimension" (assumed finite) of the data set, is an ϵ approximation. This result establishes the link between random samples and frequency estimations over several items simultaneously.

An example of ϵ-approximation algorithm is the Sticky Sampling [MM02]. The algorithm is probabilistic in nature and can be employed to handle data streams. The user needs to specify three parameters: support σ_{min}, error ϵ and probability of failure δ. Sticky Sampling uses a fixed size buffer and a variable sampling rate to estimate the counts of incoming items. The data structure S is a set of entries of the form (e, f), where f estimates the frequency of an item e belonging to the stream. Initially, S is empty, and the sampling rate r is set to 1. Sampling an item with rate r means that the item is selected with probability $\frac{1}{r}$. For each incoming item e, if an entry for e already exists in S, the corresponding frequency f will be incremented; otherwise,

sample the item with r. If this item is selected by sampling, we add an entry $(e, 1)$ to S ; otherwise, ignore e and move on to the next incoming item in the stream. Sticky Sampling is able to compute an ϵ-deficient synopsis with probability at least $1 - \delta$ using at most $\frac{2}{\epsilon}\log(S^{-1}\delta^{-1})$ expected number of entries. While the algorithm can accurately maintain the statistics of items over stable data streams, it fails to address the needs of important applications, such as network traffic control and pricing, that require information about the entire stream but with emphasis on the most recent data. Moreover, Sticky Sampling cannot be applied in FPM because it only works well for frequent single itemsets.

In [BCD+03], an algorithm called *EASE* is proposed. *EASE* can deterministically produce a sample by repeatedly halving the data to arrive at the given sample size. The sample is produced by ϵ-approximation method. *EASE* leads to much better estimation of frequencies than simple random sampling. Unlike *FAST* [CHS02], *EASE* provides a guaranteed upper bound on the distance between the initial sample and final sub-sample. In addition, *EASE* can process transactions on the fly, i.e., a transaction needs to be examined only once to determine whether it belongs to the final sub-sample. Moreover, the average time needed to process a transaction is proportional to the number of items in that transaction.

2.5.2 Progressive Sampling

Choosing a random sample of the data is one of the most natural ways of choosing a representative subset. The primary challenge in developing sampling-based algorithms stems from the fact that the support value of an itemset in a sample almost always deviates from the support value in the entire database. A wrong sample size may result in missing itemsets that have high support value in the database but not in the sample and false itemsets that are considered frequent in the sample but not in the database. Therefore, the determination of the sample size then becomes the critical factor in order to ensure that the outcomes of the mining process on the sample

generalize well. However, the correct sample size is rarely obvious. How much sample is enough really depends on the combination of chosen mining algorithms, data set and application related loss function.

Parthasarathy [Par02] attempted to determine the sample size using progressive sampling [PJO99, ND06]. Progressive sampling is a technique that starts with a small sample, and then increases the sample size until a sample of sufficient size has been obtained. While this technique eliminates the need to determine the correct size initially, it requires that there be a way to evaluate the sample to judge if it is large enough. Parthasarathy's approach relies on a novel measure of model accuracy (self-similarity of associations across progressive samples), the identification of a representative class of frequent patterns that mimics the self-similarity values across the entire set of associations, and an efficient sampling methodology that hides the overhead of obtaining progressive samples by overlapping it with useful computation. However, in streaming data context, computing the sufficiency amount can be formidably expensive especially for hill-climbing based methods.

2.5.3 Statistical Sampling

We say that a data sample is sufficient if the observed statistics, such as mean or sample total have a variance lower than the predefined limits with high confidence. Large deviation bounds such as the Hoeffding bound (sometimes called the additive Chernoff bounds) can often be used to compute a sample size n so that a given statistics on the sample is no more than ε away from the same statistics on the entire database, where ε is a tolerated error. As an example, if μ is the expected value of some amount in $[0, 1]$ taken over all items in the database, and if $\hat{\mu}$ is a random variable denoting the value of that average over an observed sample, then the Hoeffding bound states that with probability at least $1 - \delta$,

$$|\mu - \hat{\mu}| \leq \varepsilon. \qquad \text{(Eq. 2.1)}$$

The sample size must be at least n, where

$$n = \frac{1}{2\varepsilon^2} . \ln \frac{2}{\delta}. \qquad \text{(Eq. 2.2)}$$

This equation is very useful and it has been successfully applied in classification as well as association rule mining [DH00, Toi96]. Although the Hoeffding bound is often considered conservative, it gives the user at least a certain degree of confidence. Note that demanding small error is expensive, due to the quadratic dependence on ε, but demanding high confidence is affordable, thanks to the logarithmic dependence on δ. For example, in FPM, given $\varepsilon = 0.01$ and $\delta = 0.1$, $n \approx 14,979$, which means that for itemset X, if we sample 14,979 transactions from a data set, then its true support $\sigma(X)$ is beyond the range of $[\sigma(X) - 0.01, \sigma(X) + 0.01]$ with probability 0.1. In other words, $\sigma(X)$ is within $\pm\varepsilon$ of $\sigma(X)$ with high confidence 0.9.

Recent research has managed to reduce the disk I/O activity to two full scans over the database. Toivonen [Toi96] uses this statistical sampling technique in his algorithm. The algorithm makes only one full pass over the database. The idea is to pick a simple random sample, use it to determine all frequent patterns that probably hold in the whole database, and then to verify the results with the rest of the database in the second pass. The algorithm thus produces exact frequent patterns in one full pass over the database. In those rare cases where the sampling method does not produce all frequent patterns, the missing patterns can be found in a second pass. Toivonen's algorithm provides strong but probabilistic guarantees on the accuracy of its frequency counts.

In [ZPLO96], further study on the usefulness of sampling over large transactional data is carried out. They concluded that simple random sampling can reduce the I/O cost and computation time for association rule mining. This work is complementary to the approach in [Toi96], and can help in determining a better support value or sample size. The authors experimentally evaluate the effectiveness of sampling on different

databases, and study the relationship between the performance, the accuracy and confidence of the chosen sample.

In [YCLZ04], Yu et al. proposed FPDM, which is perhaps the first false-negative oriented approach for mining frequent patterns over data streams. FPDM utilizes the Chernoff bound to achieve the quality control for finding frequent patterns. Their algorithm does not generate any false positive, and has a high-probability to find patterns which are truly frequent. In particular, they use a user-defined parameter δ to govern the probability to find the frequent patterns at support threshold σ_{min}. Specifically, the mining result does not include any patterns whose frequency is less than σ_{min}, and includes any pattern whose frequency exceeds σ_{min} with probability of at least 1 - δ. Similar to the LCA [MM02], FPDM partitioned the data stream into equal sized segments (batches). The batch size is given as $k \cdot n_0$ where n_0 is the required number of observations in order to achieve Chernoff bound with the user defined δ. Note that k is a parameter to control the batch size. For each batch, FPDM uses non-streaming FPM algorithm such as the Apriori algorithm to mine all the locally frequent patterns whose support is no less than $\sigma_{min} - \varepsilon$. The set of locally frequent patterns is then merged with those frequent patterns obtained so far from the stream. If the total number of patterns kept for the stream is larger than $c \cdot n$, where c is an empirically determined float number, then all patterns whose support is less than ($\sigma_{min} - \varepsilon$) are pruned. FPDM outputs those patterns whose frequency is no less than $\sigma_{min} \times N$ where N is the number of transactions received so far.

2.5.4 Reservoir Sampling

Described by Fan et al., the classic reservoir sampling algorithm is a sequential sampling algorithm over a finite population of records, with the population size unknown [FMR62]. Assuming that a sample of size n is desired, one would commence by placing the first n encountered into the putative sample (the reservoir). Each subsequent record is processed and considered in turn. The k-th record is accepted with

probability n/k. If accepted, it displaces a randomly chosen record from the putative sample. Fan et al. showed that this will produce a simple random sample.

In practice, most stream sizes are unknown. [JMR05] suggests that it is useful to keep a fixed-size sample. The issue of how to maintain a sample of a specified size over data that arrives online has been studied in the past. The standard solution is to use reservoir sampling of J.S. Vitter [Vit85]. The technique of reservoir sampling is, in one sequential pass, to select a random sample of n transactions from a data set of N transactions where N is unknown and $n \ll N$. Vitter introduces Algo-Z, which allows random sampling of streaming data. The sample serves as a reservoir that buffers certain transactions from the data stream. New transactions appearing in the stream may be captured by the reservoir, whose limited capacity then forces an existing transaction to exit the reservoir. Variations on the algorithm allow it to idle for a period of time during which it only counts the number of transactions that have passed by. After a certain number of transactions are scanned, the algorithm can awaken to capture the next transaction from the stream. Reservoir Sampling can be very efficient, with time complexity less than linear in the size of the stream. Since 1985, there have been some modifications of Vitter's algorithm to make it more efficient. For example, in [Li94], Algo-Z, is modified to give a more efficient algorithm, Algo-K. Additionally, two new algorithms, Algo-L and Algo-M, are proposed. If the time for scanning the data set is ignored, all the four algorithms (K, L, M, and Z) have expected CPU time $O(n(1+log(\frac{N}{n})))$, which is optimum up to a constant factor.

In [BDM02], "chain-sample" is proposed for a moving window of recent transactions from a data stream. If the i^{th} transaction is selected for inclusion in the sample, another index from the range $i+1, ..., i+n$ is selected which will replace in the event of its deletion. When the transaction with the selected index arrives, the algorithm stores it in memory and chooses the index of the transaction that will replace *it* when it expires and so on, thus building a chain of transactions to use in case of the expiration of the current transaction in the sample. The sample as a result will be a

random sample. So, although this is able to overcome the problem of deletion in reservoir sampling, it still suffers from the problems that a random sample suffers from, namely, random fluctuations particularly when the sample size is small, and inability to handle noise. In [POSG04], a reservoir sampling with replacement for streaming data is proposed. It allows duplicates in the sample. The proposed method maintains a random sample with replacement at any given time. Although the difference between sampling with and without replacement is negligible with large population size, the authors argue that there are still situations when sampling with replacement is preferred.

In [Agg06], the author proposed a new method for biased reservoir based sampling of data streams. This technique is especially relevant for continual streams which gradually evolve over time, as a result of which the old data becomes obsolete for a variety of data mining applications. While biased reservoir sampling is a very difficult problem (with the one pass constraint), the author demonstrates that it is possible to design very efficient replacement algorithms for the important class of memory-less bias functions. In addition, the incorporation of bias results in upper bounds on reservoir sizes in many cases. In the special case of memory-less bias functions, the maximum space requirement is constant even for an infinitely long data stream. This is a nice property, since it allows for easy implementation in a variety of space-constrained scenarios.

2.6 Approximate Counting vs. Sampling

Earlier, we have discussed many algorithms that adopt either approximate counting or sampling. At the heart of approximate counting is the fact that for streaming data one cannot keep exact frequency count for all possible itemsets. Note that here we are concerned with data sets that have 1000s of items or more, not just trivial data sets with less than 10 items. To solve this memory problem, approximate counting maintains only those itemsets which are frequent in at least a small portion of the stream, but if the itemset is found to be infrequent it is discontinued. As approximate counting does not maintain any information about the stream that has passed by, it adds an error frequency term to make the total frequency of a "potentially" frequent itemset higher than its actual frequency. Thus, approximate counting in most cases produces 100% recall but suffer from poor precision.

Sampling is another approach that is used in [Toi96, MTV94, ZPLO96] to produce frequent patterns. The idea is very straightforward: maintain a sample over the stream and when asked, run the frequent pattern mining algorithm, such as the Apriori algorithm, to output the frequent patterns. Research in sampling focuses on how to maintain a sample over the streaming data. On one hand approximate counting such as LCA [MM02] does not keep any information about the stream that has passed by but keeps exact information starting from some point in the stream. On the other hand sampling keeps information about the whole stream (one can use a decaying factor to decrease or increase the influence of the past) but only partially. So, a sample will have both false positives and false negatives. Approximate counting will not have any false negatives, but it will have false positives. A researcher would love to know how these two compare against each other. This is the focus of our research (see Chapters 3 and 4).

2.7 Other Related Work

Due to the increasing prevalence of data streams, improvement in these areas will likely find immediate applications. As a result, recent years have witnessed a growing interest in designing algorithms for continuous data streams. Since there is a large volume of literature which has been published in this field, a thorough review of all the work is not possible in this thesis. In this section, we briefly summarize some of the prominent work in the area of data stream mining.

Clustering Cluster analysis seeks to discover groups of closely related observations so that observations that belong to the same cluster are more similar to each other than observations that belong to other clusters. Clustering is a widely studied problem in the data mining literature. However, it is more difficult to adapt arbitrary clustering algorithms to data streams because of one-pass constraints on the data set. An interesting adaptation of the k-means algorithm has been discussed in [GMMO00] which uses a partitioning based approach on the entire data set. STREAM is a single-pass, constant factor approximation algorithm. It uses an adaptation of a k-means technique in order to create clusters over the entire data stream.

CluStream is an algorithm for the clustering of evolving data streams based on user specified, on-line clustering queries [AHWY03]. CluStream contains an on-line component and off-line component. In on-line, it computes and stores summary statistics about the data stream using micro-clusters, and performs incremental on-line computation and maintenance of the micro-clusters. In off-line, it does macro-clustering and answers various user questions using the stored summary statistics, which are based on the tilted time frame model (see Section 2.4.4). CluStream claims to achieve a higher accuracy than STREAM thanks to the exploitation of micro-clustering techniques.

Classification Classification is the process of finding a model (or function) that describes and distinguishes data classes or concepts, for the intention of being able to employ the model to predict the class of objects whose class label is not available. The problem of classification in data stream is perhaps one of the most widely studied. Several methods have been proposed for streaming data classification. Domingos et al. [DH00] have developed the Very Fast Decision Tree (VFDT). VFDT is a decision tree learning system based on Hoeffding trees. It splits the tree using the current best attribute taking into consideration that the number of scanned data sets used satisfies a statistical measure which is Hoeffding bound. In addition, a later framework has been termed as CVFDT to handle concept drift in time changing data streams [HSD01].

Wang et al. [WFYH03] have proposed a generic framework for mining concept drifting data streams. An important motivation behind the framework is to deal with the expiration of old data streams. The idea is to train an ensemble of classification models (C4.5, Naive Bayes, RIPPER, and others) from sequential batches of the data stream. Whenever a batch arrives, a new classifier is built. The strength of each classifier is evaluated based on its expected classification accuracy in the time-changing environment. Only the best number of k classifiers are kept. Aggarwal et al. have adopted the idea of microclusters introduced in CluStream [AHWY03] in On-Demand classification and it shows a high accuracy [AHWY04]. The technique uses clustering results to classify data using statistics of class distribution in each cluster.

Others Research related to data stream has been very active in recent years. Some of the well known surveys in this area include Babu and Widom [BW01], Muthukrishnan [Mut03] and Babcock et al. [BBD+02]. There is also a tutorial by Garofalakis et al. [GGR02].

There have been some extensive studies on stream data management and the processing of continuous queries in data stream. Query or aggregate processing methods have been proposed by Gehrke et al. [GKS01], Chandrasekaran and Franklin [CF02] and Dobra et al. [DGGR02]. Zhu and Shasha [ZS02] developed Statstream, a statistical method for the monitoring of thousands of data streams in real time. MAIDS (Mining Alarming Incidents from Data Streams), a stream data mining system built on top of a stream data cube, was developed by Cai et al. [CCP+04]. Chen et al. introduced a multidimensional regression method for analysis of multidimensional time-series data [CDH+02].

In streaming data, the arrival rates are often high and bursty. When the load of mining exceeds the capacity of the system, load shedding is needed to keep up with the arrival rates of the input streams [SW04, DNO06, TcZ07]. Analogous to load shedding in query processing, Gaber et al. proposed data rate adaptation as a solution approach for mining data streams [GKZ05]. Dang et al. addressed 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 [DNOL07]. For other FPM problems, Karp et al. proposed a counting algorithm for finding frequent elements in data streams [KSP03]. Metwally et al. introduced a memory efficient method for computing frequent and top-k elements in data streams [MAA05].

2.8 Summary

The high complexity of the FPM problem handicaps the application of the stream mining techniques. We note that a review of existing techniques is necessary in order to research and develop efficient mining algorithms and data structures that are able to match the processing rate of the mining with high speed data streams. In this chapter, we have described the prior works related to our research. We have presented a number of the state-of-the-art algorithms on FPM and sampling over data streams. We summarize them in Table 2.2. Note that some of these general purpose sampling algorithms such as Algo-Z existed long before FPM become popular. However, they have proven to be useful tools for solving data stream problems.

Table 2.2: Overview of FPM and sampling algorithms over data streams

Authors	Algorithm Name	Window Model	Strategy
Manku and Motwani [MM02]	LCA	Landmark	Approximate Counting
Li et .al [LLS04]	DSM-FI	Landmark	Approximate Counting
Li et .al [LLS05]	DSM-MFI	Landmark	Approximate Counting
Manku and Motwani [MM02]	Sticky	Landmark	Sampling Related
Bronnimann et .al [BCD+03]	*EASE*	Landmark	Sampling Related
Toivonen [Toi96]	not provided	Landmark	Sampling Related
Yu et al. [YCLZ04]	FPDM	Landmark	Sampling Related
Vitter [Vit85]	Algo-Z	Landmark	Sampling Related
Li [Li94]	Algo-K, Algo-L and Algo-M	Landmark	Sampling Related
Park et al. [POSG04]	RSWR	Landmark	Sampling Related
Chang and Lee [CL04]	not provided	Sliding	Approximate Counting
Chang and Lee [CL03a]	*estWin*	Sliding	Approximate Counting
Chi et al. [CWYM04]	MOMENT	Sliding	Approximate Counting
Chang and Lee [CL03b]	*estDec*	Damped	Approximate Counting
Aggarwal [Agg06]	not provided	Damped	Sampling Related
Giannella et al. [HPY00]	FP-Stream	Titled-time	Approximate Counting

Chapter 3

Approximate Counting of Frequent Pattterns over Transactional Data Streams

In this chapter, we investigate the problem of finding frequent patterns in a continuous stream of transactions using approximate counting. When mining for frequent patterns, it is well recognized that the approximate solutions are usually sufficient and many existing literature explicitly trade off accuracy for speed and memory space where the quality of the final approximate counts are governed by an error parameter, ϵ. However, while implementing such algorithms, the quantification of ϵ is never simple. In this chapter, we will show that by setting a small ϵ, we achieve good accuracy but suffer in terms of efficiency. A larger ϵ improves the efficiency but seriously degrades the mining accuracy. Even a slight change in ϵ may have a big impact on the outcome. To abate the dependency on ϵ, we propose an alternative which allows user to customize a set of error bounds based on his requirement. From an implementation perspective, the experimental studies show that the proposed algorithm has improved precision, requires less memory and consumes less CPU time.

3.1 Preliminaries

In order to ensure the completeness of frequent patterns in the streaming data, it is necessary to capture not only the information related to frequent items, but also that related to the infrequent ones. If the counts about the currently infrequent patterns were not maintained, such information would be lost forever. If these patterns emerge as frequent later, it would be impossible to figure out their correct overall frequency and their connections with other patterns. However, it is totally impractical to hold all streaming data in the limited main memory. In this respect, we need to divide patterns into three categories: frequent patterns (see Chapter 2 for the definition), sub-frequent patterns and infrequent patterns.

Definition 3.2 [**Maximum Support Error Threshold,** ϵ] *We denote N as the number of transactions processed so far. Let ϵ be a given error parameter such that $\epsilon \in (0,\ \sigma_{min}]$. It is an error bound that is employed to approximate the frequency of an itemset such that the estimated frequency is less than the true frequency by at most $\epsilon \times N$.*

Definition 3.3 [**Sub-frequent Pattern**] *An itemset X is considered as a sub-frequent pattern (FP_{sub}) if its frequency is less than $\sigma_{min} \times N$ but not less than $\epsilon \times N$, where $\epsilon < \sigma_{min}$. Let AFP_{sub} be the set of all FP_{sub}.*

Definition 3.4 [**Infrequent Pattern**] *An itemset X is considered as an infrequent pattern if its frequency is less than $\epsilon \times N$. An infrequent pattern will not be maintained in the main memory and will be discarded.*

Definition 3.5 [**Significant Pattern, SP**] *We are only interested in frequent patterns. However, AFP_{sub} are the potential itemsets that may become frequent as time passes by. Therefore, we need to maintain AFP_{sub} and AFP at all times in the main memory. We consider both AFP_{sub} and AFP as significant pattern (SP) and let ASP denote the set of all significant patterns.*

We need to discard infrequent patterns since the number of all infrequent patterns are too large and the loss of frequency from infrequent pattern will not affect the computed frequency too much. At any moment, the set of frequent patterns can be obtained from a highly compact data structure that is residing in the main memory. In the next section, we will discuss how this can be practically achieved by using LCA.

3.2 Lossy Counting Algorithm

In order to provide a comprehensive understanding of the research problem we are addressing, we devote some space here for describing and analyzing the *Lossy Counting* algorithm (LCA). This algorithm is chosen for discussion due to its popularity. In fact, based on the recent survey, several algorithms have adopted the same error bound approach [JG06, CKN08, Agg07]. We refer the reader to [MM02, LTKC05] for further details on the implementation aspects of this algorithm.

LCA requires two input parameters (σ_{min} and ϵ). To find frequent patterns, incoming transactions are processed in batches. The information about the previous mining result up to the latest batch operation is maintained in a data structure called D_{struct} containing a set of entries (X, f_{est}, Δ) where X is an itemset, f_{est} is the estimated frequency of X, and Δ is the maximum possible error frequency of X. For each batch, the transactions are divided into buckets of width $w = \lceil \frac{1}{\epsilon} \rceil$ and loaded together into a fixed-sized buffer in the main memory. The buckets are labeled with

bucket *ids*, starting from 1. We let $b_{current}$ be the current bucket *id* ($current = N/w$) and β be the total number of buckets present in the current batch being processed.

D_{struct}, whose job is to maintain the frequency of AFP and AFP_{sub}, is updated as follows. In order to avoid the combinatorial explosion of itemsets, LCA applies the Apriori property [AS94] such that no superset of an itemset will be generated if the itemset has a frequency less than β in the current batch. Those itemsets whose frequency value is $f_{est} \geq \beta$ will be considered as significant patterns, SP. For better implementation of LCA, a special SETGEN module is used to selectively enumerate itemsets in a batch and to count their frequencies. There are three situations for updating D_{struct}:

1) **Insertion** If an itemset has frequency $f_{est} \geq \beta$ and is not present in D_{struct}, the itemset is added as a new entry where Δ is set to $b_{current} - \beta$. In this way, when a new itemset, X, is inserted to D_{struct}, its true frequency before the current batch can be at most $\epsilon N = b_{current} - \beta$.

2) **Maintenance** If a given itemset already exists in D_{struct}, LCA adds the number of occurrences of the itemset among the current batch of β buckets to f_{est}.

3) **Deletion** If the updated entry satisfies $f_{est} + \Delta \leq b_{current}$, LCA deletes it from D_{struct}. This itemset is considered to be an infrequent pattern and thus un-promising. We can safely removed it from D_{struct}.

In this way, the algorithm ensures that $\forall X$, if $freq(X) \geq \epsilon N$, $(X, f_{est}, \Delta) \in D_{struct}$. When user requests a list of itemsets with threshold σ_{min}, we can simply output those itemsets in D_{struct} where $f_{est} \geq (\sigma_{min} - \epsilon) \times N$. In LCA, the main features are:

(i) All itemsets whose true frequency is above $\sigma_{min} \times N$ are output. There are no false-negatives.

(ii) No itemset whose true frequency is less than $(\sigma_{min} - \epsilon) \times N$ is output.

(iii) The estimated frequency of an itemset is less than its true frequency by at most ϵN.

Although this approximate counting algorithm appears attractive for frequent pattern mining over data streams, we remark that there are two problems that are difficult to address. The computational complexity of LCA can be seriously affected by:

1) **Support Threshold (σ_{min} and ϵ)** Lowering the support threshold, σ_{min} often results in more itemsets being declared as frequent. As such, the quantification of the parameter ϵ is coupled with σ_{min}. Usually it is set at 10% of σ_{min}. Therefore, if σ_{min} is small, ϵ will be even smaller. This has an adverse effect on the computational complexity of the algorithm because more significant patterns will be generated and counted. As the maximum size of frequent patterns increases, the algorithm will need to allocate more memory space for D_{struct}.

2) **Data Sets** For dense data sets, the average transaction width can be very large. This can affect LCA in two ways. First, the maximum size of frequent patterns tends to increase as the average transaction width increases. As a consequence, more patterns need to be examined during the insertion and deletion process. This reduces the speed of LCA. Second, as the width increases, more patterns are contained in the transaction. This again will cause the size of the D_{struct} to swell and eventually drain up the memory resources.

3.3 A Closer Look at ϵ

As a recapitulation, LCA is an online algorithm that employs the error bound, ϵ, for finding frequent patterns with approximate support counts from a stream of market-basket transaction data. A major weakness of LCA is that its performance is greatly

influenced by ϵ which is the crux of this algorithm. Note that ϵ is actually a minimum support threshold used to control the quality of the approximation of the mining result. For small ϵ, LCA achieves high precision, but consumes more memory and time, while for large ϵ, LCA uses small memory and runs faster at the expense of poor precision. If $\epsilon \ll \sigma_{min}$, a large number of SP needs to be generated and maintained. We refer the reader to the super exponential growth mentioned in [ZKM01]. The paper reported that even a slight decrease in the support threshold might create a deep impact on the performance of most mining algorithms. In their experiment, a 0.02% change in σ_{min} increased the number of itemsets from less than a million to over a billion! This implies that outside a very narrow range of support values, the choice of these values is not useful.

3.3.1 Important Observations

To verify our analysis, we conducted a simple experiment to study the impact of changing the value of ϵ on frequent pattern mining. In the experiment, IBM synthetic data set [AS94] is used. A data stream of 2 millions transactions is generated. It has an average transaction length of 15 and 10k unique items. We implemented the LCA. Without loss of generality, we equate the buffer size to the number of transactions processed per batch. The algorithm is run with a fixed batch size of 200k transactions. The Apriori alogrithm [AS94] is used to generate the true frequent patterns AFP_{true} from the data set. We fixed σ_{min} to be 0.1% and the resulted size of $|AFP_{true}| = 26136$. To measure the quality of the results, we employ two metrics — the recall and the precision.

Given a set AFP_{true} of true frequent itemsets and a set AFP_{appro} of frequent itemsets obtained in the output by the algorithm,

$$Recall = \frac{|AFP_{true} \bigcap AFP_{appro}|}{|AFP_{true}|} \qquad \text{(Eq. 3.1)}$$

and

$$Precision = \frac{|AFP_{true} \bigcap AFP_{appro}|}{|AFP_{appro}|}. \quad \text{(Eq. 3.2)}$$

If *Recall* equals 1, the results returned by the algorithm contains all true results. This means no false negative. If *Precision* equals 1, all the results returned by the algorithm are some or all of the true results. This means no false positives are generated. Ideally, we would like to have an algorithm that is able to achieve both recall and precision near to 1.00.

Table 3.1: Impact of varying the value of epsilon.

ϵ (%)	$\|ASP\|$	$\|AFP_{appro}\|$	$\|AFP_{true}\|$	*Recall*	*Precision*	Processing Time(s)
0.05	363614	212488	26136	1.00	0.123	296
0.01	1711851	47694	26136	1.00	0.548	1459
0.005	2032720	35656	26136	1.00	0.733	9121
0.003	3587576	30355	26136	1.00	0.861	19950

In Table 3.1, the first column is the error parameter (ϵ), and the second is the number of itemsets (ASP) that need to be maintained in the data structure D_{struct} while running the LCA. That means, any itemset having a frequency of ϵN or above will be in ASP and ASP is maintained in the main memory. The higher the value of $|ASP|$ the more memory is consumed. The third column is the output size of the frequent itemsets $|AFP_{appro}|$. The fourth column is the true size of the frequent itemsets $|AFP_{true}|$. The next two columns are the accuracy of the output. The last column is the time spent in running the algorithm.

Remark 3.1 *The experiment reflects the dilemma of false-positive oriented approach. The memory consumption increases reciprocally in terms of ϵ where ϵ controls the*

error bound. In contrast, the number of false positives increases as ϵ increases.

One of the reason why LCA is popular is because it has the ability to uncover all frequent patterns. From the table, it is apparent that the LCA indeed ensures all itemsets whose true frequency is above $\sigma_{min} \times N$ are output (*Recall* is equal to 1 at all times). This implies $AFP_{true} \subseteq AFP_{appro}$. When ϵ is at 0.05% (half of σ_{min}), we see that its memory consumption and processing time is low. However, the precision is rather poor. This can be explained from the fact that any itemset in ASP is output as long as its computed frequency is at least $(\sigma_{min} - \epsilon) \times N$. If ϵ approaches σ_{min}, more false positives will be included in the final result.

On the other hand, when ϵ decreases, the cutoff value for ASP starts to reduce. This results in more and more entries being inserted into D_{struct} leading to higher memory consumption and processing time. However, when ϵ is low, $(\sigma_{min} - \epsilon) \times N$ increases, thus making it tougher for any itemset in ASP to get into AFI_{appro} where AFI_{appro} is the end product of LCA. Clearly, the tighter the error bound, the higher the precision. Unfortunately, there is no clear clue as to how small ϵ must be defined. From the table, even the rule of thumb of setting $\epsilon = 0.01\%$ (one-tenth of σ_{min}) does not seem to yield appealing result. The precision is still below satisfactory. For example, out of the 47694 patterns that are considered frequent, 21558 are false positives!

3.3.2 Is it Worth the Overhead?

Table 3.1 provides only one side of the story. Additionally, we further examine the behavior of ASP when it is partitioned according to the length of the itemset. Table 3.2 illustrates the breakdown sizes of $|ASP|$ when ϵ is fixed at 0.01%. From the table, the first column represents the corresponding itemset length. The second column represents the number of itemsets for each k-itemset that the LCA maintained

Table 3.2: Breakdown values of $|ASP|$ with $\epsilon = 0.01\%$

k-itemset	$\lvert ASP_k \rvert$	$\lvert Freqk_{true} \rvert$	Recall	Precision
1	7817	4763	1.00	0.609313
2	115214	3668	1.00	0.031836
3	217080	4692	1.00	0.021614
4	305438	4669	1.00	0.015286
5	338558	4060	1.00	0.011992
6	308944	2420	1.00	0.007833
7	237057	1196	1.00	0.005045
8	153389	499	1.00	0.003253
9	82957	142	1.00	0.001712
10	36996	25	1.00	0.000676
11	13458	2	1.00	0.000149
12	5053	0	NA	NA
Overall	1821961	26136	1.00	0.014345

($|ASP| = \sum |ASP_k|$). The third column denotes the break down size of all the true frequent itemsets ($|AFI_{true}| = \sum |Freqk_{true}|$ and $Freqk_{true} \subseteq ASP_k$). The fourth and fifth column represent the recall and precision respectively. The last row in the table shows the overall performance when all the k-itemset are merged. Note that unlike Table 3.1, here, the precision is a good indication of the efficiency of LCA when the length of the itemset increases. Since the recall is always 1, the precision is simply equivalent to the ratio of $|Freqk_{true}|$ to $|ASP_k|$.

Figure 3.1 gives a broader picture of the fast diminution of precision when the length of the itemset increases. We reason that this is related to the Apriori property. According to the anti-monotone property, an itemset X must have frequency either smaller than or equal to its subset Y, where $|X| > |Y|$. With increasing length of itemsets, the probability of meeting σ_{min} for the itemsets that meet ϵ decreases. Consider this example: itemset A, AB and ABC have support equal to 1.5%, 0.1% and 0.05% respectively. If σ_{min} is 0.5% and ϵ is 0.05%, all the three itemsets will be easily inserted into D_{struct}. However, only itemset A will meet σ_{min}. As shown in Figure 5.1, generation of excess itemsets particularly at the larger k leads to a smaller

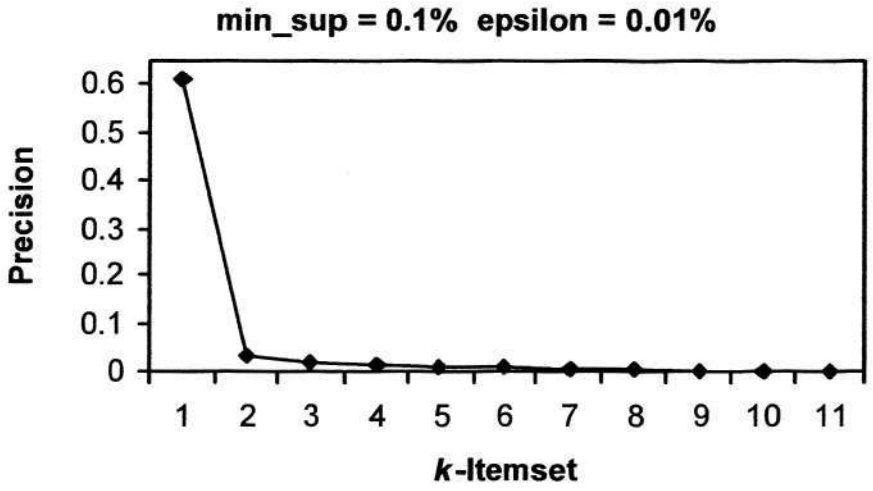


Figure 3.1: Individual precision for breakdown of $|ASP|$ with $\epsilon = 0.01\%$.

precision. This compels one to wonder why would one need to maintain, for instance when $k = 11$, 13458 entries in order to uncover only 2 true frequent itemsets!

3.4 Customized Lossy Counting Algorithm, CLCA

The example problem in the previous section has highlighted the dilemma pertaining to the definition of ϵ. In this section, we propose the customized lossy counting algorithm, denoted as CLCA, to ameliorate the problem faced when we attempt to quantify ϵ.

3.4.1 Variation of ϵ

In many existing literature, ϵ is uniformly fixed for all size of itemsets [MM02, GHP+03, CL03b]. In principle, this may be important if one is interested in uncovering the entire set of frequent patterns regardless the length of the itemsets. However, as shown in the previous section, the longer the pattern we wish to explore, the less efficient LCA becomes. In situations where user is only keen on the itemsets having

a size range from 1 to x and less interested on itemsets having length greater than x, using LCA to uncover all the frequent patterns might not be wise. For practicality, short frequent patterns are usually preferred in many applications. The rationales behind this preference are simple:

(i) short patterns are more understandable and actionable;

(ii) short patterns are more likely to reflect the regularity of data while long patterns are more prone to reflect casual distribution of data [YS04, LHM99b].

In this respect, by focusing on the shorter patterns, one will be able to obtain an overall picture of the domain earlier without being overwhelmed by a large number of detailed patterns. With this understanding, we introduce CLCA to alleviate the drawback of LCA.

The basic intuition of CLCA is as follows: use smaller error bounds to find shorter frequent patterns and larger ones to find longer frequent patterns. We assign different error bounds to different itemsets based on their length. The strategy is simple. Here, a tight error bound can be used at the beginning to explore the itemsets with lengths from 1 to x above which, the bound is progressively relaxed as the length of the itemset increases. By assigning a tight error bound, the precisions for mining short frequent patterns will be improved but the efficiency for uncovering them will be reduced. On the contrary, a relaxed error bound will make the search for long patterns fast, but reduces accuracy. In other words, we migrate the workload for finding long frequent patterns to the front. CLCA is designed to focus on mining short frequent patterns.

Example 3.1 *Imagine a user wants to mine frequent patterns whose frequency is at least 0.1% of the entire stream seen so far. Then* $\sigma_{min} = 0.1\%$. *Due to resource constraint, he focuses his search on itemsets having a size range from 1 to 4* ($x = 4$)

and willing to accept an error less than $0.1\sigma_{min}$. However, he does not wish to give up all the remaining long frequent patterns and is willing to accept them as long as their error are less than $0.5\sigma_{min}$.

The treatment for this example problem is outlined in Algorithm 1. The algorithm is similar to LCA except for the initialization. Instead of assigning a uniform ϵ for all k-itemsets, it offers an alternative, which allows the user to customize the range of error bounds based on his requirement. In CLCA, the error bound of every pattern length is stored in an array *Error* of n elements. In this dissertation, we assume CLCA will mine up to 20-itemsets ($n = 20$). To generate the content of kth element, we require a function that could map any k to a new value between $[\epsilon_1, \epsilon_2]$. This can be easily computed using a step function. Alternatively, we can use the sigmoid function $Sig(k) = \frac{1}{1+e^{s(k-h)}}$, where h is the threshold and s determines the slope of the sigmoid unit. This function is defined as a strictly increasing smooth bounded function satisfying certain concavity and asymptotic properties. In this Chapter, we adopt a sigmoid function in order to demonstrate the flexibility of assigning different error bound to different value of k. We will leave the error analysis for later part. The reason of using a sigmoid function is due to its unique feature of "squashing" the input values into the appropriate range. All small k values are squashed to ϵ_1 and higher ones are squashed to ϵ_2 (line 3). Figure 5.4 illustrates the use of the sigmoid function to compute $Error[k]$. Note that a standard sigmoid function will only output a range of 0-1, therefore we need to normalize it to $[\epsilon_1, \epsilon_2]$.

Like LCA, CLCA processes the data in batches. However, to facilitate the exposition of CLCA, we fixed the batch size instead of the buffer size. B_{size} denotes the number of transactions in a batch and N is updated after every new batch of transactions arrive (line 6). For each itemset of length k, we identify a corresponding cutoff frequency such that $cutoff[k] = Error[k] \times B_{size}$. The cutoff is used to determine whether a k-itemset should be inserted into D_{struct}.

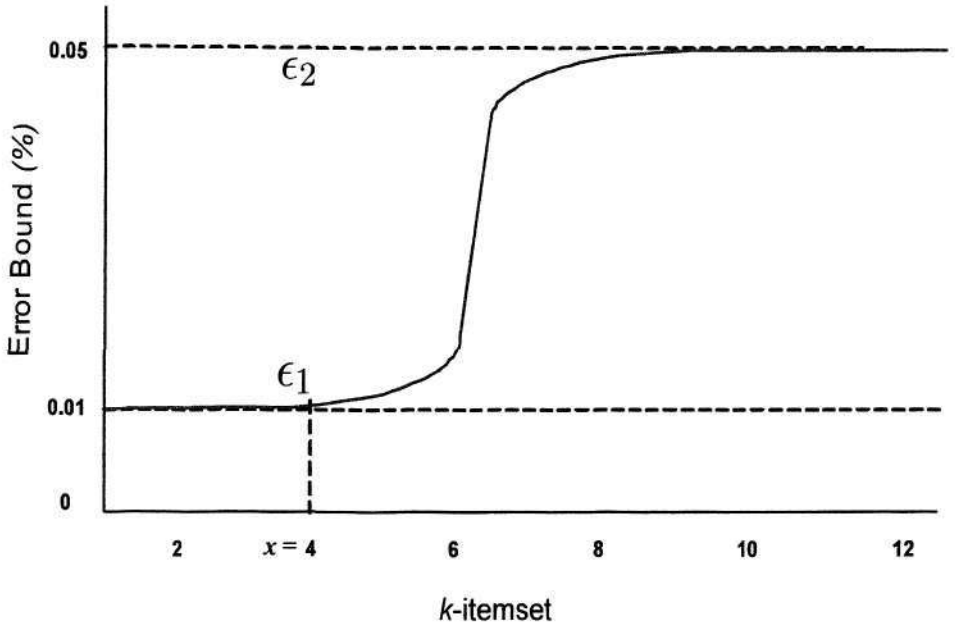


Figure 3.2: Using a sigmoid function to generate the range of error bounds.

When processing a batch, a new set of k-itemsets having frequency $\geq$ *cutoff*$[k]$ will be temporary kept in the pool *NS* (line 8). These are the newly uncovered set of significant patterns that need to be inserted into D_{struct}. D_{struct} contains a set of entries $(X, f_{est}(X), \Delta)$. For insertion, a new entry is created for every itemset in *NS* (line 9). The frequency of the itemsets will be directly transferred to $f_{est}(X)$. $f_{est}(X)$ is the estimated frequency of X. The value of Δ, which is the maximum possible error before insertion, is defined as $Error[k] \times N - cutoff[k]$. For k-itemsets which are already present in D_{struct}, we update by adding their frequency in the current batch to $f_{est}(X)$ (line 10). Deletion will be carried out after D_{struct} has been updated. For any k-itemset in D_{struct}, it will be deleted if $f_{est}(k\text{-itemset}) + \Delta \leq Error[k] \times N$. In order to avoid the combinatorial explosion of the itemsets, we apply the Apriori property so that if an itemset X does not exist in D_{struct}, then all the supersets of X need not be considered. When a user demands for mining result, CLCA outputs all entries, $(X, f_{est}(X), \Delta) \in D_{struct}$ if $f_{est}(X) \geq (\sigma_{min} - Error[|X|]) \times N$.

Algorithm 1 Customized Lossy Counting Algorithm

Input:

$\sigma_{min} \in (0, 1]$; // minimum support threshold
$\epsilon_1 \in (0, \epsilon_2]$;
$\epsilon_2 \in (\epsilon_1, \sigma_{min}]$; // range for the error bound
x; // the length of frequent patterns to focus
n; // the expected longest size of frequent patterns

Output:

On demand, output all frequent patterns after processing N transactions;

1: $N \leftarrow 0$, $D_{struct} \leftarrow \varnothing$, $NS \leftarrow \varnothing$;
2: **for** $k = 1$ to n **do**
3: $Error[k] = Sig(k)$; // the range is scaled to $[\epsilon_1, \epsilon_2]$
4: **end for**
5: **for** every B_{size} incoming transactions **do**
6: $N \leftarrow B_{size} + N$;
7: **for all** k such that $1 \leq k \leq n$ **do**
8: Uncover all k-itemsets with frequency $\geq cutoff[k]$ and store them in NS;
9: Insert NS to D_{struct};
10: Update the frequency of each k-itemsets in D_{struct};
11: **if** $f_{est}(k\text{-itemset}) + \Delta \leq Error[k] \times N$ **then**
12: Eliminate this entry from D_{struct};
13: **end if**
14: $NS \leftarrow \varnothing$;
15: **end for**
16: **if** mining results are requested **then**
17: **return** all entries with $f_{est}(X) \geq (\sigma_{min} - Error[|X|]) \times N$;
18: **end if**
19: **end for**

3.4.2 Error Analysis

We now analyse the reliability of the frequency estimates produced by our proposed algorithm. This section will derive some guarantees for CLCA. We denote the true and estimated frequency of itemset X by $f_{true}(X)$ and $f_{est}(X)$ respectively. We will show that CLCA ensures that (1) all true frequent patterns are output; (2) for any frequent patterns having length equal or smaller than x, the error in the estimate is no more than ϵ_1; and (3) for any frequent patterns having length greater than x, the

error in the estimate is no more than ϵ_2, where $\epsilon_1 \leq \epsilon_2$.

Lemma 3.1 *If an itemset X exists in D_{struct} , then its subset Y, where $|Y| < |X|$ and $Error[|Y|] \leq Error[|X|]$, must first exist in D_{struct}.*

Proof: Let us assume that both itemset X and Y do not exist in D_{struct}. For X to be inserted into D_{struct}, its frequency computed in the current batch must be at least $Error[|X|] \times B_{size}$ (or $cutoff[|X|]$). According to the Apriori property, the frequency of X must be smaller or equal to Y. If X enters D_{struct} and $Error[|Y|] \leq Error[|X|]$, then Y will definitely enter as well. Now assume that both itemsets X and Y have existed in D_{struct} for some time. If Y is eliminated from D_{struct} while processing the current batch, then X will also be eliminated. The reason is the same as before. ■

This Lemma allows one to tailor the range of error bounds according to his requirement. We can assign different error bounds to mine different frequent patterns based on their length. Note that the assignment has to be in ascending order. The Lemma will not hold if $Error[|Y|] > Error[|X|]$. Consider this example: itemset A, AB and ABC have frequency equal to 90, 65 and 21 respectively. If $B_{size} = 200k$, $Error[1] = 0.01\%$, $Error[2] = 0.03\%$ and $Error[3] = 0.05\%$, only itemset ABC will fail to enter D_{struct}. However, if the assignment is not in ascending order such that $Error[1] = 0.05\%$, $Error[2] = 0.03\%$ and $Error[3] = 0.01\%$, itemset ABC and AB will not enter D_{struct} even though they have already passed their cutoff frequency. This is because itemset A cannot make it to D_{struct} and thus no superset will be considered from then. In CLCA, Lemma 3.1 is applied so that a tighter bound (ϵ_1) can be used to explore the k-itemsets where $k \leq x$. The bound is quickly relaxed after x and eventually settle at ϵ_2.

Lemma 3.2 *Whenever an entry* $(X, f_{est}(X), \Delta)$ *is deleted, the true frequency* $f_{true}(X) \leq Error[|X|] \times N$.

Proof: We prove by induction. When the 1^{st} batch of transactions is processed, an entry $(X, f_{est}(X), \Delta)$ will have $\Delta = 0$ and $f_{est}(X) = f_{true}(X)$. Deletion will only happen if $f_{est}(X) = Error[|X|] \times B_{size}$. Since $N = B_{size}$, $f_{true}(X) = Error[|X|] \times N$.

Consider an entry $(X, f_{est}(X), \Delta)$ that gets inserted into D_{struct} at the n^{th} batch. The value of Δ assigned to this entry is the maximum number of times X could have occurred after processing the $(n-1)^{th}$ batch. On the other hand, $f_{est}(X)$ is the exact frequency since it was inserted into D_{struct}. Therefore, we can be sure that $f_{true}(X) \leq f_{est}(X) + \Delta$. Combined with the deletion rule that $f_{est}(X) + \Delta \leq Error[|X|] \times N$, we obtain $f_{true}(X) \leq Error[|X|] \times N$. ■

Lemma 3.3 *The true frequency of any itemset* X *in* D_{struct} *is bounded as follows:* $f_{est}(X) \leq f_{true}(X) \leq f_{est}(X) + Error[|X|] \times N$.

Proof: We know that $f_{est}(X)$ is the frequency since X was inserted into D_{struct}. Therefore, $f_{est}(X)$ can never exceed $f_{true}(X)$. Hence the lower limit of the inequalities is correct. Next, $f_{est}(X)$ can only be equal to $f_{true}(X)$ provided that X is inserted in the 1^{st} batch and has never been deleted from D_{struct}. Under this condition, the upper limit of the inequalities will not be more than $f_{true}(X)$ by $Error[|X|] \times N$. From Lemma 3.2, when an entry is inserted into D_{struct}, $f_{true}(X) \leq f_{est}(X) + \Delta$. The value of Δ is the maximum possible error of f_{est} before insertion. Since $\Delta \leq Error[|X|] \times N$, $f_{true}(X) \leq f_{est}(X) + Error[|X|] \times N$. ■

From Lemma 3.2 and 3.3, we know that the most an itemset X can be underestimated is $Error[|X|] \times N$. If the true support of X is σ_{min} (this is the minimum support

for it to be considered frequent), then $f_{true}(X)$ is $N\sigma_{min}$. As such, the estimated frequency, f_{est}, in D_{struct} should be at least $(N\sigma_{min} - NError[|X|])$. Therefore, if we output all the itemsets in D_{struct} having a f_{est} value of at least $(\sigma_{min} - Error[|X|]) \times N$, then all of the frequent itemset will be output. A 100% recall is thus guaranteed.

In most FPM algorithms that adopt approximate counting, the Apriori property is needed to prune infrequent patterns. The Apriori property suggests to use the possible smallest $|X|$th frequent itemsets to generate the $(|X|+1)$th candidate itemsets, and then mine the $(|X|+1)$th candidate itemsets. Here, LCA allows 1-itemsets with support below σ_{min} but above $\sigma_{min} - \epsilon$ to be counted as frequent. Consequently, when there are some false 1-itemsets in $[\sigma_{min} - \epsilon, \sigma_{min}]$, the nature of the exponential explosion causes the number of potential frequent itemsets to be very large and makes it very difficult to be managed by approximate counting oriented approach. Moreover, the lack of domain knowledge on the data set that we are handling further exacerbates the mining task. From an implementation perspective, one can consider using the property of Lemma 4.1 to handle unknown data set. Initially, we can assign a tight error bound uniformly for all $Error[k]$. Since LCA performs level wise mining, we can design a tuning mechanism such that whenever the size of the current $|X|$th sub-frequent patterns exceeds our predefined limit, we can relaxed the error bound in $Error[|X|+1]$ accordingly. Relaxing the error bound means increasing $Error[|X|+1]$, or in other words, reducing the amount of sub-frequent patterns entering D_{struct} thus improving the speed and memory. However, note that once an error bound is relaxed, it can never be tightened as the past information is lost.

3.4.3 Data Structure

For approximate counting, an efficient method of storing a large collection of itemsets and their counts is definitely needed. Like LCA, CLCA adopts the same data structure, D_{struct}. Conceptually, it is a forest of prefix trees consisting of labelled nodes. For example, assuming the first batch of transactions is obtained from Table 2.1 (Chapter 2), with σ_{min} equal to 0.5 and a uniform error bound, ϵ equal to $0.5\sigma_{min}$, we would generate a structure that is similar to Figure 3.3. The root nodes have level 0. The level of any other node is one more than that of its parent. The children of any node are ordered by their item-ids. The nodes in the prefix tree are ordered according to the lexicographic order of the items stored in the node. Each nodes in the prefix tree corresponds to an itemset (from root to node). Note that in this way we need only one item to distinguish between the item sets represented in one node, which is relevant for the implementation of CLCA. For example, in Figure 3.3 the nodes represented by solid boxes are frequent patterns, where those as dotted boxes are Sub-frequent patterns.

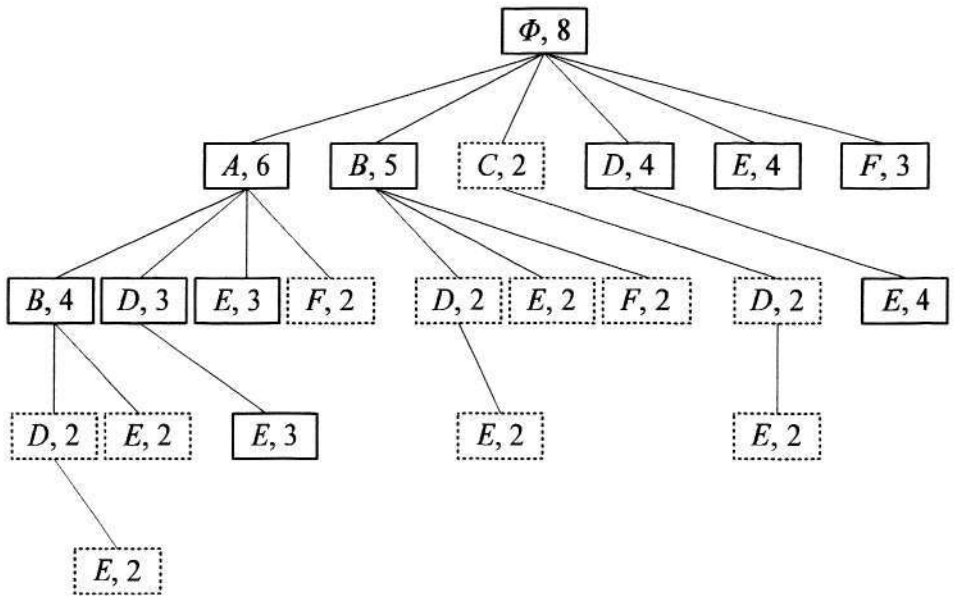


Figure 3.3: A prefix tree representation of the Table 2.1

3.5 Experimental Evaluation

This section describes the experiments conducted in order to determine the effectiveness of CLCA. All experiments were performed on a 1.7GHz CPU Dell PC with 1GB of main memory running on the Windows XP platform. The algorithm was written in C++.

Data Sets: We used synthetic data sets in our experiments and they were generated using the code from the IBM QUEST project [AS94]. The nomenclature of these data sets is of the form $TxxIyyDzzK$, where xx refers to the average number of items present per transaction, yy refers to the average length of the maximal potentially frequent itemsets and zz refers to the total number of transactions in $K(1000\text{'s})$. Items were drawn from a universe of $\mathcal{I} = 10k$ unique items. The two data sets that we used are $T9I3D2000K$ and $T15I7D2000K$. Note that $T15I7D2000K$ is a much denser data set than $T9I3D2000K$ and therefore requires more time to process.

We fixed the minimum support threshold $\sigma_{min} = 0.1\%$ and the batch size $B_{size} = 200K$. For comparison with LCA, we set $\epsilon = 0.1\sigma_{min}$ which is a popular choice for LCA. We assume CLCA will mine up to the length of 20-itemset. Two sets of error bounds, $Error_1$ and $Error_2$, were evaluated. For $Error_1$, we let $x = 2$, $\epsilon_1 = 0.03\sigma_{min}$, $\epsilon_2 = 0.5\sigma_{min}$ and the computed array is $Error_1[20] = \{0.03\sigma_{min}, 0.03\sigma_{min}, 0.05\sigma_{min}, 0.265\sigma_{min}, 0.48\sigma_{min}, 0.5\sigma_{min}, ..., 0.5\sigma_{min}\}$. As an illustration, the distribution of $Error_1$ follows a sigmoid function. 1-itemset and 2-itemset will have the same error bound (ϵ_1). After the 2-itemset, we relax the bound until the function reaches ϵ_2. Similarly for $Error_2$, we let $x = 3$, $\epsilon_1 = 0.05\sigma_{min}$, $\epsilon_2 = 0.4\sigma_{min}$ and the computed array is $Error_2[20] = \{0.05\sigma_{min}, 0.05\sigma_{min}, 0.05\sigma_{min}, 0.07\sigma_{min}, 0.225\sigma_{min}, 0.38\sigma_{min}, 0.4\sigma_{min}, ..., 0.4\sigma_{min}\}$. Tables 3.3 and 3.4 show the experiment results for the two data sets after processing two million transactions. We observe that the time spent by CLCA on both data sets is much less than LCA. This can be explained by considering two factors:

1) The value of ϵ_2 is much larger than ϵ. A majority of the error bounds in $Error_1$ and $Error_2$ are close or equal to ϵ_2. Higher error bounds will lead to fewer itemsets getting into D_{struct}.

2) The length of x is small when compared to the average length of the maximal potentially frequent itemsets of the two data sets. Thus the high computation due to ϵ_1 can be compromised by setting a larger ϵ_2.

Table 3.3: Experiment results for $T9I3D2000K$.

	$\|ASP\|$	$\|AFI_{appro}\|$	$\|AFI_{true}\|$	Recall	Precision	Time(s)
LCA	233913	12377	9418	1.00	0.761	149
CLCA $Error_1$	149577	17092	9418	1.00	0.551	96
CLCA $Error_2$	156378	13647	9418	1.00	0.751	112

Table 3.4: Experiment results for $T15I7D2000K$.

	$\|ASP\|$	$\|AFI_{appro}\|$	$\|AFI_{true}\|$	Recall	Precision	Time(s)
LCA	1821961	54499	26136	1.00	0.480	1560
CLCA $Error_1$	1123565	207297	26136	1.00	0.126	1096
CLCA $Error_2$	1400217	134221	26136	1.00	0.195	1230

As expected, the recall for CLCA and LCA is 1. This shows that CLCA indeed guarantees that all true frequent patterns are output. However, the precisions generated by $Error_1$ and $Error_2$ are weaker than LCA. Note that these represent the entire set of frequent patterns and thus they only reveal one side of the story. The main

Table 3.5: Precision values for all frequent patterns with length ranging from 1 to x.

	$1-x$	$T9I3D2000K$	$T15I7D2000K$
LCA	1-2	0.791	0.696
CLCA $Error_1$	1-2	0.933	0.876
LCA	1-3	0.774	0.601
CLCA $Error_2$	1-3	0.857	0.765

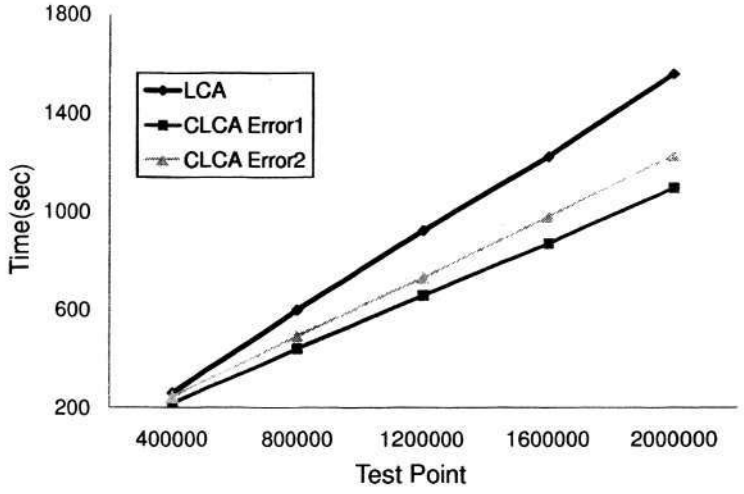


Figure 3.4: Execution time on $T15I7D2000K$.

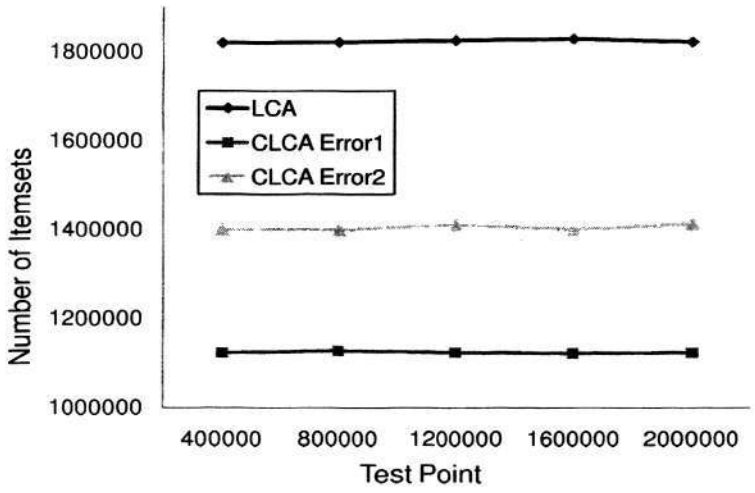


Figure 3.5: Number of itemsets to be maintained in D_{struct} for $T15I7D2000K$.

task of CLCA is to focus on mining short frequent patterns. Table 3.5 shows the real purpose of CLCA. Here, the precisions for mining all the frequent patterns with length ranging from 1 to x are recorded. Clearly, CLCA performs better here. In addition, Figure 3.4 shows the execution time on $T15I7D2000K$ where CLCA performs faster than LCA at every test points . As we can see, the cumulative execution time of CLCA and LCA grows linearly with the number of transactions processed in the streams. Figure 3.5 further demonstrates the efficiency of CLCA. It shows the comparison in terms of number of itemsets to be maintained in D_{struct}. The larger the number of itemsets, the more memory we require. Interestingly, for CLCA and

LCA, the memory consumption remains stable throughout the lifetime of processing the stream. This may suggest that the data distribution in $T9I3D2000K$ and $T15I7D2000K$ is stable. However, a data stream necessarily has a temporal dimension, and the patterns embedded in it are likely to evolve as time goes by. In Chapter 5, we will address the problem of time changing environment.

3.6 Summary

Frequent pattern mining is an important problem in applications such as data mining and computer network monitoring. In this chapter, we study LCA which generates frequent itemsets with supports accurate to within a user-specified error bound. We argue that the proper definition of ϵ is non trivial. We propose CLCA to address the problem of quantifying ϵ efficiently. We place our emphasis on mining short frequent patterns in data streams. Instead of using fixed ϵ, we use a set of error bounds. The described algorithm is simple. Our experiments show that CLCA works better than LCA if one is more concerned with mining short frequent patterns [ND08a]. In addition, this technique can be applied to several recent algorithms that adopt approximate counting approach (see Table 2.2). This research raises two open questions. The first one concerns the design of $Error[k]$: is there an "optimum" set of error bounds? The other one is, can we vary the error bounds for long patterns as well? We leave a further investigation on these two questions for future research.

Chapter 4

Data Reduction Method For Transactional Data Streams

From the previous chapter, we note that serveral existing research efforts often rely on a two-phase framework to discover frequent patterns: (1) using internal data structures to store potential frequent patterns obtained by scanning the stream data; and (2) re-mining the potentially frequent patterns to finalize and output frequent patterns. The defectiveness of such a two-phase framework lies in the fact that there is no guarantee in complexity. Resources such as memory space, CPU, and sometimes energy, are very precious in a stream mining environment. They are very likely to be exhausted when processing data streams which arrive with rapid speed and high volume. If we never monitor the availability of resources, for example the main memory, when processing the mining algorithm, data will be lost when the memory is used up. This would lead to the inaccuracy of the mining results, thus degrading the performance of the mining algorithm. However, often the result of the mining operation itself is so large that even enumerating all frequent patterns is impossible. This blow-up happens for example when we set the frequency threshold too low, or when the data is heavily correlated. In the worst case, the number of itemsets can even be exponential in the number of items. Clearly, even the most efficient algorithms cannot enumerate such huge number of itemsets. Worse still, when the computer in use is handling other tasks and may be running many other applications in parallel with the mining algorithm.

Instead of using data stream mining algorithm which is resource intensive, we consider sampling as another indirect mining approach in this chapter. Sampling methods are among the simplest methods for synopses construction in data streams [JMR05]. It is also relatively easy to use these synopsis with a wide variety of applications because their representations are not specialized and uses the same multi-dimensional representations as the original data points. In particular reservoir based sampling methods are very useful for data streams.

In this chapter, a sampling algorithm called *DSS* (Distance based Sampling for Streaming data) is proposed for streaming data [DN06]. The main contributions of this proposed method are: *DSS* elegantly addresses both the issues of very large size of the data or in other words, very small sampling ratio, and noise simultaneously. Experiments in FPM are conducted, and in all of them the results show that *DSS* outperformed simple random sampling*SRS* at the expense of a slight amount of processing time. The trade-off analysis between accuracy and time shows that it is worthwhile to invest in *DSS* rather than other approaches including *SRS* particularly when the domain is very large and noisy. Experiments are done mainly using synthetic data sets from IBM QUEST project and benchmark data set, Kosarak. Experiments are conducted to compare the performance of *DSS* with Algo-Z and LCA.

4.1 Distance Based Sampling

SRS, or its counterpart reservoir sampling in streaming data, is known to suffer from a few limitations: First, the sample generated by SRS may not adequately represent the entire data set due to random fluctuations in the sampling process. This difficulty is particularly apparent at small sampling ratios which is the case for very large databases with limited memory. Second, SRS is blind towards noisy data objects, i.e., it treats both *bona fide* and noisy data objects similarly. The proportion of noise in the SRS sample and the original data set are almost equal. Therefore, in the presence of noise, the performance of SRS degrades.

Recent work in the field of approximate aggregation processing suggests that the true benefits of sampling might be well achieved when the sampling technique is tailored to the specific problem at hand [AGP00]. In this respect, we propose a distance based sampling that is designed to work "count" data set, that is, data set in which there is a base set of "items" and each data element (or transaction) is a vector of items. Note that in this dissertation, a sample refers to a set of transactions (see Section 2.1 for the format of our transactional database). Usually, for distance based sampling, the strategy is to produce a sample whose "distance" from the complete database is minimal. The main challenge is to find an appropriate distance function that can accurately capture the difference between the sample (S) and all the transactions seen so far (Ds) in the stream.

For distance based sampling, the basic intuition is that if the distance between the relative frequency of the items in S and the corresponding relative frequency in Ds is small, then S is a good representative of Ds. Ideally, one needs to compare the frequency histograms of all possible itemsets: 1-itemsets, 2-itemsets, 3-itemsets, and so on. But experiments suggest that often it is enough to compare the 1-itemset histograms [DN06, BCD+03]. The proposed method, although based on 1-itemset histograms, can be easily extended to histograms for higher number of itemsets by simply adding them in constructing the histogram. On the flip side, histograms for higher itemsets are computationally very expensive. Our experiments (see Section 4.4) show that it becomes impractical to include even 2-itemsets.

We define a few distance functions which are all based on the frequency of each item. Here, the relative frequency of item $\mathcal{A}$ in S and Ds is given by $Sup(A;S) = \frac{freq(A;S)}{|S|}$ and $Sup(A;Ds) = \frac{freq(A;Ds)}{|Ds|}$, respectively. $freq(A;U)$ is the frequency of $\mathcal{A}$ in a set of transactions U. We note that several definitions of distance are possible, for example

$$Dist_1(S, Ds) = \sum_{A \in \mathcal{I}} |Sup(\mathcal{A}; S) - Sup(\mathcal{A}; Ds)| \quad \text{(Eq. 4.1)}$$

$$Dist_2(S, Ds) = \sum_{\mathcal{A} \in \mathcal{I}} (Sup(\mathcal{A}; S) - Sup(\mathcal{A}; Ds))^2, \quad \text{(Eq. 4.2)}$$

and

$$Dist_\infty(S, Ds) = \max_{\mathcal{A} \in \mathcal{I}} |Sup(\mathcal{A}; S) - Sup(\mathcal{A}; Ds)|. \quad \text{(Eq. 4.3)}$$

Observe that $Dist_1, Dist_2, Dist_\infty$ correspond to L_P-norm distances with $p = 1, 2$ and ∞, respectively. The sum of squared distances variant ($Dist_2$) is the most common way to measure the distance between vectors [CHS02]. Since we are aiming to minimize the distance between the item frequencies in the data stream and the sample, using the $Dist_2$ (also known as Euclidean distance) is the natural way to measure the combined distance for all items. In this dissertation, we shall use $Dist_2$ to measue the distant between S and Ds where S is the working sample (to which the new transactions are being added or removed from), and Ds is the reference set of transactions against which the accuracy of the working sample is compared.

4.1.1 Distance Based Sampling for Streaming Data (DSS)

Like any other reservoir sampling method designed for data streams, the initial step of *DSS* is to insert the first n transactions into a "reservoir". The rest of the transactions are processed sequentially; transactions can be selected for the reservoir only as they are processed. Because n is fixed, whenever there is an insertion of transaction, there is sure to be a deletion. In *DSS*, a local histogram ($Hist_\mathcal{L}$) and a global histogram ($Hist_\mathcal{G}$) are employed to keep track of the frequency of items generated by S and Ds respectively. Ideally, for a sample to be a good representation of the entire data,

the discrepancy between $Hist_{\mathcal{L}}$ and $Hist_{\mathcal{G}}$ should be small. In other words, both the structure of $Hist_{\mathcal{L}}$ and $Hist_{\mathcal{G}}$ should look similar. Any insertion or deletion of transaction on the sample will affect the shape of $Hist_{\mathcal{L}}$.

Table 4.1: Conceptual Ranking of Transactions

Rank	Distance	*Tid*
1^{st}	25.00	10
2^{nd}	15.00	20
:	:	:
n^{th}	2.00	3

To maintain the sample, *DSS* prevents an incoming transaction from entering S if its existence in S increases the discrepancy. In addition, *DSS* helps to improve the quality of the sample by deleting transaction whose elimination from S maximally reduces (or minimally increases) the discrepancy. Therefore, there is a ranking mechanism in *DSS* to rank the transactions in S so that the "weakest" transaction can be replaced by the incoming transaction if the incoming transaction is better. The transactions in the initial sample are ranked by "leave-it-out" principle, i.e., distance is calculated by leaving out the transaction. Higher ranks are assigned to the transactions removal of which leads to higher distance. Mathematically, the distance is calculated as follows for ranking:

$$Dist_t = Dist_2\left(S - \{t\}, Ds\right) \qquad \text{(Eq. 4.4)}$$

where $t \in S$. For the initial ranking, both S and Ds contain the first n transactions. Table 4.1 shows the conceptual idea of ranking. *Tid* 10 is ranked highest because its removal produces the maximum distance of 25.0. Similarly, *Tid* 3 is ranked lowest because its removal produces the minimum distance of 2.0. Let *LRT* denotes the lowest ranked transaction.

When a new transaction t_{new} arrives, $Hist_{\mathcal{G}}$ is immediately updated. A decision is made whether to keep it in the sample by comparing the distances computed when $Hist_{\mathcal{L}}$ is 'with' and 'without' t_{new}. First, we use Eq. 4.5 to calculate the distance between $Hist_{\mathcal{G}}$ and $Hist_{\mathcal{L}}$ when t_{new} is absent. Next, LRT is temporarily removed from the sample. We use Eq. 4.6 to calculate the distance between $Hist_{\mathcal{G}}$ and $Hist_{\mathcal{L}}$ when t_{new} is present. Note that because of the incoming t_{new}, the Ds in Eq. 4.5 and Eq. 4.6 is updated ($Ds = Ds + t_{new}$). If $Dist_{without\_t_{new}} > Dist_{with\_t_{new}}$, then t_{new} is selected to replace LRT in the current sample. LRT is permanently removed. t_{new} will be ranked in the sample using $Dist_{without\_t}$ value. On the other hand, if $Dist_{without\_t_{new}} \leq Dist_{with\_t_{new}}$, then t_{new} is rejected and LRT is retained in the reservoir.

$$Dist_{without\_t_{new}} = Dist_2(S, Ds) \qquad \text{(Eq. 4.5)}$$

$$Dist_{with\_t_{new}} = Dist_2((S - LRT + t_{new}), Ds) \qquad \text{(Eq. 4.6)}$$

Ideally, all transactions in the current sample should be re-ranked after each incoming transaction is processed because $Hist_{\mathcal{G}}$ is modified. Re-ranking is done by recalculating the distances using 'leave-it-out' principle for all transactions in the current sample and those which are rejected already. But this can be computationally expensive because of the nature of data stream. We need a trade-off between accuracy and speed. Thus, re-ranking is done after selecting $\mathcal{R}$ new transactions in the sample. We do not consider the rejected transactions while counting $\mathcal{R}$. A good choice of $\mathcal{R}$ is 10 (see Section 4.4.3). *DSS* tries to ensure that the relative frequency of every item in the sample is as close as possible to that of the original data stream. In the implementation, *DSS* stores the sample in an array of pointers to structures which

hold the transactions and their distance values. Initial ranking and re-ranking of the sample according to the distances involves two steps. The first step is to calculate the distances of the transactions in the sample and the second step is to sort the sample by increasing distances using the standard quick sort technique. When $Dist_2$ is well implemented, the distance based sampling can have computational cost at most $O(|t_{max}|)$, where $|t_{max}|$ is the maximal length of the transaction vector (see Section 4.2). The complete *DSS* algorithm is summarized as follows:

(i) Insert the first n transactions into S.

(ii) Initialize $Hist_{\mathcal{G}}$ and $Hist_{\mathcal{L}}$ to keep track of the number of transactions containing each item $\mathcal{A}$ in S and Ds.

(iii) Rank the initial S by 'leave-it-out' method using Eq. 4.4.

(iv) Read the next incoming t_{new}.

(v) Include t_{new} into $Hist_{\mathcal{G}}$.

(vi) Compare the distances of S 'without' and 'with' t_{new} using Eq. 4.5 and Eq. 4.6.

a. If $Dist_{without\_t_{new}} > Dist_{with\_t_{new}}$, then replace LRT with t_{new} and update $Hist_{\mathcal{L}}$.

b. Else, reject t_{new}.

(vii) If $\mathcal{R}$ new transactions are already selected, re-rank the transactions in the current sample. Eg. $\mathcal{R} = 10$.

(viii) Repeat steps (iv) to (vii) for every subsequent incoming transaction t_{new}.

4.1.2 Implementation

In our implementation, the *DSS* stores the sample in an array of pointers to structures which holds the transaction and its distance value. Initial ranking and re-ranking of the sample according to the distances involve two steps. The first step is to calculate the distances of the transactions in the sample and the second step is to sort the sample by increasing distances. Quick sort is used and it is a very efficient sorting algorithm. It consists of a partition phase and a sorting phase.

As for insertion, the selected transaction t_{new} replaces the lowest ranked transaction in the current sample by inserting t_{new} into the sample using binary insertion algorithm. Binary insertion is used for ordered array and it uses the information of the order of the transactions. Binary insertion consists of two phases and they are binary search and insertion. If t_{new} is selected to be filled in the deleted position, its distance is computed and it is placed in the right position in the ranked order among the other transactions in the sample. Let us assume that the replacement for the deleted transaction is actually not worthy to be selected based on its distance. If so, still the damage done to the quality of the sample is quickly restored. The future incoming transactions will be tested against this not-so-representative transaction and it will be replaced by any transaction that represents the data set better.

4.2 Complexity Analysis

When finding LRT, we need to compute $Dist_2$ for each $t \in S$. This is the distance between $Hist_{\mathcal{G}}$ and $Hist_{\mathcal{L}}$ when a transaction t is removed from S. Here, S_t denotes $S - \{t\}$. We let $\mathcal{F}_{\mathcal{A}S}$, $\mathcal{F}_{\mathcal{A}S_t}$ and $\mathcal{F}_{\mathcal{A}Ds}$ represent the absolute frequency of item $\mathcal{A}$ in S, S_t and Ds respectively. Note that,

$$\mathcal{F}_{\mathcal{A}S_t} = \begin{cases} \mathcal{F}_{\mathcal{A}S} - 1 & \text{if } \mathcal{A} \in t \\ \mathcal{F}_{\mathcal{A}S} & \text{else } \mathcal{A} \notin t. \end{cases}$$

To search for the weakest transaction in S, the set of n distances that are generated from Eq. 4.4 has to be compared with one another such that

$$t^* = \underset{1 \leq t \leq n}{\operatorname{argmin}} \; Dist_2(S - t, Ds) = \underset{1 \leq t \leq n}{\operatorname{argmin}} \; \sum_{\mathcal{A} \in \mathcal{I}} \left(\frac{\mathcal{F}_{\mathcal{A}S_t}}{|S_t|} - \frac{\mathcal{F}_{\mathcal{A}Ds}}{|Ds|} \right)^2. \qquad \text{(Eq. 4.7)}$$

Unfortunately, the determination of t^* can be computationally costly. The worst case time complexity is $O(n.|\mathcal{I}|)$, where $|\mathcal{I}| \gg 1$. However, we note that even though removing a transaction from S will affect relative frequencies of all items, most absolute frequencies will remain unchanged. Only those items contained within the transaction t are affected. In addition, we can make use of the fact that, in general,

$$\underset{x \in U}{\operatorname{argmin}} \, f(x) = \underset{x \in U}{\operatorname{argmin}} \, cf(x) + d. \qquad \text{(Eq. 4.8)}$$

For any constant c and real number d that we introduced, the final outcome will still remain the same. With this understanding, we can rewrite Eq. 4.7 as

$$\begin{aligned} t^* &= \underset{1 \leq t \leq n}{\operatorname{argmin}} \; \sum_{\mathcal{A} \in \mathcal{I}} \left(\left(\mathcal{F}_{\mathcal{A}S_t} - \frac{\mathcal{F}_{\mathcal{A}Ds}}{|Ds|} |S_t| \right)^2 - \left(\mathcal{F}_{\mathcal{A}S} - \frac{\mathcal{F}_{\mathcal{A}Ds}}{|Ds|} |S_t| \right)^2 \right) \\ t^* &= \underset{1 \leq t \leq n}{\operatorname{argmin}} \; \sum_{\mathcal{A} \in t} \left(1 - 2\mathcal{F}_{\mathcal{A}S} + 2\frac{\mathcal{F}_{\mathcal{A}Ds}}{|Ds|} |S_t| \right) \end{aligned} \qquad \text{(Eq. 4.9)}$$

Clearly, from the above representation of t^*, it is possible to reduce the worst-case cost of ranking from $O(n.|\mathcal{I}|)$ to $O(n.|T_{max}|)$. Similarly, for comparing between LRT and t_{new}, we can apply the same strategy. The cost will then be $O(|T_{max}|)$.

4.3 Handling Noise in *DSS*

Noise here means a random error or variance in a measured variable [HK06a]. The data object that holds such noise is a noisy data object. Such noisy transactions have very little or no similarity with other transactions. This "aloof" nature of noisy transactions is used to detect and remove them. Noise detection and removal has been studied abundantly, for example in [KM03, ZWKS07]. These methods typically employ a similarity measurement and a similarity threshold to determine whether an object is noise, that is if a data object has lesser similarity than the threshold, it is considered noise.

In this chapter the focus is to maintain a small sample of transactions over a streaming data. *DSS* not only maintains a good representative sample, it also removes noise by not selecting them in the sample. *DSS* banks on the *aloofness* of the transactions to determine whether an incoming transaction is corrupted by noise. Removal of a noisy transaction will induce smaller distance than removal of a *bona fide* transaction. As *DSS* selects transactions into the reservoir sample by the distance it induces, it naturally rejects the noisy transactions. Later experimental results show that *DSS* has a better ability to handle noise than LCA and Algo-Z.

4.4 Experimental Evaluation

This section describes the experimental comparison between *DSS* and LCA in the context of frequent pattern mining. In addition, we also compared *DSS* with simple random sampling (SRS) using Algo-Z. Even though random sampling performs poorly with respect to the established metrics, it is a good reference for comparing the performance of our deterministic sampling algorithm. All experiments were performed on a 1.7GHz CPU Dell PC with 1GB of main memory running on the Windows XP

platform. All the algorithms were written in C++. For efficient implementation, LCA uses the Trie data structure that is described in [MM02].

We used the synthetic data sets in our experiments and they were generated using the code from the IBM QUEST project [AS94]. The two data sets that we used are $T10I3D2000K$ and $T15I7D2000K$ (see Section 3.5 for the nomenclature of these two data stes). Note that $T15I7D2000K$ is a much denser data set than $T10I3D2000K$ and therefore requires more time to process. In addition, to verify the performance of all the three algorithms on real world data sets, we used a coded log of a clickstream data (denoted by Kosarak) from a Hungarian on-line news portal [Bod03]. This database contains 990002 transactions with average size 8.1. We fixed the minimum support threshold $\sigma_{min} = 0.1\%$ and the batch size $B_{size} = 200K$. For comparison with LCA, we set $\epsilon = 0.1\sigma_{min}$ which is a popular choice for LCA. To measure the accuracy of Algo-Z, *DSS* and LCA, we can apply the symmetric difference (SD) metric in Eq. 4.10. The Apriori algorithm was used to generate all the true frequent patterns (AFI_{true}) from the data sets. Given a set AFP_{true} and a set AFP_{appro} of frequent itemsets obtained in the output by the algorithms,

$$SD = \frac{|(AFI_{true} - AFI_{appro}) \cup (AFI_{appro} - AFI_{true})|}{|AFI_{true}| + |AFI_{appro}|}. \qquad \text{(Eq. 4.10)}$$

Note that the above expression is nothing more than the ratio of the sum of missed and false itemsets to the total number of AFI [BCD+03, LCK98]. The SD is zero when $AFI_{true} = AFI_{appro}$ and the SD is 1 when AFI_{true} and AFI_{appro} are disjoint. The greater the value of SD, the greater is the dissimilarity between AFI_{true} and AFI_{appro} (and vice versa). Alternately, we can define the overall accuracy with

$$Acc = 1 - SD. \qquad \text{(Eq. 4.11)}$$

4.4.1 Time Measurements

We report the experimental results for the three data sets described previously. Figure 4.1 depicts the number of frequent patterns uncovered during the operation of the three algorithms for the three different data sets. The set of frequent patterns are broken down according to the pattern length. Note that for *DSS* and Algo-Z, the size of the reservoir is 5k transactions. In the figure, ORG indicates the true size of frequent itemsets generated when the Apriori algorithm operates on the entire data set. Interestingly, we can observe that the general trends of the sampled data sets resemble the true result. This is also similar with the output from LCA. However, Algo-Z tends to drift very far away from ORG indicating more false positives are generated. LCA is caught in between the two sampling algorithms. *DSS* gives the most appealing results in terms of fidelity. The graphs from *DSS* are the closest to ORG.

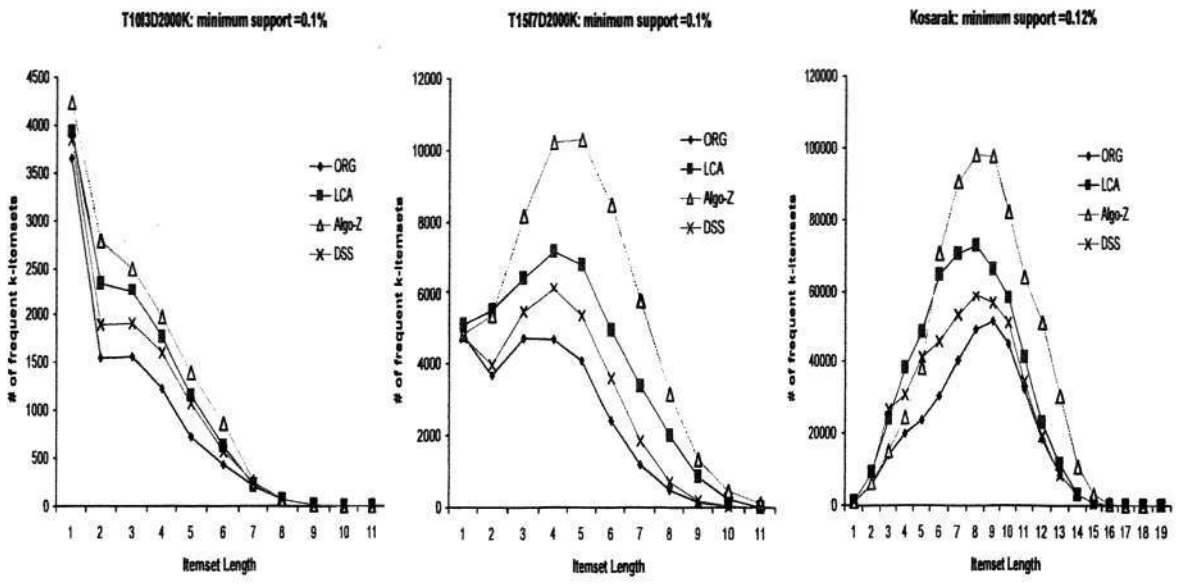


Figure 4.1: Itemset size vs Number of Frequent Itemsets.

Figure 4.2 shows the execution time on the three data sets. The results of the algorithms are computed as an average of 20 runs. Each run corresponds to a different

shuffle of the input data set. Note that for *DSS* and Algo-Z, the execution time consists of the time spent for obtaining the sample as well as using the Apriori algorithm to generate the frequent patterns (AFI_{appro}) from the sample. As we can see, the cumulative execution times of *DSS*, Algo-Z and LCA grow linearly with the number of transactions processed in the streams. In particular, Algo-Z is the fastest because it only needs to maintain a sample by randomly selecting transactions to be deleted or inserted in the reservoir. There is no real processing work to be done on any incoming transaction. Unlike Algo-Z, *DSS* processes every incoming transaction by computing the distance it may cause if included in the sample. As a result, its processing speed is slower than Algo-Z. However, the slowest algorithm is LCA. In all three data sets, LCA spent the most amount of time.

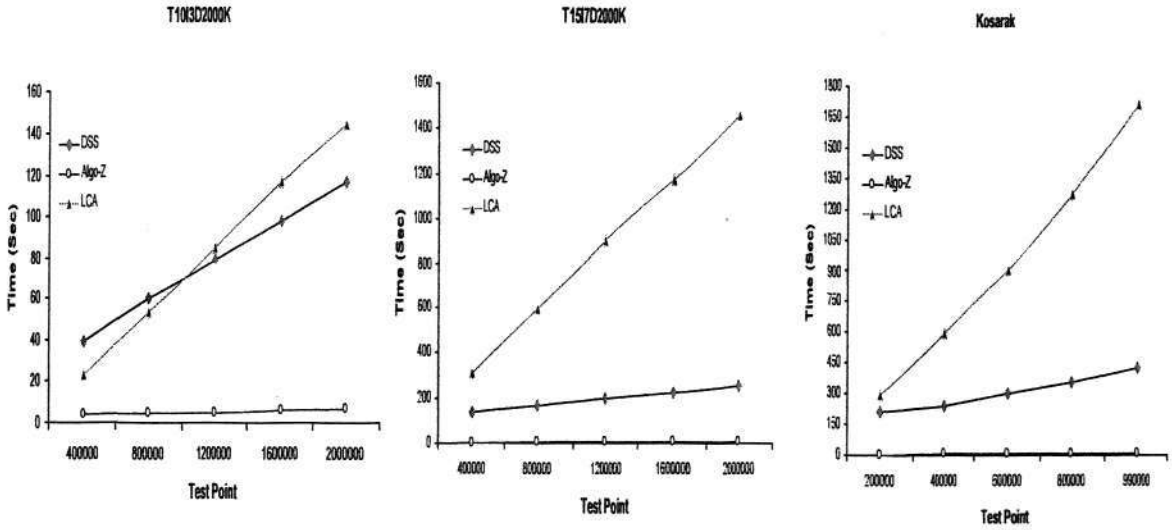


Figure 4.2: Execution time on $T10I3D2000K$, $T15I7D2000K$ and Kosarak.

4.4.2 Accuracy Measurements

Figure 4.3 displays the accuracies of the three algorithms against the number of transactions processed. Here, *Acc* is computed using Eq. 4.11. As expected, Algo-Z achieved the worst performance. As the number of transactions to be processed increases, we see that its accuracy drops significantly. Since the reservoir is a fixed size,

the sampling ratio decreases at every test point. From the graph, *DSS* achieved good accuracy even for small sampling ratios. Its accuracy remains almost stable for all the test points. Although LCA guarantees 100% recall, its performance was heavily affected by its poor precision. This can be clearly seen in the figure. From Figure 4.3, it is clear that the accuracy of LCA was dragged down due to its low precision value. For all the three data sets, its performance is lower than *DSS*.

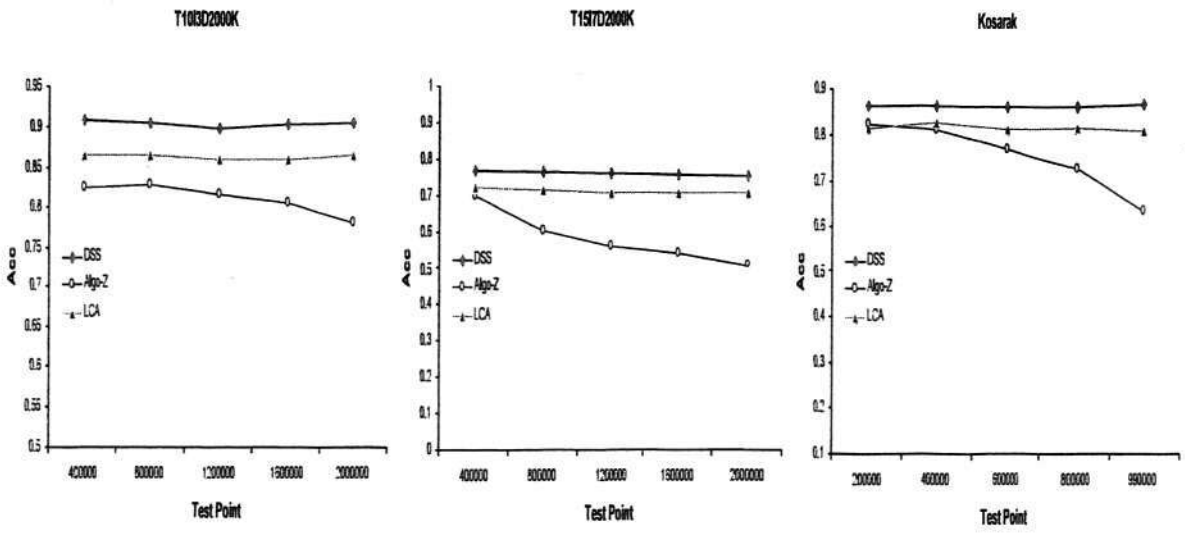
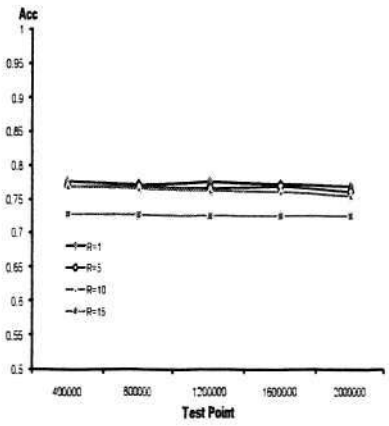


Figure 4.3: Accuracy on $T10I3D2000K$, $T15I7D2000K$ and Kosarak.

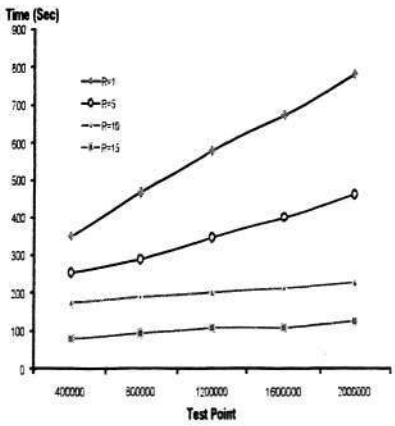
4.4.3 Varying $\mathcal{R}$

When running *DSS*, there is always a need to re-calculate the distance of the transactions in the sample because the global histogram will be slightly modified whenever a new transaction arrives. Unfortunately, to re-rank every transaction in the sample for every incoming transaction will be very costly. To cope with this problem, the parameter, $\mathcal{R}$, is introduced. The idea is very simple. If $\mathcal{R}$ new transactions have been inserted into the sample, we will activate the ranking mechanism. $\mathcal{R}$ is the only parameter in *DSS*. Figure 4.4 illustrates the impact of varying the value of $\mathcal{R}$ in *DSS*. We provide four variations (1, 5, 10 and 15) of $\mathcal{R}$ for the data set $T15I7D2000K$.

From the Figure 4.4a, we see an obvious gap for $\mathcal{R} = 10$ and $\mathcal{R} = 15$. However, the different in accuracy for $\mathcal{R} = 1$, $\mathcal{R} = 5$ and $\mathcal{R} = 10$ is not so apparent. In addition, from the Figure 4.4b, as $\mathcal{R}$ gets smaller, the execution time increases. This suggests that the value 10 is a suitable default value for $\mathcal{R}$. Unlike LCA which has a drastic effect either on its speed or memory if ϵ is wrongly set, $\mathcal{R}$ is less sensitive.



4.4.a: Quality of the samples for varying $\mathcal{R}$

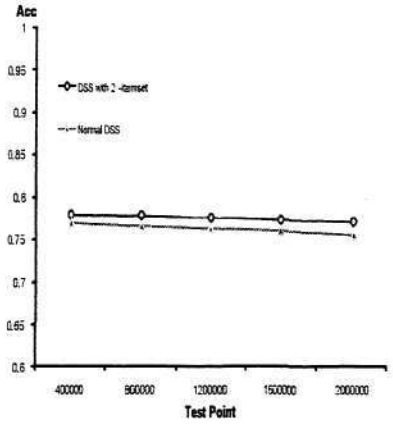


4.4.b: Performance of *DSS* for varying $\mathcal{R}$

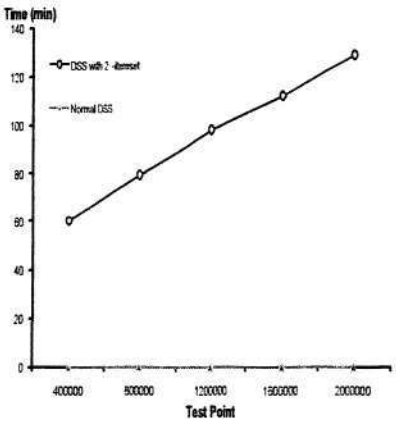
Figure 4.4: Impact of varying the value of $\mathcal{R}$

4.4.4 *DSS* with Higher Itemsets

So far, we have been discussing using the frequency of single itemsets to maintain the histogram. What about the support for 2-itemsets or k-itemsets? Logically, *DSS* having higher itemsets would perform better than one that is using only 1-itemsets. However, by considering all large itemsets in the histogram would be to generate all the $2^{\mathcal{I}}$ subsets of the universe of $\mathcal{I}$ items. It is not hard to observe that this approach exhibit complexity exponential in $\mathcal{I}$, and is quite impractical. For example, in our experiment, we have $\mathcal{I} = 10000$ items. If we consider having 2-itemsets in the histogram, we would generate about 50 million of possible combinations. Note that it is still technically possible to maintain an array of 50 million 2-itemsets in a system.



4.5.a: Performance of normal *DSS* and *DSS* with 2-itemset



4.5.b: Execution time of normal *DSS* and *DSS* with 2-itemset

Figure 4.5: Normal *DSS* vs. *DSS* with 2-itemset

However, to go up a higher order will be impossible. In this section, we shall compare the performance of *DSS* with and without 2-itemsets in the histogram.

To maintain 50 million counters in $Hist_{\mathcal{G}}$ and $Hist_{\mathcal{L}}$, we can still adopt the trimming method discussed in Section 4.2. The extra work added to *DSS* is the generation of all possible 2-itemset combinations for all incoming transactions. For example, a transaction having item A, B, C and D will generate AB, AC, AD, BC, BD and CD. In Figure 4.5, we compare the performance of *DSS* having 2-itemsets with one that uses only 1-itemsets on $T15I7D2000K$. From Figure 4.5a we see that *DSS* having 2-itemsets on average has a small increase of 2% in accuracy over normal *DSS*. In Figure 4.5b, we discover the price we need to pay for higher accuracy. The execution time for *DSS* having 2-itemsets increases tremendously. It took more than 2 hours to complete the sampling process where as a normal *DSS* finished in less than 5 minutes. Therefore, this explains why at our current research level we only focus on single itemsets.

4.4.5 Handling Noise

Removing transactions that are corrupted with noise is an important goal of data cleaning as noise hinders most types of data analysis. This section shows how *DSS* is not only able to produce high quality sample from normal data sets, but it is also able to cope with data sets having corrupted transactions by preventing these transactions from being inserted into the reservoir. To demonstrate the robustness of *DSS* against noise, we let the three algorithms operate on a noisy data set. For this experiment, we added 5% of noise to $T15I7D2000K$. Noise was added using the *rand* function. With a probability of 5%, we corrupt an item in a transaction by replacing it with any item in $\mathcal{I}$. Similar to the previous experiments, we made use of the true frequent patterns uncovered from the original data set to compare with the approximate frequent patterns uncovered from the corrupted data set. Figure 4.6 illustrates the performances of the three algorithms against noise. For reference, the results for the algorithms operating on the noise free data set are also included in the plot. From the graph, *DSS* suffers the least in terms of accuracy when noise was added to the data. Its overall accuracy is maintained at about 75% and its maximun drop in performance is at most 2% when the test point is at 400k. However, for LCA and Algo-Z, the gap between the original result without noise and the result with noise is wider when compared with the one by *DSS*. The greatest drop in accuracy is by 5% for LCA when the test point is at 2000k and by 6% for Algo-Z when the test point is at 1600k.

To make the comparison complete, we introduced 5%, 10% and 20% of noise to the three data sets. Figures 4.7, 4.8 and 4.9, show the results by varying the noise level. In all the data sets, the performances of the three algorithms are affected by noise. Their performances declines as the noise level increases. However, a closer look at the results reveals that even when the noise level is set at 20%, the performance of *DSS* does not suffer too badly as compared with Algo-Z and LCA. For example in

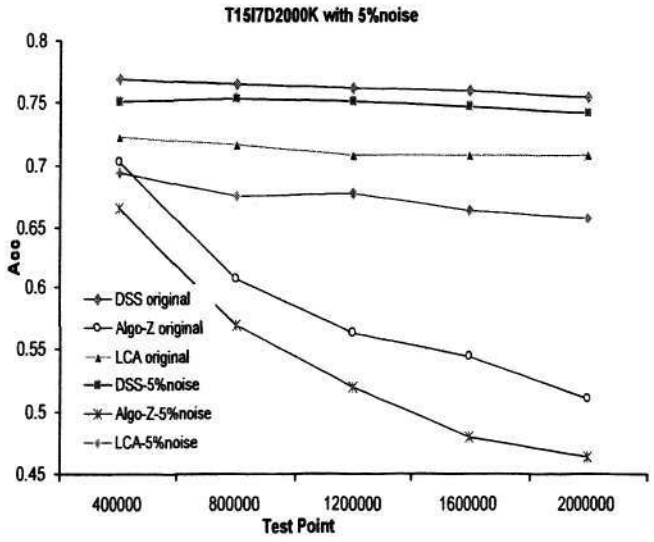


Figure 4.6: Accuracy on corrupted data set.

Figure 4.9, when processing Kosarak data set having 20% of noise level, the drop in accuracy for *DSS* is at most by 5% (test point = 400k). However, Algo-Z suffers 21% loss in accuracy (test point = 990k) and LCA suffers 11% (test point = 400k).

In these experiments, we can see that *DSS* outperforms the other two algorithms in terms of noise resistance. We reason this is due to the ranking process of *DSS*. As *DSS* is a deterministic sampling, every incoming transaction is processed and ranked according to its importance in the sample. When an incoming transaction is corrupted, its presence in the reservoir would create a larger discrepancy between $Hist_{\mathcal{L}}$ and $Hist_{\mathcal{G}}$ than when it is not corrupted and thus its chance to be inserted into the reservoir is low. In other words, employing distance based sampling is beneficial as it helps in filtering out noisy data from the database. As for Algo-Z, it uses simple random sampling and therefore it is blind towards noisy data. This explains why its performance is the most unreliable. Similarly for LCA, data is processed in batches. In a single batch of transactions, there can be a mixture of actual transactions as well as some corrupted transactions. LCA only maintains a set of entries to keep track of the counts of those itemsets that it regards as significant patterns. However,

there is no filtering process to distinguish between actual and corrupted transactions. Therefore its performance is poor particularly when a lot of noise is introduced.

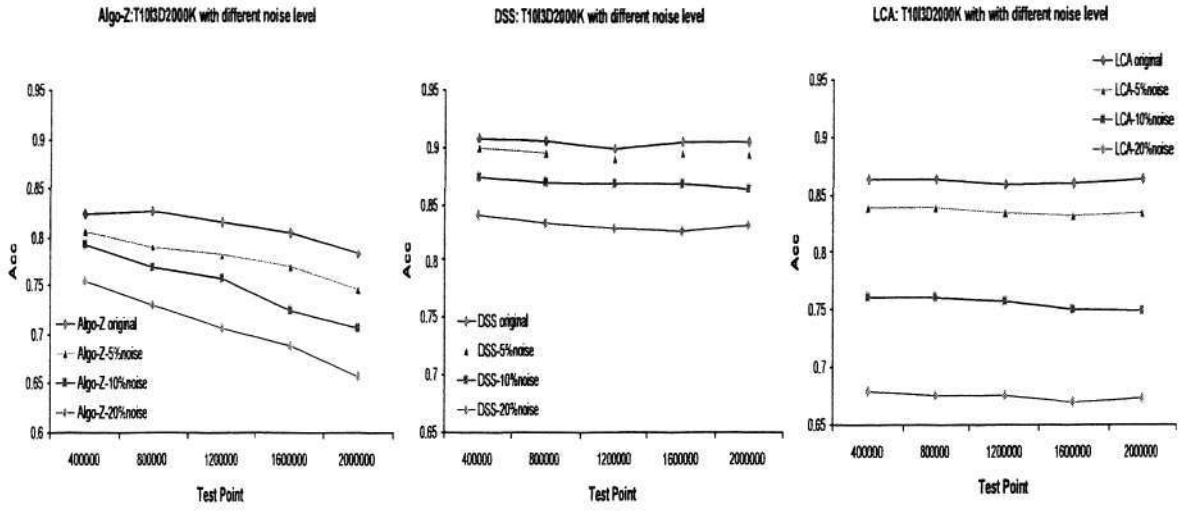


Figure 4.7: Accuracy on $T10I3D2000K$ with different noise level.

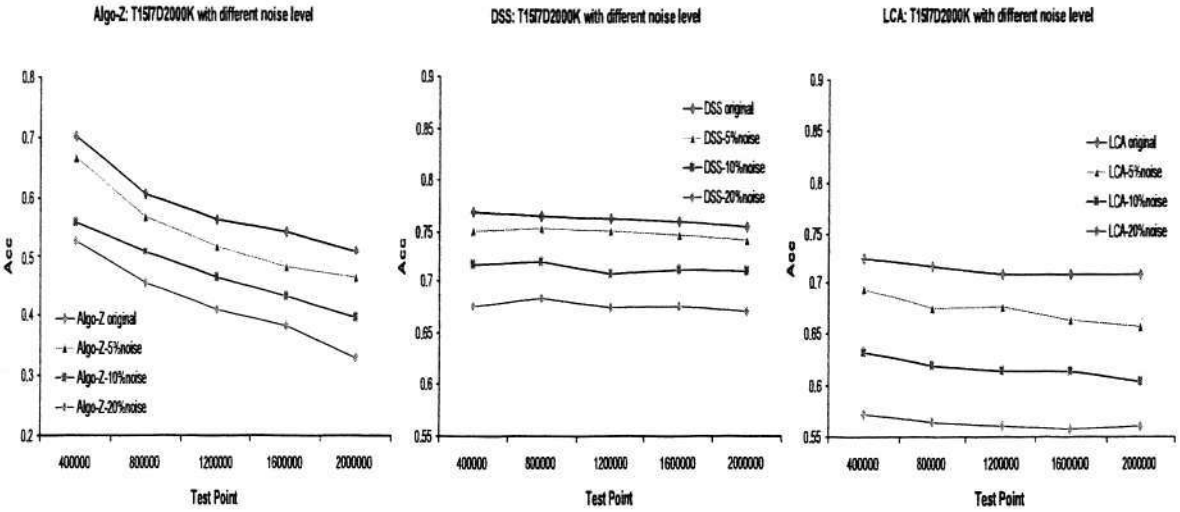


Figure 4.8: Accuracy on $T15I7D2000K$ with different noise level.

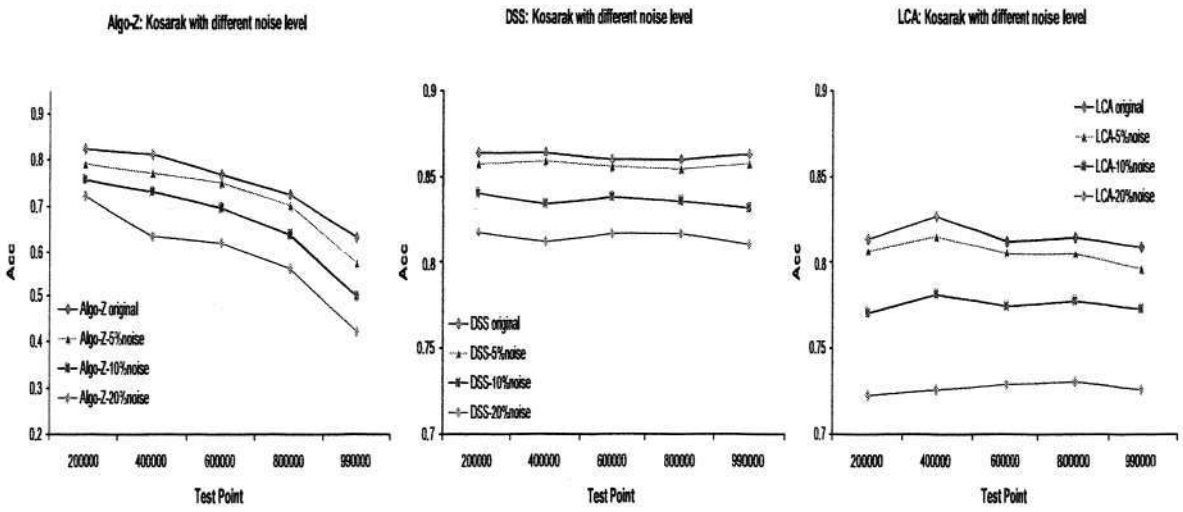


Figure 4.9: Accuracy on Kosarak with different noise level.

4.4.6 Comparison with Theoretical Bounds

Zaki et al. [ZPLO96] set the sample size using Chernoff bounds and find that sampling can speed up mining of association rules. However, the size bounds are found too conservative in practice. Moreover, Chernoff bound assumes data independence. In reality, data in a data stream is most probably dependent or correlated. When data is dependent in a transactional data stream, the quality of the sample cannot be guaranteed. In the last part of this experimental evaluation, we shall compare the performance of distance based sampling with the theoretical bounds.

Denote by $\mathbf{X}$ the number of transactions in the sample containing the itemset I. Random variable $\mathbf{X}$ has a binomial distribution of n trials with the probability of success σ_{min}. For any positive constant, $0 \leq \varepsilon \leq 1$, the Chernoff bounds [HR90] state that

$$P(\mathbf{X} \leq (1-\varepsilon)n\sigma_{min}) \leq e^{-\varepsilon^2 n\sigma_{min}/2} \qquad \text{(Eq. 4.12)}$$

$$P(\mathbf{X} \geq (1+\varepsilon)n\sigma_{min}) \leq e^{-\varepsilon^2 n\sigma_{min}/3} \qquad \text{(Eq. 4.13)}$$

Chernoff bounds provide information on how close the actual occurrence of an itemset in the sample is, compared to the expected count in the sample. *Accuracy* is given as $1-\varepsilon$. The bounds also tell us the probability that a sample of size n will have a given accuracy. We call this aspect *confidence* of the sample (defined as 1 minus the expression on the right hand side of the equations). Chernoff bounds give us two sets of confidence values. The first equation gives the lower bound – the probability that the itemset occurs less often than expected and the second one gives the upper bound – the probability that the itemset occurs more often than expected.

The following plots in Figure 4.10 show the results of comparing theoretical Chernoff bound with experimentally observed results. We show that for the databases we have considered the Chernoff bound is very conservative compared to the two sampling algorithms. Furthermore, we show that *DSS* samples are more accurate than reservoir sampling Z algorithm. We can obtain the theoretical confidence value by simply evaluating the right hand side of the equations. For example, for the upper bound the confidence $C = 1 - e^{-\varepsilon^2 n\sigma_{min}/3}$. We can obtain experimental confidence values as follows. We take s samples of size n, and for each item we compute the confidence by evaluating the left hand side of the two equations as follows. Let i denote the sample number, $1 \leq i \leq s$. Let $l_I(i) = 1$ if $(n\sigma_{min} - \mathbf{X}) \geq n\sigma_{min}\varepsilon$ in sample i, otherwise 0. Let $h_I(i) = 1$ if $(\mathbf{X} - n\sigma_{min}) \geq n\sigma_{min}\varepsilon$ in sample i, otherwise 0. The confidence can then be calculated as $1 - \sum_{i=1}^{m} h_I(i)/s$, for the upper bound. For our experiment we take $s = 100$ samples for both algorithms Z and *DSS* for each of the three data sets. We cover our discussion on all 1-itemsets. Using the theoretical and experimental approaches we determine the probabilities (1-confidence) and plot them in the following figures. Figure 4.10 compares the distribution of experimental confidence of simple random sampling and *DSS* to the one obtained by Chernoff upper bounds. The graphs show the results using $T15I7D2000K$, $n = 2000$ with $\varepsilon = 0.01$. From the figure, Chernoff bounds, with a mean probability of 99.95%, suggests that

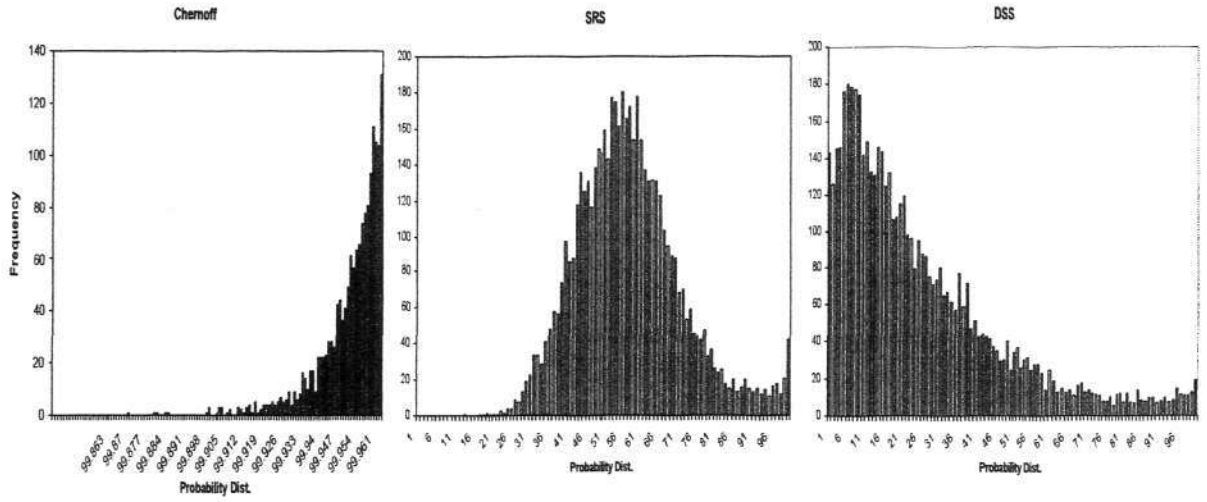


Figure 4.10: Probability Distribution.

this sample size is 'likely' unable to achieve the given accuracy. Obviously, this is very pessimistic and over conservative. In actual case, simple random sampling (or Algo-Z) and *DSS* gave a mean probability of 75% and 43% respectively.

Figures 4.11, 4.12 and 4.13, provide a broader picture of the large discrepancy between Chernoff bounds and experimental results. For the three data sets, we plot the mean of the probability distribution for different Epsilon (ε). Different values of sample size are used (from 0.1% to 10%). The higher the probability, the more conservative the approach is. So we can see that *DSS* samples are the most accurate, followed by Algo-Z samples, and the theoretical bounds are the most conservative.

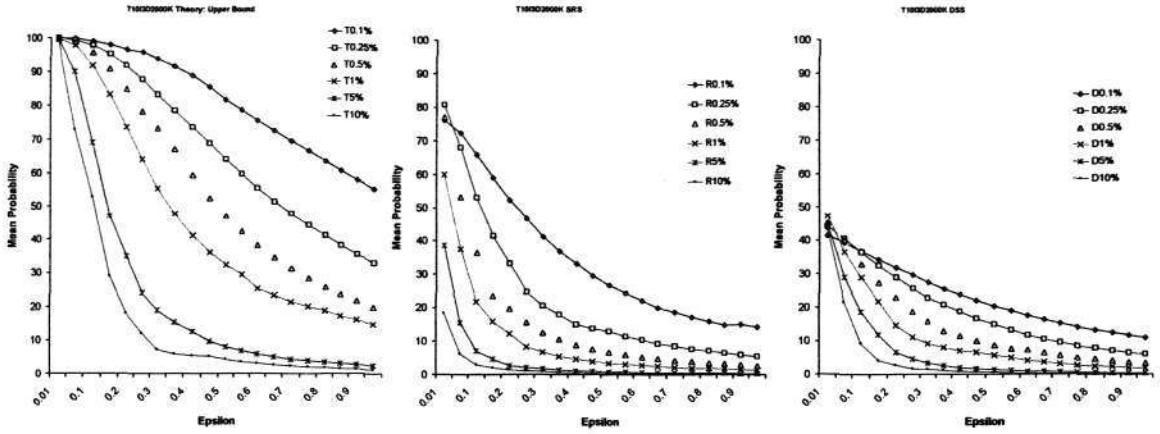


Figure 4.11: $T10I3D2000K$: Epsilon vs. mean Probability.

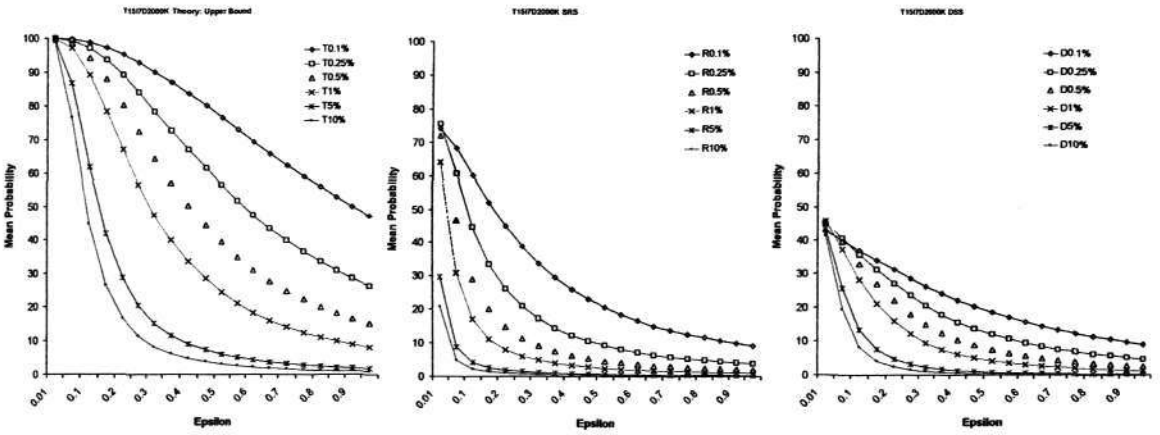


Figure 4.12: $T15I7D2000K$: Epsilon vs. mean Probability.

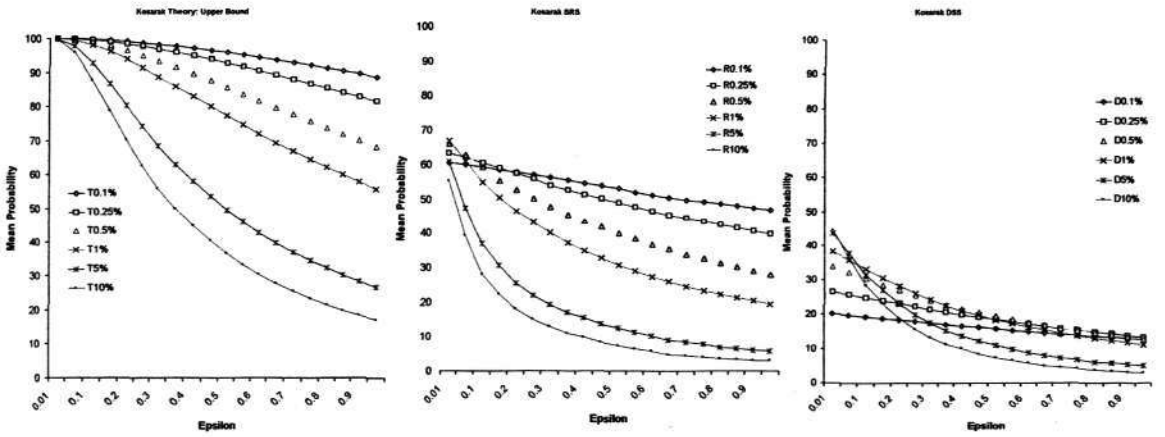


Figure 4.13: Kosarak: Epsilon vs. mean Probability.

4.5 Summary

As the volumes of available data grow too fast for the hardware to keep up, the data mining tasks encounter the scalability problem. There is a pressing need to have algorithms that handle data in an online fashion with a single scan of data stream whilst operating under resource constraints. An effective way to address this issue is to run the mining algorithms on a sample of the data. Gearing the data reduction algorithms to generate high-quality sample for the problem at hand provides better approximation results compared to simple random sampling (SRS). We have compared three algorithms LCA, Algo- Z and DSS. For sampling, random (Algo-Z) and deterministic (DSS) sampling are covered. If the sample size is small, it is advisable to adopt deterministic sampling than random sampling although deterministic sampling is more time consuming.

An empirical study using both real and synthetic databases supports our claims of efficiency and accuracy. In comparison to traditional two-phase data mining approaches (LCA), which store meta-patterns and re-mine internal data structures to finalize mining results, *DSS* has better time and space efficiencies. Random sampling produces sample of poor quality when the sample is small. Results show that DSS is significantly and consistently more accurate than both LCA and Algo-Z, whereas LCA performs consistently better than Algo-Z. The sample generated by DSS is well suited for dataset having vector of items. However, beside data mining, there are other problems that are highly relevant for data streams. For example, we can explore distance based sampling to search for unusual data. This has application to network intrusion detection and can be considered for future extension of DSS.

Chapter 5

Utility Mining over Transactional Data Streams

This chapter is based on [ND08b]. As discussed in the previous chapter, frequent pattern mining is an interesting subfield of data mining and has seen a proliferation of techniques over the past several years [AIS93, AS94]. A prominent aspect of research is the measuring of "usefulness" of the discovered patterns. Extracting meaningful patterns from data streams is important in many applications. In practice, FPM suffers from an embarrassment of richness: one often obtains too many results. The number of frequent patterns one discovers is inversely related to the support threshold. If one sets this threshold (too) high, one only discovers already well-known patterns. If one lowers the threshold, the number of patterns grows dramatically. The number of discovered frequent patterns can be easily in the thousands or tens of thousands. Getting more results than tuples in the database is not unheard of. Clearly, such large numbers of patterns are very difficult, if not impossible, to be analyzed by a human user. Furthermore, FPM only considers if an item is absent or present in a transaction. The quantity of the items present in a transaction has been tacitly ignored. External information such as the profit or cost of the item has not been considered as well.

Recently, the term *utility mining* (UM) was suggested in [YHB04] to address the shortcomings of FPM. In this context, utility refers to the measuring of how valuable an itemset is. The utility of an itemset can be measured in terms of cost, profit,

or other expressions of user preference. Thus, the main goal of UM is to mine high utility itemsets from a database. High utility itemsets differ from frequent itemsets in that the utility of the items in each itemset is taken into account (e.g., the profit associated with an item may be considered). We consider a Two-Phase algorithm to efficiently mine high utility itemsets in large data sets.

In this chapter, we extend the challenge of UM from traditional static databases to data streams. Here, we recognize two fundamental issues namely :

(i) How much data is sufficient for UM? Having access to massive data does not necessarily imply that a data mining system must use all of it.

(ii) How to deal with time-changing data streams? A data stream has necessarily a temporal dimension, and the patterns embedded in it are more likely to evolve as time goes by.

As demonstrated in the previous chapter, the method of sampling has great appeal because it generates a sample of the original multi-dimensional data representation. Thus, it can be employed with arbitrary data mining applications with little modifications to the underlying methods and algorithms. As such, sampling techniques in data mining have a long history, and they have also been recently considered as a way to practicality for FPM [Toi96, ZPLO96, Par02, BCD$^+$03, DN06]. Indeed, when faced with an infinite data set to analyze, it may be much more efficient to select some representative subset of the data and perform the mining on this subset. If the size of the subset is well defined, we can expect that the model obtained from mining the sample to be similar to one obtained from mining the entire data set. In this chapter, we employ the distance based sampling for streaming data (DSS). The algorithm is designed to produce sample that are "close" to the data set processed so far.a [DN06].

Suppose we employ a mining algorithm to create a model that describes certain aspects of the data stream. Assuming the current data generating process to be stationary, it is possible to simply feed this algorithm with a "good" sample. In such a scenario, our main problem will be to seek only the best sampling technique such that a sample can be used to represent the unbounded data stream. We can then draw upon years of research in sampling where powerful techniques have already been implemented [Coc77, OR95, JMR05]. Unfortunately, such an assumption is impractical as data streams do change. Moreover, it is a non-trivial task to determine when change really occurs. Past research on mining frequent patterns over stream data has mainly focused on adapting rather than detecting changes [GHP+03, CL03b, CWYM04]. Maintaining an up to date model is important. However, we contend that it is also imperative to alarm the user when change is detected. A user may be interested to understand the nature of the change so that he can take the appropriate action against such change in the shortest possible time. Therefore, in this spirit, we will also cover an online change detection method that is capable of working with sampling and UM algorithms.

5.1 Research Problem

Consider a data stream composed of two consecutive subparts (D_A and D_B). The transition from D_A to D_B occurs at time t_{change} (see Figure 5.1). Such a scenario is common in applications that have seasonal changes , e.g. from winter to spring. We may want to conduct sampling over certain fixed time interval. For example, we obtain the sample S_{A1} from time t_0 to t_1. To obtain S_{A1}, some techniques such as the reservoir sampling [Vit85] can be employed. We can then mine S_{A1} and construct a reference model (high utility itemsets) that represents D_A. Next, we obtain the sample S_{A2} from time t_1 to t_2. Assuming that the data are generated independently, the data distribution of S_{A1} and S_{A2} should be identical ($P(S_{A1}) \approx P(S_{A2})$). In this

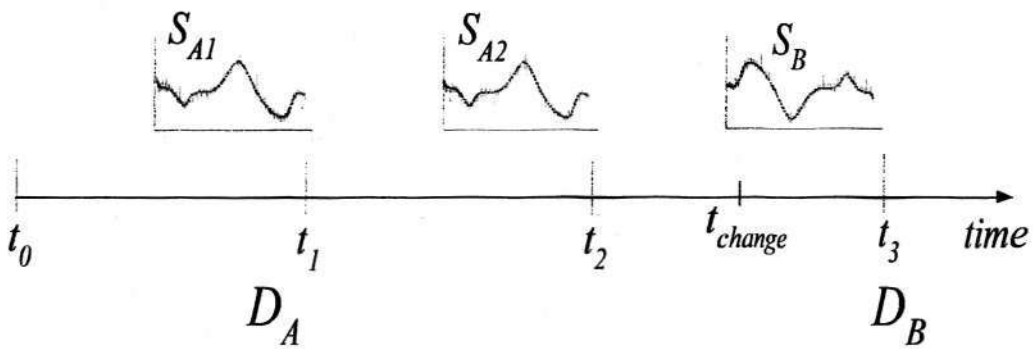


Figure 5.1: Comparing sample distribution

respect, whatever high utility itemsets found in S_{A1} are likely to be uncovered again in S_{A2}. Let S_{A1}, the sample that is used to build the reference model, be the reference sample. We can then periodically compare the reference sample with the current sample. We will need to design a test algorithm that can determine if the present and past data streams are different. If the test indicates that they are different, the mining algorithm will be triggered to alarm the user. In this case, S_B will replace S_{A1} as the new reference sample and the monitoring process continues [KBDG04]. All the samples will be processed in the main memory. In addition, to provide a mechanism for making quantitative descriptions of the detected change, we can adopt the statistical test. A statistical test will determine if there is any difference between the two models. We believe there is the opportunity for an immensely rewarding synergy between data mining and statistics.

5.1.1 Motivating Applications

We provide some examples to make the above research problem concrete.

Example 5.1 (*Utility Mining*). A click stream is the recording of what a user clicks on while Web browsing. Click stream analysis is useful for Web activity analysis, software testing, market research, and for analyzing employee productivity. A sequence of clicks as a visitor explores a Web site can be considered as a transaction. Depend-

ing solely on the number of visits to a Web page might not be sufficient to answer questions, such as whether this Web page (or item) has a high utility value. Since the number of visits to a Web page and the time spent on a particular Web page is different for different visitors, the total time spent on a page by a visitor can also be viewed as a utility. A Web site designer can study the behavior patterns of the visitors by looking at the utilities of the page combinations and then re-organizing the link structure of their website to cater to the needs of the visitors.

Example 5.2 (*Change Detection*). A sales manager may be interested in analyzing the outcome of a marketing campaign targeted at increasing the sales of some discounted products. The goal may be to assess the overall accomplishment of this campaign and the impact on other related products. The manager is likely to trace and compare the customer behavior reflected in the transactions recorded **before** the campaign with the transactions recorded **after** the campaign. To do this, he needs to know when a significant change occurs after the campaign is launched.

Example 5.3 (*Building New Model*). An analyst may wish to employ data mining techniques to summarize a large database to facilitate decision making. The data mining model does not necessarily need to be changed as long as the underlying data distribution is stable. However, if significant changes are detected in the distribution, the reliability of this model might drop drastically. Past research has focussed on adapting to changes by using a sliding window or a time decay factor that slowly dilutes the effect of outdated information. However, if the change is significant, it will be awkward to have a model built from a database that consists of a mixture of recent and outdated information. In this respect, it is advisable to forget the old transactions and rebuild a new model based on the recent transactions once a change is triggered.

5.2 Utility Mining

In the field of UM, there are several definitions of useful itemset depending on the user's objective. Research that assigns different weight to items has been introduced [CFCK98, TMF03]. The share measure is presented in [BH03] to overcome the shortcomings of support. It reflects the impact of the quantity sold on cost or profit of an itemset. [CYS03] recently proposed a new pruning strategy based on utilities that allows pruning of low utility itemsets to be done by means of a weaker but anti-monotonic condition. Our discussion follows closely on the work presented in [YHB04, LkLC05a, LkLC05b, TCL06, ND08b]. The earliest work on mining temporal high utility itemsets from data streams is by [TCL06]. They proposed an algorithm named THUI-Mine that can mine temporal high utility itemsets from data streams. The underlying idea of THUI-Mine algorithm is to integrate the advantages of the Two-Phase algorithm [TCL06] and a sliding window model [LLC05], and augment incremental mining techniques for mining temporal high utility itemsets efficiently. However, the algorithm does not provide any solution for detecting changes in data streams.

For ease of exposition, this section provides the common notations that will be found in this chapter. We adopt similar definitions as in [YHB04, LkLC05b]. The problem of UM can be formally stated as follows: Let Ds denote a data stream, which is a sequence of continuously in-coming transactions, e.g., $t_1, t_2, ..., t_N$. We denote N as the number of transactions processed so far. Each transaction has a unique identifier (Tid) and contains a set of items. Let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ be a set of distinct items.

The local transaction utility value, $lo(i_p, T_q)$, is the amount of item i_p present in transaction T_q (see Table 5.1). For example, the amount of an item sold in the transaction might be used as the transaction utility. The external utility, $ex(i_p)$, of an item i_p is based on information not available in the transaction. For example, it might be stored in a utility table, such as that shown in Table 5.2, which indicates the

Table 5.1: Transaction table. Each row is a transaction. The columns represent the amount of items in a particular transaction.

Transaction ID	*ItemA*	*ItemB*	*ItemC*	*ItemD*	*ItemE*	*Transaction Utility*
T_1	5	1	3	0	1	117
T_2	0	2	1	0	0	60
T_3	0	4	1	0	2	104
T_4	20	0	3	2	4	148
T_5	0	2	0	1	6	77
T_6	12	0	1	0	7	91

Table 5.2: The external utility table.

Item Name	Profit ($)
Item A	1
Item B	15
Item C	30
Item D	5
Item E	7

maximum profit for each item. To measure the utility of i_p in transaction T_q, we simply compute the product of $lo(i_p, T_q)$ and $ex(i_p)$. For example, for $u(A, T_1) = 5 \times 1 = 5$. An itemset X is a set of items such that $X \in (2^{|\mathcal{I}|} - \{\emptyset\})$ where $2^{|\mathcal{I}|}$ is the power set of $\mathcal{I}$. An itemset with k items is called a k-itemset. We write "ABC" for the itemset $\{A, B, C\}$ when no ambiguity arises. To measure the utility of X in transaction T_q, we simply compute the sum of all $u(i_p, T_q)$ where $i_p \in X$. For example, for $u(ABC, T_1) = u(A, T_1) + u(B, T_1) + u(C, T_1) = 110$.

Definition 5.6 [**Utility of an itemset** X] *The utility of an itemset X is the sum of utilities of X in all transactions containing X:*

$$u(X) = \sum_{T_q \in D \wedge X \subseteq T_q} u(X, T_q). \qquad \text{(Eq. 5.1)}$$

For example, $u(DE) = u(D, T_4) + u(E, T_4) + u(D, T_5) + u(E, T_5) = 85$.

Definition 5.7 [High Utility Itemset] *An itemset X is considered a high utility itemset if and only if its utility is more than or equal to some minimum utility threshold $util_{min}$. We denote the set of all the high utility itemsets by AHU.*

Thus, the job of UM is to mine for all the high utility itemsets in D. A lattice structure can be used to enumerate the list of all possible itemsets. Figure 5.2 shows an itemset lattice for $\mathcal{I} = \{A, B, C, D, E\}$. However, it should be emphasized that when computing the utility of an itemset, the itemset utility can either increase or decrease as the itemset is extended by adding items. There is no efficient strategy to find all the high utility itemsets due to the non-existence of anti-monotone property in the utility mining model. For example, $u(AC) > u(A)$.

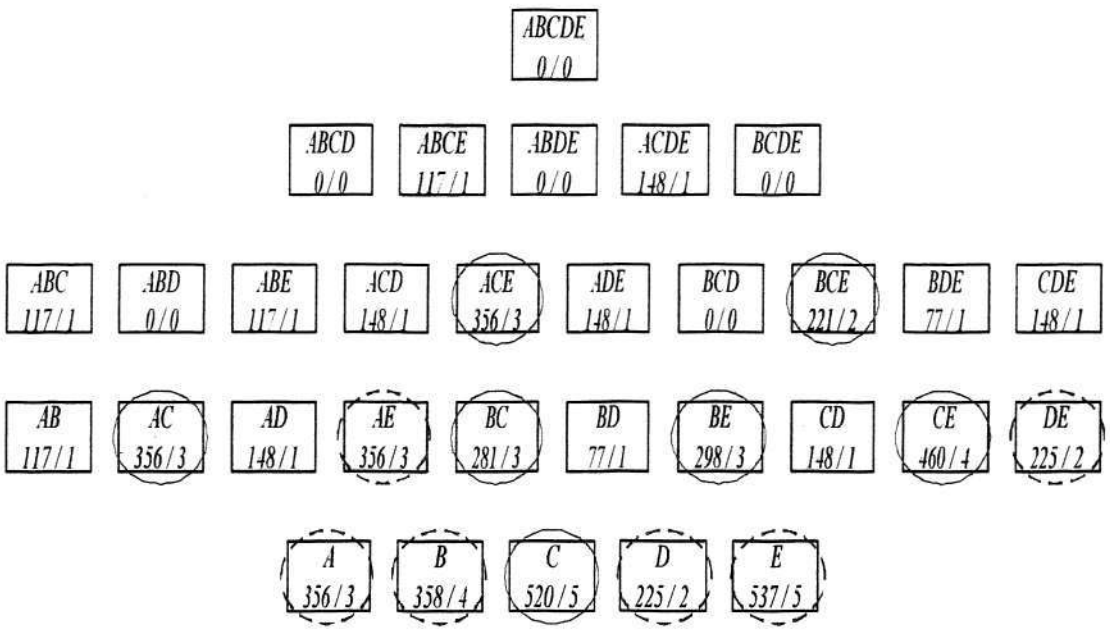


Figure 5.2: An itemset lattice generated from Table 5.1.

A brute-force approach for finding high utility itemsets is to determine the utility value for every candidate itemset in the lattice structure. To do this, we must compare each candidate against every transaction. Unfortunately, the generation of candidate itemsets is the most costly in terms of time and memory space. The real challenge of UM is in constraining the size of the candidate set and simplifying the

computation for calculating the utility. In [YHB04], a heuristics is used to predict whether an itemset should be included in the candidate set. However, the prediction usually overestimates, particularly at the initial stages, where the number of candidates approaches the number of all the combinations of items. The examination of a large combinations is impractical whenever the number of items is large or $util_{min}$ is low. To circumvent this problem, the Two-Phase algorithm is proposed by [LkLC05b].

5.2.1 Downward Closure Environment

Here, we discuss an algorithm which is able to mine high utility itemsets in large data sets efficiently. The relevant definitions and theorem are given as follows.

Definition 5.8 [**Transaction Utility**] *The transaction utility of T_q, denoted as $tu(T_q)$, is the sum of the utilities of all the items in T_q:*

$$tu(T_q) = \sum_{i_p \in T_q} u(i_p, T_q). \qquad \text{(Eq. 5.2)}$$

For example, $tu(T_1) = u(A, T_1) + u(B, T_1) + u(C, T_1) + u(E, T_1) = 117$.

Definition 5.9 [**Transaction-weighted Utilization**] *The transaction-weighted utilization of an itemset X, denoted as $twu(X)$, is the sum of the transaction utilities of all the transactions containing X:*

$$twu(X) = \sum_{X \subseteq T_q \in D} tu(T_q). \qquad \text{(Eq. 5.3)}$$

For example, $twu(AC) = tu(T_1) + tu(T_4) + tu(T_6) = 356$.

Definition 5.10 [**High Transaction-weighted Utilization itemset**] *An itemset X is considered a high transaction-weighted utilization itemset if and only if $twu(X)$ is more than or equal to the user defined threshold twu_{min}. We denote the set of all the high transaction-weighted utilization itemsets by $AHTW$.*

Theorem 5.1 *The transaction-weighted utilization is downward closed with respect to the lattice of all itemsets.*

Proof: We let itemset Y be a high transaction-weighted utilization itemset and itemset X be a subset of Y. This implies that for all transaction $T_q \supseteq Y$, $T_q \supseteq X$ too. Therefore,

$$twu(X) = \sum_{X \subseteq T_q \in D} tu(T_q) \geq \sum_{Y \subseteq T_q \in D} tu(T_q) = twu(Y) \geq twu_{min}. \quad \blacksquare$$

When $util_{min} = twu_{min}$, it is not difficult to show that a high utility itemset is also a high transaction-weighted utilization itemset ($AHU \subseteq AHTW$). Theorem 1 is useful as it allows one to create a downward closure environment so that redundant itemsets can be efficiently pruned away using conventional mining algorithms. The Two-Phase algorithm employs it for the transaction-weighted utilization mining in Phase I. Similar to the Apriori algorithm [AS94], it applies the downward closure property on the search space to expedite the identification of candidates. Thus, only the combinations of high transaction weighted utilization itemsets are added into the candidate set at each level during the level-wise search. Consider the example when $util_{min}$ equals to 150.

In Figure 5.2, the shaded boxes refer to the reduced search space. The content in each box represents transaction-weighted utilization / number of occurrences. Any itemset having transaction-weighted utilization $\geq twu_{min}$ will be considered as a high transaction-weighted utilization itemset. Note that the output from Phase I are $AHTW$ (all solid and dashed circles). We are only interested on AHU. Phase I may overestimate some low utility itemsets, but it never underestimates any itemsets. Therefore, in phase II, one database scan is performed to filter the overestimated itemsets in order to get AHU (all solid circles).

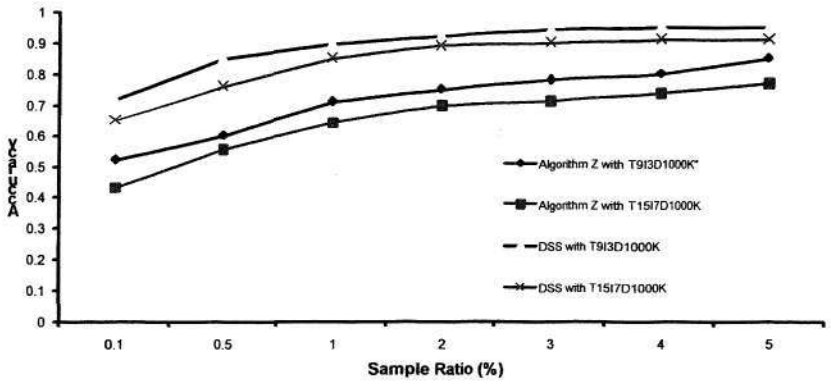
5.3 Experimental Evaluation I

Here, we present an experimental comparison between DSS and the existing random sampling, Algo-Z, in the context of UM. All experiments were performed on a 1.7GHz CPU Dell PC with 1GB of main memory running on the Windows XP platform. The algorithms were written in C++. We used the synthetic data sets in our experiments and they were generated using the code from the IBM QUEST project [AS94]. The two data sets that we used are $T9I3D2000K$ and $T15I7D2000K$ (see Section 3.5 for the nomenclature of these two data sets). Items were drawn from a universe of $\mathcal{I} = 5k$ unique items. Note that these data sets only contain quantity of 0 or 1 for each item in a transaction. To suit our need, we randomly generated the quantity of each item in each transaction, ranging from 1 to 10 [LkLC05b]. The external utility table was also synthetically generated by assigning a utility value to each item randomly, ranging from 0.1 to 100. The results of the two algorithms are computed as an average of 10 runs, each one corresponding to a different shuffle of the input.

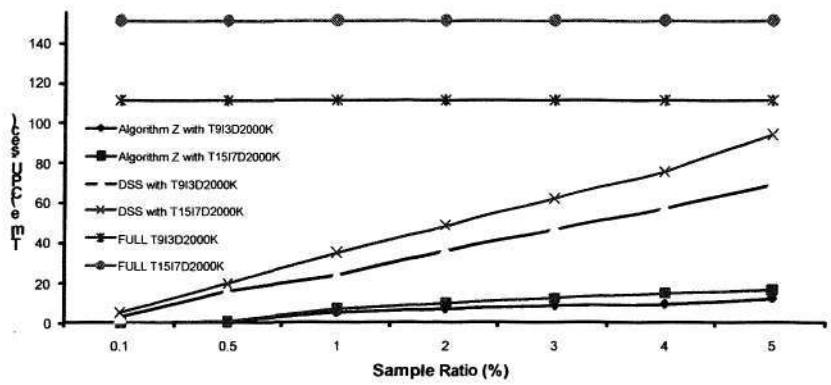
Since UM is focused on finding the high utility itemsets, the performance metric to measure the accuracy of DSS is based on the set difference between all high utility itemsets generated from the sample S and all the transactions processed so far Ds. Hence, it is sensitive to both false and missing high utility itemsets. In particular, the measure of accuracy is defined as:

$$Accuracy = 1 - \frac{|L(Ds) - L(S)| + |L(S) - L(Ds)|}{|L(S)| + |L(Ds)|} \qquad \text{(Eq. 5.4)}$$

where $L(Ds)$ and $L(S)$ denote the sets of high utility itemsets in Ds and S respectively. The threshold is set at $util_{min} = 1\%$. The experiment compared the performance of DSS and Algo-Z at different sampling ratios. We chose sample ratios ranging from 0.1% to 5%. Figure 5.3(a) displays the accuracy of DSS and Algo-Z against the



5.3.a: Accuracy *vs.* Sampling Ratio



5.3.b: Time *vs.* Sampling Ratio

Figure 5.3: Results for synthetic data sets

sampling ratio after processing 2 million transactions. From the graph, *DSS* achieved good accuracy even for small sampling ratios. Thus, with a sampling ratio of 1%, *DSS* gave an accuracy of more than 85% for both data sets while random sampling can only achieve at most 71%. For larger ratio, the accuracies of *DSS* settle at near 95% and 91% for $T9I3D2000K$ and $T15I7D2000K$ respectively. Algo-Z could not even hit 90%.

Figure 5.3(b) depicts the execution time of the two reservoir sampling algorithms *vs.* the sampling ratio. Note that the execution time consists of the time for obtaining

the sample as well as using the Two-Phase algorithm to generate the high utility itemsets. For reference, the execution times for running the entire data sets are also included in the plot. As expected, the time increased when the sampling ratio increased for the two algorithms. In particular, Algo-Z is faster because it only needs to maintain a sample by randomly selecting transactions to be deleted or inserted in the reservoir. There is no real processing work to be done on any incoming transaction. Due to the nature of DSS, which is deterministic sampling, DSS is more time-consuming than random sampling. However, even when the sampling ratio is 5%, the time spent by DSS is much lower than running the entire data sets. The execution time for DSS is still reasonable considering the high accuracy achieved. Note that once a set of transactions is selected, it can be used multiple times for data mining purposes. In this way, the extra time used by DSS to choose a better sample than Algo-Z is worthwhile.

5.4 Change Detection

For change detection, we would need to generate two sets of samples. It is not difficult to create two reservoir samples at separate time intervals. For example, from time 0 to t_1, we maintain a reservoir of size n and from t_1 to t_2, we maintain another reservoir of the same size (see Figure 5.4). The interval $[t_0, t_1]$ provides the range of transactions (N_1) for computing high utility itemsets. However, (N_1) can easily overwhelm the memory space. Even if it is possible to store them in the main memory, the time $(t_p - t_1)$ to compute all the transactions may be too expensive. To meet the real time requirement, we can allow the reservoir sample to be kept in the main memory and processed such that the approximate high utility itemsets can be uncovered at t'_p where $t'_p - t_1 \ll t_p - t_1$.

In the case of UM, there will be two sets of all high utility itemsets (AHU) for the two samples. Here, any term having a subscript of ref or cur relates to the

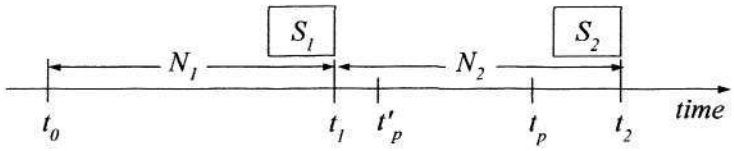


Figure 5.4: A fragment of sequence

reference and current samples respectively. For example, AHU_{cur} denotes the set of all frequent itemsets generated from the current sample. A straightforward solution for change detection is to measure the symmetric set difference (SD) between these two sets of AHU [LCK98, BCD+03]. The smaller the value of SD, the greater is the similarity between AHU_{ref} and AHU_{cur} (and vice versa). Note, however, that while computing SD, the absolute utility value of each high utility itemset has been ignored. We view the utility value of the itemsets as a potentially useful information. Consider a threshold $util_{min} = 200$. Then, an itemset ABC having the utility value of 200 in the reference sample and 2000 in the current sample will be treated as high at the same time. Obviously, there is a significant difference but this will not be reflected in SD.

Interestingly, [KBDG04] provides a good study for detecting general changes in the data distribution of the data stream. They proposed a change-detection method with statistical guarantees of the reliability of detected changes. This research can be beneficial to the user as it reduces the overhead incurred from repeatedly mining the data set. We would only need to rebuild the model if there is a change in the data stream. Unfortunately, monitoring the stream directly for changes becomes difficult when the dimensions of the data becomes an issue.

In order to determine whether there are deviations between the current and the past data stream with respect to high utility itemsets, one may construct two discrete distributions. Each bin in the distribution corresponds to an itemset. We want to know if the past and present data streams are related based on the distributions. However, it is practically impossible to establish a distribution that contains all the

possible itemsets. For example, if $\mathcal{I}$ is a set of all distinct items, to build a histogram, one would required $2^{|\mathcal{I}|}-1$ number of bins! Since we are unable to build a distribution out of the data stream, we must base our conclusions on the models produced from the samples. The ACD in [ND08b] differs from the work in [KBDG04] in another salient manner; it focuses on change detection purely on UM. It attacks the problem from another angle, that is, it detects the changes between two samples in terms of data mining models they induce. We would need a tool that can compare these two models.

Statistical tools for exploring such hypotheses are often called hypothesis testing [PG00]. The basic principle of the hypothesis testing is as follows. We start by defining two complementary hypotheses: the null hypothesis and the alternative hypothesis. In our case, the null hypothesis, denoted as H_0, might state that $Hist_{ref} = Hist_{cur}$, i.e. there has been no detectable change in the data stream, and the alternative hypothesis (H_1) might state that $Hist_{ref} \neq Hist_{cur}$, i.e. there is a detectable change. $Hist_{ref}$ denotes distribution representing the reference model and $Hist_{cur}$ denotes distribution representing the current model. Using the reference and current models, we can compute a test statistic e.g., student-t statistic. The statistic value would vary from sample to sample. Based on the test statistic, the p-value is determined. The p-value represents the probability of having a sample as extreme as or more extreme than the current sample with respect to the reference sample. If the p-value is less than the level of significance, the null hypothesis is rejected, or else the null hypothesis is accepted.

5.4.1 Algorithm for Change Detection (ACD)

In order to determine whether there are deviations between the current and the past data stream with respect to high utility itemsets, we construct two discrete distributions. Each bin in the distribution corresponds to an itemset. We want to test if the past and present data streams are related based on the histograms. The algorithm

Algorithm 2 Algorithm for Change Detection

1: $T \leftarrow t_1$;
2: load S_1 to $window_{ref}$ at t_1;
3: $AHU_{ref} \leftarrow$ FindHighUtilityItemsets($window_{ref}$);
4: $i = 1$;
5: **while** not end of stream **do**
6: $i++$;
7: load S_i to $window_{cur}$ at t_i;
8: $AHU_{cur} \leftarrow$ FindHighUtilityItemsets($window_{cur}$);
9: $AHU'_{ref} = scan(window_{ref}, Miss_{ref}) + AHU_{ref}$;
10: $AHU'_{cur} = scan(window_{cur}, Miss_{cur}) + AHU_{cur}$;
11: **if** $TEST(AHU'_{ref}, AHU'_{cur}) =$ Reject **then**
12: $T \leftarrow t_i$;
13: Report change at time T;
14: load S_i to $window_{ref}$;
15: $AHU_{ref} = AHU_{cur}$;
16: **end if**
17: **end while**

for detecting change in data steam is outlined in Algorithm 2. In the beginning, the first batch of transactions (S_1) will be loaded into the reference window ($window_{ref}$). Using the Two-Phase algorithm [LkLC05b], we can efficiently generate all the high utility itemsets, AHU_{ref}, from $window_{ref}$. In the next time interval, a new batch of transactions S_2 is loaded to the current window, $window_{cur}$. We then generate the AHU_{cur}. The absolute support value of each high utility itemset will be recorded (line 2-8).

To test for discrepancy, we compare AHU_{ref} with AHU_{cur}. Note that when we conduct the statistical test, we are actually comparing the high utility itemsets based on their utility values. In an ideal case, if the data distributions at any time interval are found to be consistent, we should be able to find a match. That is, whatever high utility itemset discovered in $window_{ref}$ can also be discovered in $window_{cur}$, and vice versa. Therefore, each high utility itemset will contain two utility values (one from each window). However, there may be a small set of itemsets whose utility values are

very near to $util_{min}$. As a consequence, even when there is no shift of patterns in the data stream, we might still encounter some itemsets that are present in AHU_{ref} but absent in AHU_{cur}, and vice versa. ACD does not end at simply comparing itemsets which are in $AHU_{ref} \bigcap AHU_{cur}$. In fact, as long as an itemset is considered high utility in either $window_{ref}$ or $window_{cur}$, it has to be included in the test.

Definition 5.11 [**Missing Itemsets**] *We define* $Miss_{ref}$ *(or* $AHU_{cur} - AHU_{ref}$*) as a set of itemsets that is considered high utility in* $window_{cur}$ *but not in* $window_{ref}$ *and* $Miss_{cur}$ *(or* $AHU_{ref} - AHU_{cur}$*) as a set of itemsets that is considered high utility in* $window_{ref}$ *but not in* $window_{cur}$.

We would like to look into why these missing itemsets are considered high utility in one window but not in another. To include these itemsets in the test, the utility value must be obtained from both the windows. Since we already have one of their utility value through the Two-Phase algorithm, what we need to do is to perform a single scan on the window that classified these itemsets as low utility (line 9-10). For example, if itemset ABC is considered high utility in $window_{cur}$ but not in $window_{ref}$, we will do a scan on $window_{ref}$ to compute its utility value. That means, at any time there are two samples residing in the memory.

Once the scanning is done, we will have two sets of itemsets, AHU'_{ref} and AHU'_{cur}, that cover $window_{ref}$ and $window_{cur}$ respectively. A test will be conducted on these two sets to see if there is any discrepancy (line 11). The discrepancy is based on the difference in utility value of the itemsets in the two windows. That means even when there is no difference in the list of high utility itemsets in the two windows, ACD may still detect change. This way the user is warned even before the major change takes place. The statistical tests are based on a level of significance. The user can set it according to the application, i.e., if the application is required to detect very fine changes then the level of significance can be set high. If there is significant

change, ACD will report a change in the current stream. The content in $window_{cur}$ and AHU_{cur} are then transferred to $window_{ref}$ and AHU_{ref} respectively. $window_{cur}$ will be cleared to accommodate the next batch of transactions. The test cycle is repeated again.

Note that in situations when the change is insignificant or the change rate is gradual, the $window_{cur}$ and $window_{ref}$ may overlap considerably. In this case, it may be difficult for any test algorithm to detect transition. Even if the test algorithm fails, the $window_{cur}$ is always in the main memory and we can output the most recent mining result to user upon request. In this way, we cover adaptation and detection at the same time.

5.5 Statistical Test

An important aspect of the proposed algorithm is the ability to test whether the frequency counts of itemsets in the reference window are different from the frequency counts of those itemsets in the current window. Statistical tests are very helpful. There are many statistical tests [PG00]; choosing the right one for the given task can be at times challenging as it is in this case. Let us consider the characteristic of the test that we wish to do:

(i) it is paired and two-sample – because for the reference and current windows we compare the frequency counts of the same itemsets,

(ii) it is not continuous, but count data,

(iii) the underlying distribution is not given, but one may argue in favor of approximate normal distribution considering the fact that the data are frequency counts – typically when it is count data, binomial distribution is used, and when the size is large (as is the case here) one may approximate it to be normal distribution,

(iv) we are interested in determining whether the differences (irrespective of positive or negative) between the corresponding counts are significant.

Considering these criteria, one may choose, paired t-test, non-parameteric tests (Sign test and Wilcoxon Signed-Rank test for paired samples), Chi-square test, test of two proportions, and Fisher's exact test. Let us consider the suitability of these tests for our task in a systematic manner. We devote some space here for describing and analyzing the suitability of these tests in order to give a comprehensive understanding. A simple data is used to compare the suitability among the different methods. Sample 1: Item A - Utility 30, Item B - Utility 70; Sample 2: Item A - Utility 70, Item B - Utility 30. Obviously, these two samples are very different. Note that this simple data is used just for making the reader understand the inherent characteristic of each statistical test. In actual testing, much larger data is used, but the inherent characteristic remain unchanged.

5.5.1 Paired t-test

The paired t-test is used for two samples that are paired. Two samples can be paired either by self-paring (the same subject has two values corresponding to the two samples) or by matching (there are actually two subjects but they are matched using some common characteristics). In our case the samples are self-paired, i.e., the utilities of the same itemset is measured in two samples – reference and current windows. The null hypothesis is that the two population (from which the two samples were drawn) means are the same, or in other words their mean difference is 0. The alternative hypothesis is that the mean difference is not 0.

Assuming that the data is approximately normally distributed, t-test is applied. The following equations mathematically depict this. $H_0 : \delta = 0$, $H_A : \delta \neq 0$ and $t = \frac{\bar{d}-\delta}{\frac{s_d}{\sqrt{n}}}$ where $\bar{d} = \frac{\sum_{i=1}^{n} d_i}{n}$, i.e., the mean difference and d_i is the difference between i^{th} values of reference and current samples, s_d is the standard deviation of the differences

d_i, n is the sample size and δ is the population mean difference. Under H_0 the mean difference is 0. But notice that by taking the mean of the differences, paired t-test is actually homogenizing the differences, both positive and negative. To understand this point, let us run through our example problem described earlier. The two differences are 30-70=-40 and 70-30=40 respectively. The mean difference is 0! Thus the null hypothesis is not rejected which is actually **not true**.

5.5.2 Nonparametric Tests

These tests are useful particularly when the underlying distributions cannot be assumed to be normal or approximately normal. For paired data two tests are commonly used: sign test and Wilcoxon signed-rank test. In the sign test, just like the paired t-test, the differences between the corresponding values between the two samples are computed and their signs (+ or -) are noted down. The null hypothesis states that the median difference is equal to 0. Note that median (not mean) is used because only the sign but not the magnitude is used. Under the null hypothesis one can expect the number of +ve and -ve signs to be equal. Thus the numbers of +ve and -ve signs are computed. Notice that as the outcome of each difference is a sign which takes only two possible values with a probability of 0.5, each of these is considered as a Bernoulli trial. Thus the mean number of +ve signs is $n/2$ and standard deviation is $\sqrt{\frac{n}{4}}$. If n is large (≥ 20) one can use a standard normal variable as given below.

$$z_+ = \frac{D - n/2}{\sqrt{\frac{n}{4}}} \qquad \text{(Eq. 5.5)}$$

If $n < 20$, binomial distribution is used. The p-value is determined by computing twice the probability of obtaining D positive difference – or some number more extreme – given that the null hypothesis is true. In our example problem with only two itemsets, binomial distribution is used. $P(D \geq 1) = P(D = 1) + P(D = 2) = 0.75$ As the

p-value is much larger than the assumed level of significance 0.05, the null hypothesis is accepted, i.e., the median difference is 0, although clearly this is not true.

The sign-rank test is a modification of the sign test where in addition to the signs of the differences, their ranks are also considered. The differences are ranked using their absolute values, and then the ranks of the positive difference are separated from the ranks of the negative differences. The sum of these two groups of ranks are taken. Under the null hypothesis which states that the median difference is 0, it is expected that the two sums of ranks (+ve ranks and -ve ranks) will be equal. Then based on the size of the sample (n) either a standard normal distribution (when $n > 12$) or a special distribution (when $n \leq 12$) is used to compute the p-value. In the example data $n = 2$, so we use the special distribution and p-value > 0.05, thus the null hypothesis is accepted, which is not correct. The reason why both the nonparametric methods failed is because they also grouped the differences without considering their absolute values.

5.5.3 Chi-square Test

Finally we discuss the Chi-square test and some of its variations which are found to be the most suitable tests. Chi-square test is applicable for count data. Data is typically arranged in a contingency table format and the observed frequencies are compared with expected frequencies. See Table 5.3 showing the contingency table for the example problem. H_0 : There is no association between the two variables ("samples" and "Itemsets"), or in other words, the proportions of the itemsets in the reference and current samples are identical. H_A : The two proportions are not identical.

To perform the test for the counts in a contingency table with r rows and c columns, we calculate the sum

$$X^2 = \sum_{i=1}^{rc} \frac{(O_i - E_i)^2}{E_i} \qquad \text{(Eq. 5.6)}$$

where O_i is the observed count, and E_i is the expected count. Expected count for cell i is equal to the row total multiplied by the column total divided by the table total. They are given inside brackets in Table 5.3. Using our example problem the p-value is computed to be 0 with 1 (r-1)*(c-1) degrees of freedom. The null hypothesis is **rejected**. Thus, it is concluded that the samples are associated with the itemsets, or in other words the proportions of the itemsets in the reference and current samples are not identical.

We can generalize this test to $c > 2$ to suit our need for comparing the reference sample with the current sample. All other computations are similar. Note that one may argue that we could have used "two proportions test". But the limitation of this test is that only **two** proportions (or in our case, items) can be compared. But our need is to compare multiple (more than two) items. One can also use **Fisher's exact test**. Fisher's exact test produces very similar results to the Chi-square test. It is more suitable when the counts in the contingency table are very small (< 5) for which Chi-square test is not very suitable. But typically data mining applications have very large sizes of data, thus Chi-square test is the best method among all. We did not use any continuity correction (e.g., Yates correction) because the number of degrees of freedom (r-1)*(c-1) is very large. This correction is useful when the number of degrees of freedom is very small, such as 1.

	Itemsets		
Sample	A	B	Total
Reference	30(50)	70(50)	100
Current	70(50)	30(50)	100
Total	100	100	200

Table 5.3: 2 X 2 Contingency table for the example problem. The values inside bracket () are the expected counts.

5.6 Experimental Evaluation II

We conducted extensive experiments to evaluate the performance of ACD using different statistical tests (paired t-test, Wilcoxon signed-rank test and Chi-square test). We used the same synthetic database in our experiments generated using the code from the IBM QUEST project [AS94]. For the experiments, we will use three different data sets (*StreamA*, *StreamB*, and *StreamC*). The parameter settings for the synthetic data generations are as follows:

StreamA: average size of transaction= 20, average size of maximal pattern= 6.

StreamB: average size of transaction= 15, average size of maximal pattern= 7.

StreamC: average size of transaction= 12, average size of maximal pattern= 5.

These data sets contain only quantity of 0 or 1 for each item in a transaction. To suit our need, we randomly generate the quantity of each item in each transaction, ranging from 1 to 5. The external utility table are also synthetically generated by assigning a utility value to each item randomly, ranging from 0.01 to 10.

5.6.1 Test for False Alarm

Before running any experiment, we examine how many false alarms each of the statistical test will generate. Without loss of generality, we equate the time duration to a fixed number of transactions. For the experiment, we set a minimum utility threshold of 1%. We let ACD scan the three data sets using different statistical tests. Each data set contains about 50 million transactions. We set our sample size to be 40k. Thus whenever the current window scan passes 500k transactions, ACD will conduct a test. There will be about 100 tests in total for each run. The result is presented in Table 5.4 where a "Fault" represents a false alarm. It shows that when there is no change in the data distribution, the t-test has the highest probability to trigger a false alarm.

	StreamA	StreamB	StreamC
	Fault	Fault	Fault
t-test	28	35	29
Wilcoxon	2	1	1
Chi-square	5	3	4

Table 5.4: Number of false alarms generated. There are 100 test points for each data set.

5.6.2 Test for Changes

In the experiment, we investigate the ability of ACD to detect changes in the distribution of AHI. To do this, we combined the three data sets into one. ACD is made to scan 50 million transactions. However, after every 1 million transactions, there will be a transition from one data set to another entirely different data set, e.g., *StreamA-StreamB-StreamC-StreamB-StreamA*.... The number of test point remains the same at 100 but out of these 100 tests, 50 are real transitions.

	Hit	Fault
t-test	28	27
Wilcoxon	10	2
Chi-square	47	5

Table 5.5: Experiment to detect change with minimum utility threshold set at 1%. There are 100 test points. Half of them are real transitions and the other half are false alarms.

	Hit	Fault
t-test	35	38
Wilcoxon	8	2
Chi-square	46	6

Table 5.6: Experiment to detect change with minimum utility threshold set at 0.1%. There are 100 test points. Half of them are real transitions and the other half are false alarms.

The experiment is reported in Table 5.5. Here, a "Hit" refers to a success in triggering the alarm whenever there is a real transition. From the table, it is obvious

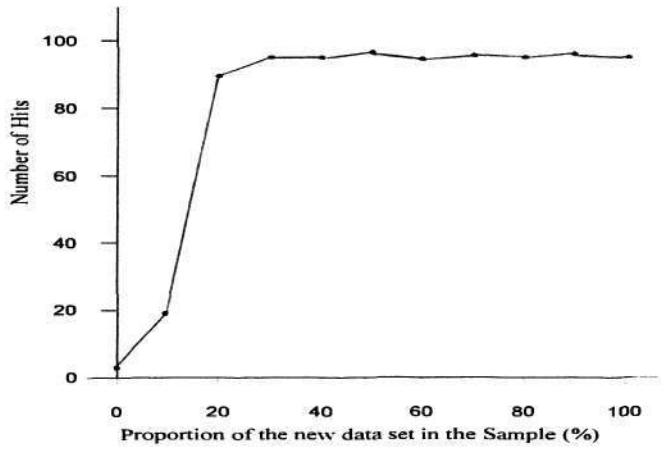
that t-test scores the worst. Although it can score 28 hits, its false alarm rate is too high. Therefore, we cannot conclude if the alarms are triggered by t-test or by random. On the contrary, Wilcoxon test receives the lowest false alarm rate. Unfortunately, it does not respond well to changes. Out of the 50 transitions, it managed to detect only 10. As for Chi-square test, it misses 3 hits and produces 5 false alarms. Clearly, among the three statistical test, Chi-square test is the most sensitive to changes. We let the reference window be fixed

Table 5.6 shows the results of the experiment when the minimum utility threshold is reduced to 0.1%. By lowering the threshold, t-test generates even more false alarms. Only slight changes occur in the other two tests. Based on the overall scores, we are comfortable in saying that when detecting transitional change in data streams, ACD with Chi-square test works reasonably well.

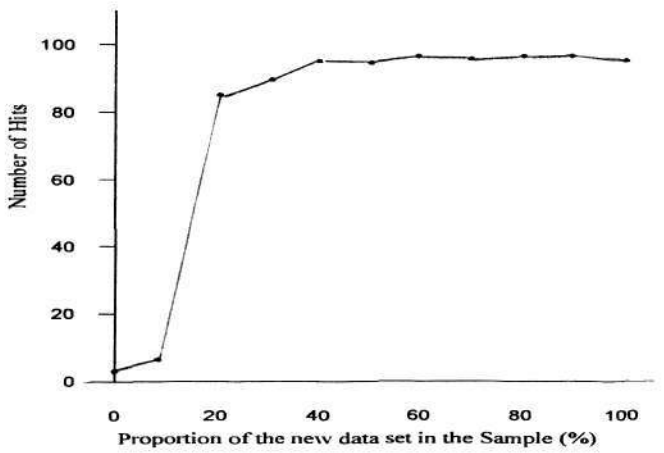
5.6.3 Test for Sensitivity

In the previous experiments, we assume that whenever there is a change, the current window is completely filled up with a new data set. However, during the early transition state, it is possible for the reservoir sample to contain a mixture of two different data sets. It will be interesting to examine how well the Chi-square test reacts to the current window if it is partially mixed with new data set.

For the experiment, we let the reference window contain the sample obtained from *StreamA*. As for the current window, we fill it with a sample having a mixture of a new data set and *StreamA*. In Figure 5.5, the X-axis in the graph represents the proportion of the new data set in the sample. We slowly increase the proportion until the current window is completely occupied by the new data set. For every 10 percent increment, we conduct a Chi-square test. This process is repeated for 100 times. The Y-axis of the graph represents the number of hits the test is able to achieve. In



5.5.a: From *StreamA* to *StreamB*



5.5.b: From *StreamA* to *StreamC*

Figure 5.5: Experiment to examine how fast the Chi-square test react when *StreamA* is slowly replaced by new data set.

Figure 5.5a and 5.5b, both the graphs converge when the proportion reaches 40%. Beyond that point, there is little increase in the number of hits. Note that even when it is at 20%, Chi-square test can still achieve more than 80 hits. This suggests that Chi-square test can be very efficient in early detection of change.

5.7 Summary

Early work in FPM rarely addressed the complex circumstances in which knowledge is extracted and applied. In this chapter, we cover three aspects of utility mining over data stream: mining, sampling and change detection. Utility mining is challenging because it does not have the anti-monotone property that is commonly employed by conventional FPM. The Two-phase algorithm is covered in this chapter. This algorithm "artificially" creates an anti-monotone environment so that redundant itemsets can be massively and efficiently removed. Unfortunately, applying this algorithm to online mining is difficult because the Two-Phase algorithm requires an extra scan of the entire data set to filter the overestimated itemsets in order to obtain the true high utility itemsets. It remains an interesting area for further research.

Although there are many work related to utility mining [WSTZ05, ZWST06], we observe that there has been little development in finding an efficient algorithm to generate high utility itemsets from high speed data stream. We reason this is due to the fact that UM is application driven. Different applications will have different objectives and thus there are several definitions of "useful" itemsets. It may not be possible to achieve a universal definition of UM. In view of this, reservoir sampling is recommended at this level. Sampling allows normal mining algorithms that deal with static data to handle dynamic data (stream). If a small sample size is required, it is advisable to adopt deterministic sampling rather than probabilistic sampling although deterministic sampling is more time consuming. This is demonstrated in the experiment where random sampling produces sample of poor quality when the sampling ratio is small.

A change detector, ACD, is presented in this chapter. ACD incorporates a statistical tool (Chi-square test) and is used to detect significant changes in a data stream. It should be noted that when we use a statistical test to compare two distributions, we are comparing all the itemsets within the histogram in general. There may be some

small group of itemsets whose behavior changes from time to time. If the majority of the itemsets remain stable, the effect caused by this group can be diluted. Therefore, the test may miss out this group. We consider this group as the outlier. We view outliers as an important issue. Future research can focus on identifying and detecting changes on a small group of itemsets.

Lastly, it is our intention to present this chapter to simulate interest in applying UM on data streams. We believe that as many new streaming applications become more mature and popular, the streaming data will become richer. UM will then play a crucial role when conducting advance analysis on data streams.

Chapter 6

Conclusions

Data stream mining is a stimulating field of study that has raised challenges and research issues to be addressed by the database and data mining communities. Streaming data is a computational challenge to data mining problems because of the additional algorithmic constraints created by the large volumes of data. In this dissertation, we have studied the discovery of frequent patterns in the transactional data streams. In particular, we devoted our study to using the two popular approaches of approximate counting and sampling. In Chapter 2, we gave an overview of FPM in data stream, which can be broadly classified into four types of data stream mining models: *landmark window*, *sliding window*, *damped window* and *tilted-time window*. Four algorithms (LCA, CLCA, *DSS* and Algo-Z) in this dissertation were discussed when comparing approximate counting methods with sampling. They have their strengths and weaknesses. For example, (1) LCA can guarantee 100% recall but is slow and hungry for memory if ϵ is small, (2) *DSS* is faster and offers better precision than LCA but cannot guarantee 100% recall, (3) Algo-Z is very fast but unreliable if the sample size is small and cannot handle noise, and (4) CLCA improves the precision for short patterns at the expense of long patterns.

In Chapter 3 we presented CLCA, which is a modified version of the lossy counting algorithm. The algorithm enables user to customize a set of error bounds depending on the cardinality of the patterns. The experimental studies show that the proposed algorithm has improved precision, requires less memory and consumes less CPU time.

In Chapter 4, we considered the use of sampling in the discovery of frequent patterns. Our distance based sampling algorithm (DSS) is able to maintain the quality of the sample on-the-fly in a streaming fashion. This is also achievable even in a noisy environment. The algorithm addressed the technical problem of selecting and rejecting transactions online with an acceptable cost. In Chapter 5, we extended the use of DSS to sample transactions for utility mining. In view of our experimental results in Chapters 4 and 5, we believe that even if the additional cost for online sampling is non-negligible, we can still obtain a significant reduction on the overall running time by using DSS.

Next, data streams can change their behavior over time and, when significant change occurs, much harm is done to the mining result if the change is not properly handled. The ability to detect and characterize change is also essential in many applications, for example intrusion detection, network traffic analysis, data streams from intensive care units etc. Detecting changes is nontrivial. In Chapter 5, an online algorithm, ACD for change detection in utility mining is proposed. In order to provide a mechanism for making quantitative descriptions of the detected change, we adopt the statistical test. Different statistical significance tests are evaluated and our study shows that the Chi-square test is the most suitable for enumerated or count data (as is the case for high utility itemsets).

6.1 Future Work

The work in this dissertation constitutes pioneering research on FPM in data stream. As such, there is a wide scope for further or related research. Below, we identify several specific future research directions.

6.1.1 Outlier

In Chapter 4, we used a distance based sampling approach to identify a good representative set of transactions from a data stream. In our context, we can consider

outlier detection in a similar manner except that we are seeking for the worst representative set of transactions for the database. Outlier detection is well studied in the context of supervised (with class label e.g., classification) and unsupervised (without class label e.g., clustering) data. To the best of our knowledge there has been no significant research on outlier detection in transactional database, e.g. market basket data where each transaction has a number of items and the number of items in each transaction is not constant. Following the definition of outlier by Barnett and Lewis [BL94], we define an outlier transaction as that which appears to deviate markedly from other transactions of the database in which it occurs. This problem is important- for instance a supermarket manager would like to know the outlier transactions so as to make proper decisions. This problem is not trivial particularly when the number of items is large which is often the case in market basket data.

For static data sets, we have proposed a novel and efficient solution DETACH which is different from other works in outlier detection. The proposed method uses a unique approach of creating a representative sample, and then subsequently determining the degree to which each transaction is an outlier considering the representative sample. The proposed method, unlike its predecessors, does not require any tricky parameters to be set. We test our proposed method using benchmark market basket data from QUEST project. Results show that DETACH is very efficient and accurate.

Although our original intention is to apply DETACH on streaming data, we are unable to achieve this due to time constraint. In order not to disturb the flow of this dissertation, we have decided not to cover this topic. Instead, we have included a recently accepted paper in the appendix. This paper will be published in the Journal of Intelligent Data Analysis in late 2010.

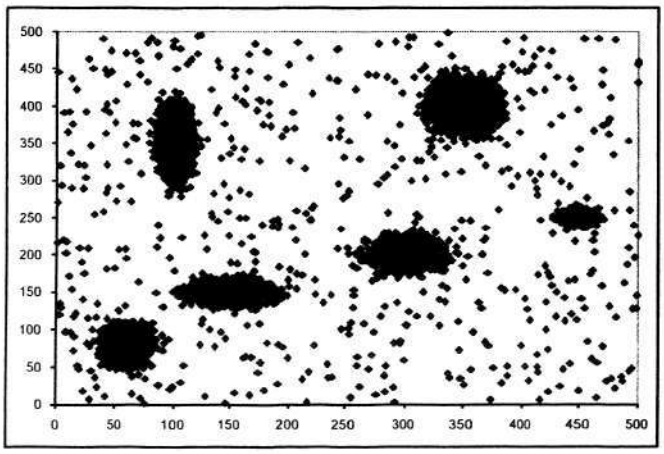
6.1.2 Extension for DSS

One of the interesting area that we can explore is to apply *DSS* on other well known data mining problem such as clustering and classification. Both clustering and clas-

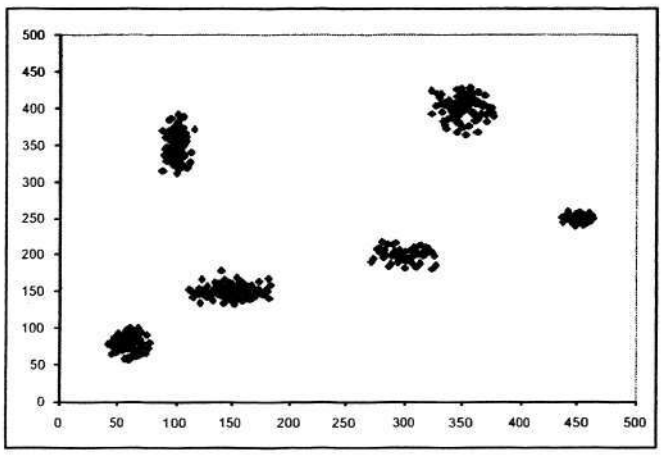
sification are indispensable to practical applications. Therefore, great savings and conveniences to the user could result if this sampling technique can be applied. We shall demonstrate the potential of applying *DSS* on clustering problem. Figure 6.1a shows a 2-D data set with six clusters and some noise. The data set is generated using the code from [DLST03]. This data set has about 6000 data points with different shapes of clusters. These shapes are formed by adjusting the mean and the variance parameter in the Gaussian distribution. For noisy data we add 10% uniformly generated data points as noise.

To use *DSS*, we need to convert the data set into a binary format similar to association rule mining. *DSS* requires "binary valued" input data set where each data point represents an itemset which in turn is a subset of the available set of attributes. Therefore, normalization and discretisation play an important role. However, to simplify our discussion, normalization is omitted in this example. For discretisation, we used the software found in [Coe05]. Here, discretisation is the process of converting the range of possible values associated with a continuous data item into a number of sub-ranges each identified by a unique integer label, and converting all the values associated with instances of this data item to the corresponding integer labels.

Logically, a simple random sampling will generate sample with about 10% of noise. Figure 6.1b shows a sample of 600 data points generated by *DSS*. We see that the noise has been filtered by *DSS*. However, it is still a long way to apply this on streaming environment. For example, discretisation in streaming data is difficult because we may not know how far the value of an attribute can reach and it may not follow a normal distribution. The underlying distribution may shift. Moreover, the complexity also increases if the dimension of the data set increases. We leave all these for future research.



6.1.a: Noisy Data set with six clusters



6.1.b: Sample of 600 data points

Figure 6.1: Simple experiment to study how *DSS* performs on noisy data set.

Appendix A

List of Publications

The work discussed in this dissertation has resulted in 9 international publications. We list these and other contributions of the author during his study in Nanyang Technological University.

Refereed Journal Papers

- **Willie Ng** and MANORANJAN DASH: A Comparison between Approximate Counting and Sampling Methods for Frequent Pattern Mining on Data Stream, accepted for publication in the Journal of Intelligent Data Analysis.
- MANORANJAN DASH and **Willie Ng**: Outlier Detection in Transactional Data, accepted for publication in the Journal of Intelligent Data Analysis.
- **Willie Ng** and MANORANJAN DASH: Discovery of Frequent Patterns in Transactional Data Streams, Transactions on Large Scale Data and Knowledge Centered Systems, 2010, pp. 1-30.

Refereed Conference and Workshop Papers

- **Willie Ng** and MANORANJAN DASH: Which is Better for Frequent Pattern Mining: Approximate Counting or Sampling?, 11th International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2009).
- **Willie Ng** and MANORANJAN DASH: Efficient approximate mining of frequent patterns over transactional data streams, 10th International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2008).

- **Willie Ng** and MANORANJAN DASH: A Change Dector for Mining Frequent Patterns over Evolving Data Stream, IEEE International Conference on Systems, Man, and Cybernetics (SMC 2008).
- **Willie Ng** and MANORANJAN DASH: A Test Paradigm for Detecting Changes in Transactional Data Streams, 13th International Conference on Database Systems for Advanced Applications (DASFAA-2008).
- **Willie Ng** and MANORANJAN DASH: An Evaluation of Progressive Sampling for Imbalanced Data Sets, IEEE ICDM Workshop on Mining Evolving and Streaming Data, 2006.
- MANORANJAN DASH and **Willie Ng**: Efficient Reservoir Sampling for Transactional Data Streams, IEEE ICDM Workshop on Mining Evolving and Streaming Data, 2006.

Other Journal Papers

- QI CAO, MENG-HIOT LIM, JUHUI LI and **Willie Ng**: A context switchable fuzzy inference chip, IEEE Transactions on Fuzzy Systems, Vol. 14, No. 4, 2006, pp. 552-567.
- MENG-HIOT LIM and **Willie Ng**: Iterative genetic algorithm for learning efficient fuzzy rule set, Journal of Artificial Intelligence for Engineering Design, Analysis and Manufacturing, Vol. 17, No. 4, 2004, pp. 335-347.
- MENG-HIOT LIM, QI CAO , JUHUI LI and **Willie Ng**: Evolvable Hardware Using Context Switchable Fuzzy Inference Processor, IEE Proceedings-Computer and Digital Techniques, Vol. 15, No. 4, 2004, pp. 301-311.

Appendix B

Outlier Detection

B.1 Abstract

Outlier detection is studied in the context of supervised (with class label) and unsupervised (without class label – e.g., clustering) data. To the best of our knowledge there has been no study on outlier detection in transactional database, e.g. market basket data where each transaction has a number of items and the number of items in each transaction is not constant. Following the definition of outlier by Barnett and Lewis, 1994, we define an outlier transaction as that which appears to deviate markedly from other transactions of the database in which it occurs. This problem is important, for instance a supermarket manager would like to know the outlier transactions so as to make proper decisions. This problem is not trivial particularly when the number of items is large which is often the case in market basket data. We propose a novel and efficient solution *DETACH* which is different from other work in this area. The proposed method uses a unique approach of creating a representative sample, and subsequently it determines the degree to which each transaction is an outlier considering the representative sample. The proposed method, unlike its predecessors, does not require any tricky parameter to be set. We test our proposed method using benchmark market basket data from QUEST project. Results show that *DETACH* is very efficient and accurate.

B.2 Introduction

Outliers are defined as, "one that appears to deviate markedly from other data objects in the data set in which it occurs", in Barnett and Lewis, 1994 [BL94]. Hence, we need

to first define what is normal behavior. Usually this is specified as a probability model from which data objects are assumed to be drawn. In some methods this probability model is explicit, whereas in others it is implicitly assumed [DS07].

There are numerous applications of outlier detection. Hodge and Austin [HA04] performed a systematic study on outlier detection and gave a list of applications of outlier detection such as fraud detection, loan application processing, intrusion detection, activity monitoring, network performance, fault diagnosis, structural defect detection, and so on. Here, we focus on detecting outliers in transactional market basket data.

In [HA04] outlier detection methods are grouped under three categories, type 1 [RL96], type 2 [Mar01], and type 3 [FP99]. The proposed study falls under the first category of detecting outliers without any knowledge about the data.

Most of the existing methods focus on numerical data sets [PKGF03, KNT00, BKNS00, RRS00]. We are particularly concerned with outliers in transactional market basket data. In [CHYC04] transactional data is clustered. But in the present paper we find outliers in the transactional data.

Outlier in the transactional data is defined as those transactions whose items occur very rarely. This definition is different from the definitions used in [DS07, NK08]. In their work an outlier in the transactional data is one that has low confidence. In [DS07] authors proposed an algorithm to detect anomalies. They defined anomalies as anything that is 'different' from 'normal' behavior. First of all they defined what is 'normal'. In [NK08] defined outlier in which many attribute values are not observed even though they should occur in association with other attribute values in the records. In neither of these works transactional data is used as market basket data to determine the outlier transactions as defined in this paper.

Suppose we are given a set of transactions where number of items in each transaction is *not fixed*. This is the typical scenario in a market basket data. A supermarket manager would like to know which are the outlier transactions. An outlier according to the manager is those transactions that appear very rarely (this indeed tallies with the outlier definition as given in [BL94]. One may think this to be a trivial problem. For example if in a transactional data, the possible number of items is three, and

the first item appears in 80% transactions, the 2^{nd} item appears in 40% transactions, and the 3^{rd} item appears in 2% transactions. Then it will not be difficult to say that those transactions that consists of the 3^{rd} item is an outlier transaction for obvious reasons. But usually number of items in a supermarket are in thousands (in the experimental set up we considering such data sets that have thousands of items). We argue that when the number of items is large it is not a trivial task to determine the outlier transactions. The reason is the frequency of occurrence of items vary, and the transaction length vary. In this paper we propose a novel penalty function that can collate the information and detect the outlier transaction.

In [KNT00, BKNS00, PES01] distance-based, density-based, and cluster-based outlier detection methods are proposed respectively. These methods are typically $O(|D|^2)$ time complexity where D is the input data set because they have to consider each data point and compare it with all other data points (using distance, density, or clustering distance respectively) in order to determine whether it is an outlier. There may be efficient techniques using partitioning and so on to reduce the complexity, but the fact is that each point has to be compared with other points to determine whether it is an outlier.

But *DETACH* is very efficient. In the heart of the algorithm is a sampling method [DS]. This sampling method is based on the classical theme that a sample closely represents the whole data. It is much more accurate in representing the whole data set compared to existing techniques such as simple random sampling (*SRS*). A penalty measure based on the item frequencies is used to select transactions for the sample. The same penalty function is used to detect whether a transaction is an outlier by evaluating its penalty vis-a-vis the selected sample. The proposed sampling method has the unique characteristic of not selecting the outlier objects. Thus, when a repeat scan of the database is performed, clearly the outliers get high penalty compared to the penalty of *bona fide* data objects.

In this context it is worth mentioning the limitations of *SRS*. In *SRS* each object is chosen randomly and entirely by chance, such that each object has the same probability of being chosen at any stage during the sampling process, and each subset of n objects has the same probability of being chosen for the sample as any other subset

of n objects. The first limitation of *SRS* is that, an *SRS* sample may not adequately represent the entire data set due to random fluctuations in the sampling process. This difficulty is particularly apparent at small sample ratios [WGC88] which is the case for very large databases with limited memory. Second, *SRS* is blind towards outliers, i.e., it treats both *bona fide* and outlier data objects similarly. The proportion of outlier in the *SRS* sample and the original data set are almost equal. So, in the presence of outliers performance of *SRS* degrades, often very drastically.

Contributions of this paper include

- introduce the problem of outlier transactions in market basket data, and
- propose a novel and efficient method to detect transactions in the market basket data.

The proposed method does not require any tricky parameters to be set. Later in the experimental section we will show sufficient evidence to convince the reader about it. But in earlier methods such as distance-based, density-based, and cluster-based outlier detection, very tricky parameters need to be set, and the results are very sensitive to such parameter values. For example, if the number of clusters is not set properly for cluster-based method, the outlier detection can be very inaccurate. Also, it is not a trivial task to set the number of clusters correctly particularly when the data is high-dimensional. Similarly, for distance-based outlier detection, the user should select the threshold distance properly, otherwise the outlier detection will be very inaccurate. Similarly, for the density-based outlier detection the number of nearest neighbors is a very tricky yet not-trivial parameter to set.

The rest of the paper is organized as follows. The next section discusses some related work. Section B.4 describes the proposed sampling algorithm. Section B.5 introduces the new outlier detection algorithm *DETACH*. Section B.6 describes the experimental details to show the efficiency of *DETACH* in detecting outliers. Finally, Section B.7 concludes the paper with discussions on future directions.

B.3 Related work

Outlier detection first gained serious attention in research work in the area of statistics. In their studies, many of the technique proposed are based on distribution based model. In such methods, the data distribution is assumed to fit into a model (e.g. Normal, Poisson, etc.) and outliers simply are those that do not fit into their probability distribution. Barnett and Lewis [BL94] discussed these techniques in great detail. While the idea seems reasonable, problems arise when data distributions are unknown, and unfeasible in high dimensional data as in many KDD applications, as the approach are univariate in nature. Assuming ideal data distribution such as normal distribution fails to produce satisfactory result.

In later years, outlier detection techniques can be found in abundance in many areas, including in computational science. In a broad category, outlier detection technique can be classified in the following major groups: distribution-based, distance-based, density-based and clustering-based model. A survey of outlier detection technique can be found in [HA04].

B.3.1 Distribution-based outlier detection

This approach is as what described earlier in statistical community.

B.3.2 Distance based outlier detection

Knorr and Ng [KNT00] proposed the distance based outlier detection using a simple yet intuitive definition: "An object O in a data set D is a $DB(p, T)$ - outlier if at least a fraction p of the data objects in D lies greater than the distance T from O". This technique enjoys a greater flexibility than earlier approach as it requires no assumption of the underlying data distribution, and has moderate computational complexity in high multidimensional.

While the approach has reasonable accuracy, a weakness is that the parameter p fraction and T distance needs to be defined by the user. Distance T is hard to define, and the author suggested a 'trial and error' approach by running it in iteration. Fraction p can be inferred by sorting in descending order the rank of their *outlier-ness* based on the measured distance, but the cutoff point still needs to be user defined.

Perhaps the main issue with the method is with clusters or regions with different densities, as distances in dense regions tend to be much smaller than those in lower densities and vice verse. This algorithm can be considered a global approach as it treat the whole data set to detect outliers without examining local densities.

In terms of complexity and cost computational, this approach is not very efficient, as it takes $O(|D|^2)$ due to each data point nearest neighbor distances calculation, where $|D|$ is the number of data points. The author also proposed a faster approach to it by using cell based grid thus making it run in linear to n, but exponential to the number of dimensions.

Ramaswamy et al [RRS00] improved on the idea and presented an interesting approach based on the calculation the distances of k-neighbor nearest to an object, and rank their outlier-ness based on the distance achieving good accuracy. This method is a local approach as it examines only the object local neighborhood.

B.3.3 Density based outlier detection

In this approach, outlier detection method is based on the local density of an object's neighbor. An interesting aspect is that it is able to identify outliers in clusters with different densities. The most influential paper in this direction is LOF-density [BKNS00].

In this setup, each data point is assigned to a degree for being an outlier. This degree is called LOF (local outlier factor), which is dependent on the local density of its k-nearest neighbor, instead of a global approach as in earlier distance-based outlier method. It requires a single parameter $MinPts$, which defines the number of nearest neighbors of an object, which the author suggest running it in a successive iteration to get the best value in a plot. Due to its nature of mining outlier from local neighborhood, it can miss out outliers whose densities are very close to the neighboring cluster density.

Since then, there have been a few improvements to the algorithm [WWW05]. In Grid-ODF [WWW05], the author proclaimed to have the idea of both density and distance based method in their approach. In their paper, the authors claimed to have improved the accuracy beyond contemporary LOF-density.

B.3.4 Cluster based model outlier detection

In this category, outlier is detected as a by-product of the clustering process. Clustering is an active research in data mining, and most of the clustering algorithms are robust to outliers. Non-robust clustering algorithms tend to be affected by the outlier and the cluster centroid will skew towards the area with high outlier level, while robust ones will not be affected.

Some of the more well-known ones are CLARANS [NH94], DBSCAN [EpKSX96] and CLIQUE [AGGR98]. Outliers detected using these methods are sensitive to the clustering algorithm used, and thus results may vary. Under this approach, outlier can be defined as"small clusters that are far away from other major clusters" [PES01] or "data points that are furthest from the cluster centroid" [LA99], [LLL00].

Since the technique uses clustering approach, it suffers from the same weakness as any clustering algorithms, which is the difficulty in estimating the number of clusters k and other parameters. Another parameter is a threshold to the top-n points of the furthest distances from the cluster center. These parameter needs to be defined by the user, and increases the difficulty of using the method. Despite this, in well defined clusters with reasonable amount of outliers, the technique can produce good results.

B.3.5 Outlier detection in supervised data

In neural network classification, outlier detection algorithms can fall into the category of supervised or unsupervised approach. The earlier technique as described above are considered unsupervised method, as there are no class label to train the algorithm. While there are unsupervised learning in neural network as well, these will not be covered here, as the algorithms explained above are already quite comprehensive.

In supervised learning for outlier detection, under the neural network approaches, model are trained to generalize and are capable of learning complex boundaries. Neural network requires a sizeable amount of data for training as it attempts to fit a surface over an object. Once this is defined, outliers are simply objects that fall outside the surface boundary. Neural network technique tends to suffer from curse of high dimensionality, and benefits from feature selection applied in preprocessing.

Some examples are Multi-Layer Perceptron (MLP), Nairac [NTC+99] and Bishop [Bis94]. MLP interpolate well, but extrapolate poorly, thus it cannot classify objects outside the region boundary.

B.3.6 Outlier detection in transactional data

While there exist multitude of outlier detection methods, traditional approaches including those described earlier deal with continuous and nominal data. There has not been much work on transactional data outlier detection, thus this area poses an active research interest. Transactional data is different from numerical data in the sense that there is no numerical ordering to the item-set under a particular transaction-id: 'yogurt' is not "larger, higher, bigger or better nor more important" than 'milk' numerically. Thus a numerical comparison is not possible. Research in this area is important because anomaly or outlier in transactional data is crucial to examining the behavior of the data set, and to know when things are not as expected.

Igor et al [CSM01] proposed a method of probabilistic mixture model in profiling to infer a most probable profile for each individual. The profile is built based on the time-stamp and item-count purchased. Thus the method is able to predict the item purchase trend for the said individual. In Song and Brown [LB06], an outlier score function (OSF) was proposed based on extremeness of cell. The more uncertainty level of an item is, the more extreme a cell is, and the higher the outlier-ness score OSF will be.

Kontrimas and Verikas [KV06] did an evaluation of the different method in order to identify the outliers in a real-estate transaction using re-sampling by half means, the smallest half volume, the closest distance to the center, ellipsoidal multivariate trimming, minimum volume ellipsoid, minimum scatter determinant, analysis of projection matrix, principal components and residuals, also influence measures, robust regression, and classification methods. Results showed that multilayer perceptron and the principal component analysis based technique achieved the best results.

In their work [DS07, NK08] an outlier in the transactional data is one that has low confidence. In [DS07] authors proposed an algorithm to detect anomalies. They defined anomalies as anything that is 'different' from 'normal' behavior. First of all

they defined what is 'normal'. In [NK08] defined outlier in which many attribute values are not observed even though they should occur in association with other attribute values in the records. In neither of these works transactional data is used as market basket data to determine the outlier transactions as defined in this paper.

B.4 Proposed Sampling Algorithm

In this section we describe the proposed sampling algorithm, and in the next section we use this sampling algorithm to detect outliers. In this section after introducing the notations that are used in this paper, the main methodology for selecting a sample is explained in Section B.4.2. A simple example to illustrate the working of the proposed sampling method is given in B.4.5. A small section B.4.3 is dedicated to explain how the proposed sampling method is successfully avoiding the outliers.

B.4.1 Notations

The notations are mainly based on association rule mining. Denote by D the database of interest, by S a sample drawn without replacement from D, and by I the set of all items that appear in D. Let $N = |D|$, $n = |S|$ and $m = |I|$. Here $|\cdot|$ means the number of data or item in the corresponding data set or itemset. Also denote by $\mathcal{I}(D)$ the collection of itemsets that appear in D; a set of items A is an element of $\mathcal{I}(D)$ if and only if the items in A appear jointly in at least one transaction $t \in D$. If A contains exactly $k(\geq 1)$ elements, then A is sometimes called a k-itemset. In particular, the 1-itemsets are simply the original items. The collection $\mathcal{I}(S)$ denotes the itemsets that appear in S; of course, $\mathcal{I}(S) \subseteq \mathcal{I}(D)$. For $k \geq 1$ we denote by $\mathcal{I}_k(D)$ and $\mathcal{I}_k(S)$ the collection of k-itemsets in D and S, respectively.

For an itemset $A \subseteq I$ and a transactions set T, let $n(A;T)$ be the number of transactions in T that contain A. The support of A in D and in S is given by $f(A;D) = n(A;D)/|D|$ and $f(A;S) = n(A;S)/|S|$, respectively. Given a threshold $s > 0$, an item is frequent in D (resp., in S) if its support in D (resp., in S) is no less than s. We denote by $L(D)$ and $L(S)$ the frequent itemsets in D and S, and $L_k(D)$ and $L_k(S)$ the collection of frequent k-itemsets in D and S, respectively.

Specifically, denote by S^i the set of all transactions in S that contains item A_i, and by r_i and b_i the number of red and blue transactions in S^i respectively. Red means the transactions will be kept in final subsample and blue means the transactions will be deleted. Q is the penalty function of r_i and b_i. f_r denotes the ratio of red transactions, i.e., the sample ratio. Then the ratio of blue transactions is given by $f_b = 1 - f_r$.

B.4.2 Selecting Sample to Minimize Distance

In order to obtain a good representation of a huge database, ε-approximation method is used to find a small subset so that the supports of 1-itemset are close to those in the entire database. The sample S_0 of S is an ε-approximation if its discrepancy satisfies $Dist(S_0, S) \leq \varepsilon$. The discrepancy is computed as the distance of 1-itemset frequencies between any subset S_0 and the superset S. It can based on L^p-norm distances, for example: [?]

$$Dist_1(S_0; S) = \sum_{A \in I_1(S)} |f(A; S_0) - f(A; S))| \quad \text{(Eq. B.1)}$$

$$Dist_2(S_0; S) = \sum_{A \in I_1(S)} (f(A; S_0) - f(A; S))^2 \quad \text{(Eq. B.2)}$$

In this paper $Dist_\infty$ metric is used as the distance metric and the discrepancy is calculated as follows:

$$Dist_\infty(S_0, S) = \max_{A \in I_1(S)} |f(A; S_0) - f(A; S)|. \quad \text{(Eq. B.3)}$$

The concept of ϵ-approximation has been widely used in machine learning. A seminal result of Vapnik and Chervonenkis shows that a random sample of size $O(\frac{d}{\epsilon^2} \log \frac{1}{\epsilon})$, where d is the "VC dimension" (assumed finite) of the data set, is an ϵ approximation. This result establishes the link between random samples and frequency estimations over several items simultaneously.

Each transaction is either selected (colored as red) or rejected (colored as blue) based on a penalty function Q_i for item A_i. A hyperbolic cosine based penalty function is applied. When the sampling ratio $f_r = \frac{|S_0|}{|S|}$ is 0.5, the penalty function has the shape depicted in Figure B.1. Q_i is low when $r_i = b_i$ approximately, otherwise Q_i increases

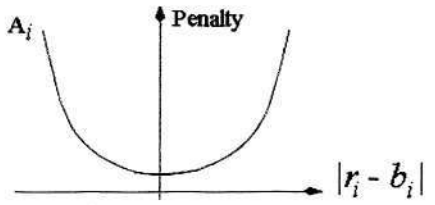


Figure B.1: Penalty as a function of $|r_i - b_i|$.

exponentially in $|r_i - b_i|$. The penalty function Q_i for each item A_i is converted into exponential equivalent shown as follows:

$$Q_i = 2\cosh(\delta_i(r_i - b_i)) \quad \text{(Eq. B.4)}$$

$$= \exp(\delta_i(r_i - b_i)) + \exp(-\delta_i(r_i - b_i)) \quad \text{(Eq. B.5)}$$

In order to directly obtain a sample set of any sample ratio f_r (not just 0.5), set the ratio of red transactions as $r_s = f_r$ and the ratio of blue transactions is $f_b = 1 - r_s$. Then we have

$$r_i = f_r \cdot |S^i| \quad \text{(Eq. B.6)}$$

$$b_i = f_b \cdot |S^i| \quad \text{(Eq. B.7)}$$

So

$$\frac{r_i}{f_r} = \frac{b_i}{f_b} = |S^i| \quad \text{(Eq. B.8)}$$

$$\frac{r_i}{2f_r} + \frac{b_i}{2f_b} = |S^i| \quad \text{(Eq. B.9)}$$

The penalty function Q_i tries to minimize:

$$\left|\frac{r_i}{2f_r} - \frac{b_i}{2f_b}\right| \quad \text{(Eq. B.10)}$$

instead of $|r_i - b_i|$.

The modified penalty function Q_i for each item A_i is

$$Q_i = 2\cosh(\delta_i(\frac{r_i}{2f_r} - \frac{b_i}{2f_b})) \quad \text{(Eq. B.11)}$$

$$= \exp(\delta_i(\frac{r_i}{2f_r} - \frac{b_i}{2f_b})) + \exp(-\delta_i(\frac{r_i}{2f_r} - \frac{b_i}{2f_b})) \quad \text{(Eq. B.12)}$$

As for a small δ_i, $\exp(\delta_i)$ and $\exp(-\delta_i)$ can be replaced by $(1 + \delta_i)$ and $(1 + \delta_i)$, Equation Eq. B.12 can be modified to:

The modified penalty Q_i of j-th transaction will be:

$$Q_i = Q_i^{(j)} = Q_{i,1}^{(j)} + Q_{i,2}^{(j)} \quad \text{(Eq. B.13)}$$

$$Q_{i,1}^{(j)} = (1+\delta_i)^{\frac{r_i}{2f_r}}(1-\delta_i)^{\frac{b_i}{2f_b}} \quad \text{(Eq. B.14)}$$

$$Q_{i,2}^{(j)} = (1-\delta_i)^{\frac{r_i}{2f_r}}(1+\delta_i)^{\frac{b_i}{2f_b}} \quad \text{(Eq. B.15)}$$

where $Q_i^{(j)}$ means the penalty of ith item in jth transaction and δ_i controls the steepness of the penalty plot. The initial values of $Q_{i,1}$ and $Q_{i,2}$ are both 1.

Suppose the $(j+1)$-th transaction is colored as r (or b), the corresponding penalty function $Q_i^{(j||r)}$ (or $Q_i^{(j||b)}$) [1]:

$$\begin{aligned} Q_{i,1}^{(j||r)} &= (1+\delta_i)^{\frac{r_i+1}{2f_r}}(1-\delta_i)^{\frac{b_i}{2f_b}} \\ &= (1+\delta_i)^{\frac{1}{2f_r}}(1+\delta_i)^{\frac{r_i}{2f_r}}(1-\delta_i)^{\frac{b_i}{2f_b}} \\ &= (1+\delta_i)^{\frac{1}{2f_r}} Q_{i,1}^{(j)} \end{aligned} \quad \text{(Eq. B.16)}$$

$$\begin{aligned} Q_{i,2}^{(j||r)} &= (1-\delta_i)^{\frac{1}{2f_r}} Q_{i,2}^{(j)} \\ Q_{i,1}^{(j||b)} &= (1-\delta_i)^{\frac{1}{2f_b}} Q_{i,1}^{(j)} \\ Q_{i,2}^{(j||b)} &= (1+\delta_i)^{\frac{1}{2f_b}} Q_{i,2}^{(j)} \end{aligned} \quad \text{(Eq. B.17)}$$

The computation process of $Q_{i,1}^{(j||r)}$ is given in Equation Eq. B.16. Other penalty functions are computed with a similar procedure and the results are shown in Equation Eq. B.17.

The penalty function of current transaction is the summation of penalties for all items. If $Q^{(j||b)} = \sum_i Q_i^{(j||b)}$ is less than $Q^{(j||r)} = \sum_i Q_i^{(j||r)}$, the $(j+1)$-th transaction will be colored blue and rejected. Otherwise, it will be colored red and added to the sample.

An important observation is that $Q_i^{(j||r)} + Q_i^{(j||b)} = 2Q_i^{(j)}$, and so when summing over all items, by linearity $Q^{(j||r)} + Q^{(j||b)} = 2Q^{(j)}$. It follows that

$$\min\left(Q^{(j||r)}, Q^{(j||b)}\right) \leq \frac{1}{2}\left(Q^{(j||r)} + Q^{(j||b)}\right) = Q^{(j)}.$$

This means that the coloring chosen by the algorithm *never* increases the penalty!

By the symmetry of the penalty function, and since $\frac{r_i}{2f_r} + \frac{b_i}{2f_b} = |S^i|$, we have

$$Q_i^{(final)} \geq (1+\delta_i)^{\left|\frac{r_i}{2f_r} - \frac{b_i}{2f_b}\right|}(1-\delta_i^2)^{|S^i|}$$

[1] Note that $j||r$ denotes $(j+1)^{th}$ transaction being painted red, i.e., transaction j followed by the next ($||$) transaction which is painted red.

From this and the fact that $Q^{(init)} = 2m$, we get that the final overall penalty is at most $2m$, hence $Q_i^{(final)} \leq 2m$ for every i, which in turns implies that

$$\left|\frac{r_i}{2f_r} - \frac{b_i}{2f_b}\right| \leq \frac{\ln(2m)}{\ln(1+\delta_i)} + \frac{|S^i|\ln(1/(1-\delta_i)^2)}{\ln(1+\delta_i)}.$$

We can choose the value of δ_i to make the right-hand side as small as possible. The first (resp., second) term in the sum is decreasing (resp. increasing) in δ_i, and so a reasonable choice is to balance the two terms, leading to

$$\delta_i = \sqrt{1 - \exp\left(-\frac{\ln(2m)}{S^i}\right)}. \qquad \text{(Eq. B.18)}$$

For that value, it can be shown that the final error on item i is at most $O(\sqrt{|S_i|\log(2m)})$ which, if S^i is not too small, is much smaller than $|S^i|$.

In order to guarantee that there are $f_r * |S|$ number of red transactions, we can add a (fictitious) item A_0 that is contained in all transactions. In that case, $\frac{S_r}{2f_r} - \frac{S_b}{2f_b}$ is also $O(\sqrt{n\log(2m)})$, and this implies that $|S_r| = n * f_r + O(\sqrt{n\log(2m)})$. Since $f(A_i; S_r) = r_i/|S_r| = r_i/(n * f_r + O(\sqrt{n\log(2m)}))$, and $f(A_i; S) = (r_i + b_i)/n$, from the above bounds on $\left|\frac{r_i}{2f_r} - \frac{b_i}{2f_b}\right|$ we get that for each i,

$$|f(A_i; S_r) - f(A_i; S)| \leq \epsilon(n, m) = O(\sqrt{\log(2m)/n}). \qquad \text{(Eq. B.19)}$$

In practice, the proposed method will work with any choice of δ_i, but the bounds on $|f(A_i; S_r) - f(A_i; S)|$ will not necessarily be guaranteed. In our implementation, we have found that setting

$$\delta_i = \sqrt{1 - \exp\left(-\frac{\ln(2m)}{n}\right)} \qquad \text{(Eq. B.20)}$$

is very effective. The advantage is that if the defining parameters of the database, i.e., the number n of transactions and number m of items, are already known then the proposed method requires a single scan of the database. It also guarantees that $\left|\frac{r_i}{2f_r} - \frac{b_i}{2f_b}\right|$ is at most $O(\sqrt{n\log(2m)})$ for any item, which is not as strong as $O(\sqrt{|S_i|\log(2m)})$, but still strong enough to guarantee the upper bound of (Eq. B.19).

B.4.3 Handling outliers

As SRS randomly selects the sample set, the percentage of outliers in the sample set is almost the same as the original data when the experiment repeats for several times. For the proposed sampling method, from Equation Eq. B.16 and Equation Eq. B.17, it can be seen that as the sample ratio f_r is usually very small, i.e.,$f_r \ll f_b$, and $Q_i^{(j||b)}$ is smaller than $Q_i^{(j||r)}$ in the beginning. That means only after painting sufficient number of transactions as blue, the proposed sampling method paints some transactions as red. For the data with noisy items, their corresponding frequency are very low. In the red sample these outlier transactions are usually not selected because first of all the blue sample has to be considerably filled with such outlier transactions for even one such outlier transaction to be painted red. But for usual *bona fide* items the corresponding frequency is maintained, thus the proposed sampling algorithm retains *bona fide* transactions and removes the outlier transactions. Usually in outlier detection one needs to input some threshold which is not easy to determine off-hand. But, note that the proposed sampling algorithm removes the outliers without any user given threshold.

B.4.4 Proposed Sampling Algorithm

The complete sampling algorithm is given below. The penalty for each item i of a transaction is calculated only once. To store the penalties memory required is $O(m)$. The time for processing one transaction is bounded by $O(T_{max})$ for the proposed method where T_{max} denotes the maximal transaction length in T. Thus, the proposed method is independent of sample size.

Algorithm 1: The Proposed Sampling Method

Input: D, n, m, f_r

Output: S_0, the transactions in red color

For each item i in D

State $\delta_i = \sqrt{1 - \exp\left(-\frac{\ln(2m)}{n}\right)}$;

State $Q_{i,1} = 1$;

State $Q_{i,2} = 1$;

EndFor

State $f_b = 1 - f_r$;

Foreach transaction j in D

State color transaction j red;

State $Q^{(r)} = 0$;

State $Q^{(b)} = 0$;

Foreach item i contained in j

State $Q_{i,1}^{(r)} = (1 + \delta_i)^{\frac{1}{2f_r}} Q_{i,1}$;

State $Q_{i,2}^{(r)} = (1 - \delta_i)^{\frac{1}{2f_r}} Q_{i,2}$;

State $Q_{i,1}^{(b)} = (1 - \delta_i)^{\frac{1}{2f_b}} Q_{i,1}$;

State $Q_{i,2}^{(b)} = (1 + \delta_i)^{\frac{1}{2f_b}} Q_{i,2}$;

State $Q^{(r)} += Q_{i,1}^{(r)} + Q_{i,2}^{(r)}$;

State $Q^{(b)} += Q_{i,1}^{(b)} + Q_{i,2}^{(b)}$;

EndFor

If $Q^{(r)} < Q^{(b)}$

State $Q_{i,1} = Q_{i,1}^{(r)}$;

State $Q_{i,2} = Q_{i,2}^{(r)}$;

Else State color transaction j blue;

State $Q_{i,1} = Q_{i,1}^{(b)}$;

State $Q_{i,2} = Q_{i,2}^{(b)}$;

EndIf

Iftransaction j is red

State set $S_0 = S_0 + \{j\}$;

EndIf

EndFor

B.4.5 Illustration of The Proposed Sampling Algorithm Using A Simple Example

In this section the proposed sampling algorithm is illustrated using a simple example. Consider that there are only two binary attributes and each data object has the value

Table B.1: An illustration to show the working of the proposed sampling algorithm

TID	Transaction	Penalty Red	Penalty Blue
1	1	3.384186	1.983128
2	1	2.930321	2.008347
3	2	4.756560	1.966732
4	1	2.554856	2.074596
5	1	2.246980	2.181708
6	2	3.756361	2.016286
7	**1**	**1.997704**	**2.330380**
8	1	5.217903	1.799605
9	2	2.983778	2.144547
10	1	4.442370	1.646608
11	2	2.390372	2.350921
12	1	3.788997	1.533810
13	1	3.239510	1.457335
14	1	2.778497	1.414213
15	1	2.392981	1.402283
16	1	2.072050	1.420116
17	**2**	**1.938617**	**2.638050**
18	2	8.045688	1.599555
19	2	6.289355	1.350980
20	2	4.920177	1.176028
21	1	1.806556	1.466961
22	1	1.588855	1.542701
23	**1**	**1.412590**	**1.647828**
24	1	3.689615	1.272513
25	1	3.141231	1.164327
26	2	3.853517	1.062094
27	2	3.023367	1.000000
28	1	2.679226	1.084567
29	1	2.290679	1.030492
30	1	1.964694	1.000000

of '1' for only one of the two attributes, i.e., in association rule mining terms, there are only two items (item 1 and item 2) and each transaction either has item 1 or item 2. Although the proposed algorithm considers all the attributes simultaneously by adding their penalties, it will be more tedious to illustrate using more attributes. There are 30 transactions (1,1,2,1,1,2,1,1,2,1,2,1,1,1,1,1,2,2,2,2,1,1,1,1, 1,2,2,1,1,1) with ten 2's and twenty 1's. Sampling ratio is 10%. Ideally the sample should consist of two 1's and one 2. Output of the proposed sampling algorithm is (1,2,1). The penalties for painting the transaction as red or blue are shown in Table B.1.

When the penalty for painting it red is less than penalty for painting it blue, the transaction is selected in the final sample, otherwise it is not selected. Note that the blue bin would ideally hold 90% of the transactions and the red bin should hold 10% of the transactions. Furthermore, as the original data has one-third 2's and two-third 1's, the blue and red bins should also hold similar portions of 1's and 2's. In the table the bold face entry indicates that the corresponding transaction is selected (painted as red, i.e., penalty for painting red is less than penalty for painting blue). So, the transactions 7, 17, and 23 are selected and the rest 27 transactions are not selected. Notice that the proposed sampling method selected two 1's and one 2's (it is the correct representation of the input data). When the algorithm begins with the transaction # 1, considering the fact that the sampling ratio is 10%, $f_r = 0.1$ and $f_b = 0.9$. In Equation Eq. B.17, although i can take two values (1 and 2), but because there is only one item in each transaction, i takes only one value. So δ_i value is initialized using Equation Eq. B.20. At the start, $Q_{i,1}$ and $Q_{i,2}$ are each set to 1. As $f_r << f_b$ in Equation Eq. B.17, so total penalty $Q_{i,1} + Q_{i,2}$ will be much smaller if the transaction is painted blue than if it is painted red.

Notice that in Equation Eq. B.17, $Q_{i,1}$ is multiplied by $(1+\delta_i)^{\frac{1}{2f_r}}$ while calculating the penalty for painting the transaction red. But $Q_{i,1}$ is multiplied by $(1-\delta_i)^{\frac{1}{2f_b}}$ while calculating the penalty for painting the transaction blue. On the other hand, $Q_{i,2}$ is multiplied by $(1-\delta_i)^{\frac{1}{2f_r}}$ while calculating the penalty for painting the transaction red. But $Q_{i,2}$ is multiplied by $(1+\delta_i)^{\frac{1}{2f_b}}$ while calculating the penalty for painting the transaction blue. Notice that $(1+\delta_i)^{\frac{1}{2f_r}}$ is larger than $(1-\delta_i)^{\frac{1}{2f_r}}$. And also notice that $(1+\delta_i)^{\frac{1}{2f_b}}$ is larger than $(1-\delta_i)^{\frac{1}{2f_b}}$. The net effect of this is that if transactions are painted blue in a sequence (as is the case here: first six transactions are painted blue), the $Q_{i,1}$ eventually decreases and $Q_{i,2}$ increases. This tilts the total penalty in favor of red as compared to blue for the transaction#7. As soon as a transaction is painted red, the new $Q_{i,1}$ increases (thus favoring the next transaction to be painted blue) and $Q_{i,2}$ decreases (thus favoring the next transaction to be painted blue).

B.5 *DETACH* Outlier Detection Algorithm

After selecting a sample using the proposed sampling algorithm described in the previous section, the database is scanned for the second time to detect the outliers. In the second scan transactions are not included in the sample, but the penalty for including it in the red bin is calculated. It is a dummy run through the database to detect outliers based on the selected sample.

Algorithm 2: *DETACH* Outlier Detection Method

Input: $D, n, m, f_r, outlierPercent$

Output: Selected outliers

Perform Algorithm 1 and select a sample S_0;

State $f_b = 1 - f_r$;

Foreach transaction j in D

  State $Q^{(r)} = 0$;

  State $Q^{(b)} = 0$;

  Foreach item i contained in j

    State $Q_{i,1}^{(r)} = (1 + \delta_i)^{\frac{1}{2f_r}} Q_{i,1}$;

    State $Q_{i,2}^{(r)} = (1 - \delta_i)^{\frac{1}{2f_r}} Q_{i,2}$;

    State $Q^{(r)} += Q_{i,1}^{(r)} + Q_{i,2}^{(r)}$;

  EndFor

  State $outlierPenaltyArray[j] = Q^{(r)}$

EndFor

Sort $outlierPenaltyArray$ in descending order;

Select transactions corresponding to the $outlierPercent * |D|$ highest outlier values from $outlierPenaltyArray$;

In *DETACH* first of all the proposed sampling algorithm selects the sample. Note that there is no need of storing the selected transactions of the sample. Only the modified penalty values for each item ($Q_{i,1}$, and $Q_{i,2}$) are used to calculate the final penalty

values for each transaction. Then the second scan is performed and the penalty values for each transaction is computed without including any more transaction into the sample, in other words without modifying the penalty values for each item ($Q_{i,1}$, and $Q_{i,2}$). Only penalty for red is computed because it is unnecessary to compute the penalty for blue. The penalty values are stored in the $outlierPenaltyArray$ in descending order, and transactions corresponding to $outlierPercent * |D|$ top penalty values are selected as outliers. Time complexity of *DETACH* is $O(|D|)$.

B.6 Performance Study

This section describes the experiments conducted in order to determine the effectiveness of *DETACH*. All experiments were performed on a 3.50GHz CPU Dell PC with 3GB of main memory and running on the Windows XP platform. The algorithm is written in C++. We use the benchmark data sets in our experiments and they were generated using the code from the IBM QUEST project [AS94]. The nomenclature of these data sets is of the form $TxxIyyDzzK$, where xx refers to the average number of items present per transaction, yy refers to the average size of the maximal potentially frequent itemsets and zz refers to the total number of transactions in $K(1000$'s). Items were drawn from a universe of $\mathcal{I} = 10k$ unique items. The two data sets that we used are $T10I4D100K$ and $T15I7D100K$.

Outliers are added using the *rand* function. We added 1%, 5%, 7% and 10% of outliers. Consider the "5% outliers": In one scan 5% transactions are randomly selected and their items are randomly changed from a range of items. We varied the range from 0 to 20k.

In Figures B.2 and B.3 we showed the distribution of the items in $T10I4D100K$ before and after adding outliers. Both follow Zipf distribution. This is to ensure that after adding outliers the distribution remains as Zipf.

To run *DETACH* we need to set the sample ratio for the sampling algorithm. We varied the sample ratios as 5%, 10% and 20%. *DETACH* outputs the possible outliers and they are compared with the true outliers. Thus, we obtain the accuracy of *DETACH*.

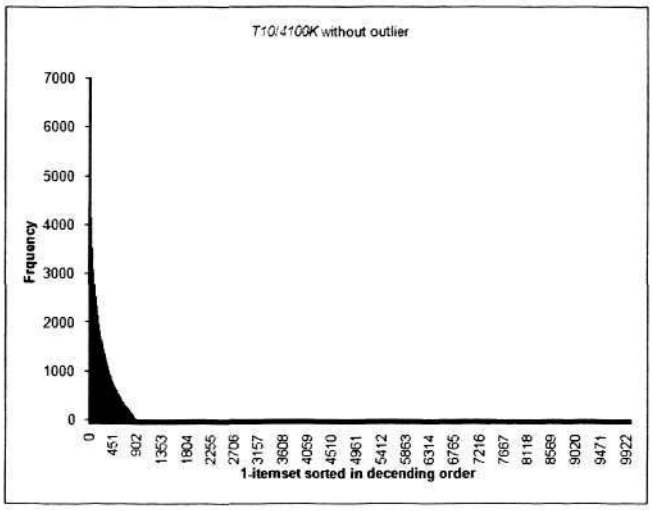


Figure B.2: Zipf distribution before adding outliers

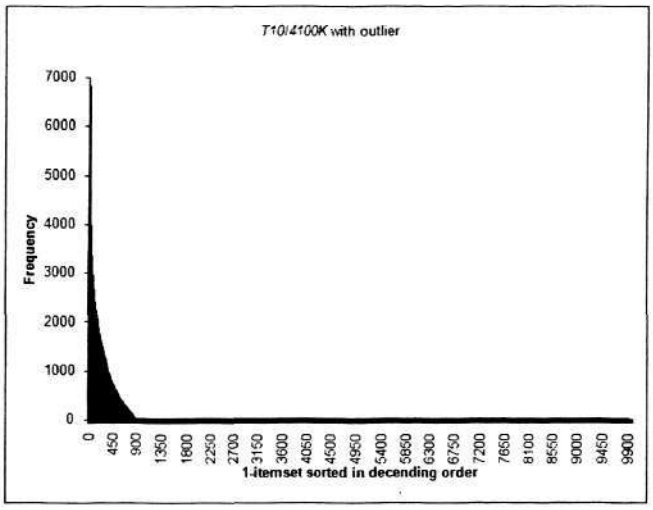


Figure B.3: Zipf distribution after adding outliers

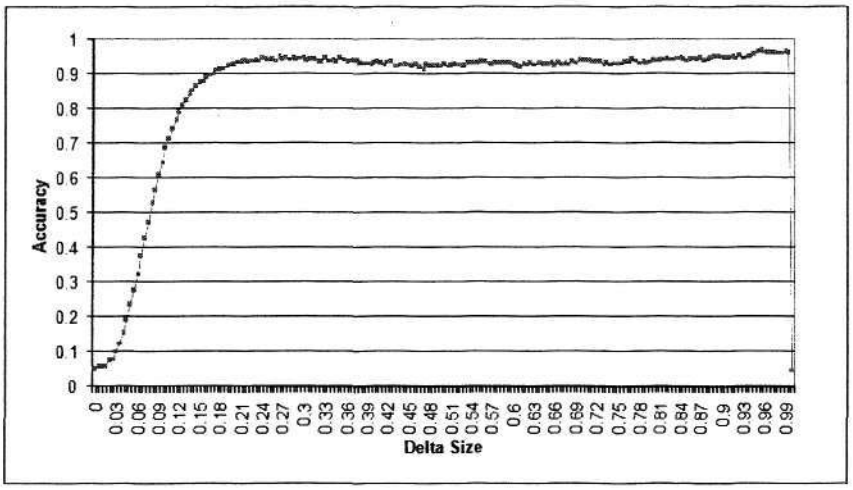


Figure B.4: Accuracy vs. δ

First of all we run an experiment to see the effect of δ upon the accuracy of *DETACH*. Figure B.4 shows that except for very small values of δ, the performance of *DETACH* does not vary with δ. Thus, in the following experiments we set δ values for each item to 0.25.

In Figures B.5 and B.6 the accuracy of *DETACH* over the two data sets are shown. *For most data sets the accuracy is close to 100%.* The accuracy for smaller sample size is lower than the accuracy for larger sample size. Thus 5% sample size scores the lowest accuracy whereas 20% sample size scores the highest accuracy, and the accuracy for 10% sample size is in between these two. These results are consistent for different percentage of outliers and for the two different data sets.

When the outlier percentage level increases the accuracy dips slightly in most cases. But for 5% sample size the accuracy dips significantly particularly for $T10I4D100K$ data set. We conclude that 5% sample size is too low for these experiments. 20% sample size scores more than 90% accuracy for all cases and thus we recommend the user to use 20% sample size.

Among the two data sets, $T15I7D100K$ is more dense than $T10I4D100K$. Its accuracy values are higher than the accuracy values for $T10I4D100K$ particularly when the outlier percentage is the highest, i.e., 10%. We reason it as follows. $T15I7D100K$

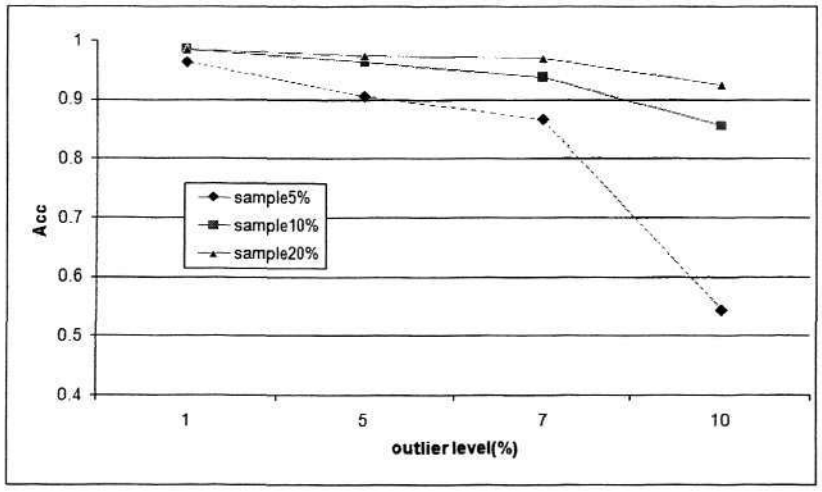


Figure B.5: Performance of DETACH on $T10I4D100K$

is more dense than $T10I4D100K$ because each transaction has the average length of 15 compared to 10 for $T10I4D100K$. Thus each item has higher frequency in $T15I7D100K$ than that in $T10I4D100K$. So, when the outlier percentage is the highest, i.e., 10%, *DETACH* could detect the outliers for the dense data set more accurately than the less dense data set. When the outlier percentage is high, it becomes more difficult to distinguish between outlier and *bona fide* transactions particularly for the less dense data set $T10I4D100K$.

B.7 Conclusion and Future Directions

In this paper we proposed a new problem of detecting outliers in transactional data. We solved this problem by a unique method *DETACH* where in the first phase a representative sample is selected from the whole data set and in the second phase the degree of outlier-ness for each transaction in the whole database is determined. The results showed that *DETACH* works very well. For most data sets the accuracy is close to 100%. The variations in the accuracy with respect to sample percentage and outlier percentage are as per common sense.

An immediate future direction is to test the effectiveness of *DETACH* over stream data. *DETACH* is linear in computational complexity which is an essential require-

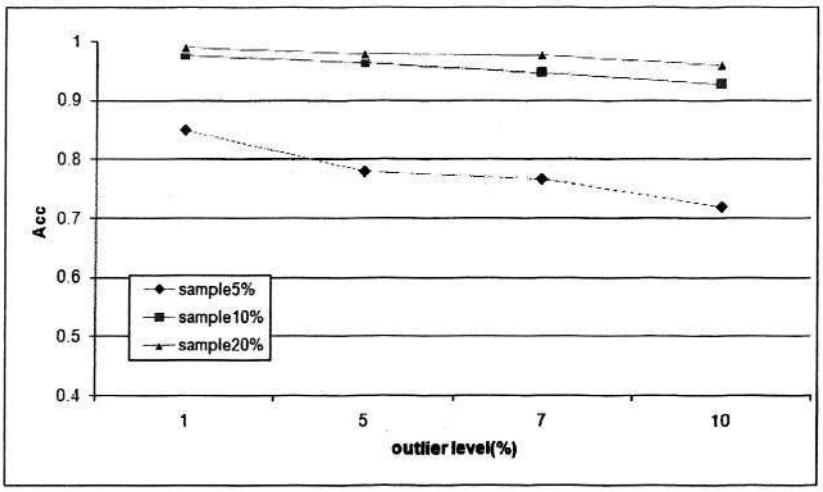


Figure B.6: Performance of DETACH on $T15I7D100K$

ment for stream data processing that has potentially infinite data size. The second scan that we performed in *DETACH* is not necessary for stream data. Instead, the sample is selected based on the already processed data, and the incoming data is tested whether it is an outlier. After testing its outlier-ness, it is considered for inclusion into the sample.

References

[Agg06] Charu C. Aggarwal. On biased reservoir sampling in the presence of stream evolution. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 607–618, 2006.

[Agg07] Charu C. Aggarwal. *Data Streams: Models and Algorithms.* Springer, 2007.

[AGGR98] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *Proceedings of ACM SIGMOD Conference on Management of Data*, 1998.

[AGP00] Swarup Acharya, Phillip B. Gibbons, and Viswanath Poosala. Congressional samples for approximate answering of group-by queries. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 487–498, 2000.

[AHWY03] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. A framework for clustering evolving data streams. In *Proceedings of 29th International Conference on Very Large Data Bases*, pages 81–92, 2003.

[AHWY04] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. On demand classification of data streams. In *Proceedings of the Tenth ACM SIGKDD International Conference in Knowledge Discovery and Data Mining*, pages 503–508, 2004.

[AIS93] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In Peter Buneman and

Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD international conference on Management of Data (SIGMOD'93)*, pages 207–216, 1993.

[AMS96] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *Journal of Computer and system sciences*, volume 58, pages 20–29, 1996.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proceedings of 20th International Conference on Very Large Data Bases*, pages 487–499, 1994.

[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.

[AY06] Charu C. Aggarwal and Philip Yu. A survey of synopsis construction in data streams. *Data Streams: Models and Algorithms*, pages 169–208, 2006.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–16, 2002.

[BCD+03] Herve Bronnimann, B. Chen, M. Dash, P. Haas, and P. Scheuermann. Efficient data reduction with ease. In *Proceedings of the Ninth ACM SIGKDD International Conference in Knowledge Discovery and Data Mining*, pages 59–68, 2003.

[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, 2001.

[BDF+97] Daniel Barbar'a, William Dumouchel, Christos Faloutsos, Peter J. Haas, Joseph M. Hellerstein, Yannis Ioannidis, H. V. Jagadish, Theodore Johnson, Raymond Ng, Viswanath Poosala, Kenneth A. Ross, and Kenneth C. Sevcik. The new jersey data reduction report. *IEEE Data Engineering Bulletin*, 20:3–45, 1997.

[BDM02] Brian Babcock, Mayur Datar, and Rajeev Motwani. Sampling from a moving window over streaming data. In *Proceedings of 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2002.

[BH03] Brock Barber and Howard J. Hamilton. Extracting share frequent itemsets with infrequent subsets. *Data Mining and Knowledge Discovery*, 7(2):153–185, 2003.

[Bis94] Chris M. Bishop. Novelty detection and neural network validation. In *IEE Proceedings on Vision, Image and Signal Processing*, pages 217–222, 1994.

[BKNS00] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Sander. LOF: identifying density-based local outliers. In *SIGMOD 2000 International Conference on Management of Data*, pages 93–104, 2000.

[BL94] Vic Barnett and Toby Lewis. *Outliers in statistical data*. Wiley Series in Probability & Statistics. Wiley, April 1994.

[BMS97] Sergey Brin, Rajeev Motwani, and Craig Silverstein. Beyond market baskets: Generalizing association rules to correlations. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 265–276, 1997.

[BMUT97] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, and Shalom Tsur. Dynamic itemset counting and implication rules for market basket data. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data*, pages 255–264, 1997.

[Bod03] Ferenc Bodon. A fast apriori implementation. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, 2003.

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

[CCP+04] Y. Dora Cai, David Clutter, Greg Pape, Jiawei Han, Michael Welge, and Loretta Auvil. Maids: Mining alarming incidents from data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 919–920, 2004.

[CDG07] Toon Calders, Nele Dexters, and Bart Goethals. Mining frequent itemsets in a stream. In *IEEE International Conference on Data Mining (ICDM07)*, pages 83–92, 2007.

[CDH+02] Yixin Chen, Guozhu Dong, Jiawei Han, Benjamin W. Wah, and Jianyong Wang. Multi-dimensional regression analysis of time-series data streams. In *Proceedings of 28th International Conference on Very Large Data Bases*, pages 323–334, 2002.

[CF02] Sirish Chandrasekaran and Michael J. Franklin. Streaming queries over streaming data. In *Proceedings of 28th International Conference on Very Large Data Bases*, pages 203–214, 2002.

[CFCK98] Chun Hing Cai, Ada Wai-Chee Fu, C. H. Cheng, and W. W. Kwong. Mining association rules with weighted items. In *International Database Engineering and Application Symposium*, pages 68–77, 1998.

[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, 2000.

[CHS02] Bin Chen, Peter J. Haas, and Peter Scheuermann. A new two-phase sampling based algorithm for discovering association rules. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 462–468, 2002.

[CHYC04] Kun-Ta Chuang Ching-Huang Yun and Ming-Syan Chen. Adherence clustering: an efficient method for mining market-basket clusters. *Information Systems*, 31(3):170–186, 2004.

[CKN06] James Cheng, Yiping Ke, and Wilfred Ng. Maintaining frequent itemsets over high-speed data streams. In *Proceedings of the 10th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 462–467, 2006.

[CKN08] James Cheng, Yiping Ke, and Wilfred Ng. A survey on algorithms for mining frequent itemsets over data streams. *An International Journal of Knowledge and Information Systems*, 2008.

[CL03a] Joong Hyuk Chang and Won Suk Lee. *estWin*: adaptively monitoring the recent change of frequent itemsets over online data streams. In *CIKM*, pages 536–539, 2003.

[CL03b] Joong Hyuk Chang and Won Suk Lee. Finding recent frequent itemsets adaptively over online data streams. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 487–492, 2003.

[CL04] Joong Hyuk Chang and Won Suk Lee. A sliding window method for finding recently frequent itemsets over online data streams. *Journal of Information Science and Engineeering*, 20(4):753–762, 2004.

[Coc77] William Gemmell Cochran. *Sampling techniques*. John Wiley & Sons, 1977.

[Coe05] Frans Coenen. The LUCS-KDD Data Discretisation/Normalisation (DN) Java Software for Classification Association Rule Mining, http://www.csc.liv.ac.uk/~frans/kdd/software/lucs-kdd-dn/lucs-kdd_dn.html, 2005.

[CS03] Edith Cohen and Martin Strauss. Maintaining time-decaying stream aggregates. In *Proceedings of the Twenty-Second ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 223–233, 2003.

[CSM01] Igor V. Cadez, Padhraic Smyth, and Heikki Mannila. Probabilistic modeling of transaction data with applications to profiling, visualization, and prediction. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 37–46, 2001.

[CWYM04] Yun Chi, Haixun Wang, Philip S. Yu, and Richard R. Muntz. Moment: Maintaining closed frequent itemsets over a stream sliding window. In *Proceedings of the 4th IEEE International Conference on Data Mining (ICDM 2004)*, pages 59–66, 2004.

[CYS03] Raymond Chan, Qiang Yang, and Yi-Dong Shen. Mining high utility itemsets. In *Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM 2003)*, pages 19–26, 2003.

[DGGR02] Alin Dobra, Minos N. Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Processing complex aggregate queries over data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 61–72, 2002.

[DH00] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *Proceedings of ACM SIGKDD International Conference in Knowledge Discovery and Data Mining*, pages 71–80, 2000.

[DH01] Pedro Domingos and Geoff Hulten. Catching up with the data: Research issues in mining data streams. In *Workshop on Research Issues in Data Mining and Knowledge Discovery*, 2001.

[DLOM02] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Frequency estimation of internet packet streams with limited space. In *Proceedings of the 10th Annual European Symposium on Algorithms*, pages 348–360, 2002.

[DLST03] Manoranjan Dash, Huan Liu, Peter Scheuermann, and Kian-Lee Tan. Fast hierarchical clustering and its validation. *Data and Knowledge Engineering*, 44(1):109–138, 2003.

[DN06] Manoranjan Dash and Willie Ng. Efficient reservoir sampling for transactional data streams. In *IEEE ICDM workshop on Mining Evolving and Streaming Data*, pages 662–666, 2006.

[DN10] Manoranjan Dash and Willie Ng. Outlier detection in transactional data. *Intelligent Data Analysis*, 2010.

[DNO06] Xuan Hong Dang, Wee Keong Ng, and Kok-Leong Ong. Adaptive load shedding for mining frequent patterns from data streams. In *Data Warehousing and Knowledge Discovery, 8th International Conference, DaWaK 2006*, pages 342–351, 2006.

[DNOL07] Xuan Hong Dang, Wee Keong Ng, Kok-Leong Ong, and Vincent C. S. Lee. Discovering frequent sets from data streams with cpu constraint. In *Data Mining and Analytics 2007, Proceedings of the Sixth Australasian Data Mining Conference (AusDM 2007)*, pages 121–128, 2007.

[DS] Manoranjan Dash and Ayush Singhania. An entropy-based fuzzy clustering method. *ACM Journal of Data and Information Quality.*

[DS07] Kaustav Das and Jeff Schneider. Detecting anomalous records in categorical datasets. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 220–229, 2007.

[EpKSX96] Martin Ester, Hans peter Kriegel, Jrg S, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise.

In *Proceedings of International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 226–231, 1996.

[FMR62] C.T. Fan, Mervin E. Muller, and Ivan Rezucha. Development of sampling plans by using sequential (item by item) selection techniques and digital computers. *Journal of the American Statistical Association*, 1962.

[FP99] Tom Fawcett and Foster Provost. Activity monitoring: Noticing interesting changes in behavior. In *Proceedings of the 5th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 53–62, 1999.

[FPSS96] Usama M. Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From data mining to knowledge discovery: An overview. In *Advances in Knowledge Discovery and Data Mining*, pages 1–34. 1996.

[GGR02] Minos N. Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Querying and mining data streams: you only get one look a tutorial. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, page 635, 2002.

[GHP+03] Chris Giannella, Jiawei Han, Jian Pei, Xifeng Yan, and Philip S. Yu. Mining frequent patterns in data streams at multiple time granularities. In *Next Generation Data Mining*. AAAI/MIT, 2003.

[GKS01] Johannes Gehrke, Flip Korn, and Divesh Srivastava. On computing correlated aggregates over continual data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 13–24, 2001.

[GKZ05] Mohamed Medhat Gaber, Shonali Krishnaswamy, and Arkady B. Zaslavsky. Resource-aware mining of data streams. *The Journal of Universal Computer Science*, 11(8):1440–1453, 2005.

[GM99] Phillip B. Gibbons and Yossi Matias. Synopsis data structures for massive data sets. *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science: Special Issue on External Memory Algorithms and Visualization*, 50, 1999.

[GMMO00] Sudipto Guha, Nina Mishra, Rajeev Motwani, and Liadan O'Callaghan. Clustering data streams. In *41st Annual Symposium on Foundations of Computer Science*, pages 359–366, 2000.

[GMS97] Dimitrios Gunopulos, Heikki Mannila, and Sanjeev Saluja. Discovering all most specific sentences by randomized algorithms. In *Database Theory - ICDT '97, 6th International Conference*, pages 215–229, 1997.

[Goe03] Bart Goethals. Survey on frequent pattern mining. Manuscript, 2003.

[GZ01] Karam Gouda and Mohammed Javeed Zaki. Efficiently mining maximal frequent itemsets. In *Proceedings of the 2001 IEEE International Conference on Data Mining (ICDM 2001)*, pages 163–170, 2001.

[GZ05] Karam Gouda and Mohammed J. Zaki. GenMax: An efficient algorithm for mining maximal frequent itemsets. *Data Mining and Knowledge Discovery*, 11:1–20, 2005.

[HA04] Victoria Hodge and Jim Austin. A survey of outlier detection methodologies. *Artificial Intelligence Review*, 2004.

[HK06a] J. Han and M. Kamber. *Data Mining: Concepts and Techniques, 2nd Ed.* Morgan Kaufmann Publishers, 2006.

[HK06b] Wontae Hwang and Dongseung Kim. Improved association rule mining by modified trimming. In *Proceedings of Sixth IEEE International Conference on Computer and Information Technology (CIT)*, 2006.

[HMS01] David Hand, Heikki Mannila, and Padhraic Smyth. *Principles of Data Mining.* MIT Press, 2001.

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *2000 ACM SIGMOD Intl. Conference on Management of Data*, pages 1–12, 2000.

[HR90] Torben Hagerup and Christine Rub. A guided tour of chernoff bounds. *Information Processing Letters*, pages 305–308, 1990.

[HSD01] Geoff Hulten, Laurie Spencer, and Pedro Domingos. Mining time-changing data streams. In *Proceedings of ACM SIGKDD International Conference in Knowledge Discovery and Data Mining*, pages 97–106, 2001.

[JAG99] Roberto J. Bayardo Jr., Rakesh Agrawal, and Dimitrios Gunopulos. Constraint-based rule mining in large, dense databases. In *Proceedings of the 15th International Conference on Data Engineering*, pages 188–197, 1999.

[JG06] Nan Jiang and Le Gruenwald. Research issues in data stream association rule mining. In *SIGMOD Record*, volume 35, pages 14–19, 2006.

[JMR05] Theodore Johnson, S. Muthukrishnan, and Irina Rozenbaum. Sampling algorithms in a stream operator. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 1–12, 2005.

[KBDG04] Daniel Kifer, Shai Ben-David, and Johannes Gehrke. Detecting change in data streams. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, pages 180–191, 2004.

[KK06] Sotiris Kotsiantis and Dimitris Kanellopoulos. Association rules mining: A recent overview. *GESTS International Transactions on Computer Science and Engineering*, 32(1):71–82, 2006.

[KM03] Jeremy Martin Kubica and Andrew Moore. Probabilistic noise identification and data cleaning. In *Proceedings of International Conference on Data Mining (ICDM)*, pages 131–138, 2003.

[KNT00] Edwin M. Knorr, Raymond T. Ng, and Vladimir Tucakov. Distance-based outliers: algorithms and applications. *The VLDB Journal*, 8:237–253, 2000.

[KSP03] Richard M. Karp, Scott Shenker, and Christos H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems*, 28(1):51–55, 2003.

[KV06] Vilius Kontrimas and Antanas Verikas. Tracking of doubtful real estate transactions by outlier detection methods: a comparative study. *Information Technology And Control*, 35(2):94–105, 2006.

[LA99] Bjornar Larsen and Chinatsu Aone. Fast and effective text mining using linear-time document clustering. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, 1999.

[LB06] Song Lin and Donald E. Brown. An outlier-based data association method for linking criminal incidents. *Decision Support System*, 41:604–615, 2006.

[LCK98] Sau Dan Lee, David Wai-Lok Cheung, and Ben Kao. Is sampling useful in data mining? a case in the maintenance of discovered association rules. *Data Mining and Knowledge Discovery*, 2(3):233–262, 1998.

[LHM98] Bing Liu, Wynne Hsu, and Yiming Ma. Integrating classification and association rule mining. In *Proceedings of the 4rd International Conference Knowledge Discovery and Data Mining (KDD-98)*, pages 80–86, 1998.

[LHM99a] Bing Liu, Wynne Hsu, and Yiming Ma. Mining association rules with multiple minimum supports. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining (KDD-99)*, pages 337–341, 1999.

[LHM99b] Bing Liu, Wynne Hsu, and Yiming Ma. Pruning and summarizing the discovered associations. In *Proceedings of the Fifth ACM SIGKDD International Conference in Knowledge Discovery and Data Mining*, pages 125–134, 1999.

[Li94] Kim-Hung Li. Reservoir sampling algorithms of time complexity o(n(1 + log(n/n))). *ACM Transactions on Mathematical Software*, 20(4):481–493, December 1994.

[LkLC05a] Ying Liu, Wei keng Liao, and Alok N. Choudhary. A fast high utility itemsets mining algorithm. In *Workshop on Utility-Based Data Mining*, 2005.

[LkLC05b] Ying Liu, Wei keng Liao, and Alok N. Choudhary. A two-phase algorithm for fast discovery of high utility itemsets. In *Advances in Knowledge Discovery and Data Mining, 9th Pacific-Asia Conference, PAKDD 2005*, pages 689–695, 2005.

[LLC05] Chang-Hung Lee, Cheng-Ru Lin, and Ming-Syan Chen. Sliding window filtering: an efficient method for incremental mining on a time-variant database. *Information Systems*, 30(3):227–244, 2005.

[LLL00] Mong Li Lee, Tok Wang Ling, and Wai Lup Low. Intelliclean: A knowledge-based intelligent data cleaner. In *Proceedings of International Conference on Knowledge Discovery and Data Mining, SIGKDD*, pages 290–294, 2000.

[LLS04] Hua Fu Li, Suh Yin Lee, and Man Kwan Shan. An efficient algorithm for mining frequent itemsets over the entire history of data streams. In *In Proc. of First International Workshop on Knowledge Discovery in Data Streams*, 2004.

[LLS05] Hua Fu Li, Suh Yin Lee, and Man Kwan Shan. Online mining (recently) maximal frequent itemsets over data streams. In *RIDE*, pages 11–18, 2005.

[LTKC05] K. K. Loo, Ivy Tong, Ben Kao, and David Cheung. Online algorithms for mining inter-stream associations from large sensor networks. In *Advances in Knowledge Discovery and Data Mining, 9th Pacific-Asia Conference*, pages 143–149, 2005.

[MAA05] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Database Theory - ICDT 2005, 10th International Conference*, pages 398–412, 2005.

[Mar01] Stephen Marsland. *On-Line Novelty Detection Through Self-Organisation, with Application to Inspection Robotics.* PhD thesis, Faculty of Science and Engineering, University of Manchester, UK, 2001.

[MG82] Jayadev Misra and David Gries. Finding repeated elements. *Scientific Computing Programming*, 2(2):143–152, 1982.

[MM02] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *Proceedings of 28th International Conference on Very Large Data Bases*, pages 346–357, 2002.

[MTV94] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Efficient algorithms for discovering association rules. In Usama M. Fayyad and Ramasamy Uthurusamy, editors, *AAAI Workshop on Knowledge Discovery in Databases (KDD-94)*, pages 181–192, 1994.

[Mut03] S. Muthukrishnan. Data streams: algorithms and applications. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 413–413, 2003.

[ND06] Willie Ng and Manoranjan Dash. An evaluation of progressive sampling for imbalanced data sets. In *IEEE ICDM workshop on Mining Evolving and Streaming Data*, pages 657–661, 2006.

[ND08a] Willie Ng and Manoranjan Dash. Efficient approximate mining of frequent patterns over transactional data streams. In *Data Warehousing*

and Knowledge Discovery, 10th International Conference, DaWaK 2008, pages 241–250, 2008.

[ND08b] Willie Ng and Manoranjan Dash. A test paradigm for detecting changes in transactional data streams. In *Database Systems for Advanced Applications, 13th International Conference, DASFAA 2008*, pages 204–219, 2008.

[NH94] Raymond T. Ng and Jiawei Han. Efficient and effective clustering methods for spatial data mining. In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 144–155, 1994.

[NK08] Kazuyo Narita and Hiroyuki Kitagawa. Detecting outliers in categorical record databases based on attribute associations. In *10th Asia-Pacific Web Conference, APWeb*, 2008.

[NTC+99] Alexandre Nairac, Neil Townsend, Roy Carr, Steve King, Peter Cowley, and Lionel Tarassenko. A system for the analysis of jet engine vibration data. *Integrated Computer Aided Engineering*, 6(1):53–66, 1999.

[OR95] Frank Olken and Doron Rotem. Random sampling from databases - a survey. *Statistics and Computing*, 5:25–42, 1995.

[Par02] Srinivasan Parthasarathy. Efficient progressive sampling for association rules. In *IEEE International Conference on Data Mining (ICDM'02)*, pages 354–361, 2002.

[PES01] Leonid Portnoy, Eleazar Eskin, and Salvatore J. Stolfo. Intrusion detection with unlabeled data using clustering. In *Proceedings of ACM CSS Workshop on Data Mining Applied to Security*, 2001.

[PG00] M. Pagano and K. Gauvreau. *Principles of biostatistics*. Duxbury, 2000.

[PH00] Jian Pei and Jiawei Han. Can we push more constraints into frequent pattern mining? In *Proceedings of the sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 350–354, 2000.

[PHL01] Jian Pei, Jiawei Han, and Laks V.S. Lakshmanan. Mining frequent itemsets with convertible constraints. In *Proceedings of the 17th International Conference on Data Engineering (ICDE-01)*, pages 433–442, 2001.

[PHM00] Jian Pei, Jiawei Han, and Runying Mao. Closet: An efficient algorithm for mining frequent closed itemsets. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 21–30, 2000.

[PJO99] Foster J. Provost, David Jensen, and Tim Oates. Efficient progressive sampling. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 23–32, 1999.

[PKGF03] Spiros Papadimitriou, Hiroyuki Kitagawa, Phillip B. Gibbons, and Christos Faloutsos. Loci: Fast outlier detection using the local correlation integral. In *Proceedings of the IEEE International Conference on Data Engimeering*, pages 315–326, 2003.

[POSG04] Byung-Hoon Park, George Ostrouchov, Nagiza F. Samatova, and Al Geist. Reservoir-based random sampling with replacement from data stream. In *Proceedings of the SIAM International Conference on Data Mining (SDM'04)*, pages 492–501, 2004.

[RL96] Peter J. Rousseeuw and Annick M. Leroy. *Robust Regression and Outlier Detection.* John Wiley & Sons, 3rd edition, 1996.

[RRS00] Sridhar Ramaswamy, Rajeev Rastogi, and Kyuseok Shim. Efficient algorithms for mining outliers from large data sets. In *SIGMOD 2000 International Conference on Management of Data*, pages 427–438, 2000.

[SBMU00] Craig Silverstein, Sergey Brin, Rajeev Motwani, and Jeffrey D. Ullman. Scalable techniques for mining causal structures. *Data Mining and Knowledge Discovery*, 4(2/3):163–192, 2000.

[Sha02] Yunyue Zhu Dennis Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of 28th International Conference on Very Large Data Bases*, pages 358–369, 2002.

[SK01] Masakazu Seno and George Karypis. Lpminer: An algorithm for finding frequent itemsets using length-decreasing support constraint. In *Proceedings of the 2001 IEEE International Conference on Data Mining (ICDM 2001)*, pages 505–512, 2001.

[SK02] Masakazu Seno and George Karypis. Slpminer: An algorithm for finding frequent sequential patterns using length-decreasing support constraint. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002)*, pages 418–425, 2002.

[SON95] Ashok Savasere, Edward Omiecinski, and Shamkant Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB'95)*, 1995.

[SW04] Utkarsh Srivastava and Jennifer Widom. Memory-limited execution of windowed stream joins. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, pages 324–335, 2004.

[TCL06] Vincent S. Tseng, Chun-Jung Chu, and Tyne Liang. Efficient mining of temporal high utility itemsets from data streams. In *Workshop on Utility-Based Data Mining*, 2006.

[TcZ07] Nesime Tatbul, Ugur Çetintemel, and Stanley B. Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 159–170, 2007.

[TMF03] Feng Tao, Fionn Murtagh, and Mohsen Farid. Weighted association rule mining using weighted support and significance framework. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 661–666, 2003.

[Toi96] Hannu Toivonen. Sampling large databases for association rules. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 134–145, 1996.

[TSK06] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining.* Pearson International Edition, 2006.

[Vit85] Jeffrey Scott Vitter. Random sampling with a reservoir. In *ACM Transactions on Mathematical Software*, volume 11, pages 37–57, 1985.

[WFYH03] Haixun Wang, Wei Fan, Philip S. Yu, and Jiawei Han. Mining concept-drifting data streams using ensemble classifiers. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 226–235, 2003.

[WGC88] R. M. Royall W. G. Cumberland. Does simple random sampling provide adequate balance? *Journal of the Royal Statistical Society. Series B (Methodological)*, 50(1):118–124, 1988.

[WSTZ05] Gary Weiss, Maytal Saar-Tsechansky, and Bianca Zadrozny. Report on ubdm-05: Workshop on utility-based data mining. *SIGKDD Explorations*, 7(2):145–147, 2005.

[WWW05] Ji Zhang Wei Wang and Hai Wang. Grid-odf: Detecting outliers effectively and efficiently in large multi-dimensional databases. In *Proceedings of International Conference on Computational Intelligence and Security*, pages 765–770, 2005.

[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 *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, 2004.

[YHB04] Hong Yao, Howard J. Hamilton, and Cory J. Butz. A foundational approach to mining itemset utilities from databases. In *Proceedings of the Fourth SIAM International Conference on Data Mining*, 2004.

[YS04] Li Yang and Mustafa Sanver. Mining short association rules with one database scan. In *Proceedings of the International Conference on Information and Knowledge Engineering*, pages 392–398, 2004.

[YSJ+00] Byoung-Kee Yi, Nikolaos Sidiropoulos, Theodore Johnson, H. V. Jagadish, Christos Faloutsos, and Alexandros Biliris. Online mining for co-evolving time sequences. In *Proceedings of the 16th International Conference on Data Engineering*, pages 13–22, 2000.

[Zak00] Mohammed Javeed Zaki. Generating non-redundant association rules. In *Proceedings of the sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 34–43, 2000.

[ZKM01] Zijian Zheng, Ron Kohavi, and Llew Mason. Real world performance of association rule algorithms. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge Discovery and Data Mining*, 2001.

[ZPLO96] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Wei Li, and Mitsunori Ogihara. Evaluation of sampling for data mining of association rules. In *Seventh International Workshop on Research Issues in Data Engineering (RIDE'97)*, 1996.

[ZPOL97] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. New algorithms for fast discovery of association rules. In *3rd Intl. Conf. on Knowledge Discovery and Data Mining*, pages 283–286. AAAI Press, 1997.

[ZS02] Yunyue Zhu and Dennis Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of 28th International Conference on Very Large Data Bases*, pages 358–369, 2002.

[ZWKS07] Xingquan Zhu, Xindong Wu, Taghi Khoshgoftaar, and Yong Shi. Empirical study of the noise impact on cost-sensitive learning. In *Proceedings of International Conference on Joint COnference on Artificial Intelligence (IJCAI)*, 2007.

[ZWST06] Bianca Zadrozny, Gary Weiss, and Maytal Saar-Tsechansky. Ubdm 2006: Utility-based data mining 2006 workshop report. *SIGKDD Explorations*, 8(2):98–101, 2006.