國立交通大學

資訊科學與工程研究所

碩 士 論 文

使用位元向量在資料串流環境探勘封閉式頻繁項目集及循序樣式之研究

Mining of Closed Frequent Itemsets and Sequential Patterns in Data Streams Using Bit-Vector Based Method

研 究 生：何錦泉

指導教授：李素瑛　教授

中 華 民 國 九 十 五 年 七 月

使用位元向量在資料串流環境探勘封閉式頻繁項目集及循序樣式之研究

Mining of Closed Frequent Itemsets and Sequential Patterns in Data Streams Using Bit-Vector Based Method

研 究 生：何錦泉　　Student：Chin-Chuan Ho

指導教授：李素瑛　　Advisor：Suh-Yin Lee

國 立 交 通 大 學
資 訊 科 學 與 工 程 研 究 所
碩 士 論 文

A Thesis
Submitted to Institute of Computer Science and Engineering
College of Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in

Computer Science

July 2006

Hsinchu, Taiwan, Republic of China

中華民國九十五年七月

使用位元向量在資料串流環境探勘封閉式頻繁項目集及循序樣式之研究

研究生：何錦泉　　　　　　　　　　　　　　　指導教授：李素瑛

國立交通大學資訊科學與工程研究所

摘要

在資料串流環境中探勘有意義的樣式是一個重要的課題，在感測網路及股市分析等許多應用中都經常採用。由於資料串流環境的限制，探勘工作將會變得比較困難。我們在此篇論文的第一部份提出 New-Moment 演算法在資料串流環境中探勘封閉式頻繁項目集，New-Moment 使用位元向量以及精簡的 closed enumeration tree 大幅改進原來 Moment 演算法的效能。在第二部分我們提出 IncSPAM 演算法在串流環境中探勘循序樣式，它提供了一個全新的滑動視窗架構。IncSPAM 利用 SPAM 演算法以及記憶體索引的方法，動態維護目前最新的樣式。實驗顯示我們的方法能夠很有效率地在資料串流環境中探勘出有意義的樣式。

檢索詞：資料串流、滑動視窗、封閉式頻繁項目集、循序樣式

Mining of Closed Frequent Itemsets and Sequential Patterns in Data Streams Using Bit-Vector Based Method

Student: Chin-Chuan Ho　　　　Advisor: Suh-Yin Lee

Institute of Computer Science and Engineering

College of Computer Science

National Chiao-Tung University

Abstract

Mining a data stream is an important data mining problem with broad applications, such as sensor network, stock analysis. It is a difficult problem because of some limitations in the data stream environment. In the first part of this paper, we propose New-Moment to mine closed frequent itemsets. New-Moment uses bit-vectors and a compact lexicographical tree to improve the performance of Moment algorithm. In the second part, we propose IncSPAM to mine sequential patterns with a new sliding window model. IncSPAM is based on SPAM and utilizes memory indexing technique to incrementally maintain sequential patterns in current sliding window. Experiments show that our approaches are efficient for mining patterns in a data stream.

Index Terms: data stream, sliding window, closed frequent itemset, sequential pattern

Acknowledgment

I greatly appreciate the kind guidance of my advisor, Prof. Suh-Yin Lee. Without her graceful suggestion and encouragement, I cannot complete this thesis.

Besides I want to express my thanks to all the members in the Information System Laboratory for their suggestion and instruction, especially Mr. Hua-Fu Li. Finally I would like to express my appreciation to my parents. This thesis is dedicated to them.

Table of Contents

Abstract (Chinese) i

Abstract (English) ii

Acknowledgment iii

Table of Contents iv

List of Figures vi

List of Tables viii

Chapter 1 Introduction 1

1.1 Overview and Motivation 1

1.2 Related Work 2

1.2.1 Mining of Frequent Itemsets 2

1.2.2 Mining of Sequential Patterns 3

1.3 Organization of Thesis 4

Chapter 2 Problem Definition and Background 5

2.1 The Sliding Window Model in Data Streams 5

2.1.1 Data Stream Environment 5

2.1.2 A Sliding Window Model 6

2.2 Definition of Mining Closed Frequent Itemsets 6

2.3 Definition of Mining Sequential Patterns 8

Chapter 3 New-Moment: Mining Closed Frequent Itemsets 10

3.1 Related Work: Moment Algorithm 10

3.2 Our Proposed Algorithm: New-Moment Algorithm 14

3.2.1 Bit-Vector 14

3.2.2 Window Sliding with Bit-Vector 15

3.2.3 Counting Support with Bit-Vector 16

3.2.4 Building the New Closed Enumeration Tree (New-CET) 17

3.2.5 Deleting the Oldest Transaction in Window Sliding 20

3.2.6 Appending the Incoming Transaction in window sliding 23

Chapter 4 Incremental SPAM (IncSPAM): Mining Sequential Patterns 25

4.1 A New Concept of Sliding Window for Sequences 25

4.2 Customer Bit-Vector Array with Sliding Window (CBASW) 26

4.3 Index Set ρ-idx 28

4.4 Maintaining Information of Items in Window Sliding with CBASW and ρ-idx 29

4.5 Building of Lexicographical Sequence Tree 31

4.6 Counting Support with ρ-idx 32

4.6.1 Counting Support in S-step 32

4.6.2 Counting Support in I-step....33
4.7 The Entire Process of Incremental SPAM (IncSPAM) 34
4.8 Weight of Customer-Sequence....40
Chapter 5 Performance Measurement 47
5.1 Performance Measurement of New-Moment 47
5.1.1 Different Minimum Support....48
5.1.2 Different Sliding Window Size 52
5.1.3 Different Number of Items 55
5.2 Performance Measurement of IncSPAM....57
5.2.1 Different Minimum Support....57
5.2.2 Different Sliding Window Size 60
5.2.3 Different Number of Customers 61
Chapter 6 Conclusion and Future Work....63
6.1 Conclusion of New-Moment 63
6.2 Conclusion of IncSPAM....64
6.3 Future Work 64
Bibliography 66

List of Figures

Fig 2-1. Processing model of data stream environment........5
Fig 2-2. A sliding window model in a data stream........6
Fig 2-3. An example of an input database D_S........8
Fig 2-4. The customer-sequences in Fig 2-3........9
Fig 3-1. CET in the first sliding window........10
Fig 3-2. Adding the new transaction with tid = 5........13
Fig 3-3. Deleting the transaction with tid = 1........13
Fig 3-4. An example database and the first three sliding windows........15
Fig 3-5. Pseudo code of building New-CET........18
Fig 3-6. New-CET in the first window after generating new candidates from item *a*........19
Fig 3-7. New-CET in the first window after checking closed frequent itemsets........19
Fig 3-8. New-CET in the first window (Window #1)........20
Fig 3-9. New-CET after deleting the oldest transaction........21
Fig 3-10. Pseudo code of deleting the oldest transaction in window sliding........22
Fig 3-11. Pseudo code of appending the incoming transaction in window sliding........23
Fig 3-12. New-CET after appending the incoming transaction (Window #2)........24
Fig 4-1. An example for the new concept of sliding window........26
Fig 4-2. An example of CBASW........27
Fig 4-3. An example of index sets........28
Fig 4-4. An example of window sliding in a CBASW........30
Fig 4-5. A lexicographic sequence tree example........31
Fig 4-6. Main function of Incremental SPAM........34
Fig 4-7. Reducing the generated candidates........35
Fig 4-8. The pseudo code of function MaintainTree........36
Fig 4-9. The pseudo code of function Generate........36
Fig 4-10. The pseudo code of function Update........37
Fig 4-11. The lexicographic sequence tree when the third transaction comes in........37
Fig 4-12. The lexicographic sequence tree after the fourth transaction comes in........38
Fig 4-13. The lexicographic sequence tree after the fifth transaction comes in........39
Fig 4-14. The lexicographic sequence tree after the sixth transaction comes in........40
Fig 4-15. The transactions of a customer with no recent records in a data stream........40
Fig 4-16. An example of calculating the weights of customers........42
Fig 4-17. When a new transaction with TID = 8 comes in........42
Fig 4-18. The lexicographic sequence tree when the third transaction comes in (with the concept of customer weight)........43

Fig 4-19. An example of support updating in IncSPAM (Case 1)..44
Fig 4-20. An example of support updating in Incremental SPAM (Case 2)...45
Fig 4-21. The pseudo code of function UpdateSupport...46
Fig 5-1. Memory usage with different minimum support (T10I8D200K) (New-Moment and Moment) 48
Fig 5-2. Loading time of the first window with different minimum support (T10I8D200K) (New-Moment and Moment) 49
Fig 5-3. Average time of window sliding with different minimum support (T10I8D200K) (New-Moment and Moment) 50
Fig 5-4. Memory usage with different minimum support (T15I12D200K) (New-Moment and Moment) 51
Fig 5-5. Loading time of the first window with different minimum support (T15I12D200K) (New-Moment and Moment) 51
Fig 5-6. Average time of window sliding with different minimum support (T15I12D200K) (New-Moment and Moment) 52
Fig 5-7. Memory usage with different sliding window size (New-Moment and Moment).................................53
Fig 5-8. Loading time of the first window with different sliding window size (New-Moment and Moment).....54
Fig 5-9. Average time of window sliding with different sliding window size (New-Moment and Moment) 54
Fig 5-10. Memory usage with different number of items (New-Moment and Moment).....................................55
Fig 5-11. Loading time of the first window with different number of items (New-Moment and Moment).........56
Fig 5-12. Average time of window sliding with different number of items (New-Moment and Moment)..........56
Fig 5-13. Memory usage with different minimum support (IncSPAM)...58
Fig 5-14. Relationship between maximum number of tree nodes and memory usage (IncSPAM)......................59
Fig 5-15. Average time of window sliding with different minimum support (IncSPAM) 59
Fig 5-16. Memory usage with different window size (IncSPAM) 60
Fig 5-17. Average sliding time with different window size (IncSPAM)..60
Fig 5-18. Memory usage with different number of customers (IncSPAM)...61
Fig 5-19. Average sliding time with different number of customers (IncSPAM) 62
Fig 6-1. The sliding window model in time units...65

List of Tables

Table 3-1. The bit-vectors of all items in each window in Figure 3-4 15
Table 4-1. Bit-vectors of all items for all customers 27
Table 5-1. Parameters of testing data for New-Moment 47
Table 5-2. Parameters of testing data for IncSPAM 57

Chapter 1

Introduction

1.1 Overview and Motivation

Many problems in mining frequent itemsets and sequential patterns focus on static databases. In recent years, dynamic environment is becoming more and more important in many applications. This dynamic environment is called *data streams*. However, there are some inherent limitations in streaming data environment. For examples, data mining in sensor networks has some limitation different from traditional data mining, e.g. battery power and capability of sensor CPU [13].

Data streams have characteristics as described below [14] [19]: (1) Unbounded size of input data; (2) Usage of main memory is limited; (3) Input data can only be handled once; (4) Fast arrival rate; (5) System can not control the order data arrives; (6) Analytical results generated by algorithms should be instantly available when users request; (7) Errors of analytical results should be bounded in a range that users can tolerate.

For conditions above, three models are adopted by many researchers in ways of time spanning: landmark model, sliding window model, and damped window model [9].

Landmark model handles data in a time interval. Starting time point is set by users, called *landmark*, and end time point is equal to current time point. So end time point is changed as time goes by. If landmark is set to the time point that the first transaction comes, this model will cover all the available data.

In sliding window model, a window with length w will be given. If current time point is t, this model handles data in the range $[t - w, t]$. So when time goes to next time point, this model has to eliminate the oldest data in the window and insert the new data. This step is

called *window sliding*. From above we can know that sliding window will cover some range of newest data in a data stream.

Damped window model also considers recent data important, but is not like sliding window model eliminating passed data. In this model all available data is kept but a user defines a weighted function for data which decreases exponentially into the past.

Three models have their own advantages and disadvantages. The sliding window model keeps the latest information in the data stream. This characteristic is useful if real-time patterns are needed, like daily or weekly stock analysis.

In this paper, an algorithm *New-Moment* that mines closed frequent itemsets and an algorithm *IncSPAM* that mines sequential patterns in data streams are proposed. These two algorithms can efficiently retrieve useful patterns in data streams.

1.2 Related Work

1.2.1 Mining of Frequent Itemsets

In sliding window model, an efficient algorithm Moment was proposed in [20, 21]. Moment uses a compact data structure, the *closed enumeration tree* (CET), to maintain a dynamically selected set of itemsets over a sliding window. These selected itemsets consist of closed frequent itemsets and a boundary between the closed frequent itemsets and the rest of the itemsets. CET can cover all necessary information because any status changes of itemsets (e.g. from infrequent to frequent) must be through the boundary in CET. Whenever a sliding occurs, it updates the counts of the related nodes in CET and modifies CET. Experiments of Moment show that the boundary in CET is stable so the updating cost is little. It outperforms algorithms Charm [8] and ZIGZAG [7] in running time.

However Moment must maintain a huge number of CET nodes for a closed frequent itemset. The ratio of CET nodes and closed frequent itemsets is about 30 : 1. If there are a large

number of closed frequent itemsets, the memory usage of Moment will be inefficient.

Our proposed algorithm, New-Moment, only maintains closed frequent itemsets and uses *bit-vector* to store the information in the window. Experiments show that memory usage of New-Moment is much less than Moment and running time of both algorithms is almost the same.

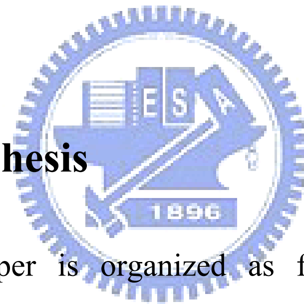
There are many researches about mining frequent itemsets over data streams. Manku et al [10] propose lossy counting algorithm to mine frequent itemsets over an entire data stream. Jin et al [15] propose hCount algorithm to maintain frequent items in the streaming environment. Li et al [19] propose an algorithm, DSM-FI, to mine frequent itemsets in the landmark model over a data stream. It is a projection-based, single-pass algorithm. Chang et al [22] propose an algorithm for mining frequent itemsets in the sliding window model. Chang et al [16] propose estDec algorithm. It uses a decay function to reduce the weight of the old transactions. Researches about mining maximal frequent itemsets and closed frequent itemsets over data streams are few. Li et al [27] propose DSM-MFI to mine maximal frequent itemsets in the sliding window model in a data stream.

1.2.2 Mining of Sequential Patterns

There are many researches about mining sequential patterns in a static database. Agrawal et al [2] introduce the concept of sequential patterns. They use apriori method that is not efficient enough. Pei et al [6] provide a efficient algorithm, PrefixSpan, to mine sequential patterns by prefix-projected pattern growth. Lin et al [12] use memory indexing to decrease the time of mining sequential patterns. The assumption is that entire sequence database can be loaded into main memory. An algorithm SPAM is provided in [11] which use a lexicographic sequence tree to check all possible frequent sequences. Bitmap representation is used for speeding up mining process.

Besides general sequential patterns, closed sequential patterns are also studied. Yan et al [17] provide CloSpan to mine closed sequential patterns in large datasets. Chen et al [23] mine multiple-level sequential patterns. A concept hierarchy is used to represent the relationship between items. For the flexibility of sequence databases, incremental mining of sequential patterns is also a research issue. Yen et al [3] and Cheng et al [24] provide researches in this area.

Researches about mining of sequential patterns in data streams are not as many as in static databases. Teng et al [18] provide FTP-DS to mine temporal patterns in a data stream. Regression-based analysis on frequent patterns is the main feature to improve the performance of FTP-DS. Chen et al [25] mine sequential patterns across many data streams. Marascu [28] use SMDS algorithm to mine web usage sequences.


E S A 1896

1.3 Organization of Thesis

The remainder of this paper is organized as follows. Some basic definitions and terminology about itemset, sequence, and sliding window model are described in Chapter 2. The New-Moment algorithm to mine closed frequent itemsets is presented in Chapter 3. The IncSPAM algorithm to mine sequential patterns is introduced in Chapter 4. Finally the experiments and performance measurements are described in Chapter 5. Conclusion and future work is in Chapter 6.

Chapter 2

Problem Definition and Background

In this chapter we introduce the basic definition of problems. We introduce the definition of the data stream environment and the sliding window model in section 2.1. Next we describe the definition of closed frequent itemsets and sequential patterns in section 2.2.

2.1 The Sliding Window Model in Data Streams

2.1.1 Data Stream Environment

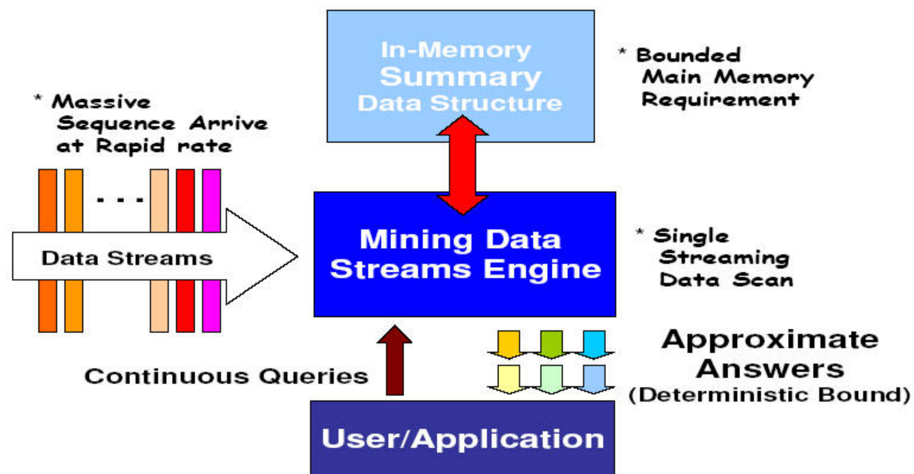


Fig 2-1. Processing model of data stream environment

A ***data stream*** DS = [T_1, T_2, …, T_M) is an infinite transaction set. In a ***data stream environment***, the input is the continuous data stream and each transaction can only be scanned once. Due to the limited memory and one-time scan of each transaction (***one-pass***), a summary data structure is needed to store compact information about the data stream. In other words, one-pass algorithms for mining data streams have to sacrifice the correctness of its analytical results by allowing some counting error. Hence traditional multi-pass techniques

for mining static databases are not feasible to be used in the data stream environment. Figure 2-1 shows a processing model of data streams [19].

2.1.2 A Sliding Window Model

Some applications in data streams emphasize the importance of the latest transactions. A ***sliding window model*** is suitable to solve this kind of problems. In the basic concept, a sliding window keeps the latest N transactions in the data streams; N is called a window size. The mining data streams engine in Figure 2-1 only mines patterns in the current sliding window. Whenever a new transaction is coming, the sliding window eliminates the oldest transaction and appends the incoming transaction. This process is called ***window sliding***. The mining data streams engine also modifies the summary data structure by the changes of sliding window. Figure 2-2 shows the sliding window in an input data stream.

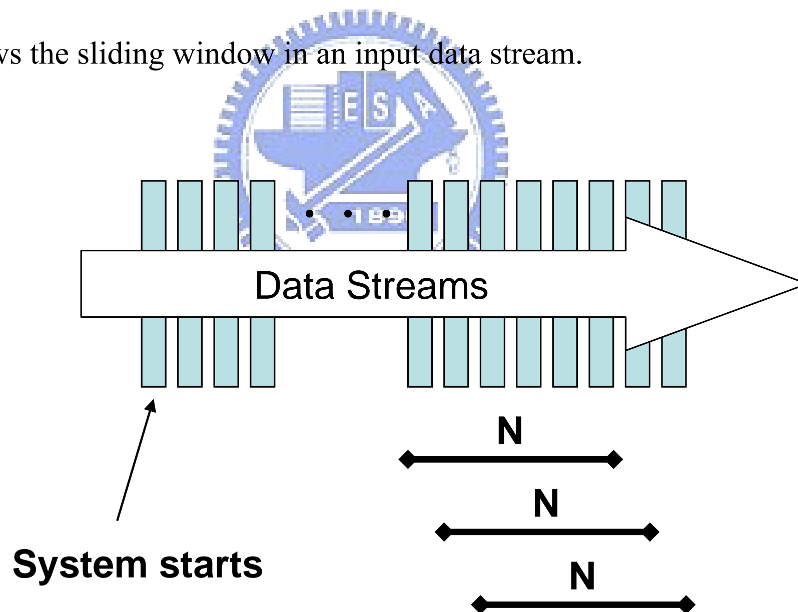


Fig 2-2. A sliding window model in a data stream

2.2 Definition of Mining Closed Frequent Itemsets

$I = \{i_1, i_2, i_3, \ldots, i_n\}$ is a set of literals, called ***items***. An ***itemset*** is a set of items. An itemset X with k items is represented in form of $X = (x_1, x_2, \ldots, x_k)$, called ***k-itemset***. Let D_I be a

database which has a set of transactions. Each ***transaction*** T consists of a set of items from I, i.e., $T \subseteq I$ and a ***transaction id (TID)*** represents the time order in the database. An itemset X is said to be contained in a transaction T if $X \subseteq T$. The ***support*** of an itemset X is the number of transactions containing X. An itemset X is a ***frequent itemset*** if the support of X is more than a user specified threshold ***minimum support*** S.

As an example, let I = {a, b, c, d}, D_I = {(a, b, c), (b, c, d), (a, b, c), (b, c)}, S = 0.5. The set of frequent itemsets F = {(a): 2, (b): 4, (c): 4, (a, b): 2, (a, c): 2, (b, c): 4, (a, b, c): 2}.The Number following the colon represents the support of the itemset.

The total number of all the frequent itemsets sometimes is too large and it is difficult to retrieve useful information. For reducing the number of output patterns, the concept of ***closed frequent itemsets*** [4] is proposed.

Definition of Closed Frequent Itemset. A frequent itemset X is ***closed*** if there is no frequent itemset X' such that (1) $X \subset X'$ and (2) $\forall$ transaction T, $X \in T \rightarrow X' \in T$.

In the above example, the set of closed frequent itemsets C = {(b, c): 4, (a, b, c): 2}, $C \subseteq F$. We observe itemsets (b), (c), and (b, c). The supports of itemsets (b) and (c) are equals to the support of itemset (b, c); and further, itemsets (b) and (c) are subsets of (b, c). That means itemsets (b) and (c) exist in the same transactions of itemset (b, c). By the definition of closed frequent itemset, (b) and (c) are not closed frequent itemsets and (b, c) is a closed frequent itemset.

All frequent itemsets can be obtained from closed frequent itemsets without losing support information. In the above example, we know that (b) and (c) are frequent itemsets by observing closed frequent itemset (b, c). Supports of (b) and (c) are the same as (b, c). Supports of frequent itemsets can be used to judge if a frequent itemset is closed.

2.3 Definition of Mining Sequential Patterns

An input *database* D_S contains ***customer-transactions***. These customer-transactions are a little different from transactions in section 2.1.1. Each customer-transaction consists of the following field: ***customer-id(CID)***, ***transaction-id(TID)***, and the ***items*** purchased in the transaction (called an ***itemset***). The concepts of TID, items and itemsets here are the same in section 2.2. The difference is that each transaction in D_S belongs to some customer. Figure 2-3 shows an example of the transaction database D_S.

Customer ID (CID)	Transaction ID(TID)	Itemset
1	1	(a, b, d)
2	2	(b)
1	3	(b, c, d)
2	4	(a, b, c)
3	5	(a, b)
1	6	(b, c, d)
3	7	(b, c, d)

Fig 2-3. An example of an input database D_S

A ***sequence*** is an ordered list of itemset and is denoted as $S = \langle s_1 s_2 s_3 \ldots s_k \rangle$, where s_j is an itemset. A sequence $\alpha = \langle a_1 a_2 a_3 \ldots a_k \rangle$ is contained in another sequence $\beta = \langle b_1 b_2 b_3 \ldots b_k \rangle$ if there exists integers $i_1 < i_2 < i_3 < \ldots < i_n$ such that $a_1 \subseteq b_{i_1}, a_2 \subseteq b_{i_2}, \ldots, a_n \subseteq b_{i_n}$.

All the transactions of a customer can be viewed as a sequence, where each transaction corresponds to a set of items, and the list of transactions, ordered by increasing transaction-id, corresponds to a sequence. We call such a sequence a ***customer-sequence***. Figure 2-4 shows the customer-sequences in Figure 2-3.

CID	Sequence
1	<(a, b, d)(b, c, d)(b, c, d)>
2	<(b)(a, b, c)>
3	<(a, b)(b, c, d)>

Fig 2-4. The customer-sequences in Fig 2-3

The absolute support of a sequence *S* is defined as the number of customer-sequences containing *S*. ***Sequential patterns*** are the sequences whose supports are more than a user-defined ***minimum support***, also called ***frequent sequences***.

Chapter 3

New-Moment: Mining Closed Frequent Itemsets

The goal of New-Moment is to improve Moment algorithm. First we introduce Moment algorithm in section 3.1. Next we introduce our proposed algorithm, New-Moment, in section 3.2.

3.1 Related Work: Moment Algorithm

Moment [20, 21] algorithm mines closed frequent itemsets with sliding window model in a data stream. It uses a *closed enumeration tree (CET)* to maintain the closed frequent itemsets in the current window. CET not only maintains closed frequent itemsets but also maintains some boundary tree nodes. Figure 3-1 shows the CET in the first window. Assume that the window size is 4 and the first four incoming transaction is listed in the left of the graph.

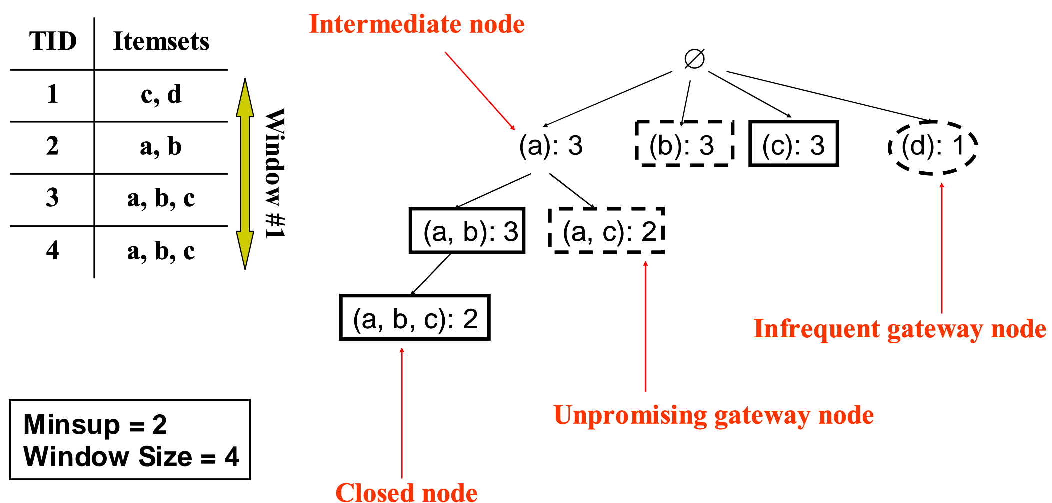


Fig 3-1. CET in the first sliding window

There are four types of tree nodes for CET:

(1) infrequent gateway nodes

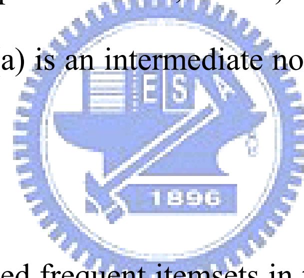
A node n_I that represents itemset I is an infrequent gateway node if i) I is an infrequent itemset, ii) n_I's parent, n_J, is frequent, and iii) I is the result of joining I's parent, J, with one of J's frequent siblings. In Figure 3-1, the tree node (d) is an infrequent gateway node.

(2) unpromising gateway nodes

A node n_I is an unpromising gateway node if i) I is a frequent itemset and ii) there exists a closed frequent itemset J such that $J \subset I$, and J has the same support as I does. In Figure 3-1, the tree nodes (a, c) and (b) are unpromising gateway nodes.

(3) intermediate nodes

A node n_I is an intermediate node if i) I is a frequent itemset, ii) n_I has a child node n_J such that J has the same support as I does, and iii) n_I is not an unpromising gateway node. In Figure 3-1, the tree node (a) is an intermediate node because its child (a, b) has the same support as (a) does.



(4) closed nodes

These nodes represent closed frequent itemsets in the current window. A closed node can be an internal node or a leaf node. In Figure 3-1, (c), (a, b), and (a, b, c) are closed nodes.

Except closed nodes, Moment keeps three types of boundary nodes. These nodes are the most possible candidates of new closed nodes in the next window. Moment keeps these nodes for speeding up modification of the closed enumeration tree.

There are three steps in Moment algorithm:

(1) Building the closed enumeration tree (CET)

When the total number of transactions coming from the data stream does not excess window size N, Moment just saves these transactions in its sliding window. As long as the

window is full, Moment builds an initial closed enumeration tree (CET). Figure 3-1 shows the tree in the first window.

Moment adopts a *depth-first* procedure to generate all possible candidate itemsets in the window and check their supports. In the procedure, if a node is found to be infrequent, it is marked as an *infrequent gateway node* and Moment does not explore its descendants further.

If a node is frequent itemset but not closed frequent itemset, the node is marked as an unpromising gateway node. Moment also does not explore its descendants, which does not contain any closed frequent itemsets. Moment uses support of a node and the tid sum of the transactions that containing the node (tid_sum) to check if the node is a closed node. Take the nodes (a, c) and (a, b, c) in Figure 3-1 as an example. The support of (a, c) is the same as (a, b, c). The tid_sum of (a, c) is 7 (the third transaction and the fourth transaction in the window). That is equal to the tid_sum of (a, b, c). By the definition of closed frequent itemsets, we can know that (a, c) is not a closed node.

If a node is found to be neither an infrequent node nor an unpromising gateway node, Moment explores its descendants. The nodes that are intermediate nodes or closed nodes are maintained in the CET.

(2) Updating the CET

Initial closed enumeration tree is built when the number of incoming transactions from the data stream is equal to the window size. After that, when a new transaction comes from the data stream, Moment updates the CET to maintain the closed frequent itemsets in the current window. There are two steps for updating the CET:

Adding the new transaction coming from the data stream

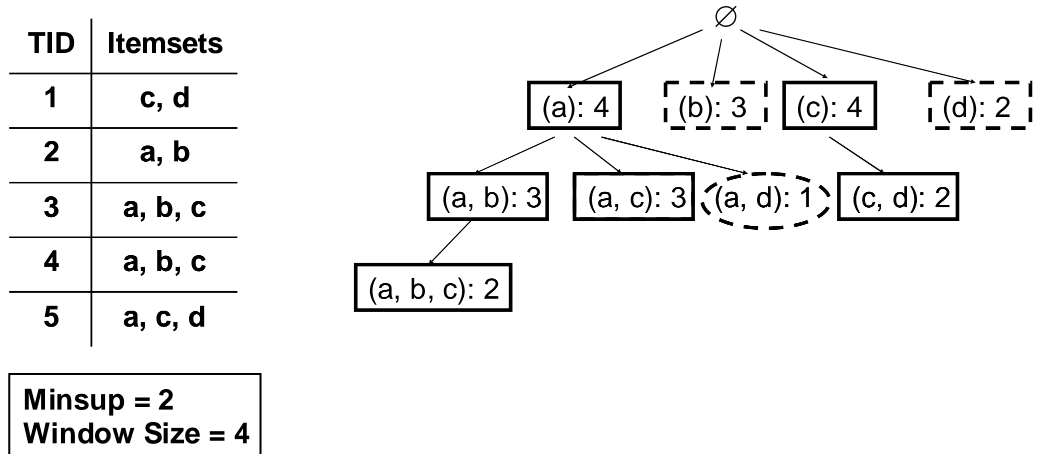


Fig 3-2. Adding the new transaction with tid = 5

In Figure 3-2, a new transaction T (tid = 5) is added to the sliding window. Moment traverses the parts of the CET that are related to transaction T. For each related node n_I in depth-first order, Moment updates its support and tid_sum. Whenever a node is updated, Moment checks if it needs to change its node type.

In Figure 3-2, the node (d) becomes a new frequent node so Moment generates the new candidates node (a, d) and (c, d). By node properties Moment know that (a, d) is an infrequent gateway node and (c, d) is a new closed node. By checking the support of the nodes (a), (a, c), and (c), Moment modifies them to closed nodes.

Deleting the oldest transaction in the window

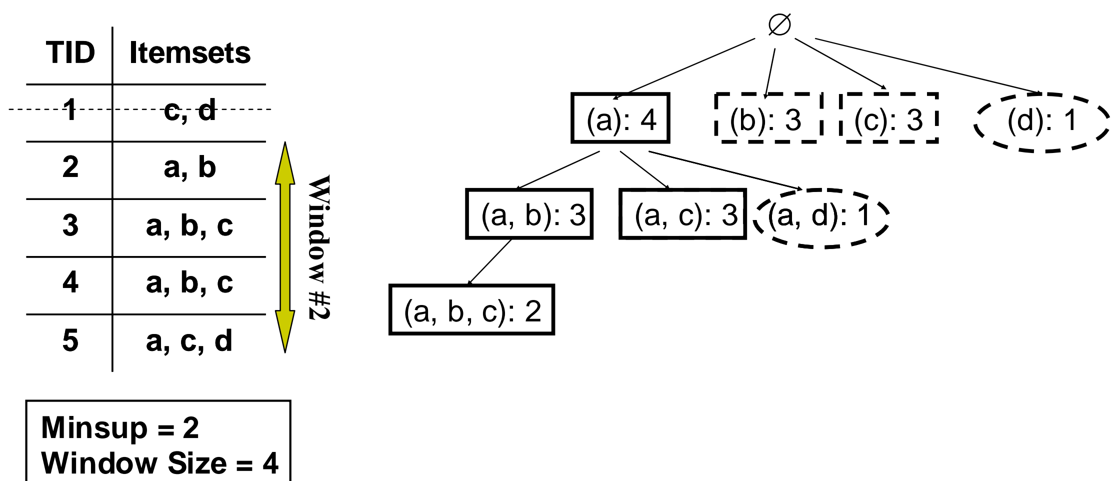


Fig 3-3. Deleting the transaction with tid = 1

In Figure 3-3, the transaction with tid = 1 is deleted. Like adding the new transaction, Moment updates support and tid_sum of each node in the CET. By checking the support of each node, Moment modifies its node type.

In Figure 3-3, node (c) becomes unpromising gateway node because it is contained by node (a, c) and supports of (c) and (a, c) are the same. Then the sub tree of node (c), (c, d), is deleted. The node (d) becomes new infrequent gateway node.

Moment maintains a huge number of boundary nodes to speed up the procedure of updating CET. The cost for a node to change its type is less. But we find that those boundary nodes are unnecessary overhead. In our proposed algorithm New-Moment, we reduce the number of tree nodes and utilize an efficient structure to store the information of the sliding window.

3.2 Our Proposed Algorithm: New-Moment Algorithm

We use *bit-vector* to store the information of a sliding window. Because of the efficiency of bit-vector in counting support and modifying transactions in window, New-Moment only maintains closed frequent itemsets in each sliding window. The *new closed enumeration tree* (New-CET) is composed of the bit-vectors of 1-itemsets, the closed frequent itemsets in current sliding window, and a hash table.

3.2.1 Bit-Vector

Definition of Bit-Vector: For a specified item *i* and a given window *w* of sliding window model in a data stream, a *bit-vector* is used to store the occurrences of item *i* in the transactions of *w*. Each bit of a bit-vector represents a transaction in *w*. If the item *i* occurs in some transaction of *w*, the corresponding bit is set to one, else set to zero.

Figure 3-4 shows an example of input database and the first three sliding windows are displayed next to it. These windows are marked from window #1 to window #3. It is assumed that the size of sliding window is 4. The example of figure 3-4 will be used in the following context.

TID	Itemsets
1	c, d
2	a, b
3	a, b, c
4	a, b, c
5	a, c, d
6	b, c

Window size N = 4

Window #1

Window #2

Window #3

Fig 3-4. An example database and the first three sliding windows

Each window in figure 3-4 can be transformed to a bit-vector by the definition of bit-vector. The bit-vectors of all items in each window are listed in Table 3-1. The most left bit represents the oldest transaction and the most right bit is the most recent transaction.

	Window #1	Window #2	Window #3
a	0111	1111	1110
b	0111	1110	1101
c	1011	0111	1111
d	1000	0001	0010

Table 3-1. The bit-vectors of all items in each window in Figure 3-4

3.2.2 Window Sliding with Bit-Vector

When the number of transactions in a data stream exceeds the size of a window, window sliding is performed to eliminate the oldest transaction and append the incoming transaction.

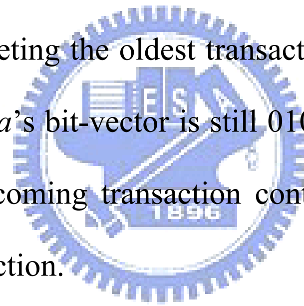
Bit-vector is efficient in window sliding process. We can separate the sliding process into two steps:

(1) Delete the oldest transaction

The only thing a bit-vector needs to do is to left-shift one bit. Take item *a* as an example. *a*'s bit-vector is 1010 in the first window. If transaction with TID = 1 is deleted, *a*'s bit-vector becomes 0100. Now the most left bit represents the transaction with TID = 2 and the most right bit is meaningless and reserved for next step.

(2) Append the incoming transaction

After deleting the oldest transaction, the most right bit of the bit-vector is set corresponding to the incoming transaction. The bit-vectors of the items contained in the incoming transaction set its most right bit to one; the others set its most right bit to zero. Take item *a* as an example. *a*'s bit-vector is 0100 after deleting the oldest transaction. The incoming transaction is (b, d) (TID = 5) not containing *a* so *a*'s bit-vector is still 0100. *b*'s bit-vector is 1110 after deleting the oldest transaction. The incoming transaction contains *b* so *b*'s bit-vector is 1111 after appending the incoming transaction.



3.2.3 Counting Support with Bit-Vector

Concept of bit-vector can be extended to itemset. For example, the bit-vector of itemset (a, b) in the first window is 1010. That means (a, b) occurs in the transactions with TID = 1 and TID = 3.

Assume there are two itemsets X and Y and their corresponding bit-vector BIT_X and BIT_Y. The bit-vector of the itemset $Z = X \cup Y$ can be obtained by bitwise AND BIT_X and BIT_Y.

For example, the bit-vector of itemset (a, b) in the first window (Window #1) is 1010 which can be obtained by bitwise AND the bit-vectors of items *a* and *b*. That means (a, b) occurs in the first and the third transactions in the first sliding window. By bitwise AND between bit-vectors, candidates can be efficiently generated when building the lexicographical tree.

The support of each itemset can be obtained by counting how many bits in the bit-vector are set to one. For example, the support of itemset (a, b) is 2.

3.2.4 Building the New Closed Enumeration Tree (New-CET)

For improving the efficiency of CET in Moment, we propose a *new closed enumeration tree (New-CET)*. New-CET is basically a lexicographical tree. There are three important parts in New-CET:

(1) Bit-vectors of all items (1-itemsets)

Moment maintains an independent sliding window for counting support of each node in CET. Instead of independent sliding window to store current *N* transactions, information of these transactions is maintained by the bit-vectors of all items.

(2) Closed frequent itemsets in current window

Each closed frequent itemset only maintains its support.

(3) Hash table

For checking whether a frequent itemset is closed or not, we need a hash table to store all closed frequent itemsets with their supports as keys. Whenever a new frequent itemset is generated, we can judge if this frequent itemset is closed by hashing its support to the hash table. How to utilize the information of support to judge if a frequent itemset is closed is introduced in section 2.2.

Building New-CET is almost the same as building CET. The major difference is that New-CET only retains bit-vectors of items and closed frequent itemsets and bit-vectors are used to count supports of generated candidates.

When the total number of incoming transactions is less than the size of sliding window, New-Moment only records all item information as introduced in section 3.2.1. When the window is full, New-Moment call function *Build* to build the initial New-CET. From the bit-vectors we can know the supports of all items. New-Moment utilize depth-first procedure

to generate all possible candidates and check their supports. Because the candidates are generated by its parent and its parent's frequent siblings, we can obtain the supports by the method introduced in section 3.2.3. Then for each frequent candidate, we use hash table to check if the frequent candidate is closed. If the candidate is closed, it is inserted in the hash table. If the candidate is not closed, the node is not maintained in New-CET. Figure 3-5 shows the pseudo code of building New-CET

Build (n_I, N, S)

1: **if** support(n_I) $\geq S \cdot N$ **then**
2:     **if** leftcheck(n_I) = false **then**
3:         **foreach** frequent sibling n_K of n_I **do**
4:             generate a new child $n_{I \cup K}$ for n_I;
5:             bitwise AND BIT_I and BIT_K to obtain $BIT_{I \cup K}$;
6:         **foreach** child n_I' of n_I **do**
7:             Build(n_I', N, S);
8:         **if** $\nexists$a child n_I' of n_I such that
              support(n_I') = support(n_I) **then**
9:             retain n_I as a closed frequent itemset;
10:            insert n_I into the hash table;

Fig 3-5. Pseudo code of building New-CET

n_I is a tree node, N is the window size and S is minimum support. Each n_I has a corresponding bit-vector BIT_I to store the information of sliding window. Except the bit-vectors of items, the BIT_I for a node n_I only exists in counting support of a new candidate.

Figure 3-6 shows the New-CET in the first window by previous example when generating new candidates from item *a*. For simplicity, hash table is not displayed in it. By the bit-vectors of items, we know that items *a*, *b*, and *c* are frequent items. Take item *a* as an example, new candidates (a, b) and (a, c) are generated. By bitwise AND bit-vectors of items *a* and *b*, we can obtain that the support of (a, b) is 3. In the same way, the support of (a, c) is 2

and the support of (a, b, c) is 2. For generating candidates below item *a*, the bit-vectors of (a, b), (a, c), and (a, b, c) are temporarily maintained in the memory.

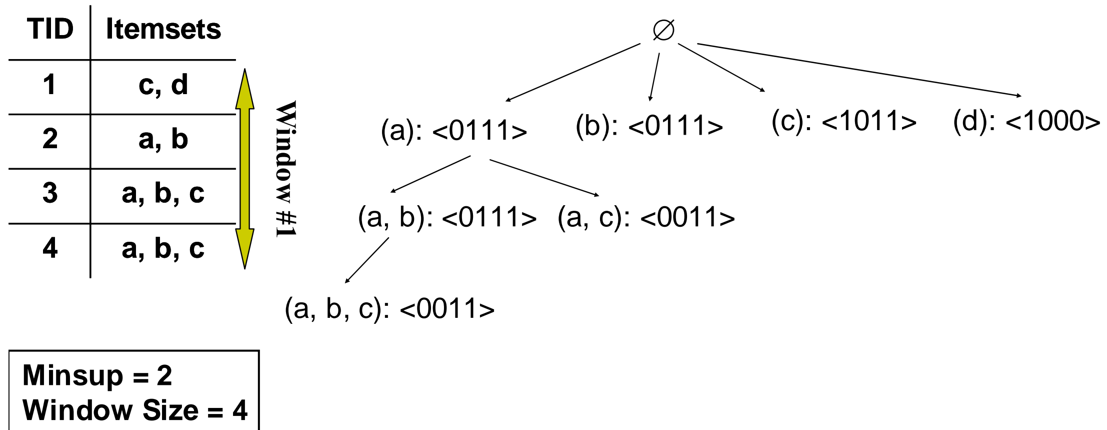


Fig 3-6. New-CET in the first window after generating new candidates from item *a*

Figure 3-7 shows the New-CET after checking if each frequent candidate is closed. The tree nodes with squares are closed frequent itemsets. By checking support with hash table, we can know that frequent itemset (a, c) is not closed. So New-Moment eliminates this node and other frequent candidates are marked as closed frequent itemsets. Although item *a* is not closed, New-Moment still maintains the bit-vector of item *a*. After the sub-tree of item *a* is checked, the bit-vectors in this sub-tree are eliminated. New-Moment only keeps the supports of closed frequent itemsets.

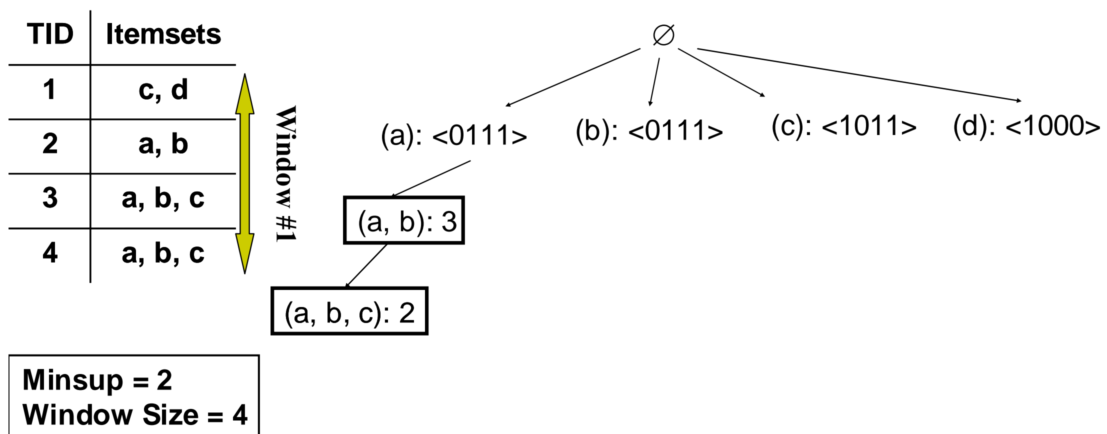


Fig 3-7. New-CET in the first window after checking closed frequent itemsets

Figure 3-8 shows the New-CET when *Build* is done. The sub-tree generations of item *b* and *c* are the same as item *a*. Item *c* is a new closed frequent itemset.

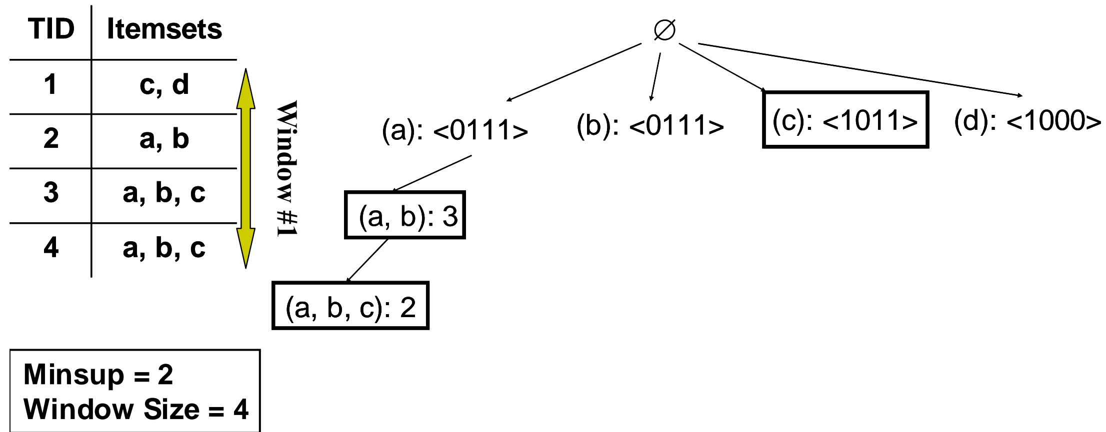


Fig 3-8. New-CET in the first window (Window #1)

3.2.5 Deleting the Oldest Transaction in Window Sliding

Deleting the oldest transaction is our first step of window sliding. All bit-vectors of items are left-shifted one bit first and all items in the deleted transaction are kept. This can be done by observing the most left bit before left-shifting. After modification of bit-vectors of items, New-Moment begins to modify New-CET.

There is only one case for deleting the oldest transaction: original closed frequent itemsets in the New-CET becomes non-closed frequent itemsets or infrequent itemsets. For checking this situation, New-Moment traverses the New-CET again to check the supports of the existing node in the New-CET. Because just the subsets of the deleted transaction are the possible infrequent itemsets, only the sub-trees of the items in the deleted transaction need to be checked. The traversing method is almost the same as building the initial New-CET, called function *Delete*. The difference is that Delete generates the entire lexicographical tree including the itemsets whose supports are $(S \cdot N - 1)$. This is because supports of some closed frequent itemsets in previous window would be $(S \cdot N)$ and then becomes $(S \cdot N - 1)$ after deletion.

Figure 3-9 shows the New-CET after deleting the oldest transaction. In the above example, the deleted transaction is (c, d). Only the sub-trees of items *c* and *d* need to be checked. We find that item *c* is no longer a closed frequent itemset. Item *d* is infrequent and we do not need to check its sub-tree. Figure 3-10 shows the pseudo code of deleting the oldest transaction after left-shifting all bit-vectors of 1-itemsets.

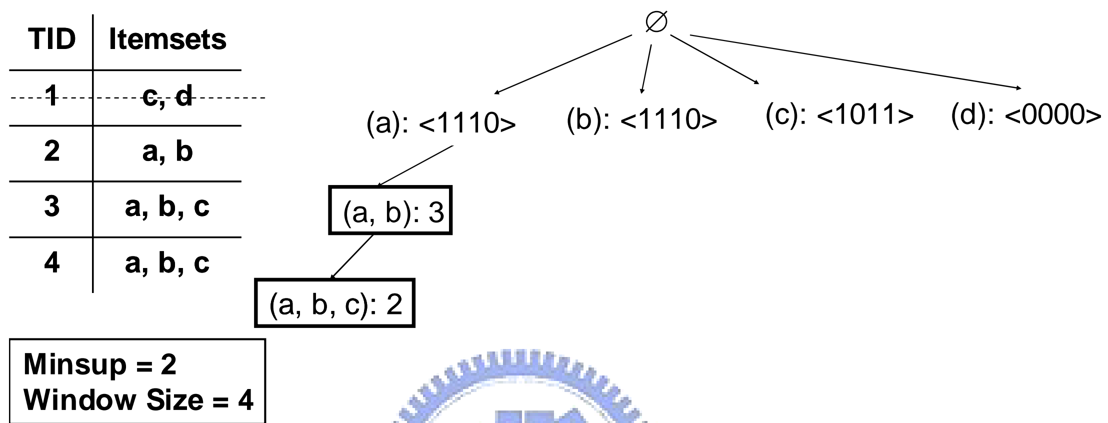


Fig 3-9. New-CET after deleting the oldest transaction

Delete (n_I, *N*, *S*)

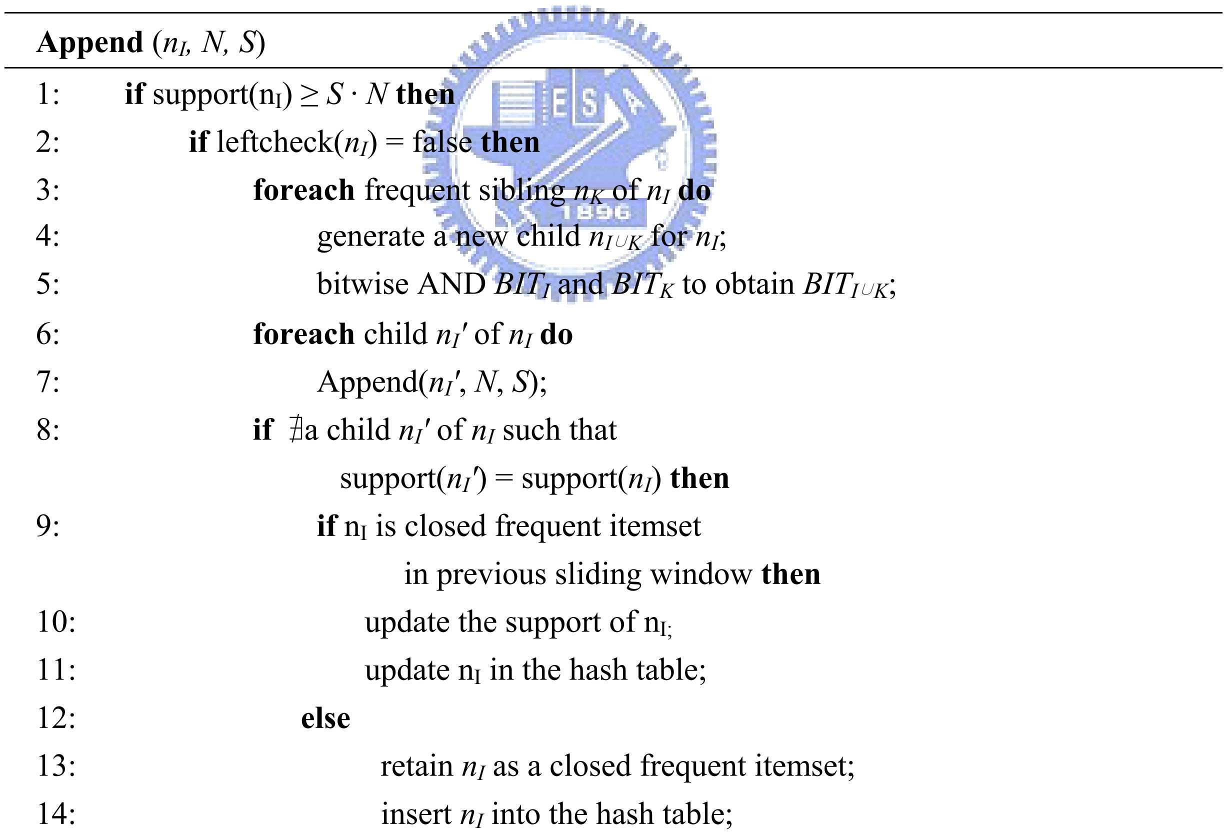
```
1:   if n_I is not relevant to the deleted transaction then
2:       return;
3:   else if support(n_I) ≥ (S · N – 1) then
4:       foreach sliding n_K of n_I
                 whose support ≥ (S · N – 1) do
5:           generate a new child n_{I∪K} for n_I;
6:           bitwise AND BIT_I and BIT_K to obtain BIT_{I∪K};
7:       foreach child n_I′ of n_I do
8:           Delete(n_I′, N, S);
9:       if support(n_I) ≥ S · N then
10:          if leftcheck(n_I) = false then
11:              if n_I is closed frequent itemset
                         in previous sliding window then
12:                  update the support of n_I;
13:                  update n_I in the hash table;
14:              else
15:                  retain n_I as a closed frequent itemset;
16:                  insert n_I into the hash table;
17:          else    //leftcheck(n_I) = true
18:              if n_I is closed frequent itemset
                         in previous sliding window then
19:                  mark n_I as non-closed frequent itemset;
20:                  eliminate n_I from the hash table;
21:      else    //support(n_I) < S · N
22:          if n_I is closed frequent itemset
                     in previous sliding window then
23:              mark n_I as non-closed itemset;
24:              eliminate n_I from the hash table;
```

Fig 3-10. Pseudo code of deleting the oldest transaction in window sliding

3.2.6 Appending the Incoming Transaction in window sliding

Appending the incoming transaction is our second step of window sliding. The most right bits of all the bit-vectors of items are set corresponding to the items contained in the incoming transaction. After modification of bit-vectors of items, New-Moment begins to modify New-CET. Only the sub-trees of the items in the inserted transaction need to be checked.

The method of traverse the New-CET for adding a new transaction, called function *Append*, is the same as function *Build*. A little difference is that the supports of existing closed frequent itemsets in the hash table need to be modified. Figure 3-11 shows the pseudo code of appending the incoming transaction after setting the most right bit in each bit-vector of 1-itemset.



Append (n_I, N, S)

1: **if** support(n_I) $\geq S \cdot N$ **then**
2:     **if** leftcheck(n_I) = false **then**
3:         **foreach** frequent sibling n_K of n_I **do**
4:             generate a new child $n_{I \cup K}$ for n_I;
5:             bitwise AND BIT_I and BIT_K to obtain $BIT_{I \cup K}$;
6:         **foreach** child n_I' of n_I **do**
7:             Append(n_I', N, S);
8:         **if** $\nexists$a child n_I' of n_I such that support(n_I') = support(n_I) **then**
9:             **if** n_I is closed frequent itemset in previous sliding window **then**
10:                 update the support of n_I;
11:                 update n_I in the hash table;
12:             **else**
13:                 retain n_I as a closed frequent itemset;
14:                 insert n_I into the hash table;

Fig 3-11. Pseudo code of appending the incoming transaction in window sliding

In the above example, the inserted transaction is (a, c, d). New-Moment checks the sub-trees of items *a*, *c*, and *d* and find that itemsets (a) and (a, c) are new closed frequent

itemsets. Figure 3-12 shows the New-CET after appending the incoming transaction. This is also the New-CET in the second sliding window.

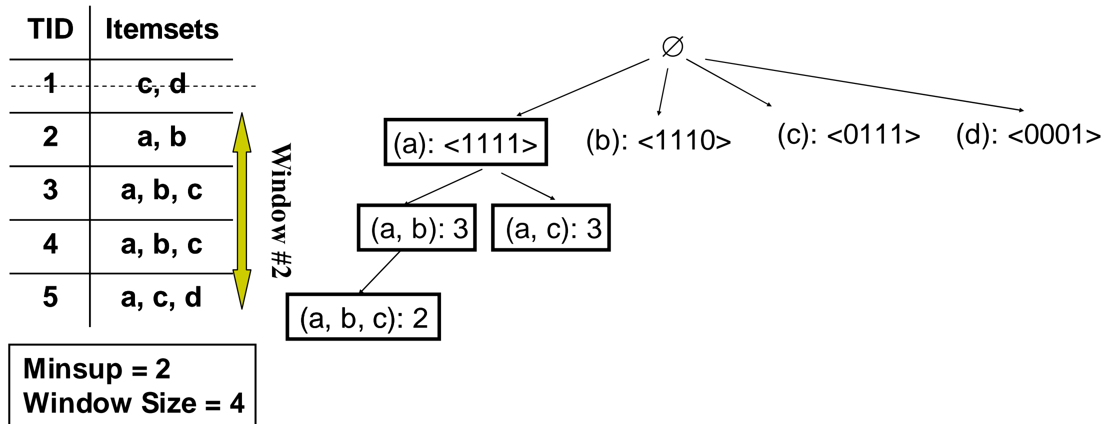


Fig 3-12. New-CET after appending the incoming transaction (Window #2)

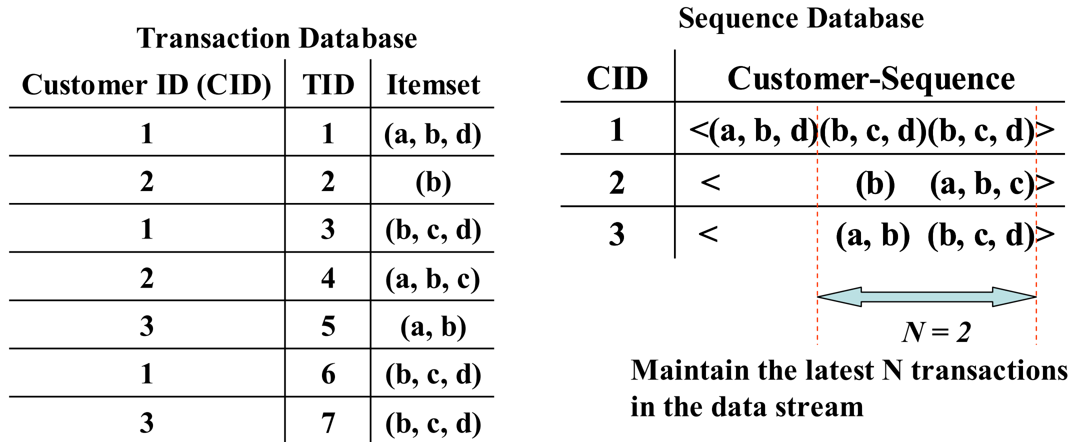
Chapter 4

Incremental SPAM (IncSPAM): Mining Sequential Patterns

Sequential pattern mining is more complicated than mining frequent itemsets, especially in the stream environment. In previous researches, there is no general processing model for handling a data stream with a transaction unit. Incremental SPAM (IncSPAM) provides a suitable sliding window model in a data stream. It receives transactions from the data stream and uses a brand-new concept of bit-vector, *Customer Bit-Vector Array with Sliding Window (CBASW)*, to store the information of items for each customer. Then IncSPAM uses a lexicographic sequence tree to maintain the sequential patterns in the current window. For speeding up the maintaining process, IncSPAM uses *index sets* to store the first occurring positions in all customer-sequences for a tree node. Whenever a new transaction comes, CBASWs and the lexicographic tree are modified incrementally. Each transaction can be analyzed in few seconds. Finally a weight function is adopted in this sliding window model. The weight function can judge the importance of a sequence and ensure the correctness of the sliding window model.

4.1 A New Concept of Sliding Window for Sequences

Original sliding window model keeps the latest transactions in a data stream. In the mining of sequential patterns, transactions in a data stream belong to many customers. IncSPAM keeps the latest *N* transactions for each customer in a data stream and *N* is called window size. Each customer maintains its own sliding window. Figure 4-1 shows an example of this sliding window model.



Transaction Database

Customer ID (CID)	TID	Itemset
1	1	(a, b, d)
2	2	(b)
1	3	(b, c, d)
2	4	(a, b, c)
3	5	(a, b)
1	6	(b, c, d)
3	7	(b, c, d)

Sequence Database

CID	Customer-Sequence
1	<(a, b, d)(b, c, d)(b, c, d)>
2	< (b) (a, b, c)>
3	< (a, b) (b, c, d)>

Fig 4-1. An example for the new concept of sliding window

In Figure 4-1, the mining system has received 7 transactions. Assume that the window size of each customer is 2 (each customer keeps the latest 2 transactions in a data stream). There are three transactions belonging to customer #1: transactions with TID = 1, 3, and 6. Only transactions with TID = 3 and 6 are stored in the sliding window of customer #1 (marked by the two-way arrow). In the same concept, the sliding windows of customer #2 and customer #3 are also displayed in Figure 3-9. This example will be used through the introduction of IncSPAM.

4.2 Customer Bit-Vector Array with Sliding Window (CBASW)

IncSPAM also uses bit-vectors to store the information of the sliding window. The concept of bit-vector is almost the same as in section 3.2.1. The difference is that each customer has his own bit-vectors for all items to store the information of his sliding window. Table 4-1 shows the bit-vectors of each customer in Figure 4-1. All these bit-vectors can be collected as a unique data structure for each customer.

	Customer #1	Customer #2	Customer #3
a	00	01	10
b	11	11	11
c	11	01	01
d	11	00	01

Table 4-1. Bit-vectors of all items for all customers

Definition of Customer Bit-Vector Array with Sliding Window (CBASW): For each customer-sequence *c*, we keep the latest *N* transactions. *N* is called *window size*. Each bit-vector of item *i* contains *N* bits to represent the occurrences of *i* in the latest *N* transactions.

Figure 4-2 shows an example of CBASW. Each bar with a customer id is a CBASW which contains all bit-vectors of all items for a customer.

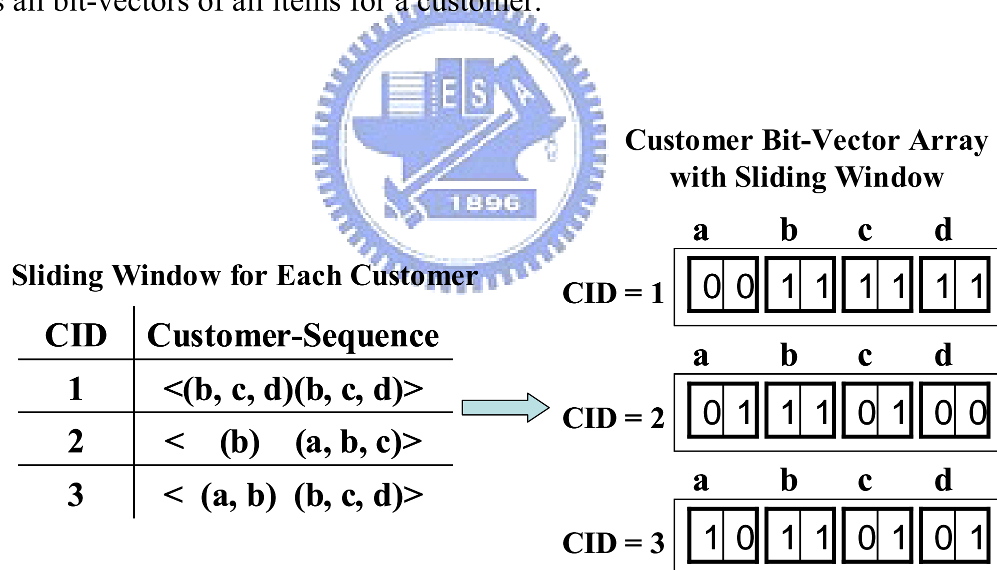


Fig 4-2. An example of CBASW

A lexicographical tree is needed in the mining process of Incremental SPAM. Although CBASWs keep all information of items (1-itemsets) for all customers, they are not efficient enough to be used to build and modify a lexicographical tree. In the next section the concept of *Index Set* will be introduced to speed up the process.

4.3 Index Set ρ-idx

Building and modifying lexicographical tree for mining sequential patterns is complicated than for mining frequent itemsets. The number of candidate sequence is huge. *Index Set* for a sequence only stores the first positions in all customer-sequences. The memory usage of these positions is less than the memory usage of bit-vectors.

Definition of Index Set: For a sequence ρ, the *first* occurring position in a customer-sequence *c* of ρ is recorded as $\rho\text{-}pos_c$. If ρ is not in *c*, $\rho\text{-pos}_c = 0$. The collection of these ρ-pos values in the order of customer id is called an *index set ρ-idx*. For convenience, $\rho\text{-pos}_c$ can be represented as ρ-idx[c].

Take the CBASWs in Figure 4-2 as an example, the index sets of 1-sequences (items) are listed in Figure 4-3. Each number in the array represents the first position of a sequence ρ in each customer. By counting the number of positions which is not zero in an index set, we can obtain the support of a sequence. Take the 1-sequence <(a)> as an example. The number of non-zero positions in <(a)>-idx is 2. That means the sequence <(a)> exists in two customer-sequence (CID = 2 and CID = 3). So the support of <(a)> is 2. The support of each sequence is records after the index set.

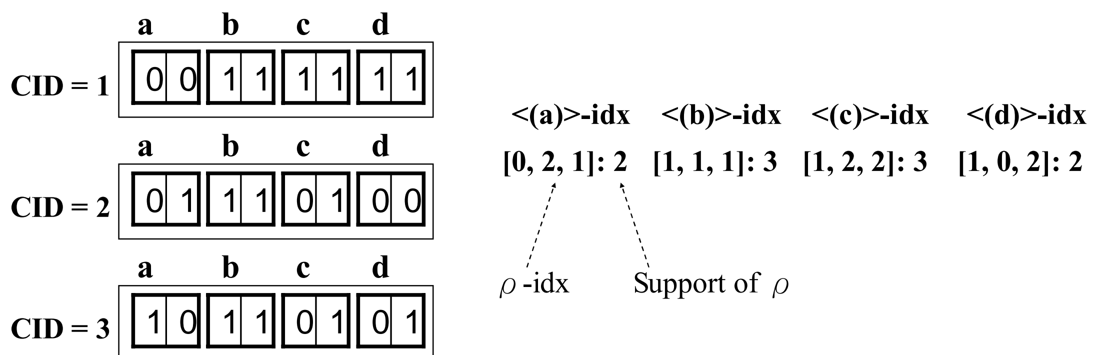


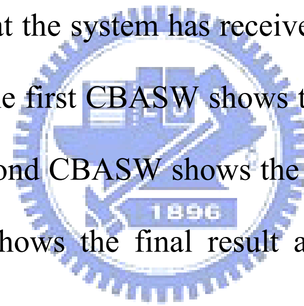
Fig 4-3. An example of index sets

4.4 Maintaining Information of Items in Window Sliding with CBASW and ρ-idx

Whenever a new transaction comes from the data stream, its CID is checked to find out which CBASW needs to be modified. Each bit-vector in this CBASW is modified by the incoming transaction. As in section 3.2.2, if the number of transactions of the customer exceeds the size of window, window sliding is performed.

Window sliding process is the same in section 3.2.2. Each bit-vector left-shifts one bit to eliminate the oldest transaction and sets the most right bit by the incoming transaction. The bit-vectors of the items in the incoming transaction set their most right bits to one; the others set their most right bits to zero. Figure 3-11 shows an example of this sliding process.

In Figure 4-4, we assume that the system has received 5 transactions and only the CBASW of customer #1 is observed. The first CBASW shows the situation before the new transaction with TID = 6 coming. The second CBASW shows the result after left-shifting each bit-vector one bit. The third CBASW shows the final result after setting the most right bit by the incoming transaction.



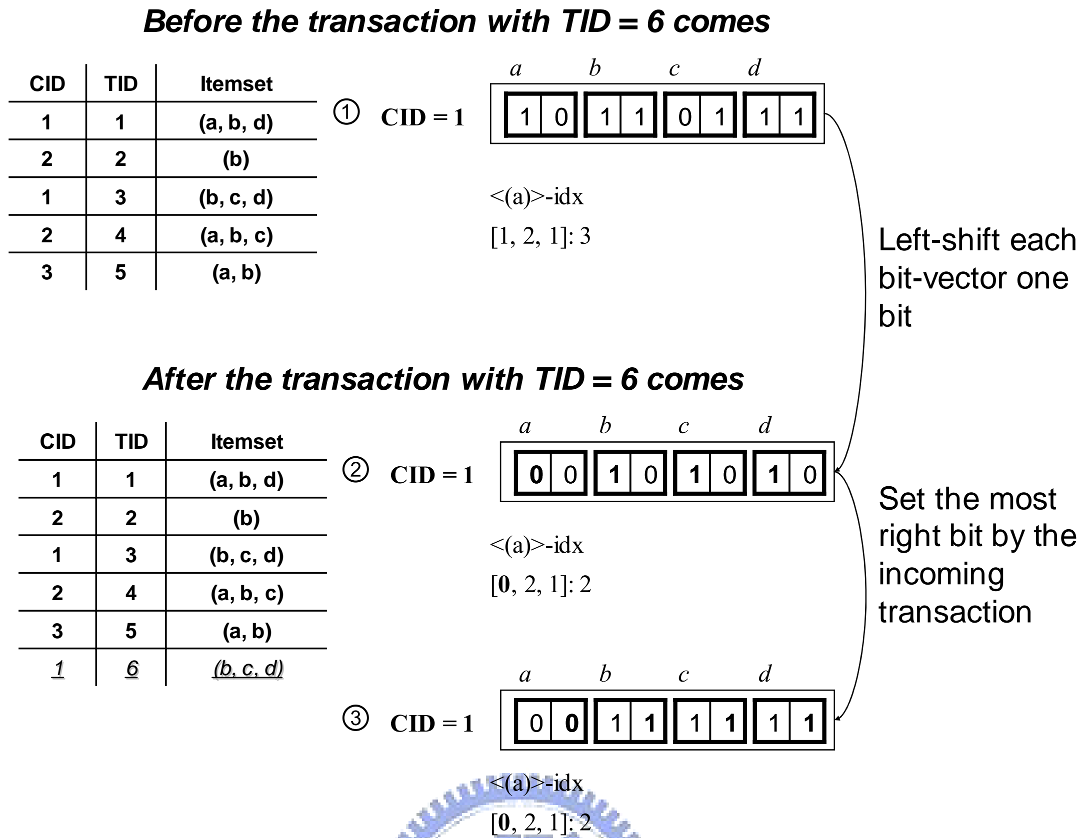


Fig 4-4. An example of window sliding in a CBASW

ρ-idx of each item (1-sequence) is maintained according to the CBASWs. Whenever a window sliding for a CBASW of a customer *c* is performed, each ρ-idx[c] is decreased once. If a ρ-idx[c] becomes zero after decreasing, the bit-vector of ρ is checked to find out the new first occurring position. If ρ does not exist in this customer-sequence anymore, the ρ-idx[c] is set to zero.

In the first CBASW of Figure 4-4, <(a)>-idx[1] is 1. After window sliding, <(a)>-idx[1] is decreased to 0. In the third CBASW of Figure 3-11, <(a)>-idx[1] is 0 because sequence <(a)> does not exist in customer-sequence of customer #1. The support of sequence <(a)> also decreases to 2.

4.5 Building of Lexicographical Sequence Tree

Section 4.4 introduces the algorithm for maintaining information of items in a data stream. After the items of the incoming transaction are processed, these items are used to generate all possible sequences and check their supports. SPAM algorithm [11] presents a concept of lexicographical sequence tree to archive this goal.

Assume that there is a ***lexicographical ordering*** **≤** of the items I in the data stream. If item i occurs before item j in the ordering, then we denote this by $i \leq_I j$. This ordering can be extended to sequences by defining $s_a \leq s_b$ if s_a is a subsequence of s_b. If s_a is not a subsequence of s_b, then there is no relationship in this ordering.

The root of the tree is labeled with ∅. Recursively if n is a node in the tree, then n's children are all nodes n' such that $n \leq n'$ and $\forall m \in T$: $n' \leq m \Rightarrow n \leq m$. Figure 4-5 shows an example of a lexicographic sequence tree. This graph is modified from [11]. Here shows a sub-tree of sequence tree for two items a and b to the fourth level.

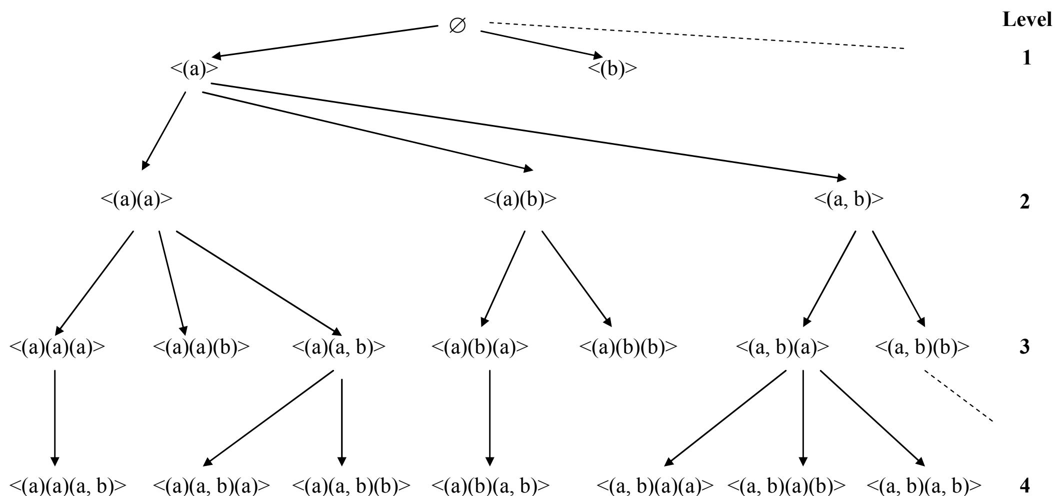
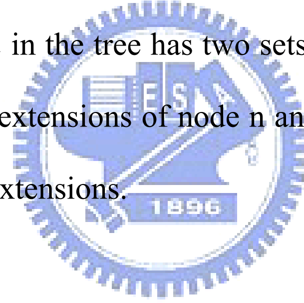


Fig 4-5. A lexicographic sequence tree example

Each sequence in the sequence tree can be considered as either a sequence-extended sequence or an itemset-extended sequence. A ***sequence-extended sequence*** is a sequence by adding a 1-itemset to the end of its parent's sequence in the lexicographical tree, like <(a)(a)(a)> and <(a)(a)(b)> generated from <(a)(a)> in Figure 4-5. An ***itemset-extended sequence*** is a sequence by adding an item to the last itemset in the parent's sequence in the lexicographical tree. The order of the item is greater than any item in the last itemset, like <(a)(a, b)> generated from <(a)(a)> in Figure 4-5.

If we generate sequences by traversing the tree, then each node in the tree can generate sequence-extended children sequences and itemset-extended children sequences. We refer to the process of generating sequence-extended sequences as the ***sequence-extension step (S-step)*** and the process of generating itemset-extended sequences as the ***itemset-extension step (I-step)***. Thus each node n in the tree has two sets: $\boldsymbol{S_n}$, the set of candidate items that are considered for possible S-step extensions of node n and $\boldsymbol{I_n}$, the set of candidate items that are considered for possible I-step extensions.



4.6 Counting Support with ρ-idx

For a sequence ρ, ρ-idx stores the first positions of ρ in all customer-sequences. In the lexicographical tree of Incremental SPAM, each node, representing ρ, uses ρ-idx to count the support of sequence ρ. We introduce the method in two different steps.

4.6.1 Counting Support in S-step

Assume there are a sequence α and an appended 1-itemset β. An S-extended sequence γ is generated by α and β. Our goal is to use α-idx and β-idx to generate γ-idx and count the support of γ.

The first step α-idx[c] and β-idx[c] for each customer *c* are checked. If either α-idx[c] or β-idx[c] is zero, γ-idx[c] is set to zero. That means γ can not exist in the customer-sequence of *c*. If there is a *c* that α-idx[c] and β-idx[c] are both not zero, γ may exist in this customer-sequence. The corresponding position values have to be checked. There are two cases for α-idx[c] and β-idx[c]:

(Case 1) α-idx[c] < β-idx[c]: That means *α* appears before *β* and *γ* does exist in the customer-sequence of *c*. γ-idx[c] is set to β-idx[c].

(Case 2) α-idx[c] ≥ β-idx[c]: In this case we do not have enough information for judging if *γ* exists. The CBASW of customer *c* needs to be further checked. We denote the bit-vector of item *i* in the CBASW of customer *c* as $CBASW_c(i)$.

A left-shit operation is performed on $CBASW_c(\beta)$ by α-idx[c] bits. If the result of shifting is a non-zero bit-vector, *γ* exists in the customer-sequence of *c*. Assume the position of the first non-zero bit in the result is *h*. The first position of *γ*, γ-idx[c], is set to α-idx[c] + *h*. Otherwise *γ* does not exist in the customer-sequence of c and γ-idx[c] is set to zero.

Finally the support of γ can be counted by the number of non-zero positions.

4.6.2 Counting Support in I-step

Assume a sequence *α*, an appended item *T*, and an I-extended sequence *γ* generated by *α* and *T*. Our goal is to use α-idx and T-idx to generate γ-idx and count the support of *γ*.

As in S-step, α-idx[c] and T-idx[c] for each customer *c* are also checked. If either α-idx[c] or T-idx[c] is zero, γ-idx[c] is set to zero. If there is a *c* that α-idx[c] and T-idx[c] are both not zero, the corresponding position values have to be checked. There are also two cases for α-idx[c] and T-idx[c] in I-step:

(Case 1) α-idx[c] = β-idx[c]: That means the last itemset of *α* and item *T* are in the same position of customer-sequence of *c*. By the definition of itemset-extended sequence, *γ* exists. γ-idx[c] is set to β-idx[c].

(Case 2) α-idx[c] ≠ β-idx[c]: The further check in CBASW of c is performed. Assume X is the last itemset of α. A bit-vector BIT_X is obtained by bitwise AND $CBASW_c(x_1)$, $CBASW_c(x_2)$, …, $CBASW_c(x_k)$, where x_i is an item contained in X. Then BIT_X is left-shifted (α-idx[c] – 1) bits. If the resultant sequence is non-zero, γ exists in this customer-sequence. Assume the position of the first non-zero bit in the result is h. The first position of γ, γ-idx[c], is set to [(α-idx[c] – 1) + h]. Otherwise γ does not exist in the customer-sequence of c and γ-idx[c] is set to zero.

As in section 4.6.1, the support of γ can be counted by the number of non-zero positions.

4.7 The Entire Process of Incremental SPAM (IncSPAM)

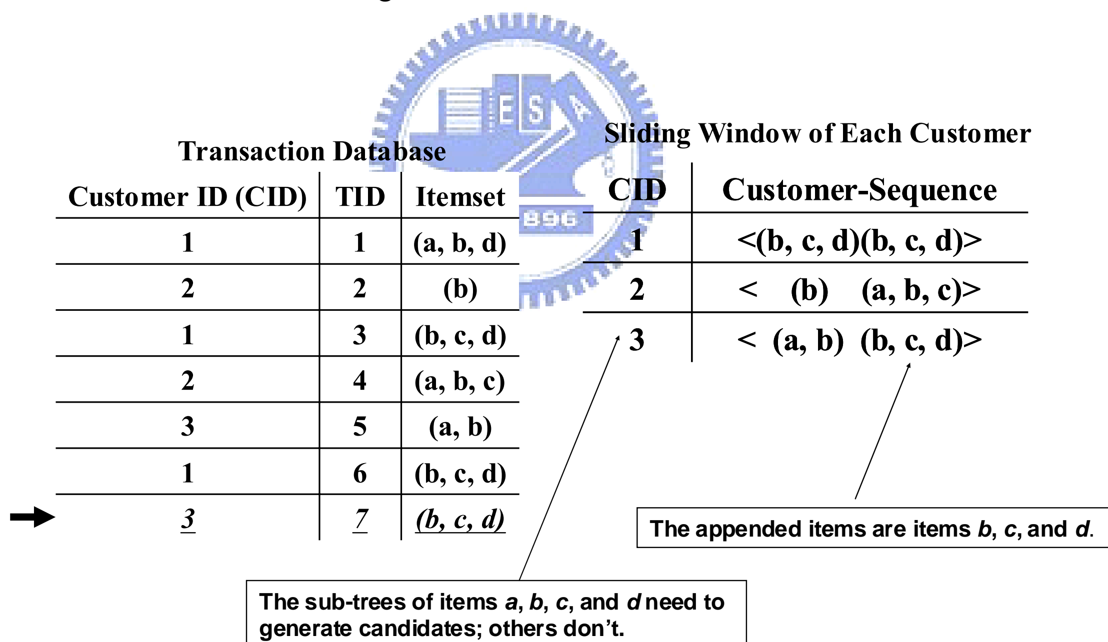
Finally we introduce entire IncSPAM algorithm for the mining of sequential patterns. Figure 4-6 shows the main function of IncSPAM.

IncSPAM (*S*, *d*, *N*)	
1:	**foreach** incoming transaction from the data stream **do**
2:	find out which customer *c* the incoming transaction belongs to;
3:	update the CBASW of this customer by the incoming transaction;
4:	store all the frequent 1-sequences to *F*;
5:	MaintainTree(*c*, *F*);

Fig 4-6. Main function of Incremental SPAM

The CBASW of each customer is modified from line 1 to line 4. After the modification of CBASWs is finished, function *MaintainTree* is called. Function *MaintainTree* maintains sequential patterns dynamically in a lexicographic sequence tree. There are some cases about incremental mining of sequential patterns. Assume that a new transaction ω comes in. ω belongs to customer c. The lexicographical tree T is updated to T':

- *A pattern which is frequent in T is still frequent in T'*: We only needs to update its ρ-idx and support
- *A pattern which is not in T appears in T'*: A new pattern is generated because of the incoming transaction. By the Apriori [1] property, since prefix of the new pattern must also be frequent, we only need to generate candidates from the leaf nodes of *T*. There are two ways to reduce the number of candidates to be generated: (1) We only consider the items in the incoming transaction to append on the leaf nodes because the new patterns must contain these items in the end. (2) The incoming transaction only belongs to a specific customer *c* so the generated candidates must begin with the items in the customer-sequence of *c*. Figure 4-7 shows an example after sliding the CBASW of customer #3. The incoming transaction is TID = 7.



Transaction Database

Customer ID (CID)	TID	Itemset
1	1	(a, b, d)
2	2	(b)
1	3	(b, c, d)
2	4	(a, b, c)
3	5	(a, b)
1	6	(b, c, d)
3	*7*	*(b, c, d)*

Sliding Window of Each Customer

CID	Customer-Sequence
1	<(b, c, d)(b, c, d)>
2	< (b) (a, b, c)>
3	< (a, b) (b, c, d)>

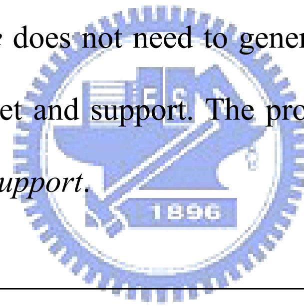
Fig 4-7. Reducing the generated candidates

- *A pattern which is in T does not exist in T'*: The pattern becomes infrequent because of window sliding. We directly delete the node and its sub-tree.

```
MaintainTree (c, F)
1:  foreach tree node n who's representing item i is in F do
2:      if i exists in the customer-sequence of c then
3:          Generate(c, n);
4:      else   //i does not exist in c
5:          Update(c, n);
```

Fig 4-8. The pseudo code of function MaintainTree

Figure 4-8 shows the pseudo code of *MaintainTree*. Function *Generate*, as shown in Figure 4-9, uses S-step and I-step to generate all possible children with the principles mentioned above for each tree node. If the child does not exist in the lexicographical tree, *Generate* creates a new tree node for this child. If the child is in the lexicographical tree, *Generate* only updates the index set and support of this child. Function *Update*, as shown in Figure 4-10, is simpler than *Generate*. *Update* does not need to generate children. *Update* only checks each tree node to update its index set and support. The process of updating the index set and the support is in Function *UpdateSupport*.



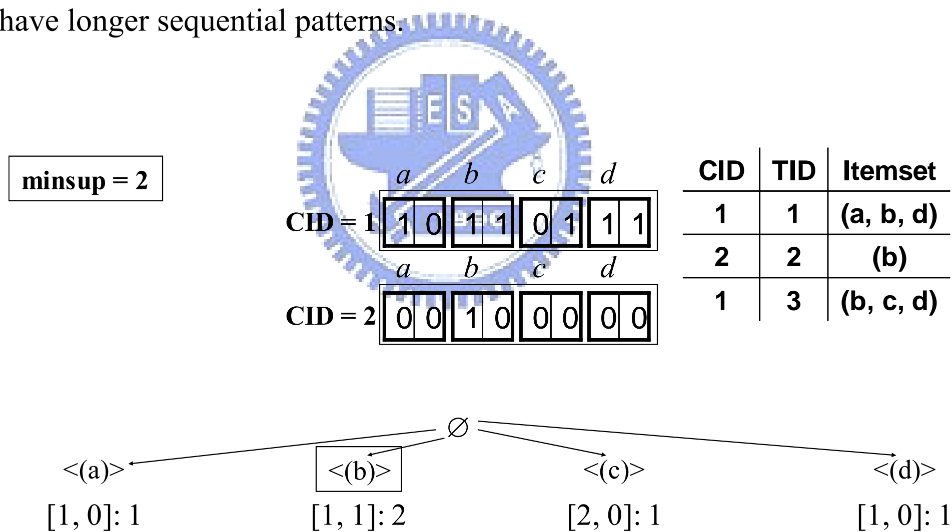
```
Generate (c, n)
1:  foreach existing child n′ of n do
2:      UpdateSupport(c, n′);
3:      if the support of n′ < S then
4:          eliminate n′ and its sub-tree;
5:  generate candidates of n by S-step and I-step;
6:  foreach generated candidate x of n do
7:      count the support of x;
8:      if the support of x ≥ S then
9:          x is a child of n;
10: foreach child n′ of n do
11:     Generate(c, n′);
```

Fig 4-9. The pseudo code of function Generate

```
Update (c, n)
1:    foreach existing child n′ of n do
2:        UpdateSupport(c, n′);
3:        if the support of n′ < S then
4:            eliminate n′ and its sub-tree;
5:    foreach child n′ of n do
6:        Update(c, n′);
```

Fig 4-10. The pseudo code of function Update

We use the previous example to show the process of IncSPAM. Assume three transactions have been received by IncSPAM. Figure 4-11 shows the CBASWs and the lexicographic sequence tree. We mark the sequential patterns with squares. Each tree node maintains an index set to record its support. In Figure 4-11, only 1-sequence <(b)> is frequent so the tree does not have longer sequential patterns.



CID	TID	Itemset
1	1	(a, b, d)
2	2	(b)
1	3	(b, c, d)

Fig 4-11. The lexicographic sequence tree when the third transaction comes in

When the fourth transaction (a, b, c) comes in, CBASW of customer 2 has been modified and 1-sequences <(a)> and <(c)> become new sequential patterns. By the extension methods, S-step and I-step, longer candidates have been generated. IncSPAM checks the support of each candidate using index set and keeps sequential patterns in the lexicographic sequence tree. Figure 4-12 shows the result after the four transaction comes.

minsup = 2

	a	*b*	*c*	*d*
CID = 1	1 0	1 1	0 1	1 1
CID = 2	0 1	1 1	0 1	0 0

CID	TID	Itemset
1	1	(a, b, d)
2	2	(b)
1	3	(b, c, d)
2	4	(a, b, c)

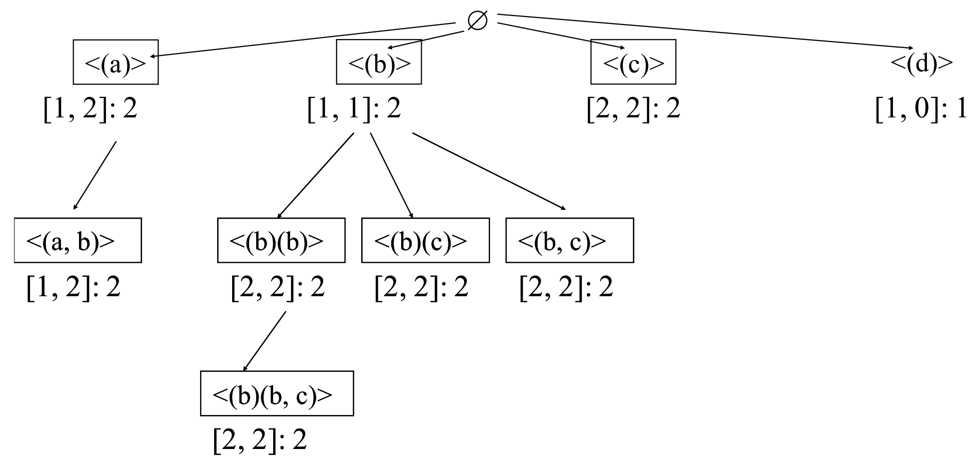


Fig 4-12. The lexicographic sequence tree after the fourth transaction comes in

When the fifth transaction (a, b) comes in, IncSPAM updates the CBASWs and the index sets in the lexicographic sequence tree. Then IncSPAM needs to generate new candidates to find if there are new sequential patterns. Figure 4-13 shows the lexicographic sequence tree and CBASWs after the fifth transaction comes. In the figure the tree nodes linked by the dotted arrows means the candidates IncSPAM needs to check. The fifth transaction belongs to customer 3 so only the sub-trees of items that exist in the customer-sequence 3 need to generate candidates. In Figure 4-13 we can know that the sub-trees of items *a* and *b* need to generate new candidates. Then we find that the new candidates <(a)(a)>, <(a)(b)>, and <(b, c)(b)> are not frequent. IncSPAM does not keep these tree nodes.

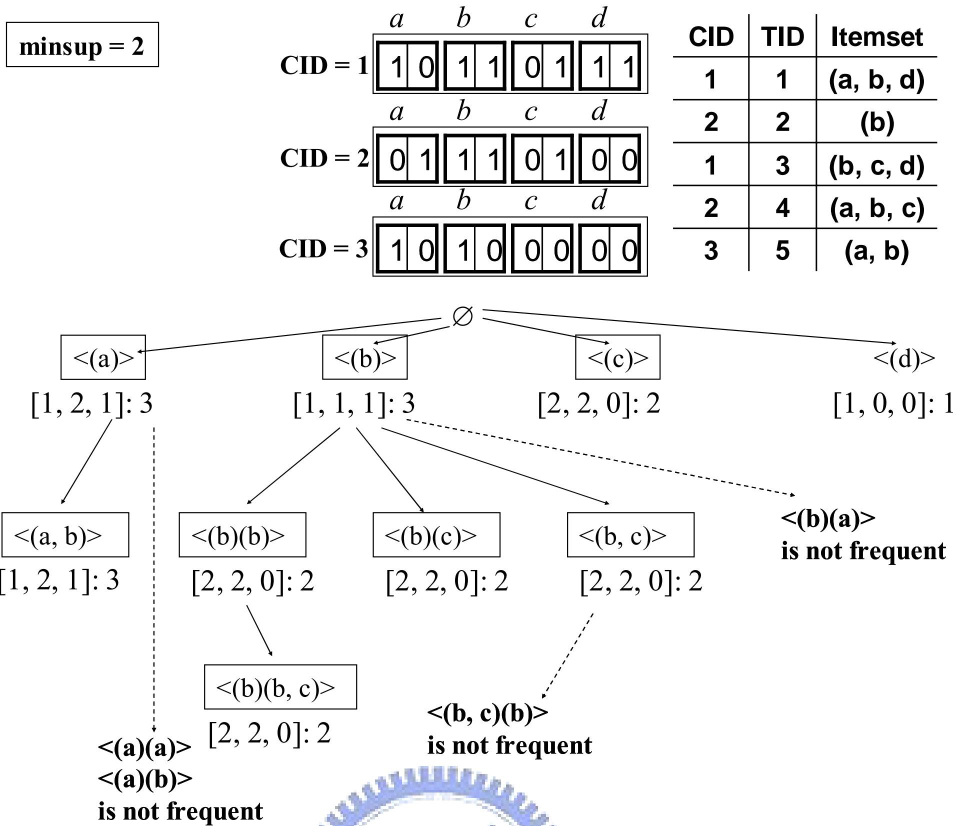
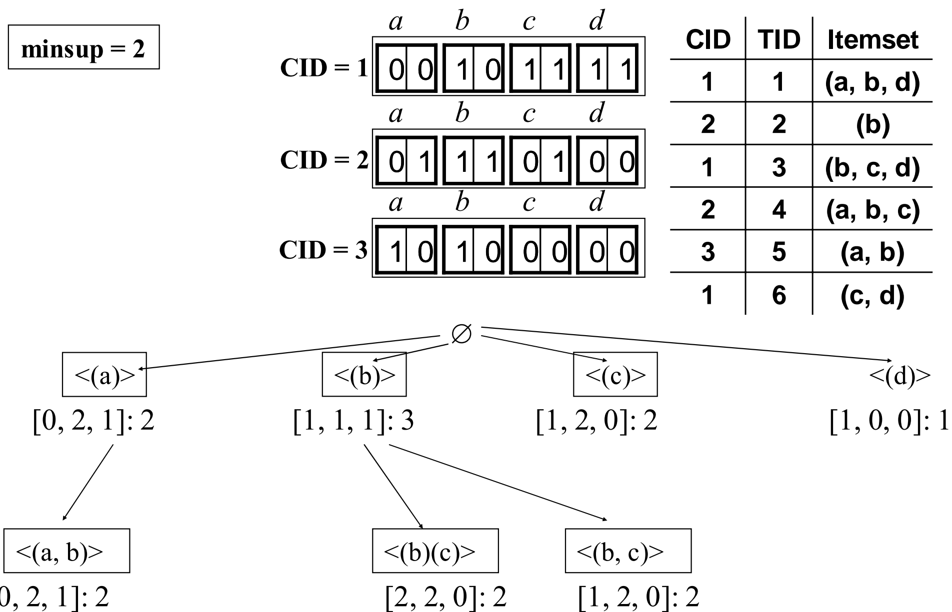


Fig 4-13. The lexicographic sequence tree after the fifth transaction comes in

Figure 4-14 shows the result after the sixth transaction comes in. IncSPAM finds that the existing tree node <(b)(b)> becomes infrequent. In this case IncSPAM directly deletes the tree node <(b)(b)> and its sub-tree <(b)(b, c)>.



CID	TID	Itemset
1	1	(a, b, d)
2	2	(b)
1	3	(b, c, d)
2	4	(a, b, c)
3	5	(a, b)
1	6	(c, d)

Fig 4-14. The lexicographic sequence tree after the sixth transaction comes in

4.8 Weight of Customer-Sequence

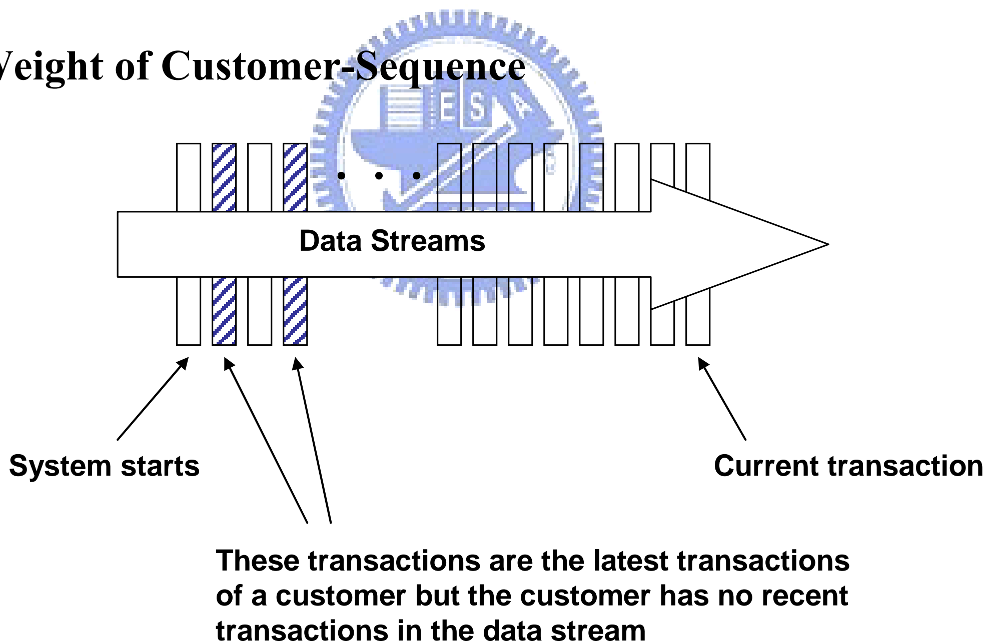


Fig 4-15. The transactions of a customer with no recent records in a data stream

In IncSPAM algorithm, each customer maintains a sliding window to keep the latest N transactions and the system mines sequential patterns from all customer-sequences. But some customers may have no transactions in recent time in the data stream. These customer-sequences with out-of-date transactions would result in a false positive problem in

our algorithm. The supports of some patterns generated by the system are overly counted. Figure 4-15 shows an example of these out-of-date transactions in a data stream. The customer-sequence with these out-of-date transactions is less important than other customer-sequences.

A concept of weight can be used to judge the importance of customers. Each customer-sequence c has its own weight w_c, $0 \leq w_c \leq 1$. Each weight w_c is decayed if the incoming transaction does not belong to c. When a transaction of c comes, the weight w_c is set to one. A decay function is used to compute the weights of customer-sequences when a new transaction is coming in:

$$w_c = 1 \times d^p$$

d, a ***decay-rate*** defined by users, can decide how fast a customer-sequence is decayed. p is a ***decay-period*** of a customer-sequence which is the number of transactions between the incoming transaction and the latest transaction of c. p can be written as below:

p = (incoming transaction TID – the latest transaction TID of c)

In our proposed algorithm, the concept of the decay-rate d is adopted from [26]. d is defined as:

$$d = b^{(-\frac{1}{h})} \quad (b > 1, h \geq 1, b^{-1} \leq d < 1)$$

Decay-base b: the amount of weight reduction per decay-unit

Decay-base-life h: the number of decay-units that makes the current weight be 1/b

Figure 4-16 shows an example of calculating the weights of customers. Assume the incoming transaction is the transaction with TID = 7 and the decay rate d = 0.9. The latest transaction of each customer is pointed by an arrow. Let us take customer #1 as an example. The latest transaction of customer #1 is the transaction with TID = 6. So the weight of customer-sequence of customer #1, w_1, is equal to $0.9^{(7-6)}$.

CID	TID	Itemset
1	1	(a, b, d)
2	2	(b)
1	3	(b, c, d)
2	4	(a, b, c)
3	5	(a, b)
1	6	(b, c, d)
3	7	(b, c, d)

Decay rate d = 0.9

customer #1: $w_1 = 0.9^{(7-6)}$

customer #2: $w_2 = 0.9^{(7-4)}$

customer #3: $w_3 = 0.9^{(7-7)}$

Fig 4-16. An example of calculating the weights of customers

In IncSPAM, we do not need to calculate decay-period when a new transaction comes in. The weight of the customer that the incoming transaction belongs to is set to one. The others decay only by a decay-rate *d*. Figure 4-17 shows an example when a new transactions with TID = 8 comes. The weight of customer #2 is set to 1 and the others decay by 0.9.

CID	TID	Itemset
1	1	(a, b, d)
2	2	(b)
1	3	(b, c, d)
2	4	(a, b, c)
3	5	(a, b)
1	6	(b, c, d)
3	7	(b, c, d)
2	*8*	*(a, b, c, d)*

Decay rate d = 0.9

customer #1: $w_1 = 0.9 \times 0.9$

customer #2: $w_2 = 1$

customer #3: $w_3 = 1 \times 0.9$

Fig 4-17. When a new transaction with TID = 8 comes in

Now the support of a sequence ρ is not just the number of non-zero positions in the ρ-idx. The support of ρ is counted by the summation of the weights of the customer-sequences which ρ is in. We take the same example in section 4.7. The CBASWs and the lexicographic sequence tree in Figure 4-11 becomes Figure 4-18. We assume the decay-rate is 0.9. In Figure 4-18, we can find that the support of the tree node <(b)> is 1.9 not 2.

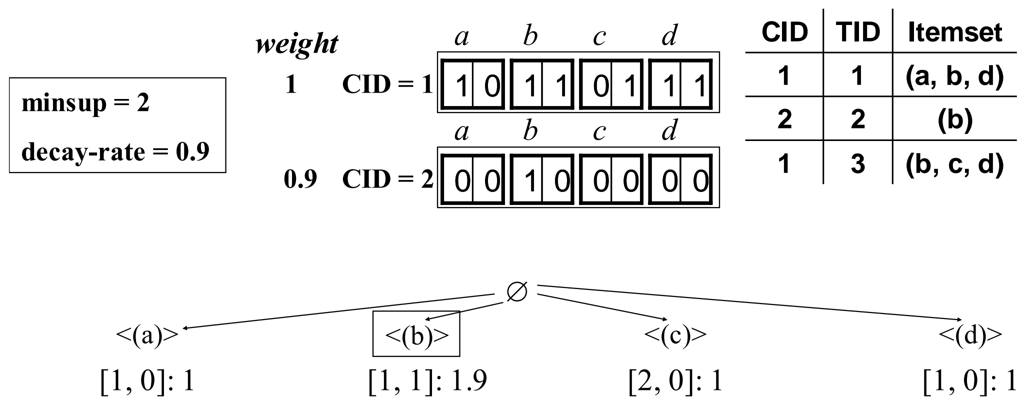


Fig 4-18. The lexicographic sequence tree when the third transaction comes in (with the concept of customer weight)

Updating support for an existing node can be easier than counting support of a new candidate. Whenever a new transaction comes in, only one customer-sequence is affected. Except the affected customer-sequence, the other customer-sequences' weights just decay by a decay-rate. We do not have to sum up all the weights one by one. The cases of updating support can be listed below:

- ***(Case 1) The incoming transaction belongs to a new customer*:** For a sequence ρ, the original support decays by a decay-rate. Then we check if ρ exists in the new customer-sequence. If ρ does exist, the decayed support increments one. If not, the decayed supports adds zero. Figure 4-19 shows an example.

Decay rate d = 0.9

CID	TID	Itemset
1	1	(a, b, d)

$w_1 = 1$

<(a)>-idx = [1]; <(a)>'s support = 1
<(b)>-idx = [1]; <(b)>'s support = 1
<(c)>-idx = [0]; <(c)>'s support = 0
<(d)>-idx = [1]; <(d)>'s support = 1

After transaction #2 comes

CID	TID	Itemset
1	1	(a, b, d)
2	2	(b)

$w_1 = 0.9$

$w_2 = 1$

<(a)>-idx = [1, 0]; <(a)>'s support = 1 × 0.9 + 0 = 0.9
<(b)>-idx = [1, 1]; <(b)>'s support = 1 × 0.9 + 1 = 1.9
<(c)>-idx = [0, 0]; <(c)>'s support = 0 × 0.9 + 0 = 0
<(d)>-idx = [1, 0]; <(d)>'s support = 1 × 0.9 + 0 = 0.9

Fig 4-19. An example of support updating in IncSPAM (Case 1)

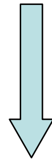
- ***(Case 2) The incoming transaction belongs to an existing customer*:** For a sequence ρ, the previous position value of this customer in the ρ-idx has to be checked. Assume the modified customer-sequence is *c*. If previous ρ-idx[c] is zero, the original support decays by a decay-rate and increments one or zero by the existence of ρ in *c*. If previous ρ-idx[c] is not zero, the original support subtracts the previous weight of customer c and then decays by a decay-rate. Finally the support increments one or zero by the same consideration. Figure 4-20 shows an example.

Decay rate d = 0.9

CID	TID	Itemset
1	1	(a, b, d)
2	2	(b)

<(a)>-idx = [1, 0]; <(a)>'s support = 1 × 0.9 + 0 = 0.9
<(b)>-idx = [1, 1]; <(b)>'s support = 1 × 0.9 + 1 = 1.9
<(c)>-idx = [0, 0]; <(c)>'s support = 0 × 0.9 + 0 = 0
<(d)>-idx = [1, 0]; <(d)>'s support = 1 × 0.9 + 0 = 0.9

$w_1 = 0.9$
$w_2 = 1$



After transaction #3 comes

CID	TID	Itemset
1	1	(a, b, d)
2	2	(b)
1	3	(b, c, d)

<(a)>-idx = [1, 0]; <(a)>'s support = (0.9 – 0.9) × 0.9 + 1 = 1
<(b)>-idx = [1, 1]; <(b)>'s support = (1.9 – 0.9) × 0.9 + 1 = 1.9
<(c)>-idx = [2, 0]; <(c)>'s support = 0 × 0.9 + 1 = 1
<(d)>-idx = [1, 0]; <(d)>'s support = (0.9 – 0.9) × 0.9 + 1 = 1

$w_1 = 1$
$w_2 = 0.9$

Fig 4-20. An example of support updating in Incremental SPAM (Case 2)

The weights of the customer-sequences do not change the entire process of maintaining a lexicographic sequence tree. Only when counting support in each tree node IncSPAM needs to consider the concept of weight. The Function *UpdateSupport* in Figure 4-8 and Figure 4-9 is changed to Figure 4-21.

```
UpdateSupport (c, n)
1:    if customer-sequence c is new then
2:        decay the support of n;
3:        if the sequence of n is in c then
4:            the support of n + 1;
5:        else
6:            the support of n + 0;
7:    else
8:        if ρ-idx[c] is 0 then      // assume the sequence in n is ρ
9:            decay the support of n;
10:           if the sequence of n is in c then
11:               the support of n + 1;
12:           else
13:               the support of n + 0;
14:       else
15:           the support of n – previous weight of c;
16:           if the sequence of n is in c then
17:               the support of n + 1;
18:           else
19:               the support of n + 0;
```

Fig 4-21. The pseudo code of function UpdateSupport

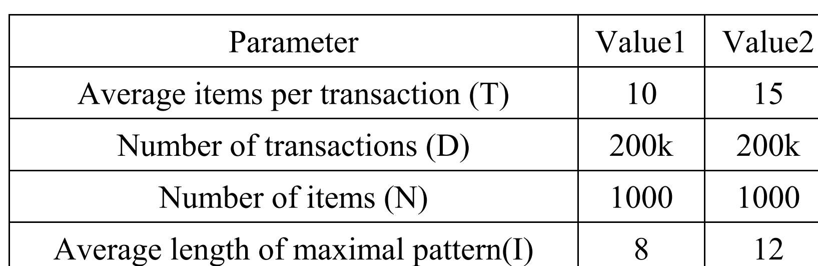
Chapter 5

Performance Measurement

5.1 Performance Measurement of New-Moment

We performed many performance measurements to compare New-Moment with Moment. Moment program (MomentFP) was provided by author. All experiments were done on a 1.3GHz Intel Celeron PC with 512MB memory and running with Windows XP system. New-Moment was implemented in C++ STL and compiled with Visual C++ .NET compiler.

All testing data was generated by the synthetic data generator provided by Agrawal et al in [1]. For testing the scalability of New-Moment and Moment, we use two set of different parameters, value1 and value2, to generate dataset. Parameters of testing data are listed in table 5-1. The dataset generated by value1 (T10I8D200K) contains general length of patterns and the dataset generated by value2 (T15I12D200K) contains longer patterns.



Parameter	Value1	Value2
Average items per transaction (T)	10	15
Number of transactions (D)	200k	200k
Number of items (N)	1000	1000
Average length of maximal pattern(I)	8	12

Table 5-1. Parameters of testing data for New-Moment

Our testing method is to execute New-Moment and Moment on the same dataset and to test their performance. The performance measurements include memory usage, loading time of the first window, and average time of window sliding. When the first window is filled by incoming transactions, both New-Moment and Moment build its initial lexicographic tree. The time of building the tree is called loading time of the first window. In the next step, both

New-Moment and Moment receive 100 continuous transactions and generate 100 consecutive sliding windows. Both New-Moment and Moment record executing time of each window. Average time of window sliding was reported over these 100 consecutive sliding windows.

We use different minimum support, different window size, and different number of item types to test these two algorithms.

5.1.1 Different Minimum Support

In the first experiment, we discuss the memory usage and executing time of New-Moment and Moment in different minimum support. Minimum support is changed from 1% to 0.1%. Sliding window size is fixed to 100,000 transactions. The number of item types is fixed to 1000. With different datasets (T10I8D200K and T15I12D200K), the results are listed below.

(1) T10I8D200K

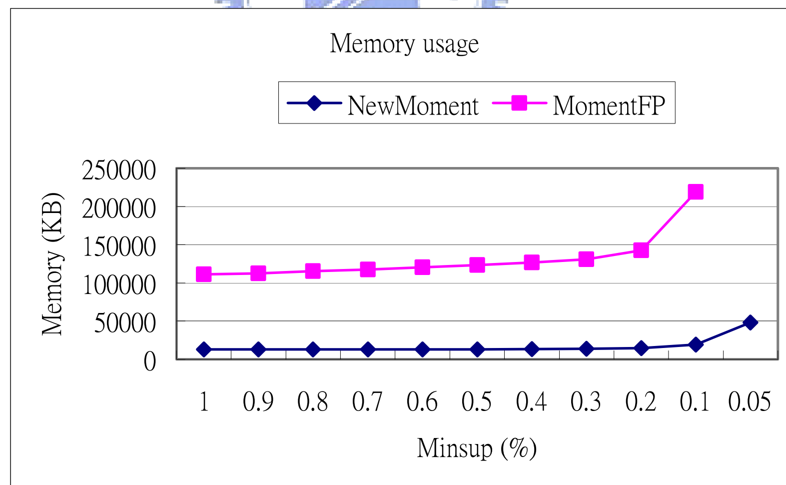


Fig 5-1. Memory usage with different minimum support (T10I8D200K) (New-Moment and Moment)

The first measurement is about memory usage of New-Moment and Moment. Figure 5-1 shows the memory usage in Kbytes. We can observe that memory used is more than 120MB in Moment but memory used in New-Moment is just about 15MB. When the minimum

support is down to 0.05%, the memory used by New-Moment is just 50MB but memory of Moment is out of bound (more than 512MB).

There are much less tree nodes in New-CET than in CET. New-Moment only maintains bit-vectors of 1-itemsets and closed frequent itemsets in the current window. Experiment shows that New-CET is more compact than CET.

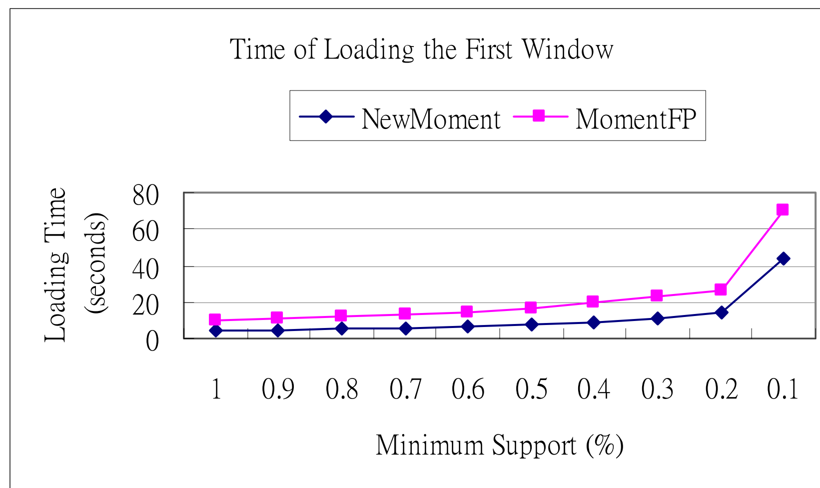


Fig 5-2. Loading time of the first window with different minimum support (T10I8D200K) (New-Moment and Moment)

The second measurement is about the loading time of the first window. Figure 5-2 shows the result. In the first window, both New-Moment and Moment need to build a lexicographic tree. We can observe that New-Moment is a little faster than Moment. The reason is that generating candidates and counting their supports with bit-vector is more efficient than with an independent sliding window (in MomentFP, a FP-tree [5] is used).

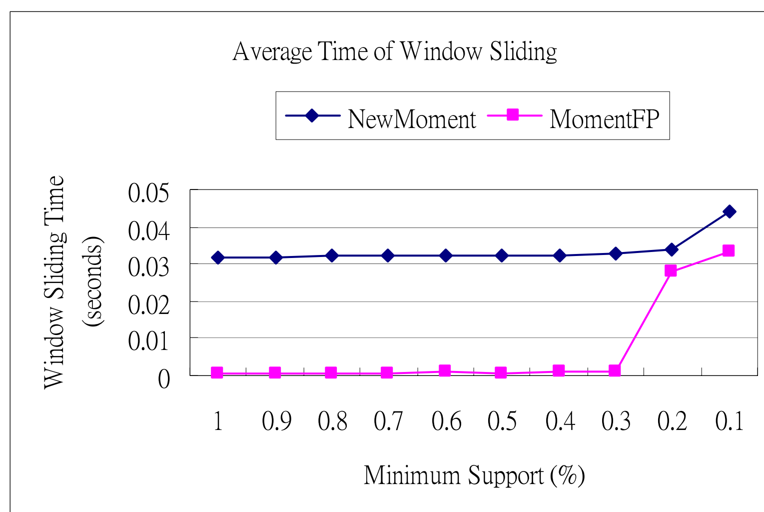


Fig 5-3. Average time of window sliding with different minimum support (T10I8D200K) (New-Moment and Moment)

The third measurement is about average time of window sliding. Figure 5-3 shows the result. In the experiment New-Moment is a little slower than Moment because New-Moment do not use *tid* sum as another key to speed up left-check step. But we can observe that the difference is little. The sliding steps can be finished in a second for both algorithms and the difference is meaningless.

(2) T15I12D200K

The patterns in this dataset are longer than the patterns in previous dataset. We also test the memory usage, loading time of the first window, and average time of window sliding. From the measurements listed below, we can observe that the scalability of New-Moment is better than Moment.

The first measurement is about memory usage in Kbytes. Figure 5-4 shows the result. We can observe that the memory used in New-Moment is still less than the memory used in Moment. By comparing the Figure 5-1 and Figure 5-4, we can also observe that the scalability of New-Moment in memory usage is better than Moment. In the dataset T10I8D200K,

memory used in Moment is about 120MB but in the dataset T15I12D200K, memory used in Moment is about 200MB. Memory used in New-Moment is under 100MB in both datasets.

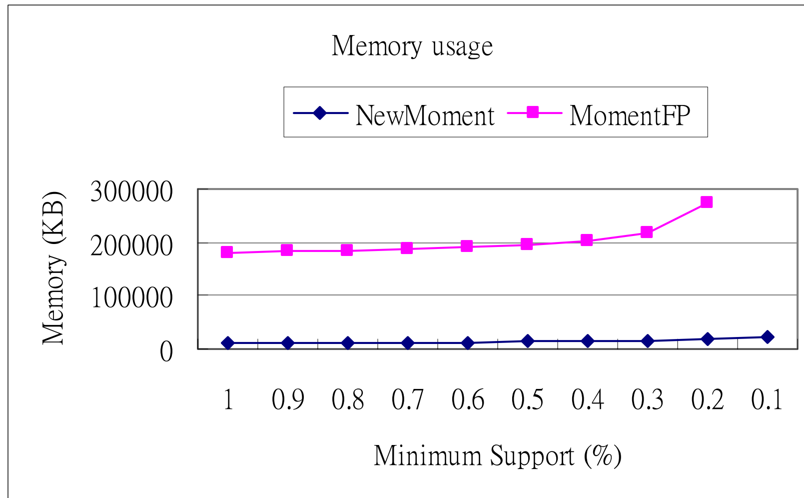


Fig 5-4. Memory usage with different minimum support (T15I12D200K) (New-Moment and Moment)

1896

The second measurement is about the loading time of the first window. Figure 5-5 shows the result. We can observe that New-Moment is still faster than Moment.

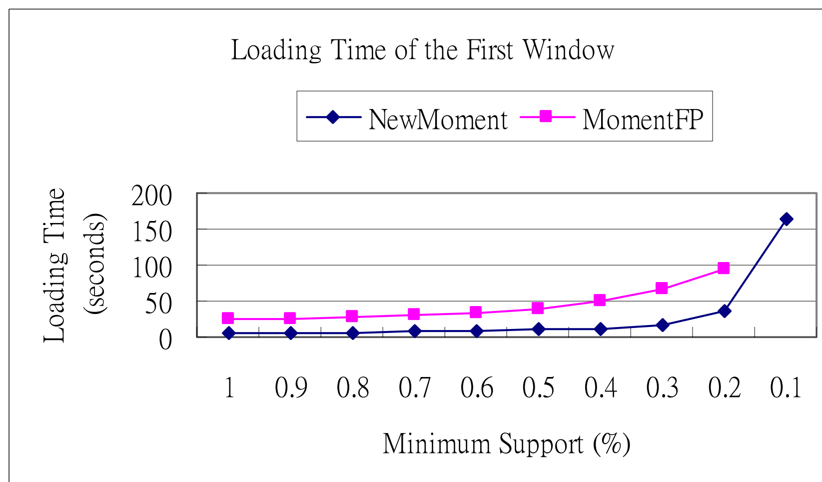


Fig 5-5. Loading time of the first window with different minimum support (T15I12D200K) (New-Moment and Moment)

The third measurement is about the average time of window sliding. Figure 5-6 shows the result. When minimum support is less than 0.3%, the average time of window sliding in New-Moment is less than Moment.

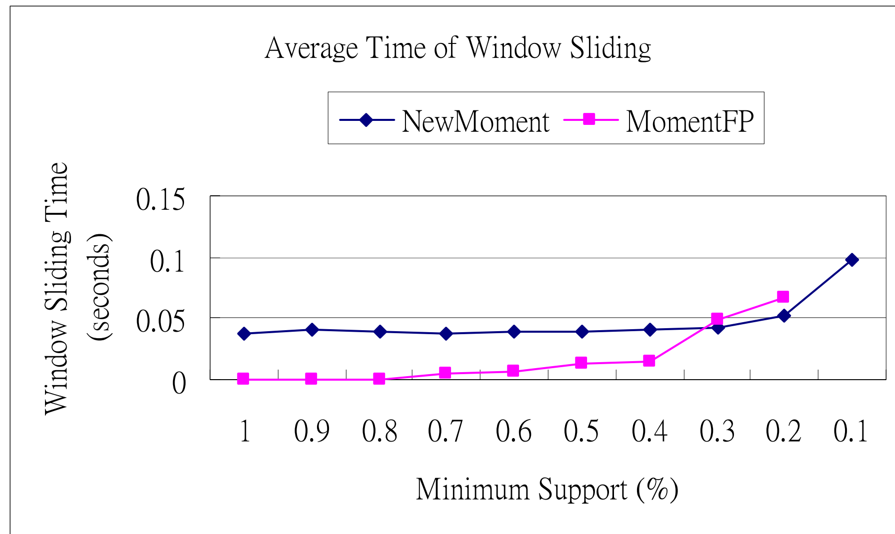


Fig 5-6. Average time of window sliding with different minimum support (T15I12D200K) (New-Moment and Moment)

By testing two different dataset with different minimum support, we can observe that New-Moment has better scalability than Moment. Although New-Moment is a little slower than Moment in window sliding, both algorithms handle a transaction in one second. In complicated dataset and low minimum support, New-Moment can even outperform Moment. New-Moment not only use less memory than Moment but also is as fast as Moment in loading the first window and window sliding.

5.1.2 Different Sliding Window Size

Sliding window size decides the length of each bit-vector. In this experiment, we want to compare New-Moment and Moment in different sliding window size. This experiment can show that using bit-vectors of items instead of independent sliding window is an efficient

strategy. In this measurement, sliding window size is changed from 10,000 transactions to 100,000 transactions. Minimum support is fixed to 0.1%. The dataset is T10I8D200K. The number of item types is fixed to 1000. We also test memory usage, loading time of the first window, and average time of window sliding.

Figure 5-7 shows the first measurement, memory usage, in Kbytes. We can observe that both New-Moment and Moment are linearly affected by sliding window size. New-Moment still outperforms Moment in memory usage. Furthermore, the memory used in Moment increases faster than New-Moment when window size becoming larger.

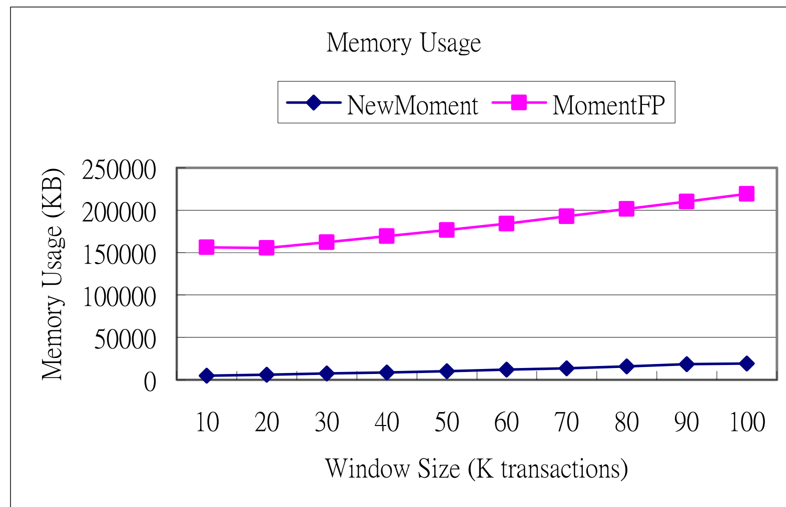


Fig 5-7. Memory usage with different sliding window size (New-Moment and Moment)

Figure 5-8 shows the result of the second measurement, time of loading the first window. Although with the increasing sliding window size each bit-vector becomes larger, New-Moment is still a little faster than Moment in loading time of the first window. The reason is that processing time of bitwise AND between bit-vectors is almost not affected by the length of bit-vector.

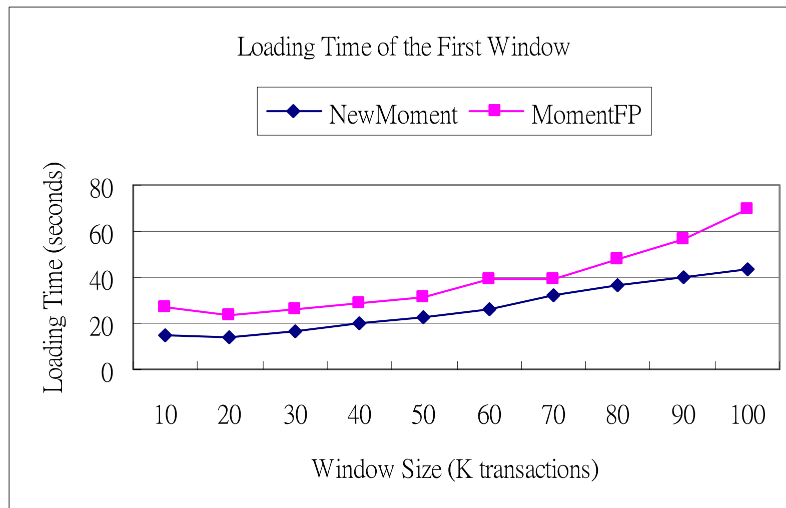


Fig 5-8. Loading time of the first window with different sliding window size (New-Moment and Moment)

Figure 5-9 shows the result of the third measurement, average time of window sliding. Window sliding time of New-Moment and Moment is almost the same. In the experiment of different window size, we can also conclude that New-Moment outperforms Moment in memory usage and retain the same executing time.

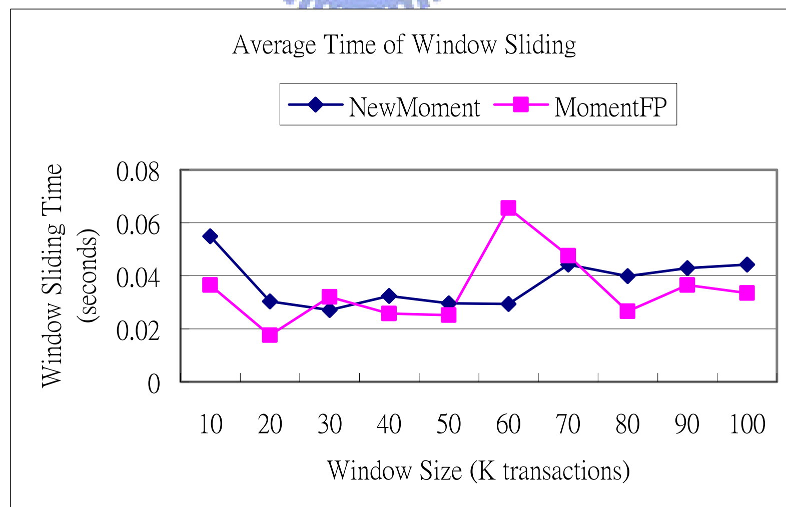


Fig 5-9. Average time of window sliding with different sliding window size (New-Moment and Moment)

5.1.3 Different Number of Items

New-Moment maintains bit-vectors of all items instead of independent sliding window structure. The more types of items, the more bit-vectors need to be maintained. The goal of this experiment is to show that even with a large number of items New-Moment also outperforms Moment in memory usage. The number of item types is ranged from 1000 to 10000. Minimum support is 0.1%. Sliding window size is 100000. Testing dataset is T10I8D200K. We also test memory usage, loading time of the first window, and average window sliding time.

Figure 5-10 shows the memory usage in Kbytes. Moment is out of memory (more than 512MB) when the number of items exceeds 3000. Memory usage of New-Moment and the number of items is linearly related. This result shows that New-Moment does not increase its memory usage suddenly when the number of items is large.

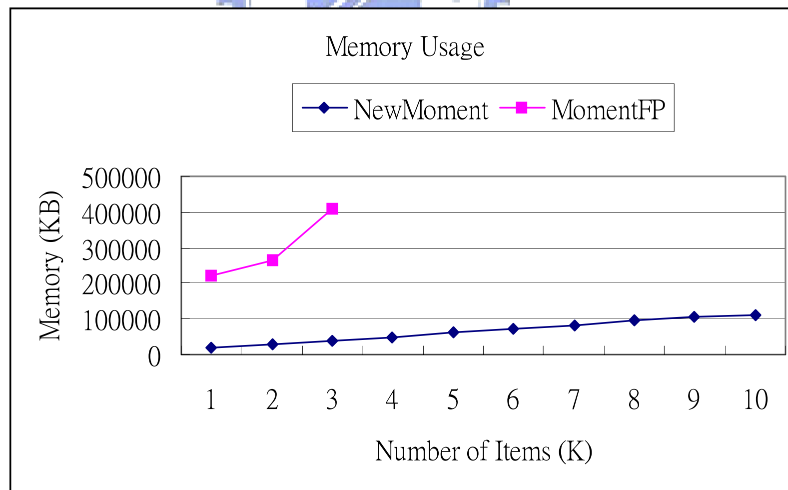


Fig 5-10. Memory usage with different number of items (New-Moment and Moment)

Next we test the executing time of both algorithms. Figure 5-11 shows the result of loading time of the first window. Figure 5-12 shows average time of window sliding. The results show that loading time and window sliding time also has linear relation with the number of items. Although loading time is more than 300 seconds when the number of items exceeds

9000, loading the first window is only executed once. Average time of window sliding is still less than 1 second. It means that New-Moment is still efficient with a large number of items.

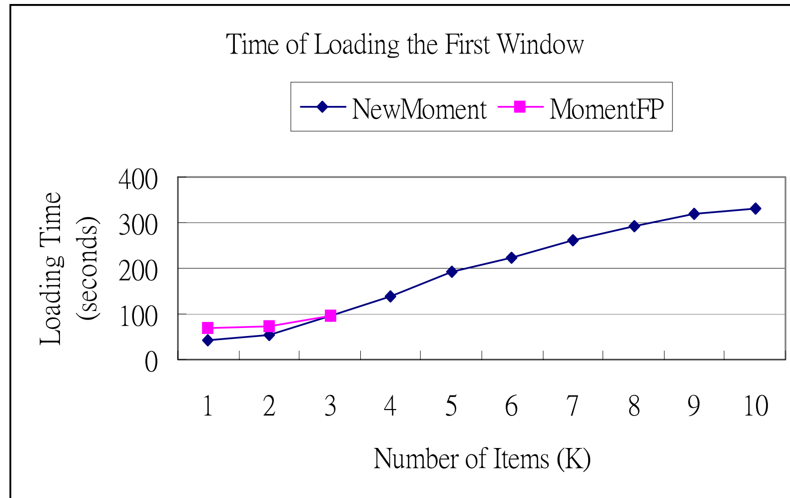


Fig 5-11. Loading time of the first window with different number of items (New-Moment and Moment)

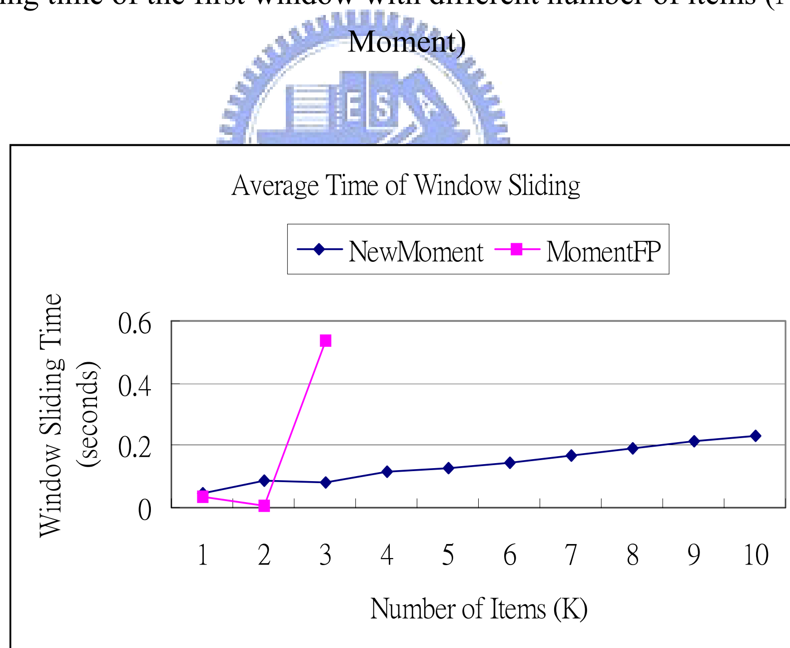


Fig 5-12. Average time of window sliding with different number of items (New-Moment and Moment)

5.2 Performance Measurement of IncSPAM

The sequence data set in transaction form is generated by IBM data generator [2]. Our program is written with C++ standard library (STL) and compiled with gcc 4.0.3 on Linux 9.0. The testing computer has 2.16GHz CPU power and 2GB main memory. Table 5-2 shows the parameters used to generate testing data.

Parameter	Value
Average Number of transactions per customer (C)	30
Average Number of items per transaction (T)	2~3
Number of Different Items (N)	1000

Table 5-2. Parameters of testing data for IncSPAM

The performance measurements include memory usage and average time of window sliding. Memory usage was tested by system tool to observe real memory variation. We run all transactions generated with parameters above and record time of handling one transaction. Average time of window sliding is over the entire dataset. All experiments are performed with decay-rate d = 0.999. For testing the scalability of IncSPAM, we test it with different minimum support, different window size of a customer-sequence, and different number of customers.

5.2.1 Different Minimum Support

The number of sequential patterns increases with lower minimum support. We want to test the memory usage and executing time of IncSPAM with different minimum support to see its scalability. In this experiment we use an absolute minimum support S. If the number of customers that support a sequence ρ is more than S, ρ is a sequential pattern. We changed S

from 3 to 10. The total number of customer is 1000. Window size of each customer is 10 transactions. Figure 5-13 shows the memory usage in Mbytes.

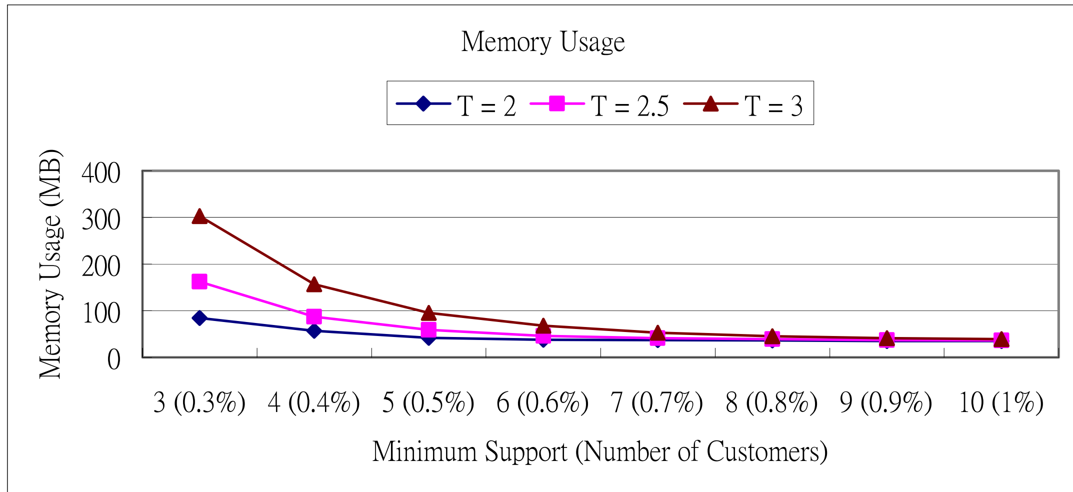


Fig 5-13. Memory usage with different minimum support (IncSPAM)

The memory usage is about 200 MB. That is reasonable in mining of sequential patterns. We can find that with lower minimum support the memory usage of IncSPAM increases rapidly. For proving the lexicographic sequence tree of IncSPAM does not generate redundant tree nodes, we test the number of tree nodes in the lexicographic sequence tree and the memory used by IncSPAM.The experiment can prove that IncSPAM is efficient in memory. Figure 5-14 shows the experiment for testing the relationship of maximum number of tree nodes and memory usage. From this graph we can observe that the relationship is linear. That means the memory usage grows up only because of increase of sequential patterns. IncSPAM does not produce additional structure when minimum support becomes small and is efficient in memory usage.

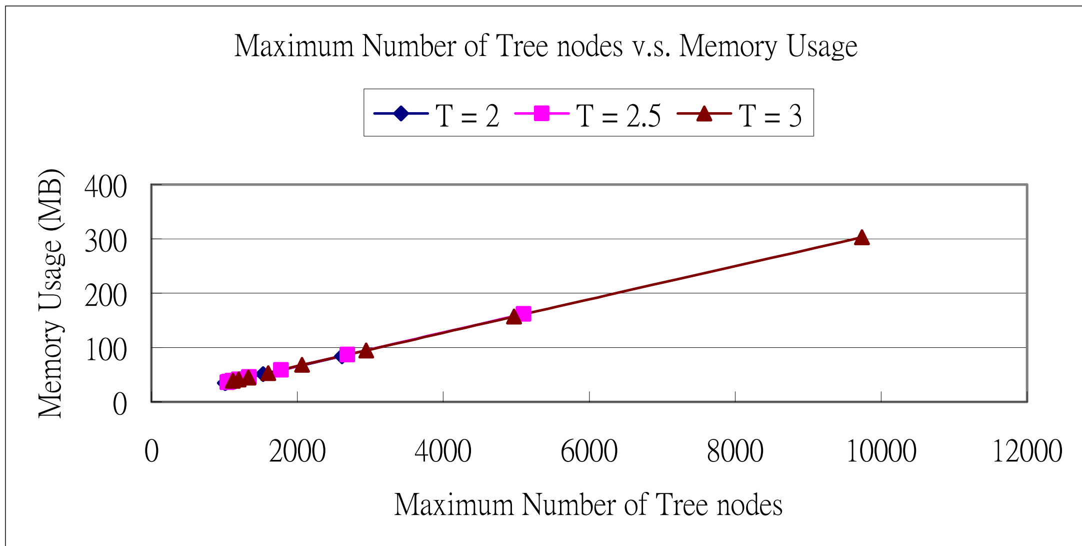
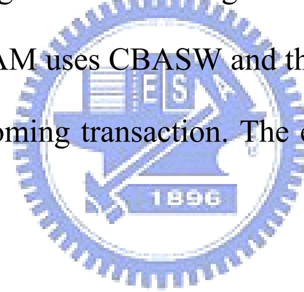


Fig 5-14. Relationship between maximum number of tree nodes and memory usage (IncSPAM)

Figure 5-15 shows the average window sliding time. The result shows that average sliding time is below 1 second. IncSPAM uses CBASW and the characteristics of incremental mining to speed up processing an incoming transaction. The experiment can prove the efficiency of IncSPAM.


1896

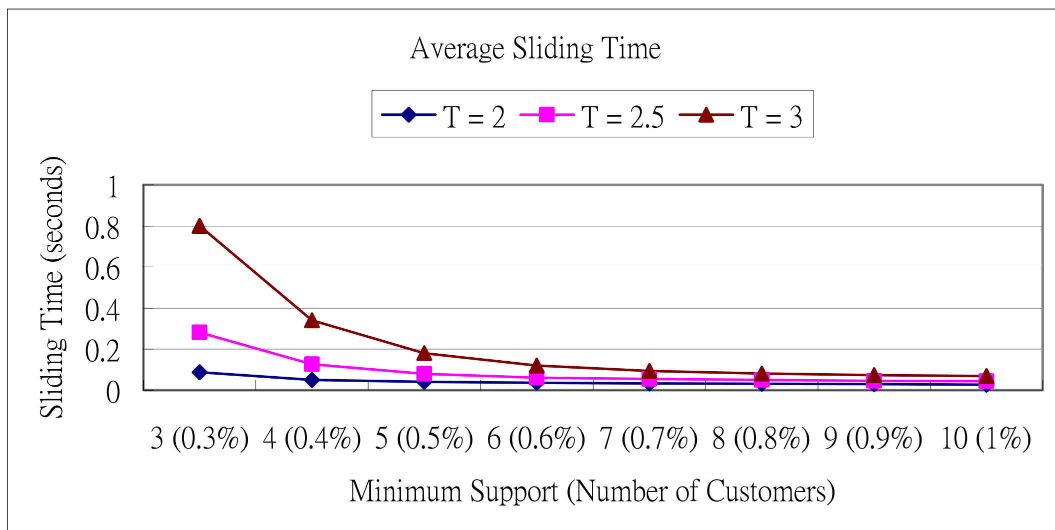


Fig 5-15. Average time of window sliding with different minimum support (IncSPAM)

5.2.2 Different Sliding Window Size

Size of sliding window is to control the number of transactions maintained by each customer. In this experiment, we test the memory usage and average sliding time by different size of sliding window. The size ranges from 10 transactions to 25 transactions. Minimum support is fixed to 10 customer-sequences. Figure 5-16 shows memory usage and Figure 5-17 shows average sliding time with different window size.

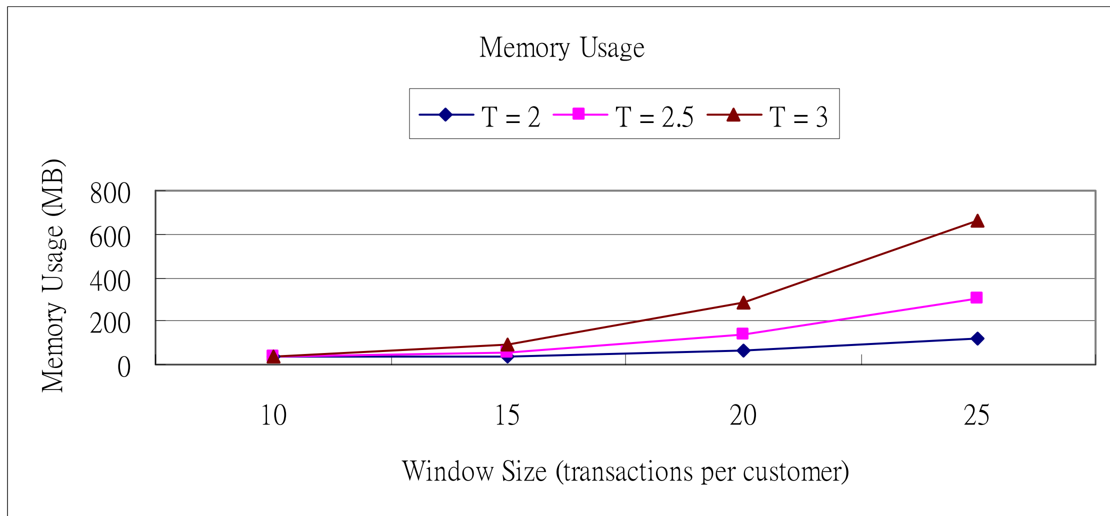


Fig 5-16. Memory usage with different window size (IncSPAM)

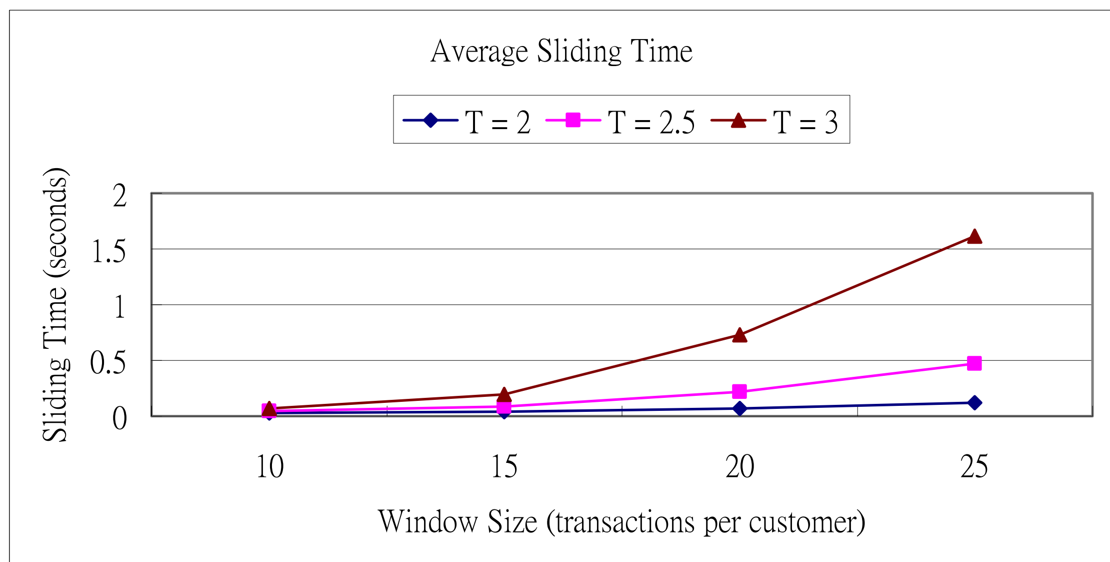


Fig 5-17. Average sliding time with different window size (IncSPAM)

When the number of transactions maintained by each customer increases, the corresponding memory usage and average sliding time also grows up. In current applications and IBM synthetic dataset, maintaining about 15 transactions for each customer is reasonable. IncSPAM can be applied in general applications and is efficient in memory usage and handing real-time transactions.

5.2.3 Different Number of Customers

IncSPAM can dynamically add a new customer to the summary data structure. In previous experiments we fix the number of customers for observing performance conveniently. The memory usage and average sliding time for different number of customers is tested in this experiment. Minimum support is also 10 customer-sequences. Figure 5-18 shows memory usage of IncSPAM in different number of customers.

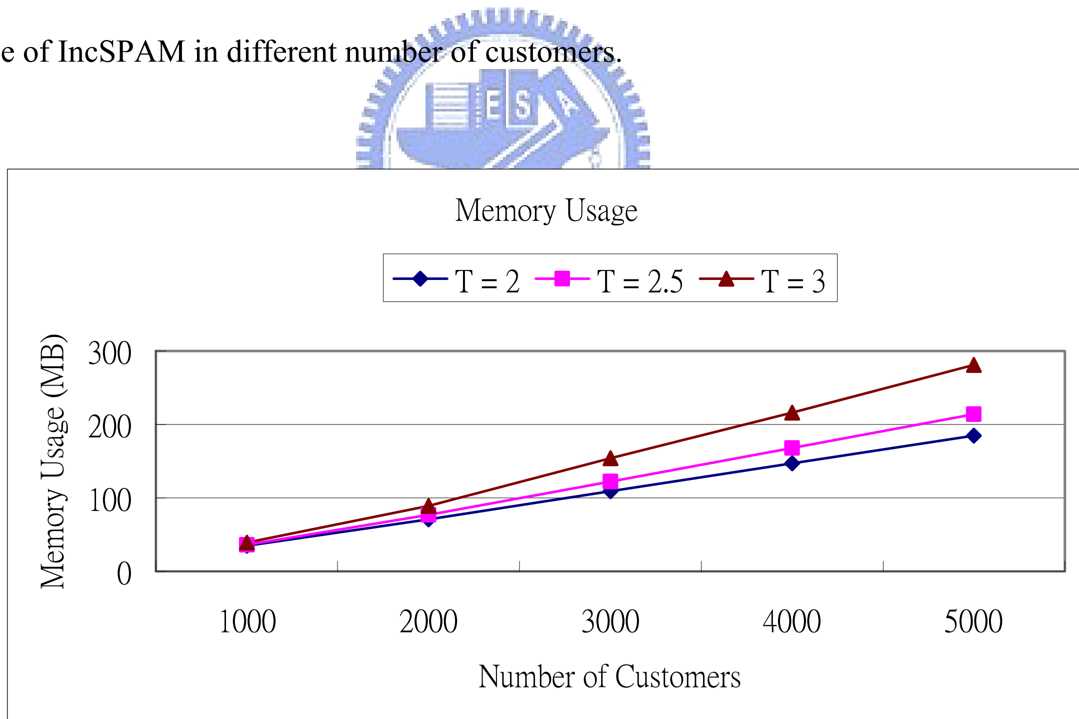


Fig 5-18. Memory usage with different number of customers (IncSPAM)

The relationship between memory usage and the number of customers is linear. IncSPAM can efficiently handle a great amount of customers with reasonable memory. In Figure 5-19 we can observe that the average sliding time is also in linear relationship.

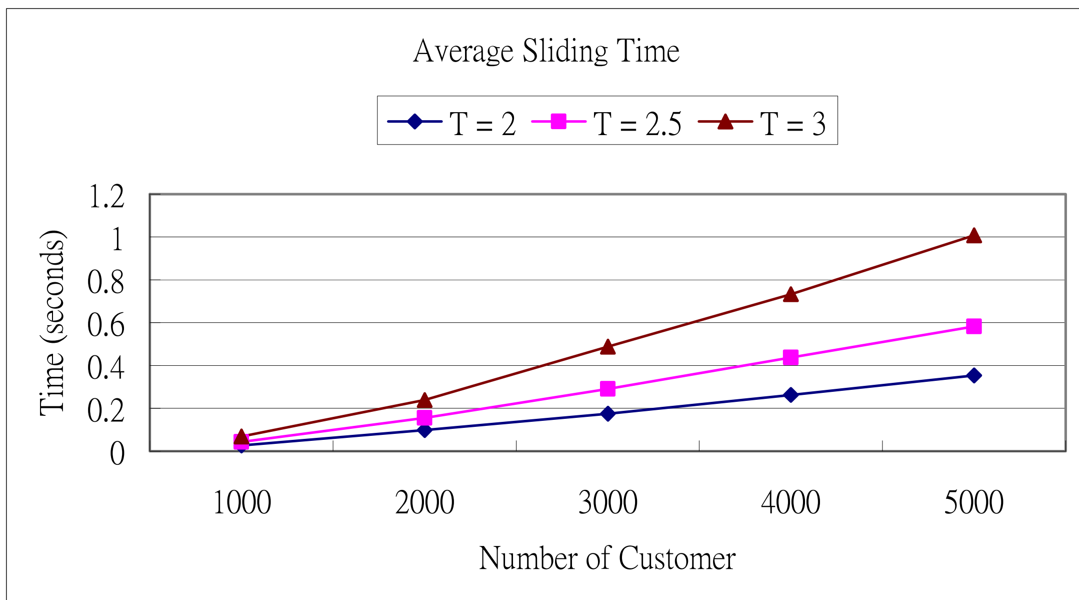


Fig 5-19. Average sliding time with different number of customers (IncSPAM)

Chapter 6

Conclusion and Future Work

Mining of frequent patterns in a data stream is more complicated than in a static database. In this paper we propose two algorithms: New-Moment to mine closed frequent itemsets and IncSPAM to mine sequential patterns with the sliding window model in the data stream environment.

The first algorithm, New-Moment, is to improve the efficiency of Moment algorithm. New-Moment utilizes bit-vectors and a smaller lexicographic tree New-CET to reduce the memory usage. Employing the characteristics of bit-vectors New-Moment is also as efficient as Moment in executing time. The second algorithm, IncSPAM, utilizes the concept of the sliding window in each customer-sequence. IncSPAM uses CBASWs and a lexicographic sequence tree to maintain the sequential patterns in the current window. In the lexicographic sequence tree IncSPAM uses an index set in each tree node to speed up counting support. IncSPAM can handle a transaction from the data stream in one second.

6.1 Conclusion of New-Moment

New-Moment reduces the memory usage by only maintaining bit-vectors of 1-itemsets and closed frequent itemsets in New-CET. In the test of different minimum support, New-Moment outperforms Moment in memory usage about 100MB. When the minimum support becomes lower, the difference of memory usage in New-Moment and Moment becomes more significant. Due to the efficiency of bit-vector in window sliding and in the generation of itemset candidates, New-Moment is faster than Moment in the loading time of the first window. Although New-Moment does not maintain the boundary tree node in the

lexicographic tree, New-Moment still has almost the same performance as Moment in execution time of window sliding. In the test of different window size, New-Moment still outperforms Moment in memory usage and running time. In the test of different number of items, Moment is even running out of memory bound. New-Moment is still efficient in memory usage and executing time.

6.2 Conclusion of IncSPAM

We test memory usage and execution time of handing a transaction in IncSPAM in different minimum support, different window size, and different number of customers. In the test of different minimum support, memory used in IncSPAM is about 300MB. The memory usage of IncSPAM increases when the minimum support becomes lower. We prove that the lexicographic sequence tree of IncSPAM does not produce redundant tree nodes. The handling time of a transaction is below 1 second. IncSPAM can be applied in the data stream environment. In the test of different window size, the memory usage and execution time of handling a transaction of IncSPAM increases when window size becomes large. We can observe that IncSPAM is still efficient when the window size is from 10 to 25. In the test of different number of customers, memory usage and time of handling a transaction of IncSPAM is linear related to the number of customers. That means IncSPAM can perform well even the number of customers becomes large.

6.3 Future Work

The concept of sliding window in this paper is based on transactions as units. In some applications the unit of a window may be a time point. The number of transactions in each time point is variable. Figure 6-1 shows the sliding window model in time units.

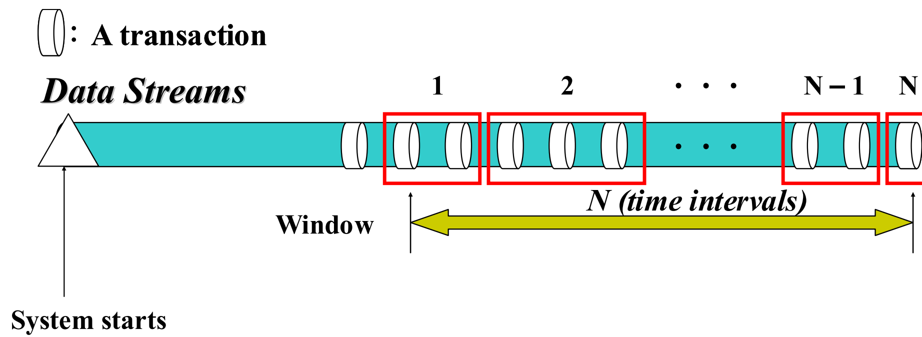
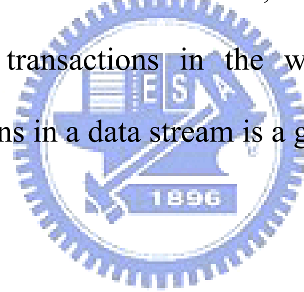


Fig 6-1. The sliding window model in time units

In this sliding window model, the system keeps the transactions in the latest N time intervals. The time interval may be one day, one week, or one month. There is different number of transactions in each time interval. In Figure 6-1, there are two transactions in the first time interval but there are three transactions in the second window. Since the number of transactions in each time interval becomes variable, using bit-vectors that store fixed number of transactions to store the transactions in the window is difficult. Mining of these complicated and flexible patterns in a data stream is a great challenge.



Bibliography

[1] R. Agrawal and R. Srikant, Fast algorithms for mining association rules, In Proc. of the 20th Intl. Conf. on Very Large Databases (VLDB) 1994.

[2] R. Agrawal and R. Srikant, Mining Sequential Patterns, In Proc. of the 7th International Conf. on Data Engineering (ICDE) 1995.

[3] M. Yen and S. Lee, Incremental Update on Sequential Patterns in Large Databases, In Proc. of the 10th IEEE Int'l Conf. on Tools with Artificial Intelligence (ICTAI98), 1998.

[4] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal, Discovering Frequent Closed Itemsets for Association Rules, Proc. Seventh Int'l Conf. Database Theory (ICDT) 1999.

[5] J. Han, J. Pei, and Y. Yin, Mining frequent patterns without candidate generation, In 2000 ACM SIGMOD Int'l. Conf. on Management of Data, 2000.

[6] J. Pei, J. Han, B Mortazavi-Asl, and H. Pinto, PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth, In Proc. of the 17th Int'l Conf. on Data Engineering (ICDE) 2001.

[7] A. Veloso, W. Meira Jr., M de Carvalho, B. Pôssas, S. Parthasarathy, and M. J. Zaki, Mining frequent itemsets in evolving databases, In SDM 2002.

[8] M. J. Zaki and C. Hsiao, Charm: An efficient algorithm for closed itemset mining, In 2nd SIAM Int'l Conf. on Data Mining 2002.

[9] Y. Zhu and D. Shasha, StatStream: Statistical Monitoring of Thousands of Data Stream in Real Time, In Proceedings of the 28th International Conference on Very Large Data Bases, pp. 358-369, 2002.

[10] G. Manku and R. Motwani, Approximate Frequency Counts over Data Streams, In Proceedings of the 28th International Conference on Very Large Data Bases, 2002.

[11] J. Ayres, J. Gehrke, T. Yiu, and J. Flannick, Sequential Pattern Mining using A Bitmap Representation, Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining, 2002.

[12] M. Lin and S. Lee, Fast Discovery of Sequential Patterns by Memory Indexing, In Proc. of the 4th Int'l Conf. on Data Warehousing and Knowledge Discovery (DaWak), 2002.

[13] E. Elnahrawy, Research Directions in Sensor Data Streams: Solutions and Challenges, DCIS Technical Report DCIS-TR-527, Rutgers University, 2003.

[14] L. Golab and M. T. Özsu, Issues in Data Stream Management, ACM SIGMOD Record, 32(2): 5-14, Jun. 2003.

[15] C. Jin, W. Qian, C. Sha, J. Yu, and A. Zhou, Dynamically Maintaining Frequent Items Over a Data Stream, In Proceedings of the 12th ACM CIKM International Conference on Information and Knowledge Management, pages 287--294, 2003.

[16] J. H. Chang and W. S. Lee, Finding Recent Frequent Itemsets Adaptively over Online Data Streams, In Proc. of the 2003 Int. Conf. Knowledge Discovery and Data Mining (SIGKDD'03), 2003.

[17] X. Yan, J. Han, and R. Afshar, CloSpan: Mining Closed Sequential Patterns in Large Datasets, In Proc. of 2003 SIAM Int'l Conf. on Data Mining (SDM03), 2003.

[18] W. Teng, M. Chen, and P. S. Yu, A Regression-Based Temporal Pattern Mining Scheme for Data Streams, Proceedings of the 29th International Conference on Very Large Data Bases , VLDB 2003.

[19] H. F. Li, S. Y. Lee and M. K. Shan, DSM-FI: An Efficient Algorithm for Mining Frequent Itemsets over the Entire History of Data Streams, In 1st International Workshop on Knowledge Discovery in Data Streams, Pisa, Italy, 2004.

[20] Y. Chi, H. Wang, P. S. Yu, and R. R. Muntz, Moment: Maintaining Closed Frequent Itemsets over a Stream Sliding Window, ICDM 2004.

[21] Y. Chi, H. Wang, P. S. Yu, and R. R. Muntz, Catch the Moment: Maintaining Closed Frequent Itemsets over a Data Stream Sliding Window, Technical Report, IBM, 2004.

[22] J. Chang and W. Lee, A Sliding Window Method for Finding Recently Frequent Itemsets over Online Data Streams, Journal of Information Science and Engineering, Vol. 20, No. 4, July, 2004.

[23] A. Chen and H. Ye, Multiple-Level Sequential Pattern Discovery from Customer Transaction Databases, International Journal of Computational Intelligence, 2004.

[24] H. Cheng, X. Yan, and J. Han, IncSpan: Incremental Mining of Sequential Patterns in Large Database, In Proc. of the 10th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining, 2004.

[25] G. Chen, X. Wu, and X. Zhu, Mining sequential patterns across data streams, Technical Report, 2004.

[26] J. H. Chang and W. S. Lee, Decaying Obsolete Information in Finding Recent Frequent Itemsets over Data Stream, IEICE Transactions on Information and System, 2004.

[27] H. F. Li, S. Y. Lee, M. K. Shan, Online Mining (Recently) Maximal Frequent Itemsets over Data Streams, In proc. Of the 15th IEEE International Workshop on Research Issues on Data Engineering (RIDE 2005), April 3-4, 2005.

[28] A. Marascu and F. Masseglia, Mining Sequential Patterns from Temporal Streaming Data, In Proc. of the 1st ECML/PKDD Workshop on Mining Complex Data 2005 (IEEE MCD05), held in conjunction with the 5th IEEE Int'l Conf. on Data Mining (ICDM05), 2005.