

國立交通大學

資訊科學與工程研究所

博士論文

在串流資料中高效率頻繁樣式探勘演算法之研究

A Study of Efficient Mining Algorithms of Frequent
Patterns on Data Streams



研究生：李華富

指導教授：李素瑛 博士

中華民國 九十五年 六月

在串流資料中高效率頻繁樣式探勘演算法之研究
A Study of Efficient Mining Algorithms of Frequent
Patterns on Data Streams

研究生：李華富

Student : Hua-Fu Li

指導教授：李素瑛博士

Advisor : Dr. Suh-Yin Lee

國立交通大學
資訊科學與工程研究所



Submitted to Department of Computer Science

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

June 2006

Hsinchu, Taiwan, Republic of China

中華民國九十五年六月

國立交通大學

博碩士論文全文電子檔著作權授權書

(提供授權人裝訂於紙本論文書名頁之次頁用)

本授權書所授權之學位論文，為本人於國立交通大學資訊科學與工程研究所 _____ 組，94 學年度第 二 學期取得博士學位之論文。

論文題目：在串流資料中高效率頻繁樣式探勘演算法之研究
指導教授：李素瑛

■ 同意

本人茲將本著作，以非專屬、無償授權國立交通大學與台灣聯合大學系統圖書館：基於推動讀者間「資源共享、互惠合作」之理念，與回饋社會與學術研究之目的，國立交通大學及台灣聯合大學系統圖書館得不限地域、時間與次數，以紙本、光碟或數位化等各種方法收錄、重製與利用；於著作權法合理使用範圍內，讀者得進行線上檢索、閱覽、下載或列印。

論文全文上載網路公開之範圍及時間：

本校及台灣聯合大學系統區域網路	■ 中華民國 96 年 7 月 3 日公開
校外網際網路	■ 中華民國 96 年 7 月 3 日公開

■ 全文電子檔送交國家圖書館

授權人：李華富

親筆簽名： 李華富

中華民國 95 年 7 月 3 日

國立交通大學

博碩士紙本論文著作權授權書

(提供授權人裝訂於全文電子檔授權書之次頁用)

本授權書所授權之學位論文，為本人於國立交通大學資訊科學與工程研究所 _____ 組，94 學年度第 二 學期取得博士學位之論文。

論文題目：在串流資料中高效率頻繁樣式探勘演算法之研究
指導教授：李素瑛

■ 同意

本人茲將本著作，以非專屬、無償授權國立交通大學，基於推動讀者間「資源共享、互惠合作」之理念，與回饋社會與學術研究之目的，國立交通大學圖書館得以紙本收錄、重製與利用；於著作權法合理使用範圍內，讀者得進行閱覽或列印。

本論文為本人向經濟部智慧局申請專利(未申請者本條款請不予理會)的附件之一，申請文號為：_____，請將論文延至____年____月____日再公開。

授權人：李華富

親筆簽名： 李華富

中華民國 95 年 7 月 3 日

國家圖書館 博碩士論文電子檔案上網授權書

(提供授權人裝訂於紙本論文本校授權書之後)

ID:GT009017811

本授權書所授權之論文為授權人在國立交通大學資訊科學與工程研究所 94 學年度第三學期取得博士學位之論文。

論文題目：在串流資料中高效率頻繁樣式探勘演算法之研究
指導教授：李素瑛

茲同意將授權人擁有著作權之上列論文全文（含摘要），非專屬、無償授權國家圖書館，不限地域、時間與次數，以微縮、光碟或其他各種數位化方式將上列論文重製，並得將數位化之上列論文及論文電子檔以上載網路方式，提供讀者基於個人非營利性質之線上檢索、閱覽、下載或列印。

※ 讀者基於非營利性質之線上檢索、閱覽、下載或列印上列論文，應依著作權法相關規定辦理。

授權人：李華富

親筆簽名：李華富

民國 95 年 7 月 3 日

國立交通大學
資訊工程系博士班

論文口試委員會審定書

本校 資 訊 工 程 系 李華富 君

所提論文 在串流資料中高效率頻繁樣式探勘演算法之研究

合於博士資格水準、業經本委員會評審認可。

口試委員： _____ 李素瑛

_____ 曾新修 _____ 陳銘慧

_____ 孫春在 _____ 張嘉惠

_____ 沈銘坤 _____ 劉文志

指導教授： _____ 李素瑛

系主任： _____ 曾煥彬

中華民國九十五年六月十五日

Department of Computer Science
College of Computer Science
National Chiao Tung University
Hsinchu, Taiwan, R.O.C.

Date: June 15, 2006

We have carefully read the dissertation entitled A Study of Efficient Mining Algorithms of Frequent Patterns on Data Streams submitted by Hua-Fu Li in partial fulfillment of the requirements of the degree of Doctor of Philosophy and recommend its acceptance.

_____	<u>Suh-Yin Lee</u>
<u>Si-Tung</u>	<u>Mhi Chen</u>
<u>Chien-Tsun</u>	<u>Chia-Hui Chang</u>
<u>Mantwan Shan</u>	<u>Wen-chih Peng</u>

Thesis Advisor: Suh-Yin Lee

Chairman: STPH

在串流資料中高效率頻繁樣式探勘演算法之研究

學生：李華富

指導教授：李素瑛博士

國立交通大學 資訊科學與工程研究所

摘要

資料串流探勘是一個快速成長的新興研究領域，但同時也帶來了新的挑戰。在眾多資料串流探勘的研究中，頻繁樣式探勘與變化探勘一直是資料串流探勘中重要的研究焦點。本論文主旨在於研發高效率的頻繁項目集合、路徑瀏覽樣式，以及項目變化樣式的單次線上掃描探勘方法。

頻繁項目集合是本論文所探討的第一個研究主題。首先，針對串流資料的標的物模型，我們提出一個可快速地探勘出所有頻繁項目集合的單次掃描演算法 DSM-FI。為了避免將每一筆新進交易中所有項目集組合都窮舉出來，我們設計了一個有效的交易投影機制，並提出一個新的摘要字首樹狀結構來儲存必要的集合項目資訊。此演算法在探勘出頻繁項目集合的同時，也可找出最大頻繁項目集合。

我們也針對串流資料的滑動窗模型提出了兩個單次頻繁項目集合探勘演算法 MFI-TransSW 與 MFI-TimeSW，可有效地在交易感知滑動窗模型以及時間感知滑動窗模型中探勘出目前存在的頻繁項目集合。MFI-TransSW 與 MFI-TimeSW 演算法主要是利用位元向量的特性來儲存單一項目在目前滑動窗中的出現位置，並利用位元向量的特性來達到快速滑動的效果。

路徑瀏覽樣式是本論文所探討的第二個主題。我們提出可快速探勘出路徑瀏覽樣式的單次掃描演算法 DSM-PLW，此演算法沿用 DSM-FI 的精神，利用快速拆解使用者瀏覽路徑，以及字首樹狀結構的特性，來達到串流資料探勘的效能要求。此外，我們進一步提出一個不需要使用者輸入最小支持度門檻值，就可以進行的 Top-K 路徑瀏覽樣式的單次掃描探勘演算法。

串流資料變化探勘是本論文的第三個研究主題。我們提出 MFC-append 演算法來找出在兩條線上交易資料串流中，穩定分布的交易項目、常常變動的項目，或無一定分佈的變化樣式。此外，針對可執行刪除運算的動態資料串流，我們提出一個以 MFC-append 為基礎的演算法 MFC-dynamic，來探勘動態資料串流中的項目變化。

我們進行了相關的實驗以評估所提方法的效能。在我們的實驗範圍中的結果顯示，對於各個不同探勘參數以及不同特性的資料集，我們的方法都優於許多著名的方法。此外，針對資料量擴充的實驗也顯示出我們所提出的探勘頻繁樣式的方法具有線性的擴充能力。

A Study of Efficient Mining Algorithms of Frequent Patterns on Data Streams

Student : Hua-Fu Li

Advisor : Dr. Suh-Yin Lee

Department of Computer Science
National Chiao Tung University

Abstract

Online mining of data streams is an important data mining problem with broad applications. However, it is a difficult problem since the streaming data possess some specific characteristics, such as unknown or unbounded length, possibly very fast arrival rate, inability to backtrack over previously arrived transactions, and a lack of system control over the order in which data arrives. Among various objectives of data stream mining, the mining of frequent patterns in data streams has been the focus of knowledge discovery. In this dissertation, the design of several core technologies for mining frequent patterns and changes of data streams is investigated.

For mining of frequent itemsets over data streams with a landmark window, we propose the DSM-FI (Data Stream Mining for Frequent Itemsets) algorithm to find the set of all frequent itemsets over the entire history of the data streams. An effective projection method is used in the proposed algorithm to extract the essential information from each incoming transaction of the data streams. A data structure based on prefix tree is constructed to store data summary. DSM-FI utilizes a top-down pattern selection approach to find the complete set of frequent itemsets. Experiments show that DSM-FI outperforms BTS (Buffer-Trie-SetGen), a state-of-the-art

single-pass algorithm, by one order of magnitude for discovering the set of all frequent itemsets over a landmark window of data streams.

For mining of frequent itemsets in data streams with a sliding window, efficient bit vector based algorithms are proposed. Two kinds of sliding windows, i.e., transaction-sensitive sliding window and time-sensitive sliding window, are discussed. MFI-TransSW (Mining Frequent Itemsets over a Transaction-sensitive Sliding Window) is developed to mine the set frequent itemsets over data streams with a transaction-sensitive sliding window. A single-pass algorithm, called MFI-TimeSW (Mining Frequent Itemsets over a Time-sensitive Sliding Window), based on MFI-TransSW algorithm and a dynamic encoding method is proposed to mine the set of frequent itemsets in a time-sensitive sliding window. An effective bit-sequence representation of items is used in the proposed algorithms to reduce the time and memory needed to slide the windows. Experiments show that the proposed algorithms not only attain highly accurate mining results, but also run significantly faster and consume less memory than existing algorithms for mining recent frequent itemsets over data streams.

For mining changes of items across two data streams, we propose two one-pass algorithms, called MFC-append (Mining Frequency Changes of append-only data streams) and MFC-dynamic (Mining Frequency Changes of dynamic data streams), to mine the set of frequent frequency changed items, vibrated frequency changed items, and stable frequency changed items across two continuous append-only and dynamic data streams, respectively. A new summary data structure, called Change-Sketch, is developed to compute the frequency changes between two data streams as fast as possible. Theoretical analysis and experimental results show that our algorithms meet the major performance requirements, namely single-pass, bounded space requirement, and real-time computing, in mining data streams.

Mining path traversal patterns from Web click streams is important in Web usage mining and Web user profiling. One of the most important We proposed two single-pass algorithms, called DSM-PLW (Data Stream Mining for Path traversal patterns in a Landmark Window) and DSM-TKP (Data Stream Mining for Top-K Path traversal patterns), to discover the path traversal patterns over Web click-streams with and without a user-defined minimum support constraint. Experiments of real data show that both algorithms successfully mine maximal reference sequences with linear scalability.

Comprehensive experiments have been conducted to assess the performance of the proposed algorithms. The empirical results show that these algorithms outperform the state-of-the-art algorithms with respect to various mining parameters and datasets of different characteristics. The scale-up experiments also verify that our algorithms successfully mine frequent patterns with good linear scalability.



Acknowledgement

(誌謝)

首先，我最感謝的是我的指導教授李素瑛老師，感謝她在這五年的研究所生涯中啟發我對學術研究的興趣，並且指導我論文的寫作上的技巧，以及生活上的幫助。在跟隨李老師研究的過程中，李老師對研究的執著及熱忱，一直是我學習的榜樣。此外，還要感謝孫春在教授和彭文志教授在計劃書口試、校內口試以及校外口試時提供許多寶貴的意見。

感謝口試委員：台大電機系陳銘憲教授、成大資工曾新穆教授、中央資工系張嘉惠教授、政大資科系沈錕坤教授在口試過程中提供許多寶貴的建議，讓我的論文能更趨完善。

諸位口試委員都是我在學術研究的道路上的最佳學習典範。

資訊系統實驗室的學長與學弟妹是我博士班研究生涯的好伙伴，謝謝大家也祝福學弟妹們早日收穫豐富的研究成果。再次謝謝你們在這段過程中對我的幫忙及鼓勵。

一直陪伴在我身旁、沒有怨言、只給我鼓勵的，就是我的太太佑青。能夠順利完成博士學位，對於佑青，我有無盡的感謝。

要感謝的人太多，僅在此對所有曾經幫助過我的朋友，致上我真切的謝意。

僅以此論文，獻給我摯愛的太太佑青。

Table of Contents

Abstract in Chinses	i
Abstract in English	iii
Acknowledgement	vi
List of Figures	ix
Chapter 1 Introduction	1
1.1 Background.....	1
1.2 Research Objectives and Contributions.....	3
1.3 Organization of this Thesis.....	4
Chapter 2 Online Mining of Frequent Itemsets in Data Streams	6
2.1 Introduction.....	7
2.2 Problem Definition.....	11
2.3 The Proposed Algorithm: DSM-FI.....	13
2.3.1 Construction and Maintenance of Summary Data structure.....	13
2.3.2 Pruning Infrequent Information from SFI-forest.....	19
2.3.3 Determining Frequent Itemsets from Current SFI-forest.....	22
2.4 Theoretical Analysis.....	25
2.4.1 Maximum Estimated Support Error Analysis.....	25
2.4.2 Space Requirement Analysis.....	26
2.5 Performance Evaluation.....	27
2.5.1 Scalability Study of DSM-FI Algorithm.....	28
2.5.2 Comparison with BTS algorithm.....	30
2.6 Conclusions.....	30
Chapter 3 Online Mining of Frequent Itemsets over Stream Sliding Windows	32
3.1 Introduction.....	33
3.2 Problem Definition: Mining Frequent Itemsets in a TransSW.....	34
3.3 The Proposed Algorithm: MFI-TransSW.....	36
3.3.1 Bit-Sequence Representation.....	36
3.3.2 The MFI-TransSW Algorithm.....	37
3.3.2.1 Window Initialization Phase.....	37
3.3.2.2 Window Sliding Phase.....	38
3.3.2.3 Frequent Itemsets Generation Phase.....	40
3.4 Problem Definition: Mining Frequent Itemsets in a TimeSW.....	41
3.5 The Proposed Algorithm: MFI-TimeSW.....	43
3.5.1 Time Unit List and Bit-Sequences of Items.....	43
3.5.2 The MFI-TimeSW Algorithm.....	44
3.5.2.1 Window Initialization Phase.....	44
3.5.2.2 Window Sliding Phase.....	45
3.5.2.3 Frequent Itemsets Generation Phase.....	45
3.6 Performance Evaluation.....	49
3.6.1 Experiments of MFI-TransSW Algorithm.....	50
3.6.2 Experiments of MFI-TimeSW Algorithm.....	54
3.7 Conclusions.....	55
Chapter 4 Online Mining of Changes of Items across Two Data Streams	56

4.1	Introduction	56
4.2	Related Work	58
4.3	Problem Definition: Mining of Changes of Items across Two Data Streams.....	58
4.4	Online Mining Changes of Items over Distributed ADSs	61
4.4.1	<i>A New Summary Data Structure: Change-Sketch</i>	61
4.4.2	<i>The MFC-append Algorithm</i>	62
4.4.3	<i>Space Analysis of Change-Sketch</i>	66
4.5	Online Mining Changes of Items over Distributed DDSs.....	67
4.6	Performance Evaluation	68
4.6.1	<i>Synthetic Data Generation</i>	68
4.6.2	<i>Experimental Results</i>	70
4.7	Conclusions	72
Chapter 5	Online Mining of Path Traversal Patterns over Web Click-Streams.....	74
5.1	Introduction	74
5.2	Problem Definition: Online Mining of Path Traversal Patterns	78
5.3	The Proposed Algorithm: DSM-PLW	79
5.3.1	<i>Construction of the In-memory Summary Data Structure</i>	80
5.3.2	<i>Pruning Mechanism of the Summary Data Structure</i>	86
5.3.3	<i>Determination of Path Traversal Patterns from SP-forest</i>	88
5.4	Performance Evaluation	89
5.4.1	<i>Experimental Results of Synthetic Data</i>	90
5.4.2	<i>Experimental Results of Real Data</i>	96
5.5	Conclusions	96
Chapter 6	Online Mining of Top-K Path Traversal Patterns over Web Click-Streams.....	98
6.1	Introduction	98
6.2	Problem Definition	99
6.3	The Proposed Algorithm: DSM-TKP.....	100
6.3.1	<i>Effective Construction of the Summary Data Structure</i>	101
6.3.2	<i>Effective Pruning of the Summary Data Structure</i>	105
6.3.3	<i>Determination of the Top-K Path Traversal Patterns</i>	106
6.4	Performance Evaluation	106
6.5	Conclusions	107
Chapter 7	Conclusions and Future Work	109
7.1	Conclusions	109
7.1.1	<i>Summary of Mining of Frequent Itemsets in Data Streams</i>	109
7.1.2	<i>Summary of Mining of Frequent Itemsets over Stream Sliding Windows</i>	110
7.1.3	<i>Summary of Mining of Changes of Items across Two Data Streams</i>	110
7.1.4	<i>Summary of Mining of Path Traversal Patterns over Web Click-Streams</i>	111
7.1.5	<i>Summary of Mining of Top-K Path Traversal Patterns</i>	111
7.2	Future Work	111
References	113
Publication List	120
Vita	124

List of Figures

Figure 2- 1. Typical processing model of data streams	7
Figure 2- 2. Algorithm SFI-forest Construction	16
Figure 2- 3. Subroutines of SFI-forest construction algorithm	18
Figure 2- 4. SFI-forest construction after processing the first transaction $\langle acdf \rangle$	18
Figure 2- 5. SFI-forest construction after processing the second transaction $\langle abe \rangle$	19
Figure 2- 6. SFI-forest construction after processing the window W_j	20
Figure 2- 7. SFI-forest after pruning all infrequent items	21
Figure 2- 8. Algorithm todoFIS	24
Figure 2- 9. Resource requirements of DSM-FI algorithm for IBM synthetic datasets: (a) execution time, (b) memory usage	29
Figure 2- 10. Comparison of DSM-FI and BTS: (a) Execution time, (b) Memory Usage.....	30
Figure 3- 1. Transaction-sensitive sliding window and time-sensitive sliding window [51].....	32
Figure 3- 2. An example transaction data stream and the frequent itemsets over two consecutive TransSWs.....	35
Figure 3- 3. Bit-sequences of items in window initialization phase of TransSW	37
Figure 3- 4. Bit-sequences of items after sliding $TransSW_1$ to $TransSW_2$	37
Figure 3- 5. Algorithm MFI-TransSW.....	39
Figure 3- 6. Steps of frequent itemsets generation in $TransSW_2$	40
Figure 3- 7. An example transaction data stream and the frequent itemsets over two time-sensitive sliding windows.....	43
Figure 3- 8. Bit-sequences of items in window initialization phase of $TimeSW_1$	46
Figure 3- 9. Bit-sequences of items after sliding $TimeSW_1$ to $TimeSW_2$	47
Figure 3- 10. Algorithm MFI-TimeSW	48
Figure 3- 11. Steps of frequent itemsets generation of MFI-TimeSW in $TimeSW_1$	49
Figure 3- 12. Memory usages in window initialization phases of algorithms SWFI-stream and MFI-TransSW ($s = 0.1\%$ and $w = 20,000$)	51
Figure 3- 13. Memory usages in window sliding phases of algorithms SWFI-stream and MFI-TransSW ($s = 0.1\%$ and $w = 20,000$)	51

Figure 3- 14. Memory usages in frequent itemset generation phases of algorithms SWFI-stream and MFI-TransSW ($s = 0.1\%$ and $w = 20,000$)	52
Figure 3- 15. Processing time in window initialization phases of algorithms SWFI-stream and MFI-TransSW under different window sizes ($s = 0.1\%$)	52
Figure 3- 16. Processing time including window sliding time and pattern generation time of algorithms SWFI-stream and MFI-TransSW under window size 200K transactions ($s = 0.1\%$)	53
Figure 3- 17. Memory usages of MFI-TimeSW algorithm in different phases ($s = 0.1\%$)	53
Figure 3- 18. Processing time of MFI-TimeSW algorithm in different phases ($s = 0.1\%$)	54
Figure 4- 1. Processing model of distributed data streams	57
Figure 4- 2. Examples of VFCIs and SFCIs	60
Figure 4- 3. Notations and conventions used in the proposed algorithms	63
Figure 4- 4. Algorithm MFC-append	64
Figure 4- 5. Algorithm MFC-dynamic	69
Figure 4- 6. Experiments on synthetic data (10^4 transactions) for <i>MFC-append</i> . Left: recall (proportion of the frequent change patterns reported). Right: precision (proportion of the output frequency change patterns which are frequent)	70
Figure 4- 7. Experiments on synthetic data (10^5 transactions) for <i>MFC-append</i> . Left: recall. Right: precision	71
Figure 4- 8. Experiments on synthetic data (10^6 transactions) for <i>MFC-append</i> . Left: recall. Right: precision	71
Figure 4- 9. Experiments on synthetic data (10^6 transactions) for <i>MFC-dynamic</i> . Left: recall. Right: precision	72
Figure 5- 1. Process of online mining of path traversal patterns in Web click streams	79
Figure 5- 2. Algorithm SP-forest construction	82
Figure 5- 3. Subroutines of SP-forest construction algorithm	83
Figure 5- 4. SP-forest after processing the first maximal forward reference $\langle acdef \rangle$	84
Figure 5- 5. SP-forest after processing the second maximal forward reference $\langle abe \rangle$	84
Figure 5- 6. SP-forest after processing the first six maximal forward references	85
Figure 5- 7. SP-forest after pruning the infrequent reference b	87
Figure 5- 8. Algorithm MRS-mining	89

Figure 5- 9. Performance comparisons of total execution time over various minimum support thresholds.....	91
Figure 5- 10. Performance comparisons of memory usage over various minimum support thresholds.....	92
Figure 5- 11. Accuracy of mining results	93
Figure 5- 12. Linear scalability of the streaming data size.....	93
Figure 5- 13. Memory usage of DSM-PLW on BMS-WebView-1 and BMS-WebView-2 over various minimum support thresholds	94
Figure 5- 14. Execution time of DSM-PLW on BMS-WebView-1 and BMS-WebView-2 over various minimum support thresholds	95
Figure 6- 1. Algorithm of TKP-forest Construction	103
Figure 6- 2. TKP-forest construction after processing the first maximal forward reference $\langle abcde \rangle$	104
Figure 6- 3. TKP-forest construction after processing the second maximal forward reference $\langle acd \rangle$	104
Figure 6- 4. Algorithm of TKP-forest pruning	105
Figure 6- 5. Example of TKP-forest	106
Figure 6- 6. Execution time and memory usage of DSM-TKP on BMS-WebView-1 and BMS-WebView-2 under various k values.....	108

Chapter 1 Introduction

1.1 Background

Data mining, which is also referred to as knowledge discovery in databases, has been recognized as *the process of extracting non-trivial, implicit, previously unknown and potentially useful information or knowledge from large amounts of data*. The typical data mining tasks include association mining, sequential pattern mining, classification, and clustering. The tasks help us to finding interesting patterns and regularities from the data. Traditional data mining techniques assume the targeting databases are disk resident or could be fit into the main memory. Hence, due to the complexity of mining tasks, almost all data mining algorithms require scanning the data several times.

Recently, database and knowledge discovery communities have focused on a new model of data processing, where data arrive in the form of continuous *streams*. It is often referred to as *data streams* or *streaming data*. The new data model addresses the data explosion from two new perspectives. First, the arrival of data streams and the volume of data are beyond our capability to store them. For example, the network traffic information of a router, though extremely important, is often impossible to record. Second, data streams processing requires real-time constraint. Generally, the need to process the data timely prohibits rescanning the data from secondary storage. For example, detecting network intrusion in real-time is the necessary condition to prevent the damage. The new model has captured a large class of important applications in current world, such as discovering the patterns of sensor data generated from sensor networks, analyzing the transactional behaviors of transaction flows in retail chains, mining user traversal behaviors from the Web record and click-streams, protecting network securities, timely finding terrorist activities, monitoring call records in

telecommunications, analyzing stock and business data, and so on [6, 33].

In order to facilitate the following discussions, we will first introduce the streaming data model in more detail. Data streams assume the data elements arrive in some order. Moreover, the amount of data is often huge and can not be held in the main memory or even disks. This means that once a new data element arrives, it must be processed quickly. In general, the period for a data element staying in the main memory is quite short. Once a data element is removed from the main memory, it is not available to be accessed again. In other words, we can only have one look at the data.

Data mining over streaming data brings many new challenges [6]. The first challenge is how to perform data mining tasks on data streams. Most of existing data mining algorithms require scanning datasets multiple times, such as Apriori algorithm of association rule mining, k-means of clustering, and C4.5 of decision tree construction. The new data model limits us to have only one look at the data, or at most to scan it once. Further, the relatively small memory compared with the large amount of streaming data results in the fact that we can only store a concise summary or partial data of the data stream. Therefore, getting precise results from data streams is commonly impossible or very difficult. The challenge is how to design efficient algorithms to get approximate results with high accuracy and confidence. The second challenge is how to understand the changes of data streams. The data streams bring us much new useful information to explore, such as the knowledge that if and when the underlying distribution has changed for continuous data streams. An example is to find such products in the retail chains that have become very popular recently in certain regions, but relatively unpopular for quite a long time before. In conclusions, how to perform data mining tasks, how to discover new knowledge, and how to mine changes of data streams make stream mining very challenging.

1.2 Research Objectives and Contributions

The research objective of this dissertation is to investigate efficient and scalable algorithms for mining frequent itemsets, path traversal patterns, and the changes of items over continuous data streams.

The first research issue of this dissertation is the online mining of frequent itemset over data streams. We propose the DSM-FI (Data Stream Mining for Frequent Itemsets) algorithm to find the set of all frequent itemsets over the entire history of the data streams. An effective projection method is used in the proposed algorithm to extract the essential information from each incoming transaction of data streams. A summary data structure based on the prefix tree is constructed. DSM-FI utilizes a top-down pattern selection approach to find the complete set of frequent itemsets. Experiments show that DSM-FI outperforms BTS (Buffer-Trie-SetGen), a state-of-the-art single-pass algorithm, by one order of magnitude for discovering the set of all frequent itemsets over a landmark window of data streams. For mining of frequent itemsets in data streams with a sliding window, we propose an online algorithm, called MFI-TransSW (Mining Frequent Itemsets over a Transaction-sensitive Sliding Window), to mine the set of frequent itemsets in streaming data with a transaction-sensitive sliding window. Moreover, another single-pass algorithm called MFI-TimeSW (Mining Frequent Itemsets over a Time-sensitive Sliding Window) based on the proposed MFI-TransSW algorithm, is proposed to mine the set of frequent itemsets in a time-sensitive sliding window. An effective bit-sequence representation of items is used in the proposed algorithms to reduce the time and memory needed to slide the windows. Experiments show that the proposed algorithms not only attain highly accurate mining results, but also run significantly faster and consume less memory than do existing algorithms for mining recent frequent itemsets over data streams.

The second research issue of the thesis is change mining of data streams. We define a new problem of the online mining of changes of items across two data streams, and propose

an one-pass algorithm, called MFC-append (Mining Frequency Changes of append-only data streams), to mine the set of frequent frequency changed items, vibrated frequency changed items, and stable frequency changed items across two continuous append-only data streams. Furthermore, a single-pass algorithm, called MFC-dynamic (Mining Frequency Changes of dynamic data streams) based on MFC-append, is proposed to mine the changes across two dynamic data streams. A new summary data structure, called Change-Sketch, is developed to compute the frequency changes between two data streams as fast as possible. Theoretical analysis and experimental results show that our algorithms meet the major performance requirements, namely single-pass, bounded space requirement, and real-time computing, in mining streaming data.

The third issue of the work is the online mining of all path traversal patterns over Web click-streams. We proposed the first single-pass algorithm, called DSM-PLW (Data Stream Mining for Path traversal patterns in a Landmark Window), to discover the path traversal patterns over Web click-streams with a user-defined minimum support constraint. Moreover, we proposed the first online algorithm, called DSM-TKP (Data Stream Mining for Top-K Path traversal patterns), to mine the set of top-K path traversal patterns without a user-specified minimum support threshold. Experiments of real click-streams show that both algorithms successfully mine maximal reference sequences with linear scalability.

All the proposed algorithms are verified by experiments of mining continuous streams of various characteristics. In the experiments comprising comprehensive comparisons, the proposed algorithms outperforms several related algorithms, and they all show excellent linear scalability with respect to the size of the streaming data.

1.3 Organization of this Thesis


The rest of this dissertation is organized as follows. In Chapter 2, we describe efficient

one-pass algorithms for mining frequent itemsets and maximal frequent itemsets in a landmark window of data streams. Efficient single-pass algorithms for mining frequent itemsets over stream sliding windows are delineated in Chapter 3. Chapter 4 addresses the problem of mining of changes of items over append-only and dynamic data streams. Efficient algorithms for mining path traversal patterns with a user-specified minimum support constraint over Web click-streams are introduced in Chapter 5. The problem of mining top-K path traversal patterns is discussed in Chapter 6. Finally, the conclusions and future work are given in Chapter 7.



Chapter 2 Online Mining of Frequent Itemsets in Data Streams

In recent years, database and knowledge discovery communities have focused on a new data model, where data arrive in the form of *continuous streams*. It is often referred to as *data streams* or *streaming data*. Data streams possess some computational characteristics, such as unknown or unbounded length, possibly very fast arrival rate, inability to backtrack over previously-arrived data elements (only one sequential pass over the data is permitted), and a lack of system control over the order in which the data arrive [6]. Many applications generate data streams in real time, such as sensor data generated from sensor networks, transaction flows in retail chains, Web record and click-streams in Web applications, performance measurement in network monitoring and traffic management, and call records in telecommunications.



Online mining of data streams differs from traditional mining of static datasets in the following aspects [6]. First, each data element in streaming data should be examined at most once. Second, the memory usage for mining data streams should be bounded even though new data elements are continuously generated from the stream. Third, each data element in the stream should be processed as fast as possible. Fourth, the analytical results generated by the online mining algorithms should be instantly available when requested by the users. Finally, the frequency errors of outputs generated by the online algorithms should be as small as possible. The online processing model of data streams is shown in Figure 2-1.

As described above, the *continuous* nature of streaming data makes it essential to use the online algorithms which require only *one scan* over the data streams for knowledge discovery. The *unbounded* characteristic makes it impossible to store all the data into the main memory or even in secondary storage. This motivates the design of *summary data structure* with small

footprints that can support both one-time and continuous queries of streaming data. In other words, one-pass algorithms for mining data streams have to sacrifice the exactness of its analytical results by allowing some tolerable counting errors. Hence, traditional *multiple-pass* techniques studied for mining static datasets are not feasible to mine patterns over streaming data.

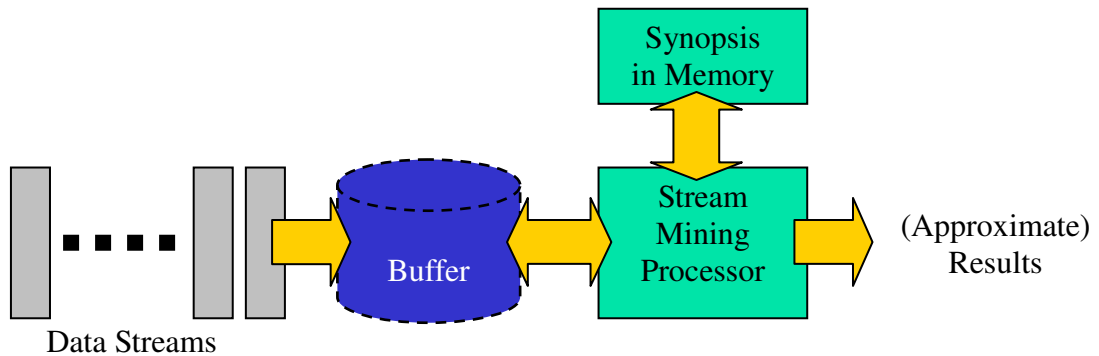


Figure 2- 1. Typical processing model of data streams

2.1 Introduction

Frequent itemsets mining is one of the most important research issues in data mining. The problem of frequent itemsets mining of *static datasets* (not *streaming data*) was first introduced by Agrawal *et al.* [2] described as follows. Let $\Psi = \{i_1, i_2, \dots, i_n\}$ be a set of literals, called *items*. Let database DB be a set of transactions, where each transaction T contains a set of items, such that $T \subseteq \Psi$. The *size* of database DB is the total number of transactions in DB and is denoted by $|DB|$. A set of items is referred to as an *itemset*. An itemset X with l items is denoted by $X = (x_1 x_2 \dots x_l)$, such that $X \subseteq \Psi$. The *support* of an itemset X is the number of transactions in DB containing the itemset X as a subset, and denoted by $sup(X)$. An itemset X is *frequent* if $sup(X) \geq minsup \cdot |DB|$, where $minsup$ is a user-specified minimum support threshold in the range of $[0, 1]$. Consequently, given a database DB and a user-defined minimum support threshold $minsup$, the problem of mining

frequent itemsets in *static datasets* is to find the set of all itemsets whose support is no less than $minsup \cdot |DB|$. In this paper, we will focus on the problem of mining frequent itemsets over the entire history of *data streams*.

Many previous studies contributed to the efficient mining of frequent itemsets in streaming data. According to the stream processing model [70], the research of mining frequent itemsets in data streams can be divided into three categories: *landmark windows*, *sliding windows*, and *damped windows*, as described briefly as follows. In the landmark windows model, knowledge discovery is performed based on the values between a specific timestamp called *landmark* and the present time. In the sliding windows model, knowledge discovery is performed over a fixed number of recently generated data elements as the target of data mining. In the damped windows model, recent sliding windows are more important than previous ones.

In [53], Manku and Motwani developed two single-pass algorithms, Sticky-Sampling and Lossy Counting, to mine frequent items over a landmark window. Moreover, Manku and Motwani proposed the first single-pass algorithm *BTS* (Buffer-Trie-SetGen) based on the Lossy-Counting [53] to mine the set of frequent itemsets (FI) from streaming data. Chang and Lee [11] proposed a *BTS*-based algorithm for mining frequent itemsets in sliding windows model. Moreover, Chang and Lee [10] also developed another algorithm, called *estDec*, for mining frequent itemsets in streaming data in which each transaction has a weight decreasing with age. In other words, older transactions contribute less toward itemset frequencies, and it is a kind of damped windows model. Teng *et al.* [63] proposed a regression-based algorithm, called *FTP-DS*, to find frequent itemsets across multiple data streams in a sliding window. Lin *et al.* [51] proposed an incremental mining algorithm to find the set of frequent itemsets in a time-sensitive sliding window. Giannella *et al.* [31] proposed a frequent pattern tree (abbreviated as *FP-tree*) [35] based algorithm, called *FP-stream*, to mine frequent itemsets at

multiple time granularities by a novel tilted-time windows technique. Yu *et al.* [68] discussed the issues of false negative or false positive in mining frequent itemsets from high speed transactional data streams. Jin and Agrawal [39] proposed an algorithm, called StreamMining, for in-core frequent itemset mining over data streams. Chi *et al.* [18] proposed an algorithm, called MOMENT, that might be the first to find *frequent closed itemsets* (FCI) from data streams. A summary data structure called CET is used in the MOMENT algorithm to maintain the information of closed frequent itemsets.

Because the focus of the chapter is on frequent itemses mining over data streams with a landmark window, we mainly address this issue by comparison with the BTS algorithm proposed by Manku and Motwani [53]. In the following, we describe the BTS algorithm in detail. In the BTS algorithm, two estimated parameters: *minimum support threshold* s , and *maximum support error threshold* ϵ , are used, where $0 < \epsilon \leq s < 1$. The incoming data stream is conceptually divided into buckets of width $w = \lceil 1/\epsilon \rceil$ transactions each, and the current length of the stream is denoted by N transactions.

The BTS algorithm is composed of three steps. In the first step, BTS repeatedly reads a *batch* of buckets into main memory. In the second step, it decomposes each transaction within the current bucket into a set of itemsets, and stores these itemsets into a summary data structure D which contains a set of entries of the form $(e, e.freq, e.\Delta)$, where e is an itemset, $e.freq$ is an approximate *frequency* of the itemset e , and $e.\Delta$ is the maximum possible error in $e.freq$.

For each itemset e extracted from the incoming transaction T , BTS performs two operations to maintain the summary data structure D . First, it counts the occurrences of e in the current batch, and updates the value $e.freq$ if the itemset e already exists in the structure D . Second, BTS creates a new entry $(e, e.freq, e.\Delta)$ in D , if the itemset e does not occur in D , but its estimated frequency $e.freq$ in the batch is greater than or equal to $|batch| \cdot \epsilon$, where the value

of maximal possible error $e.\Delta$ is set to $\lfloor |batch| \cdot \epsilon \rfloor$, and $|batch|$ denotes the total number of transactions in the current batch. To bound the space requirement of D , BTS algorithm deletes the updated entry e if $e.freq + e.\Delta \leq |batch| \cdot \epsilon$. Finally, BTS outputs those entries e_i in D , where $e_i.freq \geq (s - \epsilon) \cdot N$, when a user requests a list of itemsets with the minimum support threshold s and the support error threshold ϵ .

The motivation of the proposed work is to develop a method that utilizes some space-effective summary data structures to reduce the cost in mining frequent itemsets over data streams. In this paper an efficient, single-pass algorithm, referred to as *Data Stream Mining for Frequent Itemsets* (abbreviated as **DSM-FI**), is proposed to improve the efficiency of frequent itemset mining in data streams. A new summary data structure called *summary frequent itemset forest* (abbreviated as **SFI-forest**) is developed for online incremental maintaining of the essential information about the set of all frequent itemsets of data streams generated so far.

The proposed algorithm DSM-FI has three important features: a single pass of streaming data for counting the support of significant itemsets; an extended prefix tree-based, compact pattern representation of summary data structure; and an effective and efficient search and determination mechanism of frequent itemsets. Moreover, the frequency error guarantees provided by DSM-FI algorithm is the same as that of BTS algorithm. The error guarantees are stated as follows. First, all itemsets whose true support exceeds $s \cdot N$ are output. Second, no itemsets whose true support is less than $(s - \epsilon) \cdot N$ is output. Finally, estimated supports of itemsets are less than the true support by at most $\epsilon \cdot N$.

The comprehensive experiments show that our algorithm is efficient on both sparse and dense data, and scalable to the continuous data streams. Furthermore, DSM-FI algorithm outperforms BTS, a state-of-the-art single-pass algorithm, by one order of magnitude for discovering the set of all frequent itemsets over the entire history of the data streams.

The remainder of the chapter is organized as follows. Section 2.2 defines the problem of single-pass mining frequent itemsets in a landmark window over data streams. The proposed DSM-FI algorithm is described in Section 2.3. The extended prefix tree-based summary data structure SFI-forest is introduced to maintain the essential information about the set of all frequent itemsets of the stream generated so far. Theoretical analysis and experiments are presented in Section 2.4. We conclude the chapter in Section 2.5.

2.2 Problem Definition

Based on the estimation mechanism of the BTS algorithm, we propose a new, single-pass algorithm to improve the efficiency of mining frequent itemsets over the entire history of data streams when a user-specified minimum support threshold $s \in (0, 1)$, and a maximum support error threshold $\epsilon \in (0, s)$ are given.

Let $\Psi = \{i_1, i_2, \dots, i_m\}$ be a set of literals, called *items*. An *itemset* is a nonempty set of items. A l -itemset, denoted by $(x_1x_2\dots x_l)$, is an itemset with l items. A transaction T consists of a unique transaction identifier (*tid*) and a set of items, and denoted by $\langle tid, (x_1x_2\dots x_q) \rangle$, where $x_i \in \Psi, \forall i = 1, 2, \dots, q$. A *basic window* W consists of k transactions. The basic windows are labeled with window identifier *wid*, starting from 1.

Definition 2-1 A *data stream*, $DS = [W_1, W_2, \dots, W_N)$, is an infinite sequence of basic windows, where N is the window identifier of the “*latest*” basic window. The *current length* of DS , written as $DS.CL$, is $k \cdot N$, i.e., $|W_1| + |W_2| + \dots + |W_N|$. The windows arrive in some order (implicitly by arrival time or explicitly by timestamp), and may be seen only once.

Mining frequent itemsets in landmark windows over data streams is to mine the set of all frequent itemsets from the transactions between a specified window identifier called *landmark* and the current window identifier N . Note that the value of landmark is set to 1.

To ensure the completeness of frequent itemsets for data streams, it is necessary to store not only the information related to frequent itemsets, but also the information related to infrequent ones. If the information about the currently infrequent itemsets were not stored, such information would be lost. If these itemsets become frequent later on, it would be impossible to figure out their correct support and their relationship with other itemsets. The data stream mining algorithms have to sacrifice the exactness of the analytical results by allowing some tolerable support errors since it is unrealistic to store all the streaming data into the limited main memory. Hence, we define two types of *support* of an itemset, and divide the itemsets embedded in the stream into three categories: *frequent itemsets*, *significant itemsets*, and *infrequent itemsets*.

Definition 2-2 The *true support* of an itemset X , denoted by $X.tsup$, is the number of transactions in the data stream containing the itemset X as a subset. The *estimated support* of an itemset X , denoted by $X.esup$, is the estimated true support of X stored in the summary data structure, where $0 < X.esup \leq X.tsup$.

Definition 2-3 The *current length (CL)* of data stream with respect to an itemset X stored in the summary data structure, denoted by $X.CL$, is $(N-j+1) \cdot k$, i.e., $|W_j| + |W_{j+1}| + \dots + |W_N|$, where W_j is the first basic window with the window identifier j stored in the current summary data structure containing the itemset X , and N is the window identifier of current window.

Definition 2-4 An itemset X is *frequent* if $X.tsup \geq s \cdot X.CL$. An itemset X is *significant* if $s \cdot X.CL > X.tsup \geq \epsilon \cdot X.CL$. An itemset X is *infrequent* if $\epsilon \cdot X.CL > X.tsup$.

Definition 2-5 A frequent itemset is *maximal* if it is not a subset of any other frequent itemsets generated so far.

Therefore, given a data stream $DS = [B_1, B_2, \dots, B_N]$, a user-defined minimum support

threshold s in the range of $[0, 1]$, and a user-specified maximum support error threshold ϵ in the range of $[0, s]$, the problem of mining frequent itemsets in landmark windows over data streams is to find the set of all frequent itemsets in single scan of the data stream.

2.3 The Proposed Algorithm: DSM-FI

The proposed DSM-FI (Data Stream Mining for Frequent Itemsets) algorithm consists of four steps.

- (a) Read a basic window of transactions from the buffer in main memory, and sort the items of transaction in the lexicographical order (Step 1).
- (b) Construct and maintain the in-memory summary data structure (Step 2).
- (c) Prune the infrequent information from the summary data structure (Step 3).
- (d) Find the frequent itemsets from the summary data structure (Step 4).

Steps 1 and 2 are performed in sequence for a new incoming basic window. Step 3 is performed after every basic window has been processed. Finally, step 4 is usually performed periodically or when it is needed. Since the reading of a basic window of transactions from the buffer is straightforward, we shall henceforth focus on Steps 2, 3, and 4, and devise algorithms for effective construction and maintenance of summary data structure, and efficient determination of frequent itemsets.

2.3.1 Construction and Maintenance of Summary Data structure

In this section, we describe the algorithm which constructs and maintains the in-memory summary data structure called *SFI-forest* (Summary Frequent Itemset forest).

Definition 2-6 A *summary frequent itemset forest (SFI-forest)* is a summary data structure and is defined as follows.

1. *SFI-forest* consists of a **frequent item list (FI-list)**, and a set of **summary frequent itemset trees (SFI-trees)** of **item-prefixes**, denoted by *item-prefix.SFI-trees*.
2. Each node in the *item-prefix.SFI-tree* consists of four fields: **item-id**, **item-id.esup**, **item-id.window-id**, and **item-id.node-link**. The first field *item-id* is the item identifier of the inserting item. The second field *item-id.esup* registers the number of transactions represented by a portion of the path reaching the node with the item-id. The value of the third field *item-id.window-id* assigned to a new node is the window identifier of the current window. The final field *item-id.node-link* links up a node with the next node with the same item-id in the same SFI-tree or null if there is none.
3. Each entry in the *FI-list* consists of four fields: **item-id**, **item-id.esup**, **item-id.window-id**, and **item-id.head-link**. The *item-id* registers which item identifier the entry represents, *item-id.esup* records the number of transactions containing the item carrying the item-id, the value of *item-id.window-id* assigned to a new entry is the window identifier of current window, and *item-id.head-link* points to the root node of the *item-id.SFI-tree*. Note that each entry with *item-id* in the FI-list is an *item-prefix* and it is also the *root node* of the *item-id.SFI-tree*.
4. Each *item-prefix.SFI-tree* has a specific **opposite frequent item list (OFI-list)** with respect to the item-prefix, denoted by **item-prefix.OFI-list**. The *item-prefix.OFI-list* is composed of four fields: **item-id**, **item-id.esup**, **item-id.window-id**, and **item-id.head-link**. The *item-prefix.OFI-list* operates the same as the *FI-list* except that the field *head-link* links to the first node with the same item-id in the *item-prefix.SFI-tree*. Note that $|item-prefix.OFI-list| = |FI-list|$ in the worst case, where $|FI-list|$ denotes the total number of entries in the FI-list.

Figure 2-2 outlines the SFI-forest construction of the proposed DSM-FI algorithm. First of all, DSM-FI algorithm reads a transaction T from the current window B_N . Then, DSM-FI

projects this transaction T into many sub-transactions, and inserts these sub-transactions into the SFI-forest. The details of this projection are described as follows. A transaction T with m items, such as $(x_1x_2\dots x_m)$, in the current window should be projected by inserting m *item-prefix sub-transactions* into the SFI-forest. In other words, the transaction $T = (x_1x_2\dots x_m)$ is converted into m sub-transactions; that is, $(x_1x_2\dots x_m)$, $(x_2x_3\dots x_m)$, ..., $(x_{m-1}x_m)$, and (x_m) . These m sub-transactions are called *item-prefix transactions*, since the first item of each sub-transaction is an *item-prefix* of the original transaction T . This step, called **transaction projection**, is denoted by $TP(T) = \{x_1|T, x_2|T, \dots, x_i|T, \dots, x_m|T\}$, where $x_i|T = (x_ix_{i+1}\dots x_m)$, $\forall i = 1, 2, \dots, m$. The *projecting cost* of a transaction of length m for constructing the summary data structure SFI-forest is $(m^2+m)/2$, i.e., $m + (m-1) + \dots + 2 + 1$. Recall that the decomposing cost of a transaction with m items of BTS algorithm for constructing the summary data structure is (2^m-2) . In general, the constructing cost of summary data structure of our algorithm is extremely less than that of BTS algorithm.

After performing the transaction projection of the incoming transaction T , DSM-FI algorithm inserts T into the *FI-list*, and then removes T from the current window in the main memory. Then, the items of these item-prefix transactions are inserted into the *item-prefixes.SFI-trees* as branches, and the estimated support of the corresponding *item-prefixes.OFI-lists* are updated. If an itemset shares a prefix of an itemset already in the SFI-tree, the new itemset will share a prefix of the branch representing that itemset. In addition, an estimated support counter is associated with each node in the tree. The counter is updated when an item-prefix transaction causes the insertion of a new branch. Figure 2-3 shows the subroutines of SFI-forest construction and maintenance.

Example 2-1. Let the W_j be a window with the landmark identifier j , and it contains six transactions: $\langle acdf \rangle$, $\langle abe \rangle$, $\langle df \rangle$, $\langle cef \rangle$, $\langle acdef \rangle$ and $\langle cef \rangle$, where a, b, c, d, e and f are items in the data stream. The SFI-forest with respect to the first two transactions, $\langle acdf \rangle$

and $\langle abe \rangle$, constructed by DSM-FI algorithm is described as follows. Note that each node of the form $(id: id.esup: id.wid)$ is composed of three fields: *item-id*, *estimated support*, and *window-id*. For example, $(a: 2: j)$ indicates that, from basic window W_j to current basic window W_N ($1 \leq j \leq N$), item a appeared twice.

Algorithm SFI-forest construction

Input: A data stream, $DS = [B_1, B_2, \dots, B_N]$ with landmark 1, a user-specified minimum support threshold $s \in (0, 1)$, and a maximum support error threshold $\epsilon \in (0, s)$.

Output: A SFI-forest generated so far.

```

1: FI-list = {}; /*initialize the FI-list to empty.*/
2: foreach window  $B_j$  do /*  $j = 1, 2, \dots, N$  */
3:   foreach transaction  $T = (x_1 x_2 \dots x_m) \in B_j$  ( $j = 1, 2, \dots, N$ ) do
      /*  $m \geq 1$  and  $j$  is the current window identifier */
4:     foreach item  $x_i \in T$  do /* the maintenance of FI-list */
5:       if  $x_i \notin$  FI-list then
6:         create a new entry of form  $(x_i, 1, j, head-link)$  into the FI-list;
          /* the entry form is  $(item-id, item-id.esup, window-id, head-link)$  */
7:       else /* the entry already exists in the FI-list */
8:          $x_i.esup = x_i.esup + 1$ ;
          /* increment the estimated support of item-id  $x_i$  by one */
9:       end if
10:    end for
11:    call TP( $T, j$ );
      /* project the transaction with each item-prefix  $x_i$  for constructing the  $x_i$ .SFI-tree */
12:  end for
13:  call SFI-forest-pruning(SFI-forest,  $\epsilon, N$ ); /* Step 3 of DSM-FI algorithm */
14: end for

```

Figure 2- 2. Algorithm SFI-forest Construction

Subroutine TP /* Step 2 of DSM-FI algorithm: construct and maintain the SFI-forest */

Input: A transaction $T = (x_1x_2\dots x_m)$ and the current window-id j ;

Output: x_i .SFI-tree, $\forall i = 1, 2, \dots, m$;

```
1: foreach item  $x_i$ ,  $\forall i = 1, 2, \dots, m$ , do
2:     SFI-tree-maintenance( $[x_i|X]$ ,  $x_i$ .SFI-tree,  $j$ );
        /*  $X = x_1, x_2, \dots, x_m$  is the original incoming transaction  $T$  */
        /*  $[x_i|X]$  is an item-prefix transaction with the item-prefix  $x_i$  */
3: end for
```

Subroutine SFI-tree-maintenance /* Step 2 of DSM-FI algorithm */

Input: An item-prefix transaction $(x_ix_{i+1}\dots x_m)$, the current window-id j , and x_i .SFI-tree, where $i=1, 2, \dots, m$;

Output: A modified x_i .SFI-tree, where $i=1, 2, \dots, m$;

```
1: foreach item  $x_l$  do /*  $l = i+1, i+2, \dots, m$  */
2:     if  $x_l \notin x_i$ .OFI-list then /*  $x_i$ .OFI-list maintenance */
3:         create a new entry of form  $(x_l, 1, j, head-link)$  into the  $x_i$ .OFI-list;
            /* the entry form is  $(item-id, item-id.esup, item-id.window-id,$ 
             $item-id.head-link)$  */
4:     else /* the entry already exists in the  $x_i$ .OFI-list */
5:          $x_l.esup = x_l.esup + 1$ ;
            /* increment the estimated support of item-id  $x_l$  by one */
6:     end if
7: endfor
8: foreach item  $x_i$ ,  $\forall i = 1, 2, \dots, m$ , do /*  $x_i$ .SFI-tree maintenance */
9:     if SFI-tree has a child node with item-id  $y$  such that  $y.item-id = x_i.item-id$  then
10:         $y.esup = y.esup + 1$ ; /*increment  $y$ 's estimated support by one*/
11:     else create a new node of the form  $(x_i, 1, j, node-link)$ ;
        /* initialize the estimated support of the new node to one, and link its parent link to
        SFI-tree, and its node-link linked to the nodes with same item-id via the node-link structure.
        */
12:     end if
13: end for
```

Subroutine SFI-forest-pruning /* Step 3 of DSM-FI algorithm: prune the infrequent information from the SFI-forest */

Input: A *SFI-forest*, a user-specified maximum support error threshold ϵ , and the current window identifier N ;

Output: A SFI-forest which contains the set of all significant and frequent itemsets.

- 1: **foreach** entry $x_i (i=1, 2, \dots, d) \in \text{FI-list}$, where $d = |\text{FI-list}|$ **do**
- 2: **if** $x_i.\text{esup} < \epsilon \cdot x_i.CL$ **then** /* if x_i is an infrequent item */
- 3: delete x_i .SFI-tree;
- 4: delete the entry x_i from the FI-list;
- 5: delete x_i from other x_j .OFI-list if it exists in x_j .OFI-list ($j = 1, 2, \dots, d; j \neq i$);
- 6: delete those nodes ($\text{item-id} = x_i$) in other SFI-trees via node-link structures and merge the fragmented sub-trees;
- /* a simple way is to reinsert or to join the remainder sub-trees into the SFI-tree */;
- 7: **end if**
- 8: **end for**

Figure 2- 3. Subroutines of SFI-forest construction algorithm

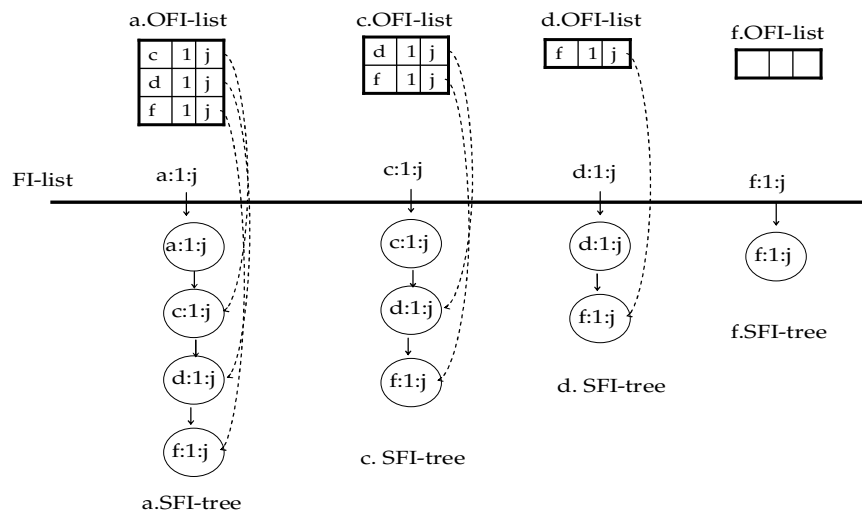


Figure 2- 4. SFI-forest construction after processing the first transaction $\langle acdf \rangle$

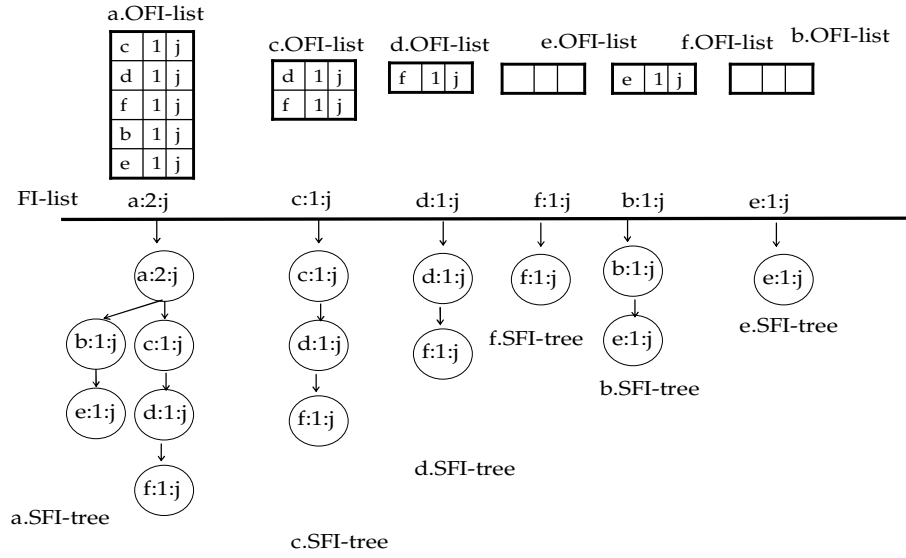


Figure 2- 5. SFI-forest construction after processing the second transaction $\langle abe \rangle$

- (a) First transaction $\langle acdf \rangle$: First of all, DSM-FI algorithm reads the first transaction and calls the Transaction-Projection($\langle acdf \rangle$). Then, DSM-FI inserts four item-prefix transactions: $\langle acdf \rangle$, $\langle cdf \rangle$, $\langle df \rangle$, and $\langle f \rangle$ into the FI-list, $[a.SFI-tree, a.OFI-list]$, $[c.SFI-tree, c.OFI-list]$, $[d.SFI-tree, d.OFI-list]$, and $[f.SFI-tree, f.OFI-list]$, respectively. The result is shown in Figure 2-4. In the following steps, the head-links of each *item-prefix*.OFI-list are omitted for concise presentation.
- (b) Second transaction $\langle abe \rangle$: DSM-FI algorithm reads the second transaction and calls the Transaction-Projection($\langle abe \rangle$). Next, DSM-FI inserts three item-prefix transactions: $\langle abe \rangle$, $\langle be \rangle$, and $\langle e \rangle$ into the FI-list, $[a.SFI-tree, a.OFI-list]$, $[b.SFI-tree, b.OFI-list]$, and $[e.SFI-tree, e.OFI-list]$, respectively. The result is shown in Figure 2-5. After processing all the transactions of window W_j , the SFI-forest generated so far is shown in Figure 2-6.

2.3.2 Pruning Infrequent Information from SFI-forest

According to the *Apriori* principle, only the frequent 1-itemsets are used to construct candidate k -itemsets, where $k \geq 2$. Thus, the set of candidate itemsets containing the

infrequent items stored in the summary data structure is pruned. The pruning is usually performed periodically or when it is needed.

Let the maximum support error threshold be ϵ in the range of $[0, s]$, where s is a user-defined minimum support threshold in the range of $[0, 1]$. The space pruning method of DSM-FI is that the item x and its supersets are deleted from SFI-forest if $x.esup < \epsilon \cdot x.CL$. For each entry $(x, x.esup, x.window-id, x.head-link)$ in the FI-list, if its $x.esup$ is less than $\epsilon \cdot x.CL$, it can be regarded as an *infrequent item*. In this case, three operations are performed in sequence. First, DSM-FI deletes the $x.OFI-list$, $x.SFI-tree$, and the infrequent entry x from the FI-list. Second, DSM-FI removes the infrequent item x of other OFI-lists by traversing the FI-list. Third, DSM-FI deletes the infrequent item x from other SFI-trees, and reconstructs these SFI-trees. After pruning all infrequent items from SFI-forest, SFI-forest contains the set of all frequent itemsets and significant itemsets of the data stream generated so far.

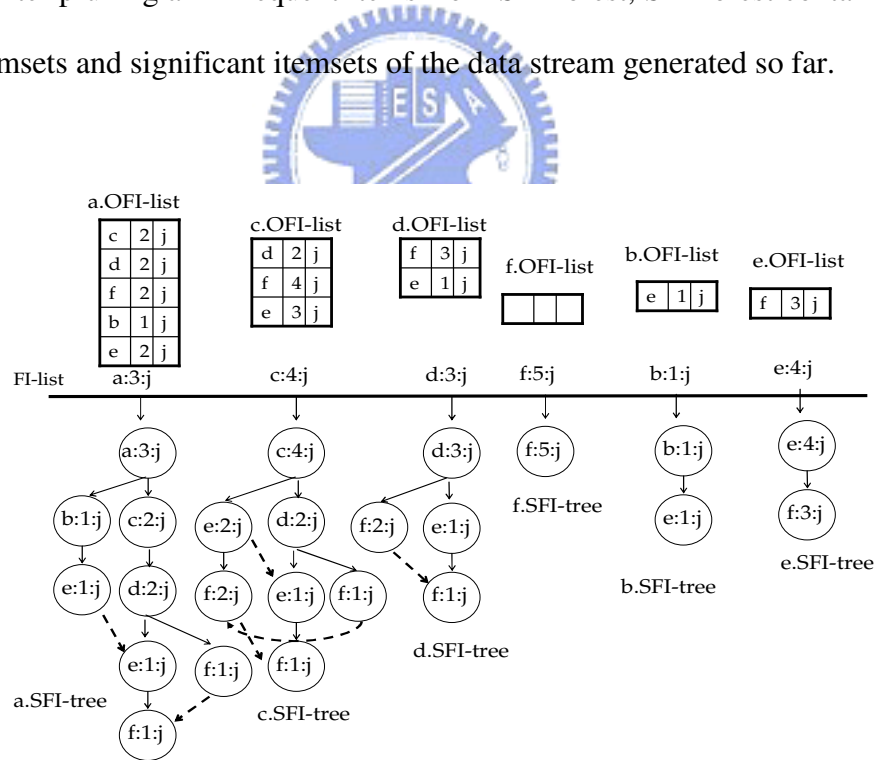


Figure 2- 6. SFI-forest construction after processing the window W_j

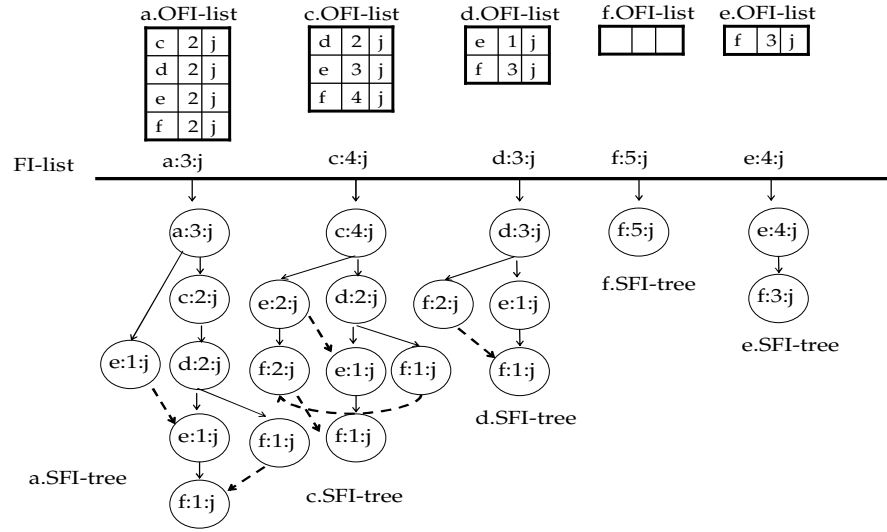


Figure 2- 7. SFI-forest after pruning all infrequent items

Example 2-2: Let the maximum support error threshold ϵ be 0.2. Hence, an itemset X is *infrequent* in Figure 2-6 if $X.esup < \epsilon \cdot X.CL$. Note that $\epsilon \cdot X.CL = 0.2 \cdot 6 = 1.2$. After computing the current window W_j , the next step of DSM-FI is to prune all the infrequent items from the current SFI-forest. At this time, DSM-FI deletes the b .SFI-tree, b .OFI-list, and item b itself from the FI-list, since item b is an infrequent item; that is, $b.esup = 1 < 1.2$. Then, DSM-FI updates the a .OFI-list and reconstructs a .SFI-tree, because a .OFI-list and a .SFI-tree contains the infrequent item b . The result is shown in Figure 2-7.

The next step of DSM-FI is to determine the set of all frequent itemsets from SFI-forest constructed so far. The step is performed only when the current results of the data stream is requested. Note that the number of candidate 2-itemsets is a performance bottleneck in the Apriori-based frequent itemset mining algorithms [3, 35]. DSM-FI algorithm can avoid this performance problem. This is because DSM-FI can generate all frequent 2-itemsets immediately by combining the frequent items in the FI-list with the frequent items in the corresponding OFI-list.

2.3.3 Determining Frequent Itemsets from Current SFI-forest

Once SFI-forest containing all the frequent items of the data stream generated so far is constructed, we can derive all the frequent itemsets by traversing the SFI-forest according to the *Apriori* principle. Therefore, we propose an efficient mechanism called *top-down frequent itemset selection (todoFIS)*, as shown in Figure 2-8, for mining frequent itemsets. It is especially useful in mining long frequent itemsets. The method is described as follows.

Assume that there are k frequent items, namely e_1, e_2, \dots, e_k , in the current FI-list, and each item $e_i, \forall i = 1, 2, \dots, k$, has an associated e_i .OFI-list, where the size of e_i .OFI-list is denoted by $|e_i$.OFI-list|. Note that the items, namely o_1, o_2, \dots, o_j , within the e_i .OFI-list are denoted by $e_i.o_1, e_i.o_2, \dots, e_i.o_j$, respectively, where the value j equals to $|e_i$.OFI-list|. For each entry $e_i, \forall i = 1, 2, \dots, k$, in the current FI-list, DSM-FI algorithm first generates a maximal candidate itemset with $(j+1)$ items, i.e., $(e_i e_i.o_1 e_i.o_2 \dots e_i.o_j)$ by combining the item-prefix e_i with all frequent items in e_i .OFI-list. Then, DSM-FI uses the following scheme to count its estimated support.

First, we start with a specific frequent item $e_i.o_l (1 \leq l \leq j)$, whose estimated support is smallest, and traverse the paths containing $e_i.o_l$ via node-links of e_i .SFI-tree to count the estimated support of the candidate $(e_i e_i.o_1 e_i.o_2 \dots e_i.o_j)$. If the estimated support of the candidate is greater than or equal to $(s-\epsilon) \cdot e_i.CL$, then it is a frequent itemset. All subsets of this frequent itemset are also frequent itemsets according to the *Apriori* principle. Hence, the complete set of the frequent itemsets stored in the e_i .SFI-tree can be generated by enumeration of all the combinations of the subsets of frequent $(j+1)$ -itemset, $(e_i e_i.o_1 e_i.o_2 \dots e_i.o_j)$. On the other hand, if the estimated support of the candidate $(j+1)$ -itemset is less than the threshold $(s-\epsilon) \cdot e_i.CL$, then it is not a frequent itemset. Now, we need to use the same mechanism to test all the subsets of the $(j+1)$ -itemset until the candidate 3-itemsets. This is because all frequent 2-itemsets can be generated by combining the item e_i and the frequent items of the e_i .OFI-list.

Note that a $(j+1)$ -itemset can be decomposed into $C(j+1, j)$ j -itemsets. We decompose one candidate j -itemset from the $(j+1)$ -itemset at a time, and use the same scheme described above to count the estimated support of this candidate j -itemset. Finally, all the maximal frequent itemsets are maintained in a temporal MFI-list, called MFI_{temp} -list, for efficient generation of the set of all frequent itemsets. If such a MFI_{temp} -list is obtained, all the frequent itemsets can be generated efficiently by enumerating the set of all maximal frequent itemsets in the current MFI_{temp} -list without any candidate generation and support counting. Note that if the user request is just to find the set of all *maximal frequent itemsets* so far, DSM-FI algorithm can output all maximal frequent itemsets efficiently by scanning the MFI_{temp} -list.

Example 2-3. Let the minimum support threshold s be 0.5. Therefore, an itemset X is *frequent* in Figure 2-7 if $X.esup \geq s \cdot X.CL$. Note that $s \cdot X.CL = 0.5 \cdot 6 = 3$ in this case. The online mining steps of DSM-FI algorithm are described as follows.

- (1) First of all, DSM-FI starts the frequent itemset mining scheme from the first frequent item a (from left to right). At this moment, only item a is a frequent itemset, since the estimated support of items c , d , e , and f in the $a.OFI$ -list are less than $s \cdot a.CL$, where $s \cdot a.CL = 3$. Now, DSM-FI stores the maximal frequent 1-itemset (a) into the MFI_{temp} -list.
- (2) Next, DSM-FI starts on the second entry c for frequent itemset mining. DSM-FI generates a candidate maximal 3-itemset (cef), and traverses the $c.SFI$ -tree to count its estimated support. As a result, the candidate (cef) is a maximal frequent itemset, since its estimated support is 3 and it is not a subset of any other frequent itemsets in the MFI_{temp} -list. Now, DSM-FI stores the maximal frequent itemset (cef) into the MFI_{temp} -list.
- (3) Next, DSM-FI starts on the third entry d and generates a candidate maximal 2-itemset (df). DSM-FI stores the itemset (df) into the MFI_{temp} -list without traversing $d.SFI$ -tree because (df) is a frequent 2-itemset and is not a subset of any other maximal frequent itemsets stored in the MFI_{temp} -list.

(4) On the fourth entry f , DSM-FI algorithm generates one frequent 1-itemset (f) directly, since the f .OFI-list is empty. DSM-FI does not store it into the MFI_{temp} -list, because (f) is a subset of a generated maximal frequent itemset (cef).

Finally, on the fifth entry e , DSM-FI generates a frequent 2-itemset (ef) directly. However, the frequent 2-itemset (ef) is a subset of a maximal frequent itemset (cef) stored in the MFI_{temp} -list. DSM-FI algorithm does not store it into the MFI_{temp} -list.

Algorithm todoFIS

Input: A current SFI-forest, the current window identifier N , a minimum support threshold s , and a maximum support error threshold ϵ .

Output: A set of all frequent itemsets.

```

1:   $MFI_{temp}$ -list =  $\emptyset$ ;
    /*  $MFI_{temp}$ -list is a temporary list used to store the set of maximal frequent itemsets */
2:  foreach entry  $e$  in the current FI-list do
3:      construct a maximal candidate itemset  $E$  with size  $|E|$  /*  $|E| = 1 + |e.OFI-list|$  */
4:      count  $E.esup$  by traversing the  $e$ .SFI-tree;
5:      if  $E.esup \geq (s - \epsilon) \cdot E.CL$  then
6:          if  $E \notin MFI_{temp}$ -list and  $E$  is not a subset of any other patterns in the
               $MFI_{temp}$ -list
          then
7:              add  $E$  into the  $MFI_{temp}$ -list;
8:              remove  $E$ 's subsets from the  $MFI_{temp}$ -list;
9:          end if
10:         else /* if  $E$  is not a frequent itemset */
11:             enumerate  $E$  into itemsets with size  $|E| - 1$ ;
12:         end if
13:     until todoFIS finds the set of all frequent itemsets with respect to entry  $e$ ;
14: end for

```

Figure 2- 8. Algorithm todoFIS

After processing all the entries in the FI-list, the MFI_{temp} -list generated by DSM-FI algorithm contains the set of current maximal frequent itemsets: $\{(a), (cef), (df)\}$. Therefore, the set of all frequent itemsets can be generated by enumerating the set: $\{(a), (cef), (df)\}$. Consequently, the set of all frequent itemsets in Figure 2-7 are $\{(a), (cef), (ce), (cf), (ef), (c), (e), (f), (df), (d)\}$.

2.4 Theoretical Analysis

In this section, we discuss the upper bound of estimated support error of frequent itemsets generated by DSM-FI algorithm, and the space upper bound of prefix-tree-based summary data structure.

2.4.1 Maximum Estimated Support Error Analysis

In this section, we discuss the maximum estimated support error of all frequent itemsets generated by DSM-FI algorithm. Let $X.wid$ be the *window-id* of itemset X stored in the current SFI-forest. Assume that the window contains k transactions. Let the maximum support error threshold be ϵ . Let the current *window-id* of the incoming stream be $wid(N)$. Now, we have the following theorem of *maximum estimated support error guarantee* of frequent itemsets generated by the proposed algorithm.

Theorem 2-1 $X.tsup - X.esup \leq \epsilon \cdot (X.wid - 1) \cdot k$.

Proof: We prove by induction. Base case ($X.wid = 1$): $X.tsup = X.esup$. Thus, $X.tsup - X.esup \leq \epsilon \cdot (X.wid - 1) \cdot k$.

Induction step: Consider an itemset of the form $(X, X.esup, X.wid)$ that gets deleted for some $wid(N) > 1$. The itemset is inserted in the SFI-forest when $wid(N+1)$ is being processed. The itemset X whose *window-id* is $wid(N+1)$ in the FI-list could possibly have been deleted as

late as the time when $X.esup \leq \varepsilon \cdot (wid(N+1) - X.wid+1) \cdot k$. Therefore, $X.tsup$ of X when that deletion occurred is no more than $\varepsilon \cdot (wid(N+1) - X.wid+1) \cdot k$. Furthermore, $X.esup$ is the estimated true support of the itemset X since it is inserted. It follows that $X.tsup$, which is the true support of X in the first window containing X though the current window, is at most $X.esup + \varepsilon \cdot (wid(N) - 1) \cdot k$. Thus, we have $X.tsup - X.esup \leq \varepsilon \cdot (X.wid - 1) \cdot k$. As a result, DSM-FI generates *no false negative*.

□

Because our algorithm is a *false-positive* algorithm, the answers produced by DSM-FI will have the following guarantees the same as those of BTS algorithm [53]:

- (a) All itemsets whose true frequency exceeds $s \cdot N$ are output. There are no false negatives.
- (b) No itemsets whose true frequency is less than $(s - \varepsilon) \cdot N$ is output.
- (c) Estimated frequencies are less than the true frequencies by at most εN .

If it is desired that the error dose not increase linearly with the value of window id, we can modify the line 5 of algorithm todoFIS from “**if** $E.esup \geq (s - \varepsilon) \cdot N$ **then**” to “**if** $E.esup \geq s \cdot N$ **then**”. After that DSM-FI algorithm becomes a *false-negative* algorithm.

Note that a *false-positive* approach returns a set of itemsets that includes all frequent itemsets but also some infrequent itemsets. A *false-negative* algorithm returns a set of itemsets that does not include any infrequent itemsets but misses some frequent itemsets.

2.4.2 Space Requirement Analysis

In this section, we discuss the space upper bound of any single-pass algorithm for constructing a summary data structure based on a prefix tree structure.

Theorem 2-2. A prefix tree-based summary data structure has at most 2^m nodes for storing the set of all frequent itemsets of data streams, when m frequent items are given.

Proof: Let m be the number of frequent items, i.e., 1-itemsets, in the data stream generated so far. Hence, the number of potential frequent itemsets is $C(m, 1)$ regarding one item, $C(m, 2)$ regarding two items, ..., $C(m, i)$ regarding i items, ..., and $C(m, m)$ regarding m items according to the *Apriori* heuristic. In a prefix tree-based summary data structure, an itemset is represented by a path and its appearance support is maintained in the last node of the path. Thus, there are $C(m, 1)$ nodes in the first level, $C(m, 2)$ nodes in the second level, ..., $C(m, i)$ nodes in the i -th level, ..., and $C(m, m)$ nodes in the m -th level. There are totally $C(m, 1) + C(m, 2) + \dots + C(m, i) + \dots + C(m, m)$ nodes in the prefix tree-based summary data structure. Consequently, the space upper bound of a prefix-tree based summary data structure is $O(2^m)$. □

The construction cost of summary data structure of DSM-FI algorithm is extremely less than that of BTS algorithm although theoretically, their worst case space complexities are same, i.e., $O(2^m)$, when m frequent items are given.

2.5 Performance Evaluation

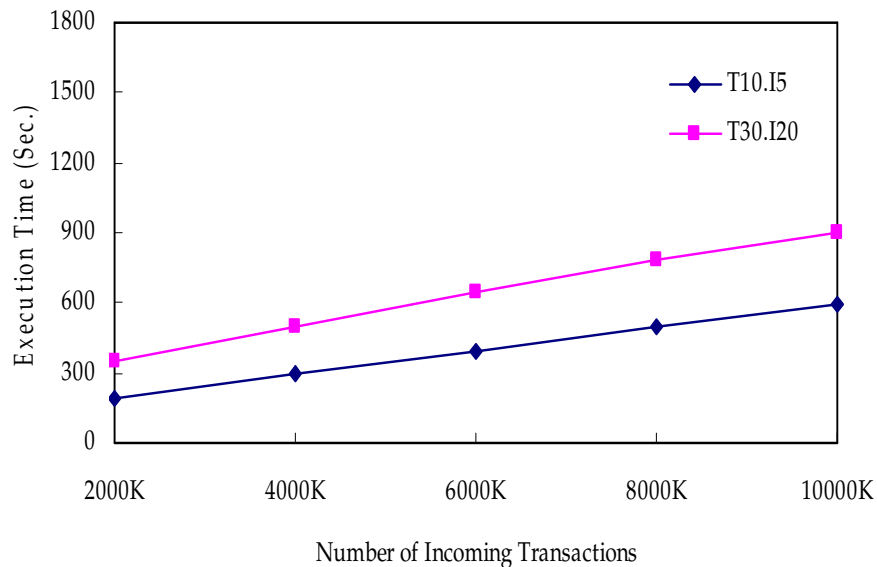
All the experiments are performed on a 1GHz IBM X24 with 384MB, and the program is written in Microsoft Visual C++ 6.0. To evaluate the performance of algorithm DSM-FI, we conduct the empirical studies based on the synthetic datasets. In Section 2.6.1, we report the scalability study of algorithm DSM-FI. In Section 2.6.2, we compare the memory and execution time requested by DSM-FI with BTS algorithm. The parameters of synthetic data generated by IBM synthetic data generator [3] are described as follows.

IBM Synthetic Dataset: $T10.I5.D1M$ and $T30.I20.D1M$. The first synthetic dataset $T10.I5$ has average transaction size T of 10 items and the average size of maximal frequent itemset I is 5-items. It is a *sparse* dataset. In the second dataset $T30.I20$, the average transaction size T

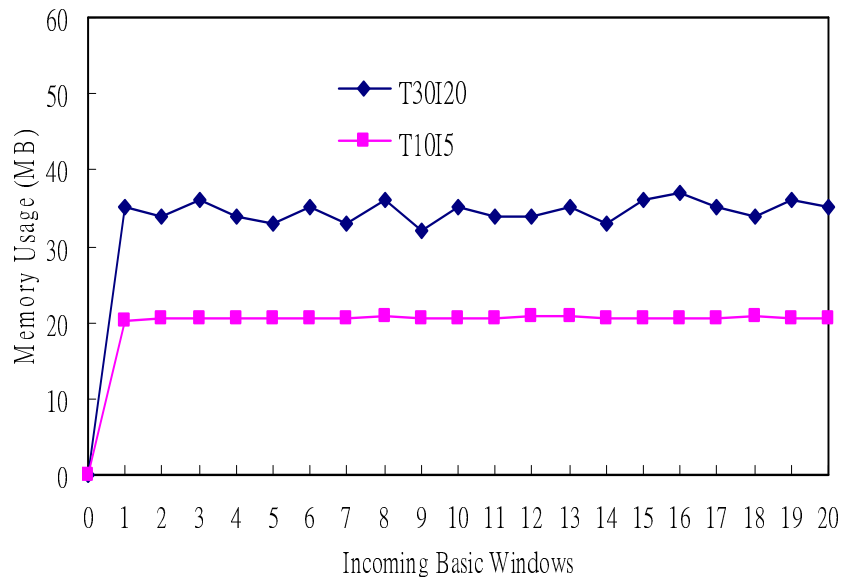
and average size of maximal frequent itemset I are set to 30 and 20, respectively. It is a *dense* dataset. Both synthetic datasets have 1,000,000 transactions. Items were drawn from a universe of 10K distinct items. In the experiments, the synthetic data stream is broken into basic windows of size 50K (i.e., 50,000 transactions) for simulating the continuous characteristic of streaming data. Hence, there are total 20 windows in these experiments.

2.5.1 Scalability Study of DSM-FI Algorithm

In this experiment, we examine the two primary factors, *execution time* and *memory usage*, to discover frequent itemsets in a data stream environment, since both should be bounded online as time advances. Therefore, in Figure 2-9(a), the execution time grows smoothly as the dataset size increases from 2,000K to 10,000K. The default value of minimum support threshold s is 0.01%. The memory usage in Figure 2-9(b) for both synthetic datasets is stable as time progresses, indicating the scalability and feasibility of algorithm DSM-FI. Notice that, the synthetic data stream used in Figure 2-9(b) is divided into 20 basic windows each of 50K.

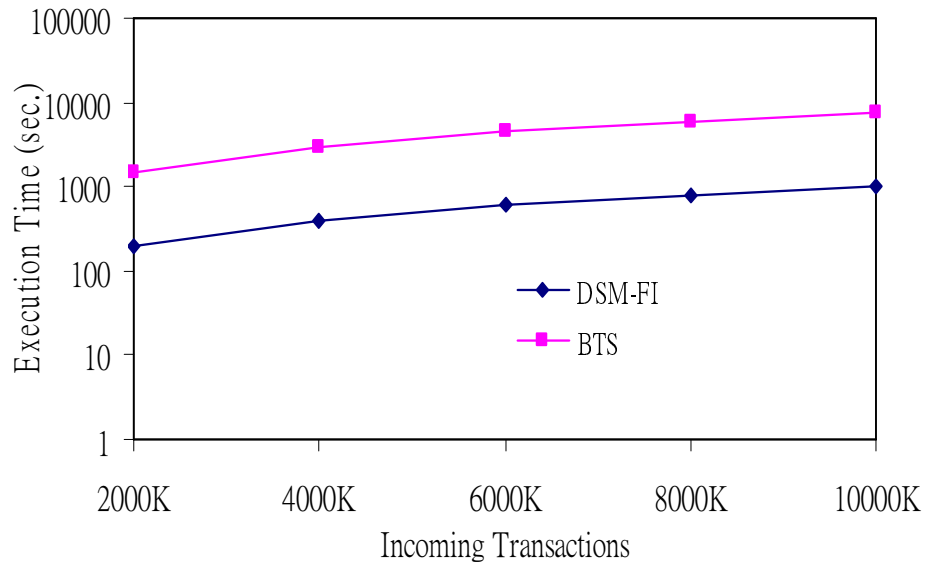


(a) Linear scalability of DSM-FI algorithm ($s = 0.01\%$)

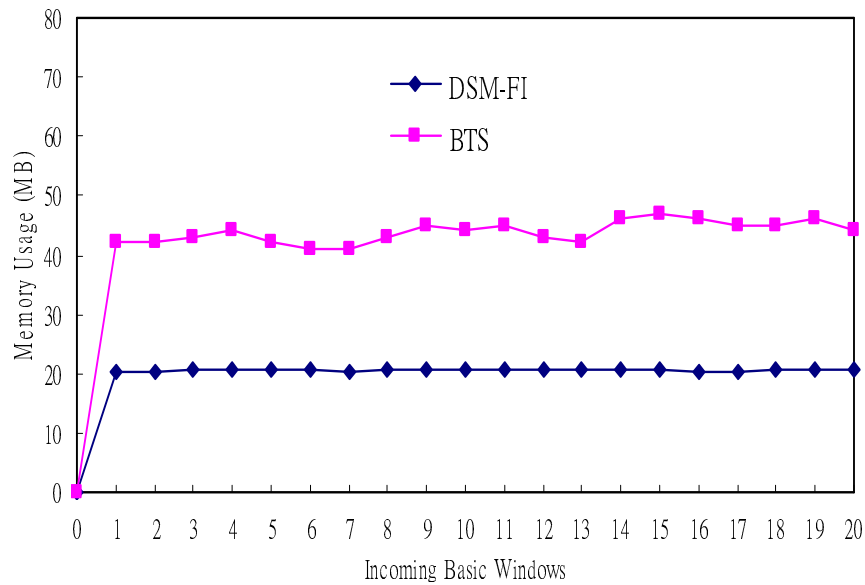


(b) Space requirement of DSM-FI algorithm ($s = 0.01\%$)

Figure 2- 9. Resource requirements of DSM-FI algorithm for IBM synthetic datasets: (a) execution time, (b) memory usage

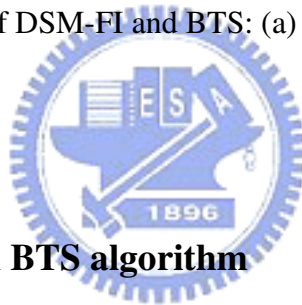


(a) Execution time compared with DSM-FI and BTS ($s = 0.01\%$)



(b) Space requirement compared with DSM-FI and BTS ($s = 0.01\%$)

Figure 2- 10. Comparison of DSM-FI and BTS: (a) Execution time, (b) Memory Usage



2.5.2 Comparison with BTS algorithm

In this experiment, we examine the execution time and memory usage between DSM-FI and BTS by dataset *T30.I20.D1M*. In Figure 2-10 (a), we can see that the execution time incurred by DSM-FI is quite steady and is less than that of BTS. The experiment shows that DSM-FI performs more efficiently than BTS algorithm. In Figure 2-10 (b), the memory usage of DSM-FI is more stable and extremely less than that of BTS. It also shows that DSM-FI algorithm is more suitable for mining frequent itemsets in large-scale data streams.

2.6 Conclusions

In this chapter, we proposed a new, single-pass algorithm, called DSM-FI (Data Stream Mining for Frequent Itemsets), which mines the set of all frequent itemsets in the landmark

model of data streams. In the DSM-FI algorithm, a new in-memory summary data structure, called SFI-forest (Summary Frequent Itemset forest), is constructed for storing the frequent and significant itemsets of the streaming data generated so far. An efficient frequent itemset search mechanism, called todoFIS (top-down Frequent Itemset Selection), is developed to find the set of all frequent itemsets from the current SFI-forest. Experiments tested on synthetic data streams show that DSM-FI is efficient on both sparse and dense datasets, and demonstrates linear scalability to very long data streams. Moreover, DSM-FI outperforms the well-known, single-pass algorithm - BTS - for mining frequent itemsets over the entire history of the streaming data.



Chapter 3 Online Mining of Frequent Itemsets over Stream Sliding Windows

Many previous studies contributed to the efficient mining of frequent items [12, 20, 22, 39, 40, 54] and frequent itemsets (FI) in streaming data [10, 11, 18, 21, 31, 51, 53, 63, 64, 68]. According to the stream processing model [70], the research of mining frequent itemsets in data streams can be divided into three categories: *landmark windows*, *sliding windows*, and *damped windows*, as described briefly as follows. In the landmark window model, knowledge discovery is performed based on the values between a specific timestamp called landmark and the present. In the sliding window model, knowledge discovery is performed over a fixed number of recently generated data elements which is the target of data mining. Two types of sliding window, i.e., *transaction-sensitive sliding window* (TransSW) and *time-sensitive sliding window* (TimeSW), are used in mining data streams. The basic processing unit of window sliding of first type is an expired transaction while the basic unit of window sliding of second type is a time unit, such as a minute or an hour. The sliding windows are shown in Figure 3-1. In the damped window model, recent sliding windows are more important than previous ones.

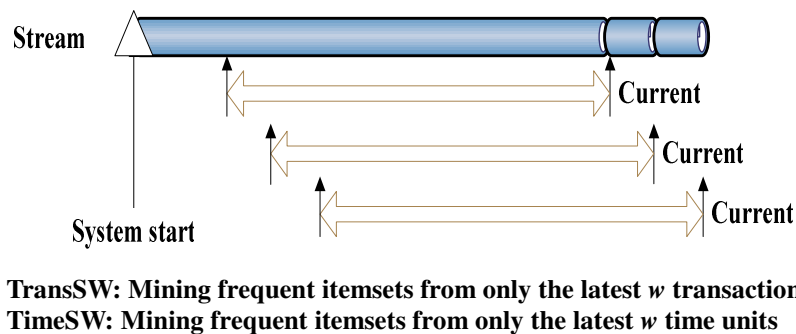


Figure 3- 1. Transaction-sensitive sliding window and time-sensitive sliding window [51]

3.1 Introduction

In [53], Manku and Motwani developed two single-pass algorithms, Sticky-Sampling and Lossy Counting, to mine frequent items over *offline* data streams with a landmark window. Moreover, Manku and Motwani proposed the BTS (Buffer- Trie-SetGen) algorithm based on Lossy Counting for mining the set of frequent itemsets from *offline* data streams. Jin and Agrawal [39] proposed an algorithm, called StreamMining, for in-core frequent itemset mining over *online* data streams. Yu et al. [68] discussed the issues of false negative or false positive in mining frequent itemsets from high speed *offline* transactional data streams.

Chang and Lee [11] proposed a BTS-based algorithm, called SWFI-stream, for mining frequent itemsets in *online* data streams with a *transaction-sensitive* sliding windows model. Teng et al. [63] proposed a regression-based algorithm, called FTP-DS, to find temporal patterns (frequent inter-transaction itemsets) across *multiple online* data streams in a time-sensitive sliding window. Teng et al. [64] proposed a resource-aware algorithm called RAM-DS, to mine temporal patterns in *multiple online* data streams with a time-sensitive sliding window. Lin et al. [14] proposed an incremental mining algorithm to find the set of frequent itemsets in *offline* data streams with a *time-sensitive* sliding window. Chi et al. [18] proposed a *transaction-sensitive* sliding window based algorithm, called MOMENT, which might be the first to find *frequent closed itemsets* (FCI) from *online* data streams with a transaction- sensitive sliding window. A summary data structure, called CET, is used in the MOMENT algorithm to maintain the information of closed frequent itemsets.

Chang and Lee [10] developed a damped window based algorithm, called estDec, for mining frequent itemsets in *online* streaming data in which each transaction has a weight decreasing with age. In other words, older transactions contribute less toward itemset frequencies, and it is a kind of damped windows model. Giannella et al. [31] proposed a frequent pattern tree (abbreviated as FP-tree) [35] based algorithm, called FP-stream, to mine

frequent itemsets at multiple time granularities by a novel tilted-time windows technique. FP-stream focuses on *offline* data streams.

The first target of this chapter is on frequent itemsets mining over online data streams with a *transaction-sensitive* sliding window. An efficient algorithm, called MFI-TransSW (Mining Frequent Itemsets over Transaction-sensitive Sliding Windows), is proposed to mine frequent itemsets over *online* data streams with a *transaction-sensitive* sliding window. The experiments show that the MFI-TransSW algorithm not only attain highly accurate mining results, but also run significant faster and consume less memory than that of SWFI-stream algorithm [11] for mining frequent itemsets over recent data streams. The second purpose of the chapter is to mine frequent itemsets over online data streams with a *time-sensitive* sliding window. A MFI-TransSW based algorithm, called MFI-TimeSW (Mining Frequent Itemsets over Time-sensitive Sliding Windows), is developed for mining frequent itemsets over *online* data streams with a *time-sensitive* sliding window.

The remainder of this chapter is organized as follows. The problem of frequent itemsets mining in a transaction-sensitive sliding window is defined in Section 3.2. The algorithm MFI-TransSW is proposed in Section 3.3. Experiments of MFI-TransSW algorithm are discussed in Section 3.4. The issue of mining in a time-sensitive sliding window is defined and algorithm MFI-TimeSW is proposed in Section 3.5 and Section 3.6, respectively. Finally, we conclude this chapter in Section 3.7.

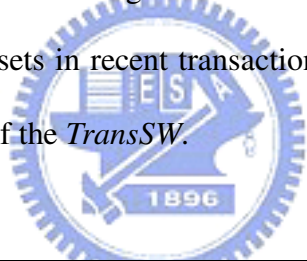
3.2 Problem Definition: Mining Frequent Itemsets in a TransSW

Let $\Psi = \{i_1, i_2, \dots, i_m\}$ be a set of **items**. A **transaction** $T = (tid, x_1x_2 \cdots x_n)$, $x_i \in \Psi$, for $1 \leq i \leq n$, is a set of items, while n is called the **size** of the transaction, and tid is the unique identifier of the transaction. An **itemset** is a non-empty set of items. An itemset with size k is called a *k-itemset*. A **transaction data stream** $TDS = T_1, T_2, \dots, T_N$ is a continuous sequence of

transactions, where N is the tid of latest incoming transaction T_N .

A **transaction-sensitive sliding window** ($TransSW$) in the transaction data stream is a window that slides forward for every transaction. The window at each slide has a fixed number, w , of transactions, and w is called the *size* of the window. Hence, the *current transaction-sensitive sliding window* is $TransSW_{N-w+1} = [T_{N-w+1}, T_{N-w+2}, \dots, T_N]$, where $N-w+1$ is the window id of current $TransSW$. The **support** of an itemset X over $TransSW$, denoted as $\mathbf{sup}(X)^{TransSW}$, is the number of transactions in $TransSW$ containing X as a *subset*. An itemset X is called a **frequent itemset** (FI) if $\mathbf{sup}(X)^{TransSW} \geq s \cdot w$, where s is a user-defined minimum support threshold (MST) in the range of $[0, 1]$. The value $s \cdot w$ is called the **frequent threshold** of $TransSW$ ($FT^{TransSW}$).

Given a transaction-sensitive sliding window $TransSW$, and a MST s , the problem of online mining of frequent itemsets in recent transaction data streams is to mine the set of all frequent itemsets by *one scan* of the $TransSW$.



Transaction Data Stream	FIs in $TransSW_1$	FIs in $TransSW_2$
$\langle T_1, (acd) \rangle$	$(a), (b), (c), (e), (ac),$	$(b), (c), (e), (bc), (be),$
$\langle T_2, (bce) \rangle$	$(bc), (be), (ce), (bce)$	$(ce), (bce)$
$\langle T_3, (abce) \rangle$		
$\langle T_4, (be) \rangle$		

A transaction data stream is formed by transactions arriving in series

Figure 3- 2. An example transaction data stream and the frequent itemsets over two consecutive $TransSW$ s

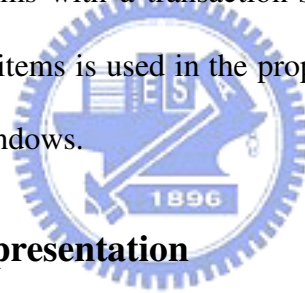
Example 3-1 Let the first four transactions in a transaction data stream be $\langle T_1, (acd) \rangle$, $\langle T_2, (bce) \rangle$, $\langle T_3, (abce) \rangle$, and $\langle T_4, (be) \rangle$, where T_1, T_2, T_3 , and T_4 are transactions and a, b, c, d , and e are items. Let the size of sliding window w be 3 and the user-defined minimum support threshold s be 0.6. Hence, the transaction data stream consists of two transaction-sensitive sliding windows, i.e., $TransSW_1 = [T_1, T_2, T_3]$ and $TransSW_2 = [T_2, T_3, T_4]$, where first

window $TransSW_1$ contains the transactions T_1 , T_2 , and T_3 , and the second window $TransSW_2$ contains the transactions T_2 , T_3 , and T_4 . The example is shown in Figure 3-2.

In Figure 3-2, the frequent itemsets in $TransSW_1$ are (a) , (b) , (c) , (e) , (ac) , (bc) , (be) , (ce) , and (bce) , and the frequent itemsets in $TransSW_2$ are (b) , (c) , (e) , (d) , (bc) , (be) , (ce) , and (bce) . In this instance, we can find that itemsets (a) and (ac) are frequent itemsets in $TransSW_1$, but not frequent ones in $TransSW_2$.

3.3 The Proposed Algorithm: MFI-TransSW

In this section, we proposed an efficient single-pass algorithm, called **MFI-TransSW** (Mining Frequent Itemsets over a Transaction-sensitive Sliding Window), to mine the set of all frequent itemsets in data streams with a transaction-sensitive sliding window. An effective bit-sequence representation of items is used in the proposed algorithm to reduce the time and memory needed to slide the windows.



3.3.1 Bit-Sequence Representation

In the proposed *MFI-TransSW* algorithm, for each item X in the current transaction-sensitive sliding window $TransSW$, a *bit-sequence* with w bits, denoted as $\mathbf{Bit}(X)$, is constructed. If an item X is in the i -th transaction of current $TransSW$, the i -th bit of $\mathbf{Bit}(X)$ is set to be 1; otherwise, it is set to be 0. The process is called *bit-sequence transform*.

For example, in Figure 3-2, the first sliding window $TransSW_1$ consists of three transactions: $\langle T_1, (acd) \rangle$, $\langle T_2, (bce) \rangle$, and $\langle T_3, (abce) \rangle$, but the $TransSW_2$ consists of transactions: $\langle T_2, (bce) \rangle$, $\langle T_3, (abce) \rangle$, and $\langle T_4, (be) \rangle$. Because item a appears in the 1st and 3rd transactions of $TransSW_1$, the bit-sequence of a , $\mathbf{Bit}(a)$, is 101. Similarly, $\mathbf{Bit}(b) = 011$, $\mathbf{Bit}(c) = 111$, $\mathbf{Bit}(d) = 100$, and $\mathbf{Bit}(e) = 011$.

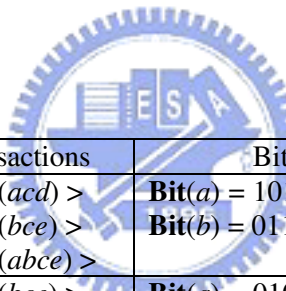
3.3.2 The MFI-TransSW Algorithm

MFI-TransSW algorithm consists of three phases, *window initialization* phase, *window sliding* phase, and *frequent itemsets generation* phase.

3.3.2.1 Window Initialization Phase

The phase is activated while the number of transactions generated so far in a transaction data stream is less than or equal to a user-predefined sliding window size w . In this phase, each item in the new incoming transaction is transformed into its bit-sequence representation.

For instance, in Figure 3-3, the first sliding window $TransSW_1$ contains three transactions: T_1 , T_2 , and T_3 . The bit-sequences of items of $TransSW_1$ in the window initialization phase are shown in Figure 3-4.



Window-id	Transactions	Bit-Sequences of items
$TransSW_1$	$\langle T_1, (acd) \rangle$ $\langle T_2, (bce) \rangle$ $\langle T_3, (abce) \rangle$	Bit(a) = 101, Bit(c) = 111, Bit(d) = 100, Bit(b) = 011, Bit(e) = 011
$TransSW_2$	$\langle T_2, (bce) \rangle$ $\langle T_3, (abce) \rangle$ $\langle T_4, (be) \rangle$	Bit(a) = 010, Bit(c) = 110, Bit(d) = 000, Bit(b) = 111, Bit(e) = 111

Figure 3- 3. Bit-sequences of items in window initialization phase of TransSW

tid	Items	Bit-Sequences in current $TransSW_1$
T_1	(acd)	Bit(a)=100, Bit(c)=100, Bit(d)=100
T_2	(bce)	Bit(a)=100, Bit(c)=110, Bit(d)=100, Bit(b)=010, Bit(e)=010
T_3	$(abce)$	Bit(a)=101, Bit(c)=111, Bit(d)=100, Bit(b)=011, Bit(e)=011

Figure 3- 4. Bit-sequences of items after sliding $TransSW_1$ to $TransSW_2$

3.3.2.2 Window Sliding Phase

The phase is activated after the sliding window *TransSW* becomes full. A new incoming transaction is appended to the sliding window, and the oldest transaction is removed from the window.

For removing oldest information, an efficient method is used in the proposed algorithm. Based on the bit-sequence representation, MFI-TransSW algorithm uses the *bitwise left shift* operation to remove the aged transaction from the set of items in the current sliding window. After sliding the window, an effective pruning method, called *Item-Prune*, is used to improve the memory usage. The pruning approach is that *an item X in the current transaction-sensitive sliding window is dropped if $\text{sup}(X)^{\text{TransSW}} = 0$* .

For example, in Figure 3-2, before the fourth transaction $\langle T_4, (be) \rangle$ is processed, the first transaction T_1 must be removed from the current window using bitwise left shift on the set of items. Hence, **Bit(a)** is modified from 101 to 010. Similarly, **Bit(c)**=110, **Bit(d)**=000, **Bit(b)**=110, and **Bit(e)**=110. Then, the new transaction $\langle T_4, (be) \rangle$ is processed by bit-sequence transform. The result is shown in Figure 3-4. Note that item *d* is dropped since **Bit(d)**=000, i.e., $\text{sup}(d)^{\text{TransSW}} = 0$.

Algorithm MFI-TransSW

Input: *TDS* (a transaction data stream), *s* (a user-defined minimum support threshold in the range of [0, 1]), and *w* (the user-specified sliding window size).

Output: a set of frequent itemsets, *FI-Output*.

Begin

TransSW = NULL; /* TransSW consists of *w* transactions */

Repeat:

for each incoming transaction T_i in TransSW **do**


```

if TransSW = FULL then
    Do bitwise-shift on bit-sequences of all items in TransSW;
else
    for each item  $X$  in  $T_i$  do
        Do bit-sequence transform( $X$ );
    end for
end if
end for
for each bit-sequence  $\mathbf{Bit}(X)$  in TransSW do
    if  $\text{sup}(X) = 0$  then
        Drop  $X$  from TransSW;
    end if
end for
/* The following is the frequent itemsets generation phase. The phase is performed only when
    requested by users. */
 $\mathbf{FI}_1 = \{\text{frequent 1-itemsets}\};$ 
for ( $k=2$ ;  $\mathbf{FI}_{k-1} \neq \text{NULL}$ ;  $k++$ ) do
     $\mathbf{CI}_k = \text{CIGA}(\mathbf{FI}_{k-1});$ 
    Do bitwise AND to find the supports of  $\mathbf{CI}_k$ ;
    for each candidate  $c_k \in \mathbf{CI}_k$  do
        if  $\text{sup}(c_k)^{\text{TransSW}} \geq w \cdot s$  then
             $\mathbf{FI}_k = \{c_k \in \mathbf{CI}_k \mid \text{sup}(c_k)^{\text{TransSW}} \geq w \cdot s\};$ 
        end if
    end for
end for
 $\mathbf{FI}\text{-Output} = \bigcup_k \mathbf{FI}_k;$ 
End

```

Figure 3- 5. Algorithm MFI-TransSW

Transactions in TransSW ₂	Bit-Sequences in TransSW ₂	FI ₁ in TransSW ₂ ($s = 0.6$)	sup
$\langle T_2, (bce) \rangle$	Bit(a) = 010	$\{(b) \mid \mathbf{Bit}(b) = 111\}$	3
$\langle T_3, (abce) \rangle$	Bit(c) = 110	$\{(c) \mid \mathbf{Bit}(c) = 110\}$	2
$\langle T_4, (be) \rangle$	Bit(b) = 111	$\{(e) \mid \mathbf{Bit}(e) = 111\}$	3
	Bit(e) = 111		

CI ₂ in SW ₂	FI ₂ in TransSW ₂	sup
$\{(bc) \mid \mathbf{Bit}(b) = 111 \text{ AND } \mathbf{Bit}(c) = 110\}$	$\{(bc) \mid \mathbf{Bit}(bc) = 110\}$	2
$\{(be) \mid \mathbf{Bit}(b) = 111 \text{ AND } \mathbf{Bit}(e) = 111\}$	$\{(be) \mid \mathbf{Bit}(be) = 111\}$	3
$\{(ce) \mid \mathbf{Bit}(c) = 110 \text{ AND } \mathbf{Bit}(e) = 111\}$	$\{(ce) \mid \mathbf{Bit}(ce) = 110\}$	2

CI ₃ in TransSW ₂	FI ₃ in TransSW ₂	sup
$\{(bce) \mid \mathbf{Bit}(bc) = 110 \text{ AND } \mathbf{Bit}(be) = 111 \text{ AND } \mathbf{Bit}(ce) = 110\}$	$\{(bce) \mid \mathbf{Bit}(bce) = 110\}$	2

Figure 3- 6. Steps of frequent itemsets generation in $TransSW_2$

3.3.2.3 Frequent Itemsets Generation Phase

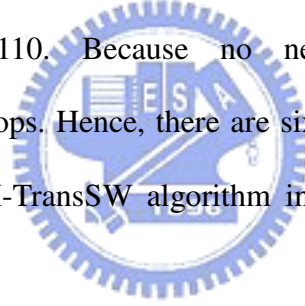
The phase is performed only when the up-to-date set of frequent itemsets is requested. In this phase, MFI-TransSW algorithm uses a level-wise method to generate the set of candidate itemsets CI_k (candidate itemsets with k items) from the pre-known frequent itemsets FI_{k-1} (frequent itemsets with $k-1$ items) according to the *Apriori* property [3]¹. The step is called CIGA (Candidate Itemset Generation using Apriori property). Then, the proposed algorithm uses the *bitwise AND* operation to compute the support (the number of bit 1) of these candidates in order to find the frequent k -itemsets FI_k . The candidate-generation-then-testing process stops when no new candidates with $k+1$ items (CI_{k+1}) are generated. The MFI-TransSW algorithm is shown in Figure 3-5.

For instance, consider the bit-sequences of $TransSW_2$ in Figure 3-4, and let the minimum

¹ It is a **downward closure property**, i.e., *if a pattern is frequent, all of its sub-patterns will also be frequent.*

support threshold s be 0.6. Hence, an itemset X is *frequent* if $\mathbf{sup}(X)^{TransSW} \geq 0.6 \cdot 3 = 1.8$. In the following, we discuss the step of frequent itemset mining of $TransSW_2$. The generated patterns are shown in Figure 3-2.

First, MFI-TransSW algorithm generates three candidate 2-itemsets, (bc) , (be) and (ce) , by combining frequent 1-itemsets: (b) , (c) and (e) , where $\mathbf{Bit}(b) = 111$, i.e., $\mathbf{sup}(b) = 3$, $\mathbf{Bit}(c) = 110$, i.e., $\mathbf{sup}(c) = 2$, and $\mathbf{Bit}(e) = 110$, i.e., $\mathbf{sup}(e) = 2$. 1-itemset (a) is an *infrequent* itemset, since its $\mathbf{Bit}(a) = 010$, i.e., $\mathbf{sup}(a) = 1$. All other candidates are frequent itemsets after using bitwise AND operations to count the supports of these candidates. Because the $\mathbf{Bit}(bc)$ is 110, the support of candidate 2-itemset bc are 2, i.e., $\mathbf{sup}(bc) = 2$. Similarity, $\mathbf{sup}(be) = 3$, and $\mathbf{sup}(ce) = 2$. Second, MFI-TransSW generates one candidate 3-itemset (bce) according to Apriori property and uses bitwise AND operation to count the $\mathbf{sup}(bce) = 2$, i.e., $\mathbf{Bit}(bc)$ AND $\mathbf{Bit}(be)$ AND $\mathbf{Bit}(ce) = 110$. Because no new candidates are generated, the generation-then-test process stops. Hence, there are six frequent itemsets, (b) , (c) , (bc) , (be) , (ce) , (bce) , generated by MFI-TransSW algorithm in $TransSW_2$. The process is shown in Figure 3-6.



3.4 Problem Definition: Mining Frequent Itemsets in a TimeSW

Let $\Psi = \{i_1, i_2, \dots, i_m\}$ be a set of **items**. An **itemset** is a non-empty set of items. An itemset with size k is called a *k-itemset*. A **transaction data stream** $TDS = T_1, T_2, \dots, T_N$ is a continuous sequence of transactions, where N is the transaction identifier of latest incoming transaction T_N . A **transaction** $T = (TUid, Tid, itemset)$, where $TUid$ is the identifier of the time unit, and Tid is the identifier of the transaction.

A **time-sensitive sliding window** (*TimeSW*) in the transaction data stream is a window that slides forward for every *time unit* (TU). Each time unit TU_i consists of a variable number, $|TU_i|$, of transactions, and $|TU_i|$ is also called the *size* of the time unit. Hence, the *current*

time-sensitive sliding window with w time units is $TimeSW_{N-w+1} = [TU_{N-w+1}, TU_{N-w+2}, \dots, TU_N]$, where $N-w+1$ is the id of time unit of current *TimeSW*, and N is the *TU*id of latest time unit TU_N . The window at each slide has a fixed number, w , of time units. The value $w = |TU_{N-w+1}| + |TU_{N-w+2}| + \dots + |TU_N|$ is called the *size* of the time-sensitive sliding window and denoted as $|TimeSW|$.

The **support** of an itemset X over *TimeSW*, denoted as $\mathbf{sup}(X)^{TimeSW}$, is the number of transactions in *TimeSW* containing X as a *subset*. An itemset X is called a **frequent itemset (FI)** if $\mathbf{sup}(X)^{TimeSW} \geq s \cdot |TimeSW|$, where s is a user-defined minimum support threshold (MST) in the range of $[0, 1]$. The value $s \cdot |TimeSW|$ is called the **frequent threshold** of *TimeSW* (FT^{TimeSW}).

Given a time-sensitive sliding window *TimeSW*, and a MST s , the problem of online mining of frequent itemsets in recent transaction data streams is to mine the set of all frequent itemsets by *one scan* of the *TimeSW*.



Example 3-2 Let the size of the time-sensitive sliding window w be 3 and the user-defined minimum support threshold s be 0.5. Figure 3-7 records the transactions that arrive in the stream in two successive windows, $TimeSW_1 = [T_1, T_2, T_3, T_4, T_5, T_6, T_7]$ and $TimeSW_2 = [T_4, T_5, T_6, T_7, T_8, T_9]$. The first window $TimeSW_1$ contains *seven* transactions and the frequent threshold $FT = 0.6 \cdot 7 = 3.5$. The second window $TimeSW_2$ contains *six* transactions and the $FT = 0.5 \cdot 6 = 3$.

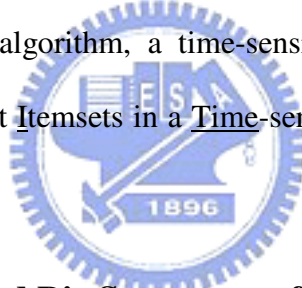
In Figure 3-7, the frequent itemsets in $TimeSW_1$ are (b) , (c) , (e) , (bc) , (be) and (ce) , and the frequent itemsets in $TimeSW_2$ are (a) , (c) , (d) , (e) , (ac) , (ae) and (ce) .

	Transaction Data Stream	FIs in TimeSW ₁	FIs in TimeSW ₂	
TimeSW ₁	$\left\{ \begin{array}{l} \langle TU_1, T_1, (be) \rangle \\ \langle TU_1, T_2, (bce) \rangle \\ \langle TU_1, T_3, (bce) \rangle \\ \langle TU_2, T_4, (acd) \rangle \\ \langle TU_2, T_5, (abce) \rangle \\ \langle TU_3, T_6, (abce) \rangle \\ \langle TU_3, T_7, (ace) \rangle \\ \langle TU_4, T_8, (bcde) \rangle \\ \langle TU_4, T_9, (cde) \rangle \end{array} \right.$	$\left\{ \begin{array}{l} (a), (b), (c), (e), (ac), \\ (bc), (be), (ce), (bce) \end{array} \right.$	$\left\{ \begin{array}{l} (a), (b), (c), (d), \\ (e), (ac), (ae), (ce) \end{array} \right.$	
				TimeSW ₂

Figure 3- 7. An example transaction data stream and the frequent itemsets over two time-sensitive sliding windows

3.5 The Proposed Algorithm: MFI-TimeSW

Based on the MFI-TransSW algorithm, a time-sensitive sliding window-based algorithm *MFI-TimeSW* (Mining Frequent Itemsets in a Time-sensitive Sliding Window) is proposed in this section.



3.5.1 Time Unit List and Bit-Sequences of Items

For mining frequent itemsets over a time-sensitive sliding window, a *time unit list* (*TU-list*) is developed in the MFI-TimeSW algorithm. A TU-list consists of a list of time unit entries, where each time unit entry records the size of the time unit, i.e., $TU\text{-list} = \langle (TU_{id}, |TU_{N-w+1}|), (TU_{id}, |TU_{N-w+2}|), \dots, (TU_{id}, |TU_N|) \rangle$.

The *bit-sequence transform* process of *MFI-TimeSW* algorithm is described as follows. For each item X in the current time-sensitive stream sliding window $TimeSW_{N-w+1}$, a *bit-sequence* with $|TimeSW_{N-w+1}|$ bits, denoted as $\mathbf{Bit}(X)^{TimeSW_{N-w+1}}$, is constructed. If an item X is in the i -th transaction of $TimeSW_{N-w+1}$, the i -th bit of $\mathbf{Bit}(X)^{TimeSW_{N-w+1}}$ is set to be 1; otherwise, it is set to be 0.

For example, in Figure 3-7, the first sliding window $TimeSW_1$ consists of seven transactions: $\langle TU_1, T_1, (be) \rangle$, $\langle TU_1, T_2, (bce) \rangle$, $\langle TU_1, T_3, (bce) \rangle$, $\langle TU_2, T_4, (acd) \rangle$, $\langle TU_2, T_5, (abce) \rangle$, $\langle TU_3, T_6, (abce) \rangle$, and $\langle TU_3, T_7, (ace) \rangle$, but the second window $TimeSW_2$ consists of six transactions: $\langle TU_2, T_4, (acd) \rangle$, $\langle TU_2, T_5, (abce) \rangle$, $\langle TU_3, T_6, (abce) \rangle$, $\langle TU_3, T_7, (ace) \rangle$, $\langle TU_4, T_8, (bce) \rangle$, and $\langle TU_9, T_2, (cde) \rangle$. Because item a appears in the fourth, fifth, sixth and seventh transactions of $TimeSW_1$, the bit-sequence of a , $\mathbf{Bit}(a)^{TimeSW_1}$, is 0001111. Similarly, $\mathbf{Bit}(b)^{TimeSW_1} = 1110110$, $\mathbf{Bit}(c)^{TimeSW_1} = 0111111$, $\mathbf{Bit}(d)^{TimeSW_1} = 0001000$, and $\mathbf{Bit}(e)^{TimeSW_1} = 1110111$. After sliding one time unit of TimeSW, the set of bit-sequences of items is changed, i.e., $\mathbf{Bit}(a)^{TimeSW_2} = 111100$, $\mathbf{Bit}(b)^{TimeSW_2} = 011010$, $\mathbf{Bit}(c)^{TimeSW_2} = 111111$, $\mathbf{Bit}(d)^{TimeSW_2} = 100011$, and $\mathbf{Bit}(e)^{TimeSW_2} = 011111$.

3.5.2 The MFI-TimeSW Algorithm

The MFI-TimeSW algorithm is composed of three phases, *window initialization phase* (phase 1), *window sliding phase* (phase 2), and *frequent itemsets generation phase* (phase 3).

3.5.2.1 Window Initialization Phase

The window initialization phase of MFI-TimeSW algorithm is activated while the number of time units generated so far in a transaction data stream is less than or equal to a user-predefined time-sensitive sliding window size w (i.e., w time units). In this phase, each item X of a new incoming transaction is transformed into its bit-sequence representation $\mathbf{Bit}(X)^{TimeSW}$.

For example, in Figure 3-7, the sliding window $TimeSW_1$ contains seven transactions: T_1 , T_2 , T_3 , T_4 , T_5 , T_6 , and T_7 . The bit-sequence transform of items of $TimeSW_1$ are shown in Figure 3-8.

3.5.2.2 Window Sliding Phase

The window sliding phase of MFI-TimeSW algorithm is activated after the sliding window $TimeSW$ becomes full, i.e., $TimeSW$ contains w time units. A new time unit TU_{N+1} is appended to the time-sensitive sliding window, and the oldest time unit TU_{N-w+1} is removed from the window.

For removing oldest information, an efficient method is used in the proposed algorithm. Based on the bit-sequence representation, MFI-TimeSW algorithm uses the *bitwise left shift* operation to remove the aged time unit from current time-sensitive sliding window. If the aged time unit TU_{N-w+1} contains d transactions, MFI-TimeSW performs d times of bitwise left shift operation on the current sliding window. After sliding the window, an effective pruning method, called *Item-Prune*, is used to improve the memory usage. The pruning approach is that *an item X in the current time-sensitive sliding window is dropped if $sup(X)^{TimeSW} = 0$.*

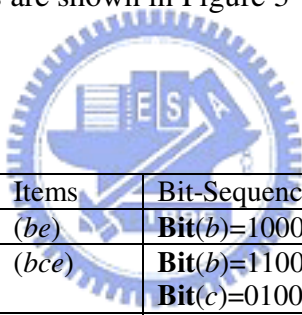
For example, in Figure 3-7, before processing the fourth time unit which consists of two transactions, $\langle TU_4, T_8, (bcde) \rangle$ and $\langle TU_4, T_9, (cde) \rangle$, the first time unit (TU_1) which consists of three transactions (T_1, T_2 , and T_3) must be removed from the current $TimeSW_1$ using bitwise left shift operation on the set of items. Therefore, $\mathbf{Bit}(a)^{TimeSW}$ changes from 0001111 to 1111. Similarly, $\mathbf{Bit}(c)^{TimeSW}$ changes from 0111111 to 1111, $\mathbf{Bit}(d)^{TimeSW}$ changes from 0001000 to 1000, $\mathbf{Bit}(b)^{TimeSW}$ changes from 1110110 to 0110, and $\mathbf{Bit}(e)^{TimeSW}$ changes from 1100111 to 0111. Then, the new time unit (TU_4) is processed by bit-sequence transform. Hence, $\mathbf{Bit}(a)^{TimeSW}$ changes from 1111 to 111100, $\mathbf{Bit}(c)^{TimeSW}$ changes from 1111 to 111111, $\mathbf{Bit}(d)^{TimeSW}$ changes from 1000 to 100011, $\mathbf{Bit}(b)^{TimeSW}$ changes from 0110 to 011010, and $\mathbf{Bit}(e)^{TimeSW}$ changes from 0111 to 011111. The result is shown in Figure 3-9.

3.5.2.3 Frequent Itemsets Generation Phase

The frequent itemsets generation phase of MFI-TimeSW algorithm is also performed only

when requested by users. In this phase, MFI-TimeSW uses the same method CIGA as used in MFI-TransSW algorithm to generate the set of candidate itemsets CI_k (candidate itemsets with k items) from the frequent itemsets FI_{k-1} (frequent itemsets with $k-1$ items). Then, the proposed algorithm uses the *bitwise AND* operation to compute the support (the number of bit 1) of these candidates in order to find the frequent k -itemsets FI_k . The candidate-generation-then-testing process stops when no new candidates with $k+1$ items (CI_{k+1}) are generated. The MFI-TimeSW algorithm is shown in Figure 3-10.

For example, consider the bit-sequences of $TimeSW_1$ in Figure 3-9, and let the minimum support threshold s be 0.5. Therefore, an itemset X is *frequent* in $TimeSW_1$ if $\mathbf{sup}(X)^{TimeSW_1} \geq 0.5 \cdot 7 = 3.5$. In the following, we discuss the steps of frequent itemsets generation of $TimeSW_1$. The generated frequent itemsets are shown in Figure 3-7.



TUid	tid	Items	Bit-Sequences of Items in $TimeSW_1$
1	T_1	(be)	Bit(b)=1000000, Bit(e)=1000000
1	T_2	(bce)	Bit(b)=1100000, Bit(e)=1100000, Bit(c)=0100000
1	T_3	(bce)	Bit(b)=1110000, Bit(e)=1110000, Bit(c)=0110000
2	T_4	(acd)	Bit(b)=1110000, Bit(e)=1100000, Bit(c)=0111000, Bit(a)=0001000, Bit(d)=0001000
2	T_5	(abce)	Bit(b)=1110100, Bit(e)=1100100, Bit(c)=0111100, Bit(a)=0001100, Bit(d)=0001000
3	T_6	(abce)	Bit(b)=1110110, Bit(e)=1100110, Bit(c)=0111110, Bit(a)=0001110, Bit(d)=0001000
3	T_7	(ace)	Bit(b)=1110110, Bit(e)=1100111, Bit(c)=0111111, Bit(a)=0001111, Bit(d)=0001000

Figure 3- 8. Bit-sequences of items in window initialization phase of $TimeSW_1$

Window-id	Transactions	Bit-Sequences of items
TimeSW ₁	$\langle TU_1, T_1, (be) \rangle$ $\langle TU_1, T_2, (bce) \rangle$ $\langle TU_1, T_3, (bce) \rangle$ $\langle TU_2, T_4, (acd) \rangle$ $\langle TU_2, T_5, (abce) \rangle$ $\langle TU_3, T_6, (abce) \rangle$ $\langle TU_3, T_7, (ace) \rangle$	Bit(b) = 1110110 Bit(e) = 1100111 Bit(c) = 0111111 Bit(a) = 0001111 Bit(d) = 0001000
TimeSW ₂	$\langle TU_2, T_4, (acd) \rangle$ $\langle TU_2, T_5, (abce) \rangle$ $\langle TU_3, T_6, (abce) \rangle$ $\langle TU_3, T_7, (ace) \rangle$ $\langle TU_4, T_8, (bcde) \rangle$ $\langle TU_4, T_9, (cde) \rangle$	Bit(b) = 011010 Bit(e) = 011111 Bit(c) = 111111 Bit(a) = 111100 Bit(d) = 100011

Figure 3- 9. Bit-sequences of items after sliding $TimeSW_1$ to $TimeSW_2$

First, MFI-TimeSW algorithm generates candidate 2-itemsets, (ab) , (ac) , (ae) , (bc) , (be) , and (ce) , by combining frequent 1-itemsets, (a) , (b) , (c) and (e) . Only one 1-itemset (d) is an *infrequent* itemset, since its $\mathbf{Bit}(d)^{TimeSW} = 0001000$, i.e., $\mathbf{sup}(d)^{TimeSW_1} = 1$. All these candidates are frequent itemsets after using bitwise AND operations to count the supports (the number of 1) of these candidates. Therefore, the support of 2-itemset (ab) is 2, since $\mathbf{Bit}(ab)^{TimeSW}$ is 0000110. Similarity, $\mathbf{sup}(ac)^{TimeSW_1} = 4$, $\mathbf{sup}(ae)^{TimeSW_1} = 3$, $\mathbf{sup}(bc)^{TimeSW_1} = 4$, $\mathbf{sup}(be)^{TimeSW_1} = 4$ and $\mathbf{sup}(ce)^{TimeSW_1} = 4$. Hence, four frequent 2-itemsets, (ac) , (bc) , (be) , and (ce) , are found.

Algorithm MFI-TimeSW

Input: TDS (a transaction data stream), $TU-list$ (a time unit list), s (a user-defined minimum support threshold in the range of $[0, 1]$), and w (the user-specified sliding window size, i.e., w time units).

Output: a set of frequent itemsets, *FI-Output*.

Begin

```

TimeSW = NULL; /* TimeSW consists of  $w$  time units */
Repeat: /*  $N$  is the id of current time unit*/
  for each new time unit  $TU_N$  from  $TDS$  do /*  $N \geq 1$ */
    if TimeSW = FULL then
      Do  $|TU_{N-w+1}|$  times of bitwise-shift operation on bit-
      sequences of all items in TimeSW;
    else
      for each transaction  $T_i$  of  $TU_N$  do
        for each item  $X$  in  $T_i$  do
          Do bit-sequence transform( $X$ );
        end for
      end for
    end if
  end for
  for each bit-sequence  $Bit(X)$  in TimeSW do
    if  $sup(X) = 0$  then
      Drop  $X$  from TimeSW;
    end if
  end for
   $N = N + 1$ ;

```

/* The following is the frequent itemsets generation phase. The phase is performed only when requested by users. */

```

   $FI_1 = \{ \text{frequent 1-itemsets} \}$ ;
  for ( $k=2$ ;  $FI_{k-1} \neq \text{NULL}$ ;  $k++$ ) do
     $CI_k = CIGA(FI_{k-1})$ ;
    Do bitwise AND to find the supports of  $CI_k$ ;
    for each candidate  $c_k \in CI_k$  do
      if  $sup(c_k)^{TimeSW} \geq |TimeSW| \cdot s$  then
         $FI_k = \{ c_k \in CI_k \mid sup(c_k)^{TimeSW} \geq |TimeSW| \cdot s \}$ ;
      end if
    end for
  end for
   $FI\text{-Output} = \cup_k FI_k$ ;

```

End

Figure 3- 10. Algorithm MFI-TimeSW

Next, two candidate 3-itemsets, (*ace*) and (*bce*), are generated by MFI-TimeSW according to Apriori property. After using the bitwise AND operation to count the supports of (*ace*) and (*bce*), respectively, only one 2-itemset (*bce*) is a frequent itemset. Because no new

candidates are generated, the candidate-generation-then-testing process stops. Consequently, there are nine frequent itemsets, (a) , (b) , (c) , (e) , (ac) , (bc) , (be) , (ce) , and (bce) , generated by MFI-TimeSW algorithm in $TimeSW_1$. The process is shown in Figure 3-11.

Transactions in $TimeSW_1$	Bit-Sequences in $TimeSW_1$	FI_1 in $TimeSW_1$ ($s = 0.5$ and $FT=3.5$)	sup
$\langle T_1, (be) \rangle$	Bit (b)=1100110,	$\{(b) \mathbf{Bit}(b)=1100110\}$	4
$\langle T_2, (bce) \rangle$	Bit (e)=1100111,	$\{(e) \mathbf{Bit}(e)=1100111\}$	5
$\langle T_3, (bce) \rangle$	Bit (c)=0111111,	$\{(c) \mathbf{Bit}(c)=0111111\}$	6
$\langle T_4, (acd) \rangle$	Bit (a)=0001111,	$\{(a) \mathbf{Bit}(a)=0001111\}$	4
$\langle T_5, (abce) \rangle$	Bit (d)=0001000		
$\langle T_6, (abce) \rangle$			
$\langle T_7, (ace) \rangle$			

CI_2 in $TimeSW_1$	FI_2 in $TimeSW_1$	sup
$\{(ab) \mathbf{Bit}(a) \text{ AND } \mathbf{Bit}(b)\}$	$\{(ac) \mathbf{Bit}(ac) = 0001111\}$	4
$\{(ac) \mathbf{Bit}(a) \text{ AND } \mathbf{Bit}(c)\}$	$\{(bc) \mathbf{Bit}(bc) = 0100110\}$	3
$\{(ae) \mathbf{Bit}(a) \text{ AND } \mathbf{Bit}(e)\}$	$\{(be) \mathbf{Bit}(be) = 1100110\}$	4
$\{(bc) \mathbf{Bit}(b) \text{ AND } \mathbf{Bit}(c)\}$	$\{(ce) \mathbf{Bit}(ce) = 0100111\}$	4
$\{(be) \mathbf{Bit}(b) \text{ AND } \mathbf{Bit}(e)\}$		
$\{(ce) \mathbf{Bit}(c) \text{ AND } \mathbf{Bit}(e)\}$		

CI_3 in $TimeSW_1$	FI_3 in $TimeSW_1$	sup
$\{(bce) \mathbf{Bit}(bc) \text{ AND } \mathbf{Bit}(be) \text{ AND } \mathbf{Bit}(ce)\}$	$\{(bce) \mathbf{Bit}(bce) = 0100110\}$	3

Figure 3- 11. Steps of frequent itemsets generation of MFI-TimeSW in $TimeSW_1$

3.6 Performance Evaluation

In this section, we report the experimental results of the proposed algorithm MFI-TransSW. All the programs are implemented using Microsoft Visual C++ Version 6.0 and performed on a 1.80 GHz Pentium(R) PC machine with 512 MB memory running on Windows 2000. For testing frequent itemsets mining over sliding windows, we generate online data streams using IBM synthetic data generator proposed by Agrawal and Srikant [2, 3]. The synthetic data stream, denoted by T5.I4.D1000K, of size 1 million transactions (D1000K) has an average transaction size of 5 items (T5) with average maximal frequent itemset size of 4 items (I4). In

all experiments, the transactions of T5.I4D1000K are looked up in sequence to simulate the environment of an online data stream.

3.6.1 Experiments of MFI-TransSW Algorithm

In this section, we compare the results of mining by SWFI-stream algorithm [11] and MFI-TransSW algorithm. The experiments of memory usage are shown in Figures 3-12, 3-13, and 3-14, and the processing times are shown in Figures 3-15 and 3-16. The minimum support threshold s and the size of a sliding window w are set to 0.1% and 20,000, respectively. As shown in these experiments, MFI-TransSW significantly outperforms SWFI-stream for both memory consumption and CPU cost.

Figure 3-12 shows the memory usage of the window initialization phase. As shown in Figure 3-12, MFI-TransSW algorithm requires only about 2.1 MB in window initialization phase, but the memory requirement of SWFI-stream increases linearly from 11.2 MB to 109.7 MB. Figure 3-13 shows the memory usage of the window sliding phase. In this phase, the memory requirement of MFI-TransSW is also approximately 2.1 MB, but that of SWFI-stream is between 109.7 MB to 120.3 MB. Figure 3-14 gives the memory usage of the frequent itemsets generation phase. In this phase, the memory requirement of MFI-TransSW is between 33.5MB to 39MB. As shown in Figures 3-12 through 3-14, MFI-TransSW algorithm outperforms SWFI-stream for memory consumption.

Figure 3-15 shows the processing time of window initialization phase under different window sizes from 20,000 (200K) transactions to 100,000 (1,000K) transactions. Figure 3-16 shows the total time of window sliding time and pattern mining time at each 100K transactions using various window sizes from 200K transactions to 1000K transactions. As shown in Figures 3-15 and 3-16, MFI-TransSW algorithm outperforms SWFI-stream for processing time consumption.

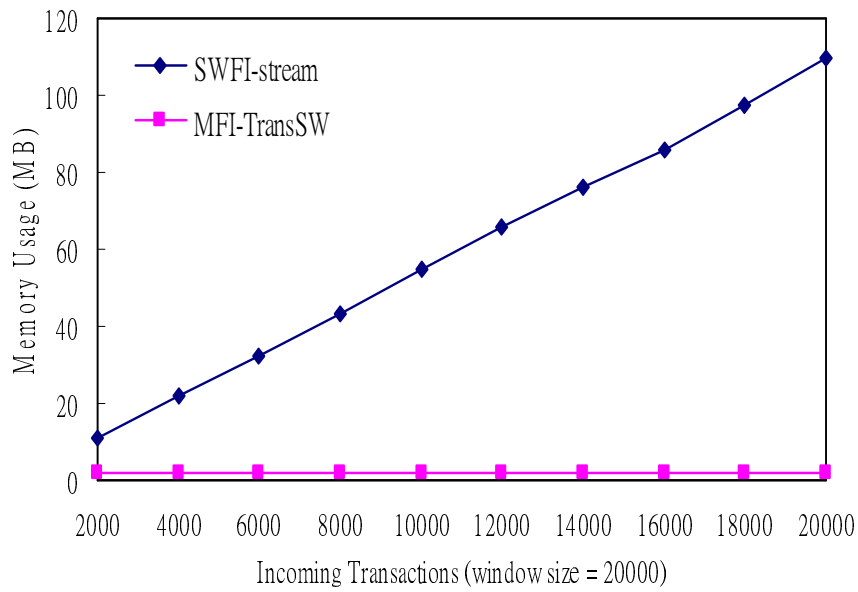


Figure 3- 12. Memory usages in window initialization phases of algorithms SWFI-stream and MFI-TransSW ($s = 0.1\%$ and $w = 20,000$)

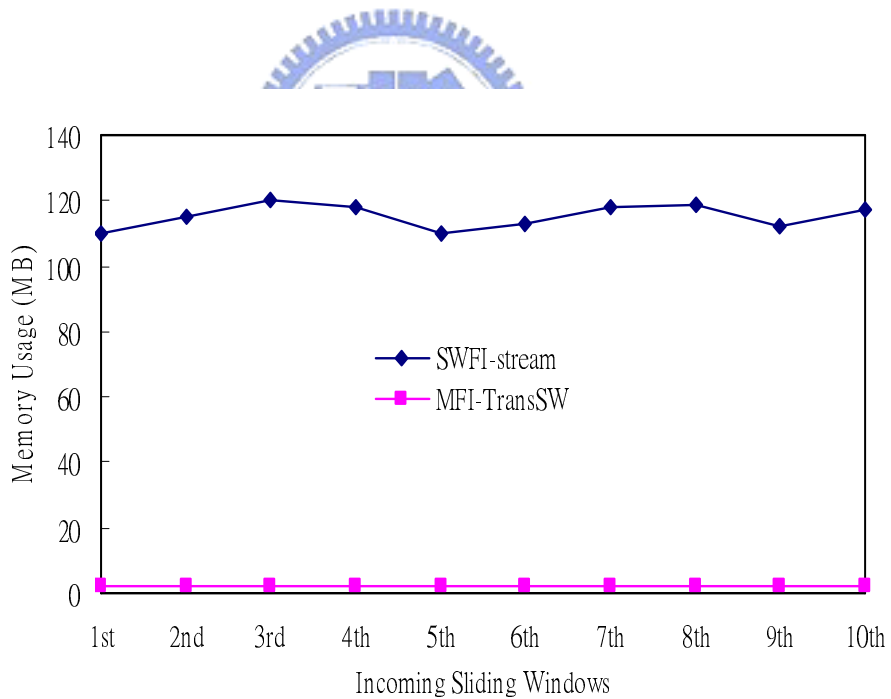


Figure 3- 13. Memory usages in window sliding phases of algorithms SWFI-stream and MFI-TransSW ($s = 0.1\%$ and $w = 20,000$)

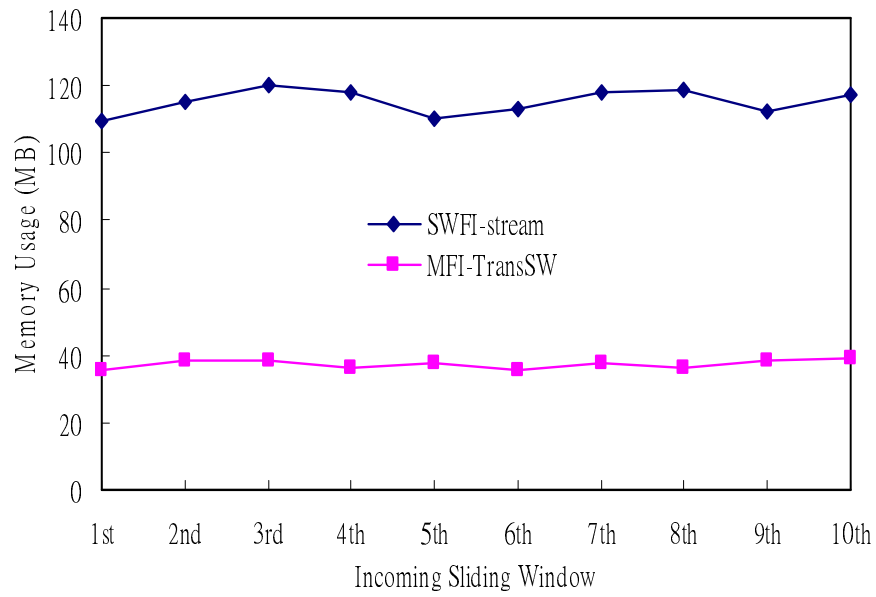


Figure 3- 14. Memory usages in frequent itemset generation phases of algorithms SWFI-stream and MFI-TransSW ($s = 0.1\%$ and $w = 20,000$)

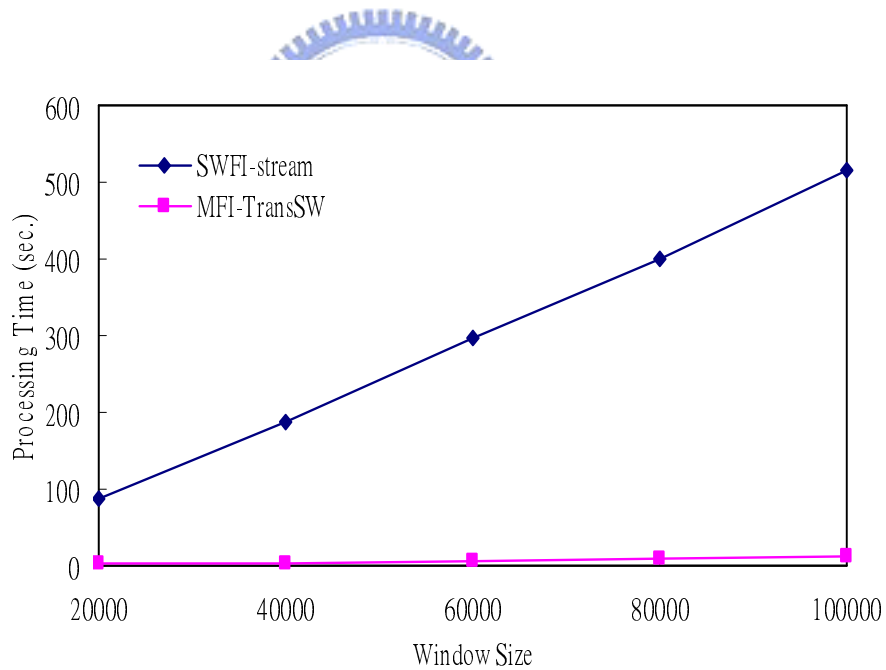


Figure 3- 15. Processing time in window initialization phases of algorithms SWFI-stream and MFI-TransSW under different window sizes ($s = 0.1\%$)

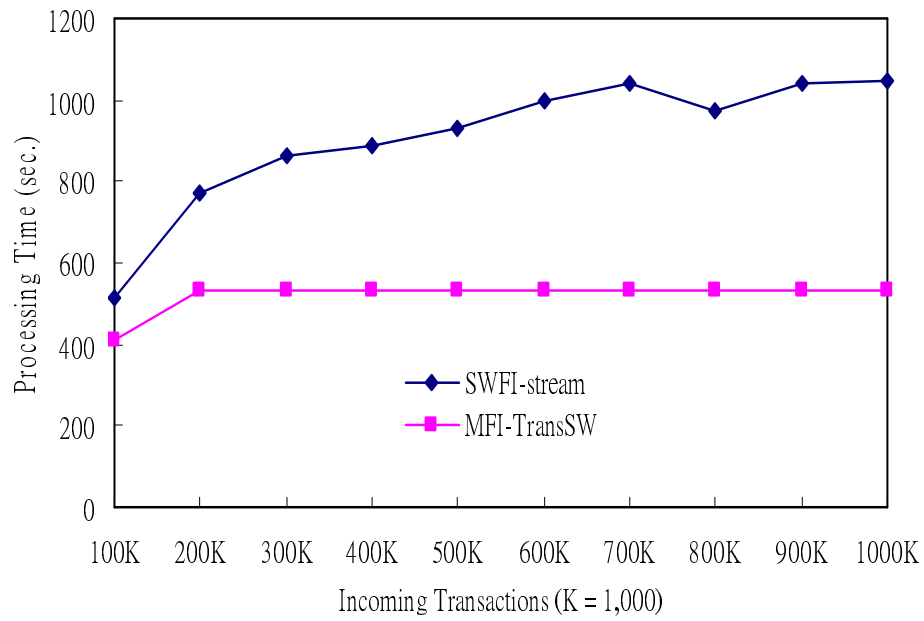


Figure 3- 16. Processing time including window sliding time and pattern generation time of algorithms SWFI-stream and MFI-TransSW under window size 200K transactions ($s = 0.1\%$)

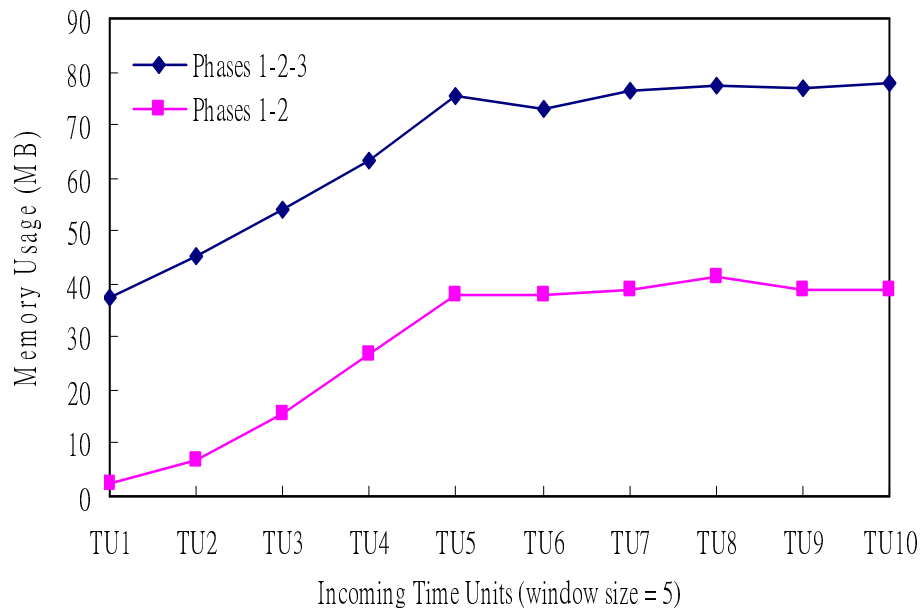


Figure 3- 17. Memory usages of MFI-TimeSW algorithm in different phases ($s = 0.1\%$)

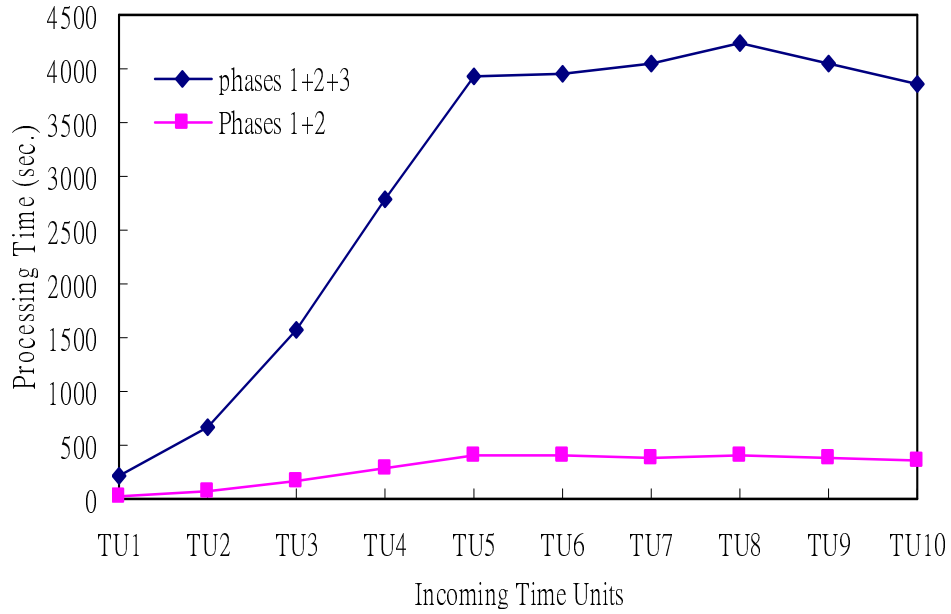


Figure 3- 18. Processing time of MFI-TimeSW algorithm in different phases ($s = 0.1\%$)

3.6.2 Experiments of MFI-TimeSW Algorithm

Because the proposed MFI-TimeSW algorithm is the first single-pass approach for mining frequent itemsets over online data streams with a time-sensitive sliding window, we only report the experimental results of MFI-TimeSW in the section. The experiments of memory usage of the proposed algorithm MFI-TimeSW is shown in Figure 3-17, and the processing time of the algorithm is shown in Figure 3-18. The minimum support threshold s is set to 0.1%. In order to simulate a time-sensitive sliding window over an online data streams, the size of a time-sensitive sliding window w is set to 5, where $|TU_1| = 200K$, $|TU_2| = 400K$, $|TU_3| = 800K$, $|TU_4| = 1,000K$, $|TU_5| = 1,000K$, $|TU_6| = 200K$, $|TU_7| = 500K$, $|TU_8| = 1,000K$, $|TU_9| = 800K$, and $|TU_{10}| = 800K$. Note that 1K transactions equals to 1,000 transactions.

Figure 3-17 shows the memory usage of phases 1-2 (window initialization phase + window sliding phase) and phases 1-2-3 (window initialization phase + window sliding phase

+ frequent itemsets generation phase) of MFI-TimeSW algorithm. As shown in Figure 3-17, the memory usage of MFI-TimeSW is increased linearly as the window size increased.

Figure 3-18 shows the processing time of phases 1-2 (window initialization phase + window sliding phase) and phases 1-2-3 (window initialization phase + window sliding phase + frequent itemsets generation phase) of MFI-TimeSW algorithm. As shown in Figure 3-18, the processing time of phases 1 and 2 of MFI-TimeSW is increased linearly as the window size increased.


3.7 Conclusions

In this chapter, we proposed two efficient one-pass algorithms, called MFI-TransSW and MFI-TimeSW, for mining frequent itemsets over online data streams with a transaction-sensitive sliding window and a time-sensitive sliding window, respectively. Experiments show that the proposed algorithms not only attain highly accurate mining results, but also run significant faster and consume less memory than existing algorithms for mining frequent itemsets over recent online data streams.

Chapter 4 Online Mining of Changes of Items across Two Data Streams

As data streams are gaining prominence in a growing number of emerging applications, advanced analysis and mining of data streams is becoming increasingly important. While there are some recent studies on mining data streams, we would like to ask the following essential question: What are the distinct features of mining data streams compared to mining other kinds of data? *Online mining of the changes in data streams* is one of the core issues [24]. In this chapter, we propose a new interesting research problem and propose efficient algorithms for this problem.

4.1 Introduction



The motivation of the problem of online mining changes of items between distributed data streams comes from the context of online transaction flows in large organizations. These companies generate the millions of records every day. For example, Google handles 70-110 millions searches, AT&T produces 250-300 million call records, and WallMart which consists of thousands of branch stores, and records 20-40 million transactions in a single day. With the computation model of distributed data streams presented in Figure 4-1, a data stream processor and the in-memory summary data structure are two major components in the distributed data streaming environment. The streams in questions are sequences of transaction data which is composed of the records in the form of $\langle Store-ID, Timestamp, Transaction-ID, Items \rangle$. In other words, a transaction record is a purchasing log generated by a customer in a specific time and store. These transaction flows are sent to the server, and we are interested in finding the frequent frequency changes in items between pairs of data streams purchased by the most customers in some period of time. Note that the buffer mechanism can be optionally

set for temporary storage of recent transactions from the transaction data streams.

In this chapter, we study the problem of online mining frequent frequency changes of items between pairs of continuous, high-volume, open-ended data streams. Three types of frequency change are defined: *frequent changed-item* (or **FCI** in short), *vibrated frequent changed-item* (or **VFCI** in short), and *stable frequent changed-item* (or **SFCI** in short). A new summary data structure, called *change-sketch*, is developed to store the essential information over the pairs of data streams. The *MFC-append* (Mining Frequent Changes of append-only data streams) algorithm is proposed to find the changes across two append-only data streams. The *MFC-dynamic* algorithm based on MFC-append is developed to find the changes over two dynamic data streams. The best space bound we achieve is $\Omega(m \log(n/m))$, where n is the size of the union of two data streams, and m is the size of the working bucket for frequent changed-items mining. Moreover, the proposed algorithms take $O(\log(n/m))$ time in the worst case to process each new data element, but only $O(1)$ amortized time per data element.

The remainder of the chapter is organized as follows. We review some related work in Section 4.2 and formulate the problem in Section 4.3. Algorithms MFC-append and MFC-dynamic are described in Section 4.4. Performance evaluation is presented in Section 4.5. We conclude the work in Section 4.6.

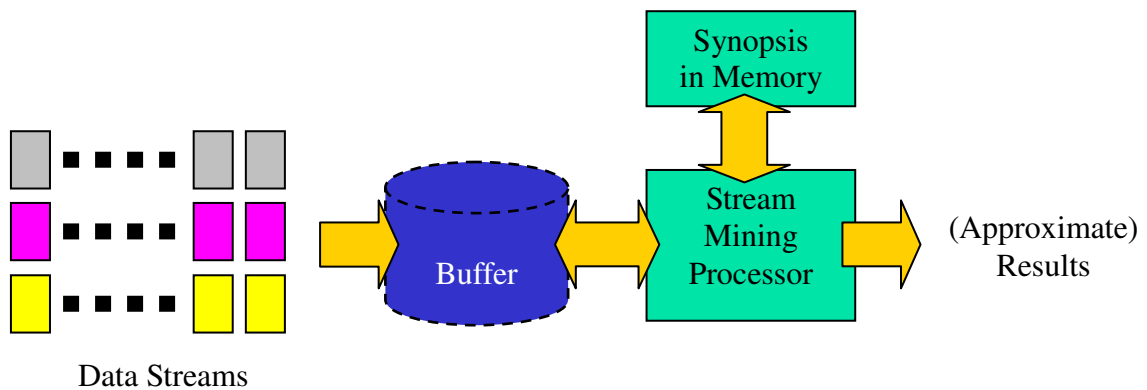
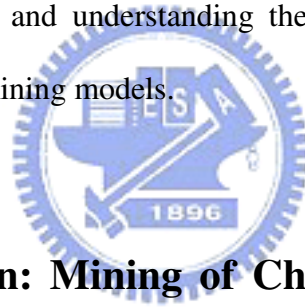


Figure 4- 1. Processing model of distributed data streams

4.2 Related Work

Change mining on static datasets has been studied in the last ten years [29, 25, 52]. Ganti et al. [29] proposed a framework to quantify the deviation in the induced models, such as two decision tree classifiers, clusters, and frequent itemsets, in the large datasets. The quantitative measure is the amount of work required to transform one model into the other. Dong et al. [25] proposed an algorithm to find the emerging patterns, and used these patterns to characterize the changes from one dataset to the other. Liu et al. [52] proposed a method to discover the changes in the new data with respect to the old data, and the old decision tree models, and generate the exact changes that have occurred to the user. These studies are focused on the effects of data changes of data mining models and algorithms, whereas this chapter is focused on the problem of measuring and understanding the changes of data directly rather than measuring the effects of data mining models.



4.3 Problem Definition: Mining of Changes of Items across Two Data Streams

Let $\Psi = \{i_1, i_2, \dots, i_m\}$ be a set of literals, called *data items* (or *items* in short). A *data stream* is an infinite sequence of data items, where the items arrive in some order, and may be seen only once. It is also referred to as *item-stream*. In the item-stream model, we focus on two performance issues: *workspace* required in main memory, which is measured as a function of the input union size n of two data streams, and the *time* to process an incoming data item over the streams. In this chapter, we assume that the data arrives in the *unordered*² form, and the same value can appear multiple times within the streaming data. This is termed the *unordered cash register, unordered aggregated model* [6, 33].

² The streaming data items from various domains arrive in no particular order and without any pre-processing.

Definition 4-1. A data stream is called an *append-only data stream* (or **ADS** in short) if it has no updates and deletions. A data stream is called a *dynamic data stream* (or **DDS** in short) if there are removal as well as addition of data items.

Definition 4-2. Two parallel item-streams are $P = \langle p_1, p_2, \dots, p_i, \dots \rangle$, and $Q = \langle q_1, q_2, \dots, q_j, \dots \rangle$ with time-varying data rates, where $p_i, q_j \in \Psi$. The *frequency* of a data item x in an item-stream S over a time period T is the number of items in T in which x occurs, and is denoted as $frequency(x, S, T)$. The size of T is n , the total number of data items so far in T .

Definition 4-3. The *changed support* of a data item x is the difference in frequency between two data streams P and Q divided by the total data items observed in T , and is denoted as $changeSup(x, T)$.

Definition 4-4. The *changed rate* of a data item x is the number of *frequency vibration* divided by the total time-points observed in T , and is denoted as $changeRate(x, T)$, where the *time-point* is a basic unit of time over which the system collects data, e.g., second or minute. *Frequency vibration* is the ratio of frequency change which exceeds a user-specified threshold, *vibrate rate*. In this research, we assume that the rate is 100% for simplicity, i.e., frequency vibration is a frequency change from positive one to negative one, or vice versa.

Definition 4-5. A data item x is called a *frequent frequency changed item* (or **FFCI** in short) if $changeSup(x, T) \geq mcs$, where mcs is a user-defined minimum changed support threshold in the range of $[0, 1]$. It is a *sub-frequent frequency changed item* (or **SFFCI** in short) if $ase \leq changeSup(x, T) < mcs$, where ase is a user-defined approximate support error threshold in the range of $[0, mcs]$. It is an *infrequent frequency changed item* (or **IFFCI** in short) if $changeSup(x, T) < ase$.

Definition 4-6. A data item x over a time period T is called a *vibrated frequency changed item* (or **VFCI** in short) if its changed rate and changed support are greater than or equal to a

user-defined *minimum* changed rate (or *mincr* in shot) and *ase*, respectively. It is a stable frequency changed item (or **SFCI** in short) if its changed rate is less than a user-specified *maximal* changed rate (or *maxcr* in short), and $changeSup(x, T) \geq mcs$, where *mincr* is a real number in the range of [0, 1] and $maxcr > mincr$.

For example, there are *ten* time-points ($T = [t_1: t_{10}]$, where t_1 is the starting time-point and t_{10} is the current time-point) in Figure 4-2, and we assume that $mincr = 0.1$, and $maxcr = 0.5$. In Figure 4-2, data item *a* and *b* are VFCIs, where $changeRate(a, T) = 9/10 = 0.9 > 0.5$, and $changeRate(b, T) = 6/10 = 0.6 > 0.5$, and items *c*, *d*, *e* are SFCIs, where $changeRate(c, T) = 0/10 = 0 \leq 0.1$, $changeRate(d, T) = 0/10 = 0 \leq 0.1$, and $changeRate(e, T) = 1/10 = 0.1 \leq 0.1$.

The goal of this chapter is to find the changes of items (FFCIs, VFCIs, and SFCIs) over the pairs of data streams, either in ADS or DDS.

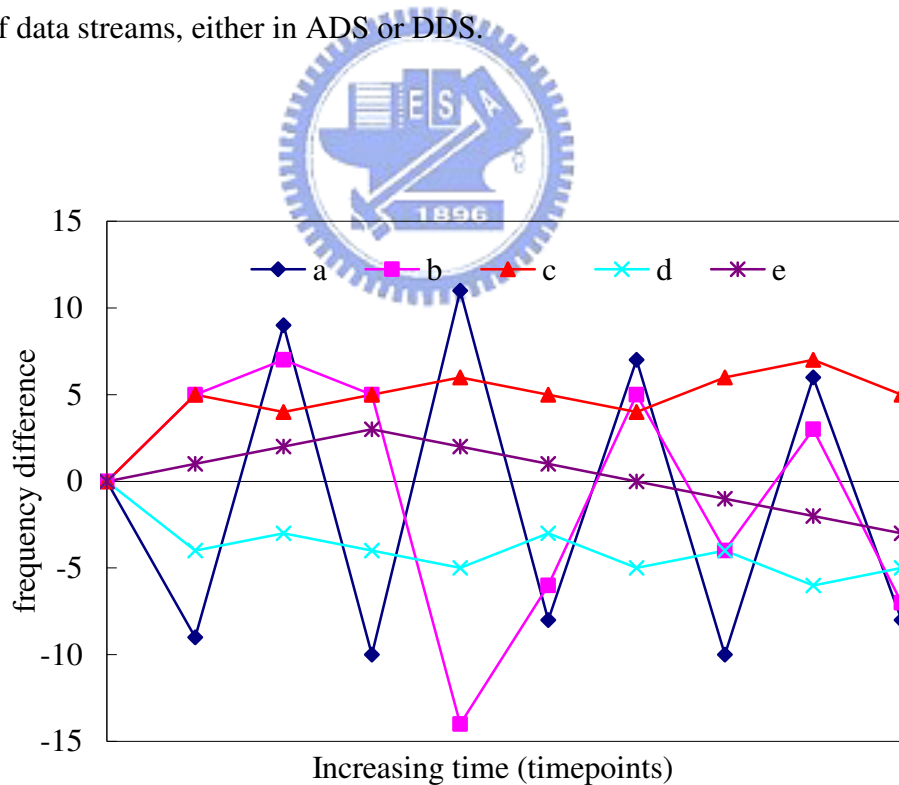


Figure 4- 2. Examples of VFCIs and SFCIs

4.4 Online Mining Changes of Items over Distributed ADSs

In this section, a new summary data structure, called *Change-Sketch*, is developed to maintain the essential information about the set of all FFCIs, VFCIs, and SFCIs embedded in data streams. A deterministic single-pass algorithm *MFC-append* (Mining Frequency Changes of append-only data streams) is proposed to find the changes of items over the pairs of data streams. The proposed algorithm uses at most $m \log(n/m)$ space, where n is the size of the union of the estimated data streams, and m is the size of working bucket.

4.4.1 A New Summary Data Structure: Change-Sketch

The proposed in-memory summary data structure *Change-Sketch* is a list of entries of the form $(q, q.count, q.wid, q.rate)$, where q is a data item in the streams, $q.count$ is an integer representing its estimated support, the value of $q.wid$ assigned to a new entry q is the window identifier of current window, and $q.rate$ is the number of frequency vibration of item q . An item q is stored in the current *Change-Sketch* if $q.count \geq ase \cdot m \cdot (w_{current-id} - q.wid)$, where m is the window size and $m = \lceil 1/ase \rceil$. Note that the parameter *ase* is an acronym of the user-specified approximate support error threshold.

Two operations are used to maintain the *Change-Sketch*:

- (1) **Update Change:** For each entry $(q, q.count, q.wid, q.rate) \in \text{Change-Sketch}$, *MFC-append* increases $q.count$ by computing the frequency changes of q in the current window. If the updated entry q takes place a frequency vibration, its $q.rate$ is increased by one. If the changed support of updated entry q is less than the user-specified minimum changed support threshold mcs , the entry is deleted from the current *Change-Sketch*.

(2) **New Change:** If an item $p \notin \text{Change-Sketch}$, and its changed support is larger than or equal to the threshold $ase \cdot m \cdot (w_{\text{current-id}} - p \cdot w_{id})$, a new entry of the form $(p, 1, p \cdot w_{\text{current-id}}, 0)$ is created into the current Change-Sketch.

4.4.2 The MFC-append Algorithm

Algorithm MFC-append uses the notations and conventions illustrated in Figure 4-3. In the framework of mining changes of items over data streams, the streaming data is divided into fixed sized buckets $B_1, B_2, \dots, B_i, \dots, B_N$, where B_N is the “latest” bucket with bucket identifier N , and B_1 is the “oldest” one. Note that each bucket contains k items. The *bucket length* from B_i to B_j is denoted as $B(i, j)$, where $i \geq j$. Let t_1, t_2, \dots, t_n be the *timepoints* (the smallest unit of time) which group the buckets so far in the streams, where t_n is the most recent timepoint, and t_1 is the oldest one. The form of bucket B_i is $(StreamID, t_i, items)$, where t_i is the timepoint when the *items* appeared in the stream with identifier *StreamID*.

The *window-id* of t_i is denoted as w_i , and the number of buckets arrived from t_{i-1} to t_i is $|w_i|$, and the number of items (i.e., *size*) in w_i is denoted as $|w_i|$. The size of buckets arrived in T equals $|w_k| + |w_{k+1}| + \dots + |w_n|, \forall k = 1, 2, \dots, n$. As described above, the goal is to find the set of all FFCIs, VFCIs, and SFCIs in a time period $T = t_k \cup t_{k+1} \cup \dots \cup t_n, \forall k = 1, 2, \dots, n$. Hence, the pair of input data streams P and Q are divided into two sequences of basic windows, i.e., $P = w_1[B_{P_1} + B_{P_2} + \dots + B_{P_i}] + w_2[B_{P_{i+1}} + B_{P_{i+2}} + \dots + B_{P_j}] + \dots + w_m[B_{P_k} + B_{P_{k+1}} + \dots + B_{P_{\text{currentid}-1}}]$, and $Q = w_1[B_{Q_1} + B_{Q_2} + \dots + B_{Q_i}] + w_2[B_{Q_{i+1}} + B_{Q_{i+2}} + \dots + B_{Q_j}] + \dots + w_m[B_{Q_k} + B_{Q_{k+1}} + \dots + B_{Q_{\text{currentid}-1}}]$. The notation $w_i[B_{StreamID_j} + B_{StreamID_{j+1}} + \dots + B_{StreamID_k}]$ denotes that the buckets of data stream with id *StreamID* arrived at timepoint t_i , and the current bucket id is denoted as $B_{StreamID_{\text{current}}}$. Note that $B_{StreamID_{\text{current}}} = \lceil n/m \rceil + 1$. For

example, there are five buckets in the first window w_1 of Figure 4-1, in which two buckets (B_{P_1} and B_{P_2}) in stream P , and three buckets (B_{Q_1} , B_{Q_2} , and B_{Q_3}) in stream Q .

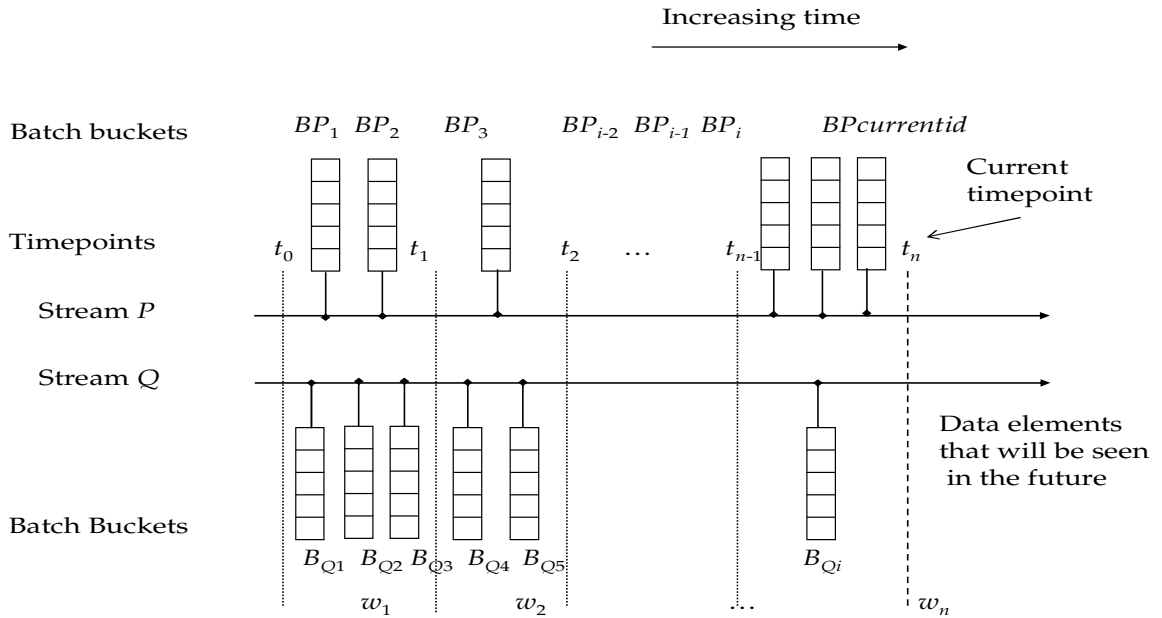


Figure 4- 3. Notations and conventions used in the proposed algorithms

The algorithm description of MFC-append is shown in Figure 4-4. Four parameters are used in MFC-append algorithm: mcs , ase , $maxcr$, and $mincr$, where mcs is an acronym of the minimum changed support threshold, ase is an acronym of the approximate error support threshold, $maxcr$ is an acronym of the maximum changed rate, and $mincr$ is an acronym of the minimum changed rate. At any moment, a list of FFCIs with their estimated changed supports and changed rates is generated by the proposed algorithm. These approximate answers (i.e., a list of FFCIs) have the following guarantees. First, all items whose changed support exceed $mcs \cdot n$ are output, i.e., *no false negative*. Second, no items whose changed support is less than $(ase - mcs) \cdot n$ are output. Third, estimated changed supports are less than the true changed

supports by at most $ase \cdot n$. Finally, all items whose changed rate exceed $mcr \cdot n$ or less than $mcr \cdot n$ are output, respectively.

Algorithm MFC-append

Input: (1) Two continuous append-only data streams, $P = \langle p_1, p_2, \dots, p_n, \dots \rangle$ and $Q = \langle q_1, q_2, \dots, q_n, \dots \rangle$ with time-varying data rate, (2) A user-defined approximate support error threshold, ase , i.e., the window size m is $\lceil 1/ase \rceil$, (3) A user-defined minimum changed support threshold, mcs , (4) A user-specified maximum changed rate $maxcr$, (5) A user-specified minimum changed rate $minicr$.

Output: A list of FFCIs, VFCIs, and SFCIs.

Begin

$Change-Sketch() \leftarrow \{ \}$;

Repeat:

for each bucket from the data streams (P and Q) **do**

for each item q in $w_i(C, B_i)$ **do** /* $i = 1, 2, \dots, \lceil n/m \rceil + 1$ */

$Change-Sketch(q, q.count++, q.wid, q.rate)$;

for each item q in $w_i(D, B_i)$ **do**

$Change-Sketch(q, q.count--, q.wid, q.rate)$;

while $Change-Sketch(q, q.count, q.wid, q.rate) \neq \emptyset$ **then**

if $|q.count| \geq mcs \cdot m \cdot (w_{current} - q.wid)$ **then**

item q is a frequent frequency change pattern in $Change-Sketch$;

else if $|q.count| \geq ase \cdot m \cdot (w_{current} - q.wid)$ **then**

preserve q in $Change-Sketch$;

else remove q from $Change-Sketch$;

if $q.w_i$ change its symbol (either from positive frequency to negative one or from negative one to positive one)

then $q.rate++$;

End

Figure 4- 4. Algorithm MFC-append

The maintenance process of Change-Sketch is described as follows. Let the window identifier of current window be k . Initially, Change-Sketch is empty. For each item q in the current window of item-stream P , MFC-append first checks Change-Sketch to see whether an entry with id q already exists or not. If the entry exists in the current Change-Sketch, the frequency of q (i.e., $q.count$) is increased by one. Otherwise, a new entry of the form $(q, 1, k, 0)$ is created in the current Change-Sketch. After processing all items in w_k of stream P , MFC-append computes all the items in w_k of another stream Q to maintain the changed information in Change-Sketch. The computation first checks Change-Sketch to see whether an entry q already exists or not in the Change-Sketch. If the search succeeds, the proposed algorithm updates the entry with id q by decreasing its frequency $q.count$ by one. Otherwise, a new entry of the form $(q, -1, k, 0)$ is created in the current Change-Sketch. Now, if the updated entry q take place frequency vibration, $q.rate$ is increased by one, i.e., from zero to one.

In order to bound the memory usage in mining changes of items over data streams, a pruning mechanism of Change-Sketch is proposed. The technique deletes some entries of Change-Sketch before MFC-append computes the next working window with window-id $k+1$. It is a trade-off between the accuracy of the outputs and the memory requirement of Change-Sketch. The pruning is described as follows. An entry of the form $(q, q.count, q.w_i, q.rate)$ is deleted, if $|q.count| < ase \cdot m \cdot (w_{current-id} - q.w_{id})$. After the pruning, MFC-append computes the next working windows with window-id w_{k+1} of data streams P and Q in the same way as described above.

When a user requests the results of the set of all FFCIs, VFCIs, and SFCIs embedded in the data streams, MFC-append algorithm outputs the entries whose $|q.count| \geq mcs \cdot m \cdot (w_{current-id} - q.w_{id})$, $|q.rate| \geq mincr \cdot m \cdot (w_{current-id} - q.w_{id})$, and $|q.rate| \geq maxcr \cdot m \cdot (w_{current-id} - q.w_{id})$, respectively, by one scan of the current Change-Sketch.

4.4.3 Space Analysis of Change-Sketch

In this section, we prove that MFC-append algorithm uses at most $O(m \log(n/m))$ space, where n denotes the current length of the estimated data streams, and $m = \lceil 1/ase \rceil$ is the size of working bucket.

Theorem 4-1: *The space requirement of MFC-append algorithm is $O(m \log(n/m))$.*

Proof: Let $w_{current-id}$ be the current window-id, i.e., $w_{current-id} = \lceil n/m \rceil$, where m is the size of working bucket. Let c_i denote the number of items in *Change-Sketch*, whose window id is $w_{current-id} - i + 1$. Since the size of each working bucket is m , we get the following constraints:

$$\sum_{i=1}^k ic_i \leq km \quad \text{for } k = 1, 2, \dots, w_{current-id}. \quad (1)$$

We claim that

$$\sum_{i=1}^k c_i \leq \sum_{i=1}^k \frac{m}{i} \quad \text{for } k = 1, 2, \dots, w_{current-id}. \quad (2)$$

We prove Inequality (2) by induction on k . If $k = 1$, then the claim is true because $c_1 \leq m$, i.e., we prove it from Inequality (1) directly. We now assume that Inequality (2) is true for $k = 1, 2, \dots, j-1$, and prove that this assumption implies that it is true for $k = j$. We now add Inequality (1) for $k = j$ to $j-1$ instances of Inequality (2) and we have

$$\begin{aligned} & \sum_{i=1}^j ic_i + \sum_{i=1}^1 c_i + \sum_{i=1}^2 c_i + \dots + \sum_{i=1}^{j-1} c_i \leq jm + \sum_{i=1}^1 \frac{m}{i} + \sum_{i=1}^2 \frac{m}{i} + \dots + \sum_{i=1}^{j-1} \frac{m}{i}. \\ \Rightarrow & c_1 + 2c_2 + \dots + (j-1)c_{j-1} + jc_j + [c_1 + (c_1 + c_2) + \dots + (c_1 + c_2 + \dots + c_{j-1})] \leq jm + [m \\ & + (m + m/2) + \dots + (m + m/2 + \dots + m/(j-1))]. \\ \Rightarrow & jc_1 + jc_2 + \dots + jc_{j-1} + jc_j \leq jm + [(j-1)m + (j-2)m/2 + \dots + m/(j-1)] \\ \Rightarrow & j \sum_{i=1}^j c_i \leq jm + \sum_{i=1}^{j-1} \frac{(j-i)m}{i}. \end{aligned}$$

Upon rearrangement, we get $j \sum_{i=1}^j c_i \leq jm + \sum_{i=1}^{j-1} \frac{(j-i)m}{i}$, which can be easily simplified

to Inequality (2) for $k = j$. Then we can complete the induction.

Since $|Change-Sketch| = \sum_{i=1}^{w_{current}} c_i$, from Inequality (2), we get $|Change-Sketch| \leq \sum_{i=1}^{w_{current}} \frac{m}{i} \leq$

$m \log(w_{current-id}) = m \log(n/m)$.

□

Note that, if $ase \leq (1/m)$, the space is effectively $\Omega(m \log(n/m))$. If we set $ase = (d/m)$ for some small d , then it requires time at most $O(m \log(n/m))$. However, this occurs only every $1/m$ items, and so the total time is $O(n \log(n/m))$.

4.5 Online Mining Changes of Items over Distributed DDSs

In this section, a MFC-append based algorithm, called *MFC-dynamic* (Mining Frequency Changes of dynamic data streams), is proposed to mine the set of all FFCIs, VFCIs, and SFCIs over dynamic data streams. Note that a data stream is called a *dynamic data stream* (or **DDS** in short) if there are removal as well as addition of data items.

An effective encoding method is used in the proposed algorithm to distinguish the inserted items and deleted items over DDSs, and is described as follows. If an item q is an inserted item, MFC-dynamic encodes it to be a “*positive*” item, denotes as $+q$. Otherwise, a deleted items q is encoded as a “*negative*” item, denotes as $-q$. After processing the encoding, MFC-append algorithm is used to find the set of all FFCIs, VFCIs, and SFCIs over dynamic data streams. Figure 4-5 presents the description of MFC-dynamic algorithm. From the interpretation of MFC-dynamic, a space usage guarantee, which is similar to Theorem 4-1, is given as follows.

Claim 4-1. Whenever the deletions of item q occurs, $frequency(q)_{Deleted} \leq frequency(q)$, where $frequency(q)_{Deleted}$ is the frequency of item q needed to be drop.

Claim 4-2. An item $q \notin \text{Change-Sketch}$, if $|q.count| < ase \cdot m \cdot (w_{current-id} - q.w_{id})$.

Theorem 4-2. The space requirement of *MFC-dynamic* algorithm is $O(m \log(n/m))$.

Proof: According to the pruning rule, only items with frequency f or larger within the last updated f windows age are not pruned. Thus, at most m/f items could have been survived from that window which gives $m \sum_{i=1}^{n/m} \frac{1}{i}$ as the upper-bound on the number of items we are keeping

track of. Now, using the well know inequality $\sum_{i=1}^p \frac{1}{i} \leq \log(p)$, the result follows directly.

□



4.6 Performance Evaluation

4.6.1 Synthetic Data Generation

In the experiments of *MFC-append*, we generated three datasets $|D|$ of 10,000, 100,000, and 1,000,000 transactions of single-item, and searched for frequent frequency changes while varying the Zipf parameter from 0 (uniform) to 3 (highly skewed), and the *ase* from 1% to 0.001%. In order to evaluate algorithm *MFC-dynamic*, we use the generation approach of synthetic data from [20]. The generated data consists of three parts: (1) a sequence of insertions distributed uniformly over a small range; (2) a sequence of insertions was drawn from a Zipf distribution with varying parameter (from 0 to 3); (3) a sequence of deletions was distributed uniformly over the same range as the starting sequence. We examine

MFC-dynamic in the fourth dataset of 1,000,000 transactions of single-item, Zipf parameter from 0 to 3, and *ase* from 1% to 0.001%.

Algorithm *MFC-dynamic*

Input: (1) Two dynamic data streams, $C=\{c_1, c_2, \dots, c_n, \dots\}$ and $D=\{d_1, d_2, \dots, d_n, \dots\}$ with time-varying data rate, (2) A minimum change support threshold, *mcs*, (3) An approximation support error threshold, *ase*, (4) A maximum change rate threshold, *maxcr*, (5) A minimum change rate threshold, *minicr*.

Output: A list of change patterns $\{q_i, \dots, q_j\}$ over dynamic data streams.

Begin

Dynamic_Encode_Streaming_Items(C, D);

MFC-append(C, D, mcs, ase, maxcr, minicr);

End

Procedure *Dynamic_Encode_Streaming_Items(C, D);*

Begin

for each bucket w_{C_i} of stream *C* and bucket w_{D_i} of stream *D*

if the item *q* is an inserted item **then**

Set it to be a positive (+*q*) item;

else

Set it to be a negative (-*q*) item;

end

endfor

End

Figure 4- 5. Algorithm *MFC-dynamic*

4.6.2 Experimental Results

In this following experimental testing (results as shown in Figure 4-6 through Figure 4-9), we use threshold $mcs = 0.01$, and $ase = 0.1 \cdot mcs$. First, we computed recall and precision for *MFC-append*, with the results shown in Figure 4-6. In this Figure, we can see that *MFC-append* algorithm has excellent precision (0.90-1.00) and recall (0.6-0.81) on the synthetic data $|D|=10,000$ transactions, and the recall decreases as the parameter ase increases, while the precision increases as the ase decreases. An important observation is that the Zipf parameters (from 0 to 3) do not affect the recall and precision of *MFC-append*.

In Figure 4-7, we can see that *MFC-append* has precision (0.93-1.00) and recall (0.57-0.76) on the synthetic data $|D|=100,000$ transactions. In Figure 4-8, we can see that *MFC-append* has precision (0.92-1.00) and recall (0.51-0.71) on the synthetic data $|D|=1,000,000$ transactions.

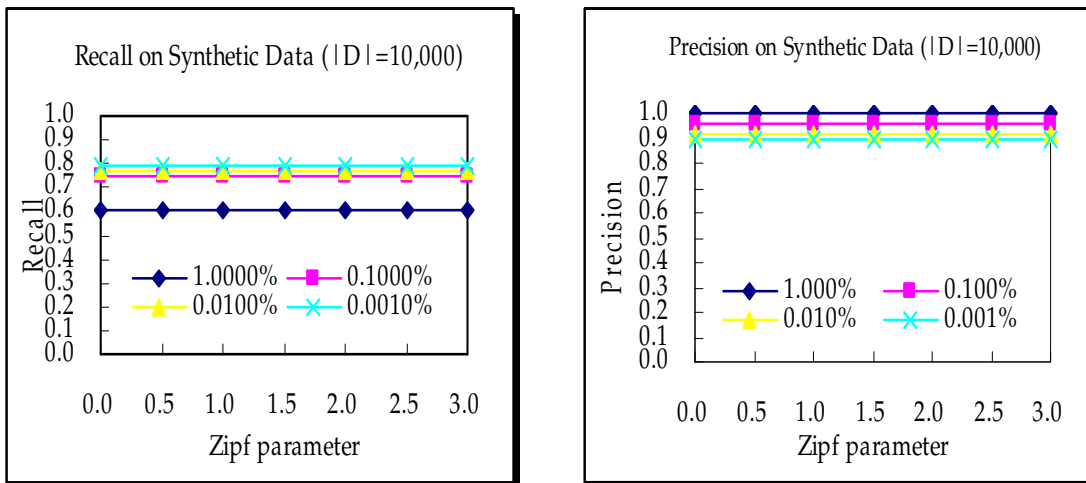


Figure 4- 6. Experiments on synthetic data (10^4 transactions) for *MFC-append*. Left: recall (proportion of the frequent change patterns reported). Right: precision (proportion of the output frequency change patterns which are frequent)

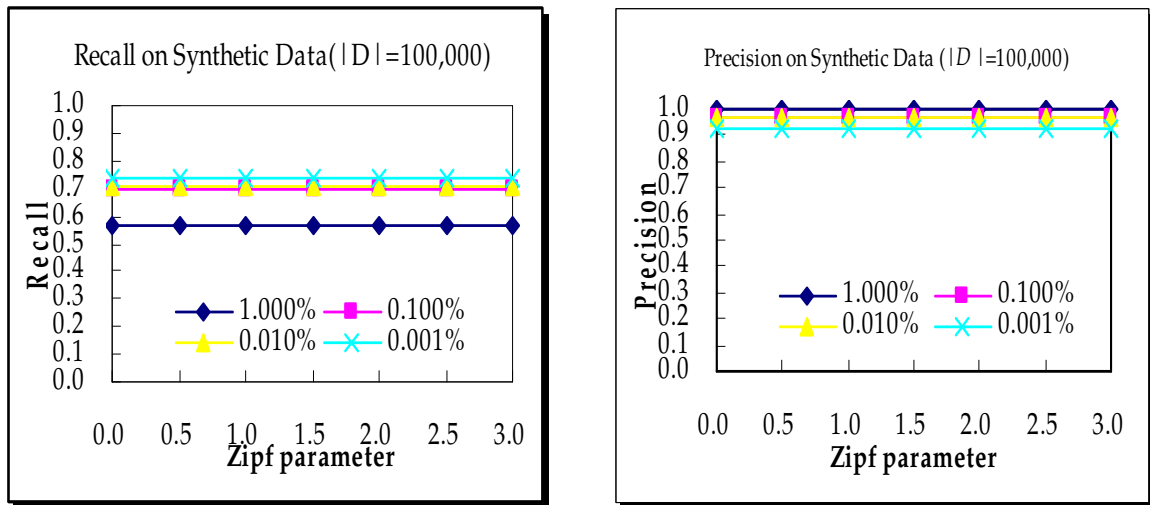


Figure 4- 7. Experiments on synthetic data (10^5 transactions) for *MFC-append*. Left: recall. Right: precision

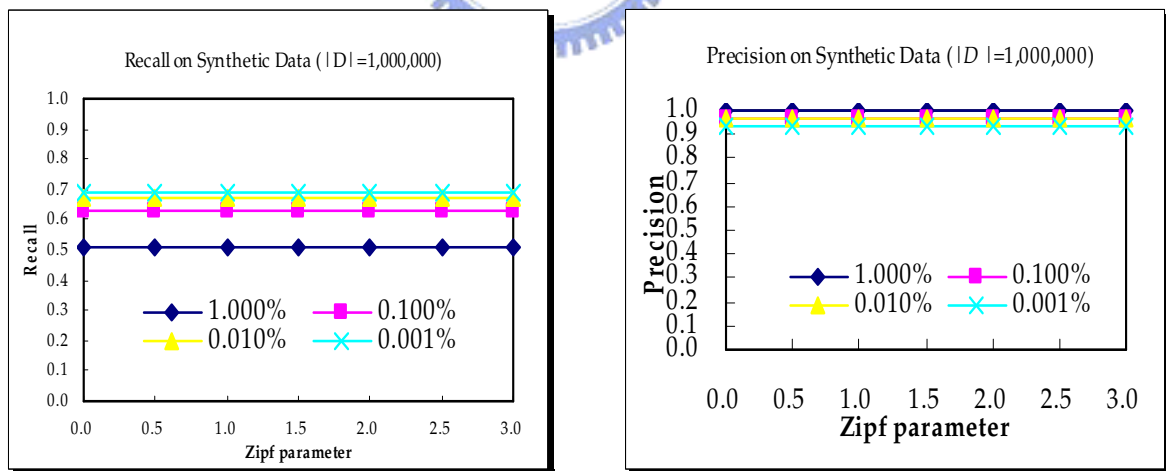


Figure 4- 8. Experiments on synthetic data (10^6 transactions) for *MFC-append*. Left: recall. Right: precision

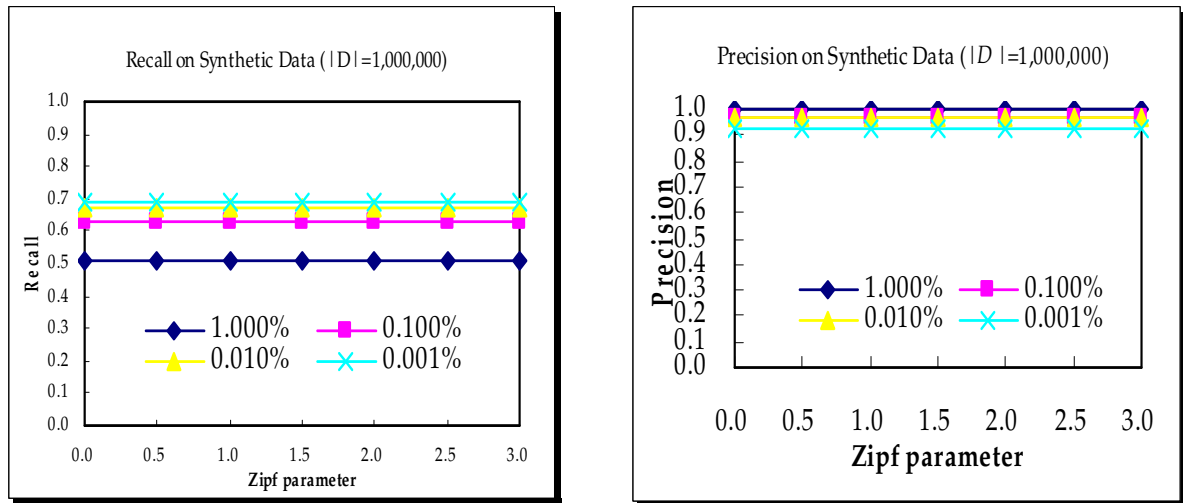


Figure 4- 9. Experiments on synthetic data (10^6 transactions) for *MFC-dynamic*. Left: recall. Right: precision

In Figure 4-9, we can see that the *MFC-dynamic* has the similar experimental results as algorithm *MFC-append*. The recall increases as the *ase* decreases while the precision decreases as the *ase* increases, and the various Zipf parameters do not influence the recall and precision of *MFC-dynamic*.

4.7 Conclusions

In this chapter, we propose two single-pass algorithms, called *MFC-append* and *MFC-dynamic*, for mining frequent frequency changed items, vibrated frequency changed items, and stable frequency changed items over continuous append-only and dynamic data streams, respectively. A new summary data structure, called *Change-Sketch*, is developed to store the essential changed patterns of data streams. The space complexity of *Change-Sketch* is $O(m \log(n/m))$, and the proposed algorithms take $O(\log(n/m))$ time in the worst case to compute each newly arrived item, but only $O(1)$ amortized time per item. The experimental

results show that our algorithms have linear scalability and high accuracy in the analytical outputs.



Chapter 5 Online Mining of Path Traversal Patterns over Web Click-Streams

Mining of path traversal patterns is one of the most important research issues of Web usage mining. The problem of mining of *path traversal patterns* from a large *static* Web click dataset was presented and two *multiple-pass* algorithms, FS (Full Scan) and SS (Selective Scan), are proposed by Chen et al. [13]. However, multiple-pass algorithms FS and SS are not feasible to mine the set of path traversal patterns in a streaming Web click-sequence environment. Hence, in this chapter, we modified the path traversal pattern mining problem proposed by Chen et al. [13] into a new research problem of Web usage mining.

5.1 Introduction



Cooley et al. [19] and Srivastava et al. [62] have surveyed the major technical advances and research problems in Web data mining. In general, Web data mining can be divided into three categories: Web structure mining, Web content mining and Web usage mining. The goal of Web structure mining is to generate structural summary about the Web site and Web page. The goal of Web content mining is to describe the automatic search of information resource available online, and to discover Web data content. Web usage mining is the process of automatic discovery of user navigation patterns from Web server logs. In this section, a brief review of Web user navigation pattern mining is described as follows.

Chen et al. [13] defined a problem of mining path traversal patterns in a large Web-log dataset. Two algorithms, FS (Full Scan) and SS (Selective Scan), were proposed. These algorithms use level-by-level methods, i.e., Apriori-based approach [3], to discover maximal reference sequences in a static Web click dataset. Although FS and SS mine path traversal

patterns in a static Web log dataset efficiently, they are not feasible in the mining of streaming Web click-sequences. This is because FS and SS algorithms need to scan the dataset at least twice.

Spiliopoulou et al. [61] proposed a navigation pattern discovery miner, called WUM (Web Utilization Miner), and proposed an algorithm for building an aggregating tree from static Web logs. Then, WUM mines the Web access patterns by MINT mining language. Borges and Levene [9] proposed a model of hypertext that captures the user navigation behavior patterns. The set of user navigation sessions is modeled as a HPG (Hypertext Probabilistic Grammar), and the set of strings which are generated with higher probability correspond to the navigation trials preferred by users. Pei et al. [57] proposed a WAP-tree (Web Access Pattern tree) to store the frequent Web page-sequences of user navigation behaviors, and proposed an efficient pattern-growth WAP-mine algorithm to mine the Web access patterns from the WAP-tree. WAP-mine is a two-pass algorithm. Shan and Li [60] proposed a two-pass algorithm Fast-Walk to mine the Web traversal walks. A Web traversal walk is a structural sequence of forward and backward traversal paths. In Fast-Walk algorithm, an extended prefix-tree structure is constructed in main memory from Web logs, and the frequent Web traversal walks are generated from the in-memory tree structure efficiently.

Pabarskaite [56] suggested several hypotheses that could help improve the retention of Web site and proposed decision trees for Web user behaviour analysis. The decision tree package C4.5 is used in [56], and showed reasonable computational performance and accuracy. Xing and Shen [67] proposed two efficient algorithms UAM (User Access Matrix) and PNT (Preferred Navigation Tree) based on the concepts of selection and time preference for the mining of user preferred navigation patterns. Considering the Web site topology, UAM algorithm can obtain user preferred access paths by the page-page transition statistics of all the users' behaviours. The PNT is similar to WAP-tree. However, each node of PNT records

the support, which is the frequency and the time of user's visiting the node along the same route, and the preference represents how users prefer visiting this node to the previous nodes.

Web prefetching and prediction of HTTP requests are important applications of Web usage mining [15, 59]. Chen et al. [15] proposed a popularity-based PPM (Prediction by Partial Match model) for Web prefetching. The popularity-based mode uses grades (grades 3, 2, 1 and 0) to rank URL access patterns and builds these patterns into a predictor tree to aid Web prefetching. The popularity-based PPM uses only the most popular URLs as root nodes and makes space optimizations to the completed tree by removing non-root nodes and those nodes accessed only once. Schechter et al. [59] introduced the use of path profiles for describing HTTP request behavior and proposed an algorithm for creating these path profiles efficiently.

Association rule and sequential pattern mining algorithms are also common for mining Web visitors behaviours [3, 35, 16, 58, 50]. Agrawal and Srikant [3] proposed the well-known *Apriori* property, i.e., *all nonempty subsets of a frequent itemset must also be frequent*, and developed three multiple-pass algorithms based on the Apriori property for mining frequent itemsets by using candidate-generation-and-testing approaches. Han et al. [35] proposed a prefix-tree structure FP-tree (Frequent Pattern tree) and a two-pass pattern-growth algorithm FP-growth to discover the set of frequent itemsets without generating candidate itemsets. Chenug and Zaïane [16] proposed a data structure called CATS Tree (Compressed and Arranged Transaction Sequence Tree), an extension of FP-tree, to discover the set of frequent itemsets. The CATS tree is a prefix tree structure and it contains all elements of FP-tree including the header, the item links etc.

Pei et al. [58] proposed a two-pass, pattern-growth algorithm PrefixSpan (Prefix-projected Sequential pattern mining) to mine sequential patterns. PrefixSpan finds frequent 1-sequences, i.e., length-1 sequential patterns, after scanning the sequence database once.

Then, the database is projected into smaller datasets according to the frequent 1-sequences. Finally, the set of sequential patterns is found recursively by growing subsequence fragments in each projected database. Although PrefixSpan discovers sequential patterns efficiently, the cost of disk I/O might be high due to the creation and processing of the projected sub-databases. Hence, the two-pass algorithm PrefixSpan is not practical for mining streaming data. Lin and Lee [50] proposed a memory-indexing algorithm MEMISP (MEMory Indexing for Sequential Pattern mining) for fast discovery of sequential patterns. MEMISP reads data sequences into memory in one pass if the memory is enough to store these sequences. Then MEMISP discovers the sequential patterns by using a recursive find-then-index technique. Although MEMISP is a single-pass algorithm, it is still not feasible for mining patterns in a streaming data. This is because the MEMISP is not an incremental mining algorithm while the data stream is a continuous sequence of data elements.

In this chapter, an efficient, single-pass algorithm, called DSM-PLW (Data Stream Mining for Path traversal patterns in a Landmark Window), is proposed to mine the set of path traversal patterns in the landmark window of an online, continuous stream of Web click sequences. The purpose of mining patterns in a landmark window of data streams is to discover patterns over the entire history of the data streams [70]. An effective in-memory summary data structure, called SP-forest (Summary Path traversal pattern forest), is proposed for storing the essential information about the frequent reference sequences of the stream so far. Finally, the set of all maximal reference sequences, i.e., path traversal patterns, is determined from the SP-forest by a depth-first-search mining mechanism, called MRS-mining (Maximal Reference Sequence mining). To the best of our knowledge, this is the first study of online, single-pass mining path traversal patterns over streaming Web click-sequences.

The remainder of the chapter is organized as follows. The problem is defined in Section 5.2. In Section 5.3, we describe the proposed algorithm DSM-PLW. Theoretical analysis and

performance results are presented in Section 5.4. Finally, we conclude the chapter in Section 5.5.

5.2 Problem Definition: Online Mining of Path Traversal Patterns

Let S be an infinite sequence of Web clicks, where a Web click wc consists of a Web user identifier (Uid) and a Web page reference r accessed by the user, i.e., $wc = (Uid, r)$. In a streaming environment, a segment of Web click stream arrived at timestamp t_i can be divided into a set of Web click-sequences (or *click-sequences* in short). For example, a fragment of stream, $S = [t_i, (100, a), (100, b), (200, a), (100, c), (200, b), (200, c), (100, d), (100, e), (200, a), (200, e)]$, arrived at timestamp t_i , can be divided into two click-sequences: $\langle 100, abcde \rangle$, and $\langle 200, abcae \rangle$, where 100, 200 are user identifiers of Web users, and a, b, c, d, e are references accessed by these users. A **(Web) click-sequence**, CS , consists of a sequence of forward references and backward references accessed by a Web user. A **backward reference** means revisiting a previously visited reference by the same user.

A **maximal forward reference (MFR)** is a forward reference path without any backward references. Hence, a click-sequence with l backward references can be divided into $(l+1)$ maximal forward references. For example, a click-sequence $\langle abcae \rangle$ can be divided into two **MFRs**: $\langle abc \rangle$ and $\langle ae \rangle$, because the second reference a is a backward reference in this click-sequence. Therefore, we can map the problem of mining path traversal patterns into the one of finding frequent occurring consecutive sequences, called **reference sequences (RSs)**, among all maximal forward references. The **estimated support (esup)** of a reference sequence RS , denoted as $RS.esup$, is the number of maximal forward references in the stream containing RS as a substring.

A reference sequence RS is called a **frequent reference sequence** if $RS.esup \geq s \cdot N$, where s is a user-defined minimum support threshold in the range of $[0, 1]$, and N is the

current length of stream, i.e., the number of maximal forward references so far. A reference sequence s_1, s_2, \dots, s_n , is called a **super-sequence** of another reference sequence r_1, r_2, \dots, r_k if there exists an i such that $s_{i+j} = r_j$, for $1 \leq j \leq k$. A frequent reference sequence is called **maximal frequent reference sequence** (abbreviated as **maximal reference sequence** in the context of this chapter) if it is not a *substring* of any other frequent reference sequences.

Consequently, the problem of online, single-pass mining path traversal patterns in a landmark window over Web click-sequence streams is to mine maximal reference sequences by *one scan* of a continuous stream of maximal forward references when the value of minimum support threshold s is given.

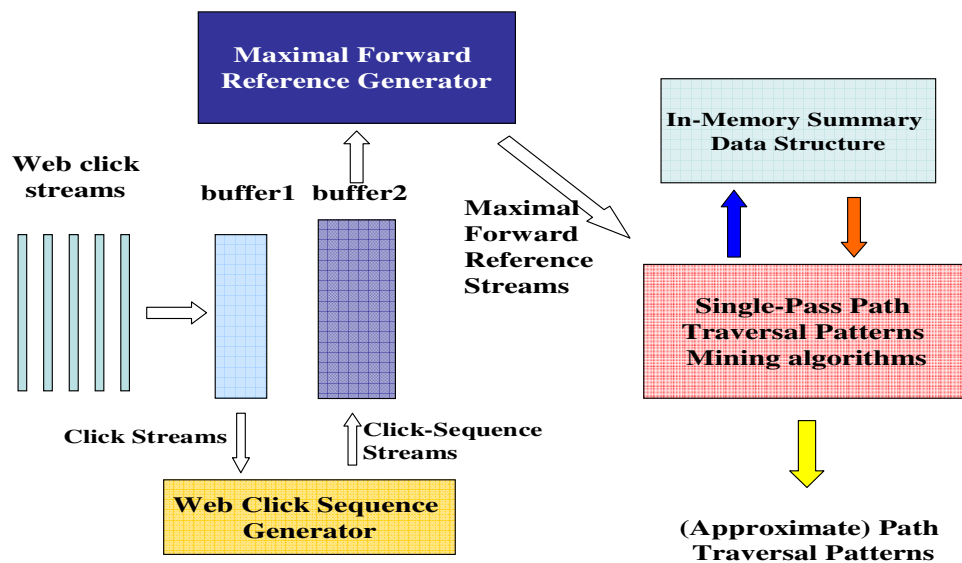


Figure 5- 1. Process of online mining of path traversal patterns in Web click streams

5.3 The Proposed Algorithm: DSM-PLW

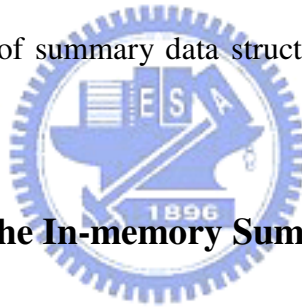
The process of mining path traversal patterns in Web click streams is shown in Figure 5-1.

Algorithm DSM-PLW (Data Stream Mining for Path traversal patterns in a Landmark

Window) is composed of four steps.

- (a) Read a basic window which consists of a fixed sized maximal forward references from the buffer in the main memory (Step 1).
- (b) Construct an in-memory summary data structure by processing each incoming basic window (Step 2).
- (c) Prune and maintain the summary data structure (Step 3).
- (d) Find the set of path traversal patterns from the current summary data structure (Step 4).

Steps 1 and 2 are performed in sequence for a new basic window. Steps 3 and 4 are usually performed periodically or when it is needed. Since the step 1 is straightforward, we shall henceforth focus on Steps 2, 3, and 4, and devise algorithms for the effective construction and maintenance of summary data structure, and efficient determination of the set of path traversal patterns.



5.3.1 Construction of the In-memory Summary Data Structure

In this section, a new in-memory summary data structure, called **SP-forest** (Summary Path traversal pattern forest), is proposed to store the essential information about path traversal patterns of each incoming basic window, and an efficient algorithm is proposed to construct the summary data structure. Then, we use a running example to illustrate.

Definition 5-1 A Summary Path traversal pattern forest (abbreviated as **SP-forest**) is a prefix tree-based summary data structure defined below.

1. SP-forest consists of a list of frequent references (denoted by **FR-list**), such as r_1, r_2, \dots, r_k , where $r_i.esup \geq s \cdot N$, and a set of Path traversal pattern tree (abbreviated as **Path-tree**) of references r_i , denoted by $r_i.Path-tree, \forall i = 1, 2, \dots, k$.
2. Each node in the $r_i.Path-tree, \forall i = 1, 2, \dots, k$, consists of four fields: $fr_id, esup, mfr_id,$

and *node-link*, where *fr_id* is the *identifier* of the incoming *forward reference*, *esup* registers the number of maximal forward references represented by a portion of the path reaching the node with the *fr_id*, the value of *mfr_id* assigned to a new node is the *identifier* of current *maximal forward reference*, and *node-link* links up a node with the next node with the same *f_id* in the SP-forest or null id if there is none.

3. Each entry $r_i, \forall i = 1, 2, \dots, k$, in the FR-list consists of four fields: *fr_id*, *esup*, *mfr_id*, and *head-link*, where *fr_id* registers the forward reference identifier the entry represents, *esup* records the number of maximal forward references in the stream so far containing the reference with identifier *fr_id*, *mfr_id* assigned to a new entry is the identifier of the current maximal forward reference, and *head-link* is a pointer pointing to the root node of the *fr_id*.Path-tree.

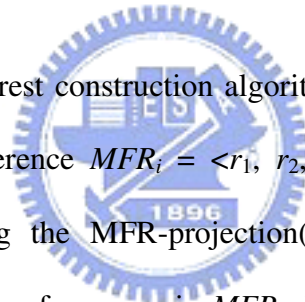


Figure 5-2 gives the SP-forest construction algorithm. First of all, DSM-PLW algorithm reads a maximal forward reference $MFR_i = \langle r_1, r_2, \dots, r_j, \dots, r_m \rangle$ from the buffer and maintains the SP-forest using the $MFR\text{-projection}(MFR_i)$. The maintenance process is described as follows. For each reference r_j in MFR_i , if the reference r_j exists in the current FR-list, the estimated support of the reference, i.e., $r_j.esup$, is increased by one. Otherwise, a new entry of the form $(r_j, 1, i, \rightarrow r_j)$ is created in the FR-list. Note that the notation $\rightarrow r_j$ indicates the *head-link* of r_j , and i is the current MFR's identifier. Next, MFR_i is projected into m *reference-suffix maximal forward references* (denoted by *rs-MFRs*) according to the order of references in the MFR_i . The step is called a *maximal forward reference projection*, and is denoted by $MFR\text{-projection}(MFR_i) = \{r_1|MFR_i, r_2|MFR_i, \dots, r_j|MFR_i, \dots, r_m|MFR_i\}$, where $r_j|MFR_i = \langle r_j r_{j+1} \dots r_m \rangle, \forall j = 1, 2, \dots, m$.

For example, a maximal forward reference $\langle acdef \rangle$ is projected into five reference-suffix maximal forward references: $\langle acdef \rangle$, $\langle cdef \rangle$, $\langle def \rangle$, $\langle ef \rangle$, and $\langle f \rangle$. Note that the cost of maximal forward reference projection is $(m^2+m)/2$, i.e., $m + (m-1) + \dots + 1$. Next, these

rs-MFRs with prefix r_i , $\forall i = 1, 2, \dots, m$, are inserted into the respective r_i .Path-tree as branches. If an rs-MFR shares a prefix with an MFR already in the Path-tree, the new MFR will share a prefix of the branch representing that MFR. In addition, an estimated support counter is associated with each node in the Path-tree. The counter is updated when a reference-suffix maximal forward reference causes the insertion of a new branch. Figure 5-3 shows the subroutines of SP-forest construction and maintenance.

Algorithm SP-forest construction

Input: A stream of maximal forward references, $MFR_1, MFR_2, \dots, MFR_N$, and a user-defined minimum support threshold $s \in (0, 1)$.

Output: A SP-forest so far.

1. FR-list = {}; /* initialize the FR-list to empty */
 2. **foreach** $MFR_i = \langle r_1, r_2, \dots, r_k \rangle$ **do** /* $\forall i = 1, 2, \dots, N$, where N is the identifier of current MFR */
 3. **foreach** reference $r_j \in MFR_i$ **do** /* $\forall j = 1, 2, \dots, k$ */
 4. **if** $r_j \notin$ FR-list **then**
 5. create a new entry of form $(r_j, 1, i, \rightarrow r_j)$ into the FR-list;
 6. **else**
 7. $r_j.esup = r_j.esup + 1$;
 8. **end if**
 9. **call** MFR-projection(MFR_i, r_j);
 10. **end for**
 11. **end for**
 12. **call** SP-pruning(SP-forest, N, s);
-

Figure 5- 2. Algorithm SP-forest construction

Subroutine MFR-projection

Input: A maximal forward reference $MFR_i = \langle r_1, r_2, \dots, r_j, \dots, r_m \rangle$.

Output: r_j .Path-tree, $\forall j = 1, 2, \dots, m$.

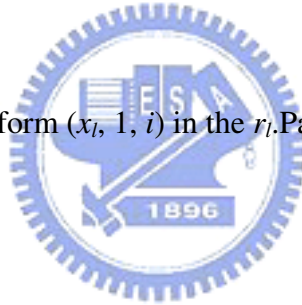
1. **foreach** reference r_j , $\forall j = 1, 2, \dots, m$, in MFR_i **do**
2. **call Path-tree-maintenance**($r_j|MFR_i, r_j$.Path-tree, i);
3. **end for**

Subroutine Path-tree-maintenance

Input: A reference-suffix maximal forward reference $r_j|MFR_i = \langle r_j r_{j+1} \dots r_m \rangle$, r_j .Path-tree, and the identifier of current maximal forward reference i ;

Output: A modified r_j .Path-tree, $\forall j = 1, 2, \dots, m$.

1. **foreach** reference r_l , $\forall l = j, j+1, \dots, m$, in $r_j|MFR_i$ **do**
2. **if** r_l .Path-tree has a child node with id y such that $y.fr_id = r_l.fr_id$ **then**
3. $y.esup = y.esup + 1$;
4. **else**
5. create a new node of form $(x_l, 1, i)$ in the r_l .Path-tree;
6. **end if**
7. **end for**



Subroutine SP-pruning

Input: A SP-forest, a user-defined minimum support threshold s in the range of $[0, 1]$, and the identifier of current maximal forward reference N .

Output: A SP-forest containing the set of all path traversal patterns.

1. **foreach** entry $r_j \in$ FR-list **do**
 2. **if** $r_j.esup < s \cdot N$ **then**
 3. delete r_j .Path-tree;
 4. delete r_j from FR-list;
 5. delete the sub-trees of a node whose fr_id is j in other r_l .Path-tree ($l \neq j$) by traversing the node-links in the SP-forest;
 6. **end if**
 7. **end for**
-

Figure 5- 3. Subroutines of SP-forest construction algorithm

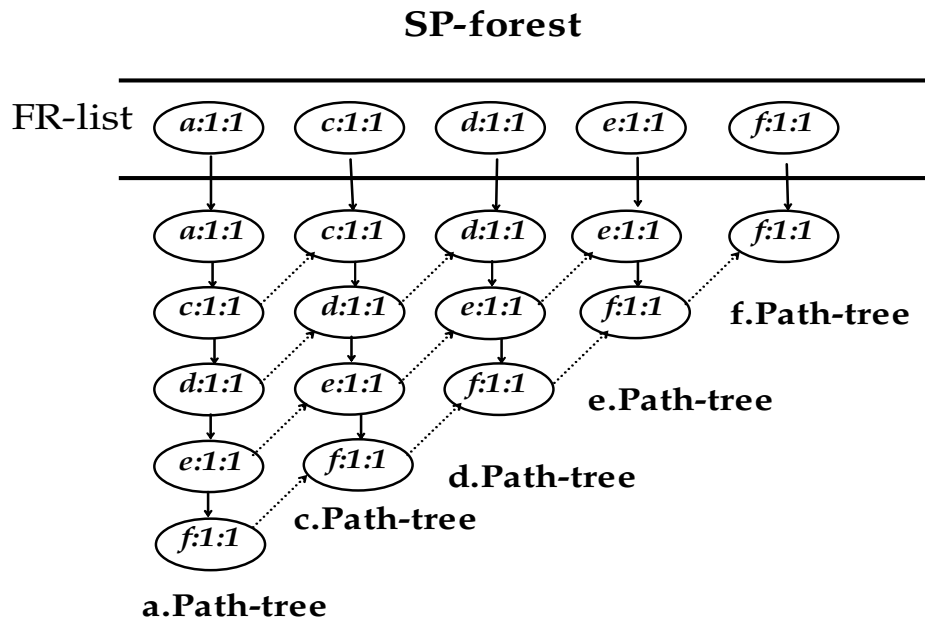


Figure 5- 4. SP-forest after processing the first maximal forward reference $\langle acdef \rangle$

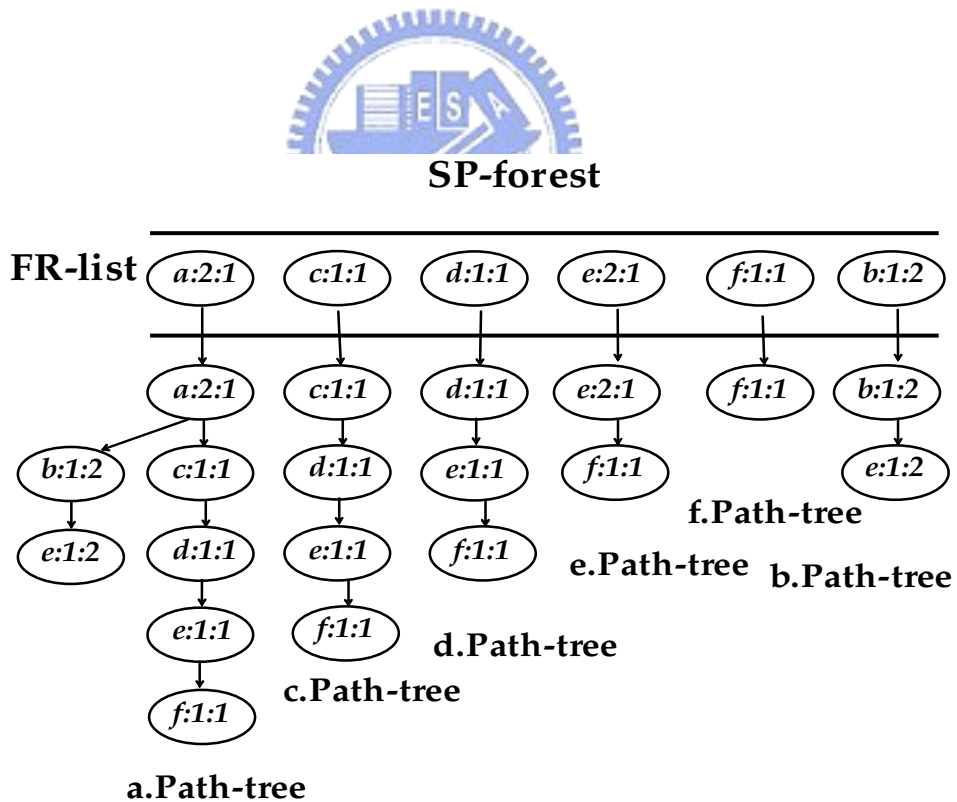


Figure 5- 5. SP-forest after processing the second maximal forward reference $\langle abe \rangle$

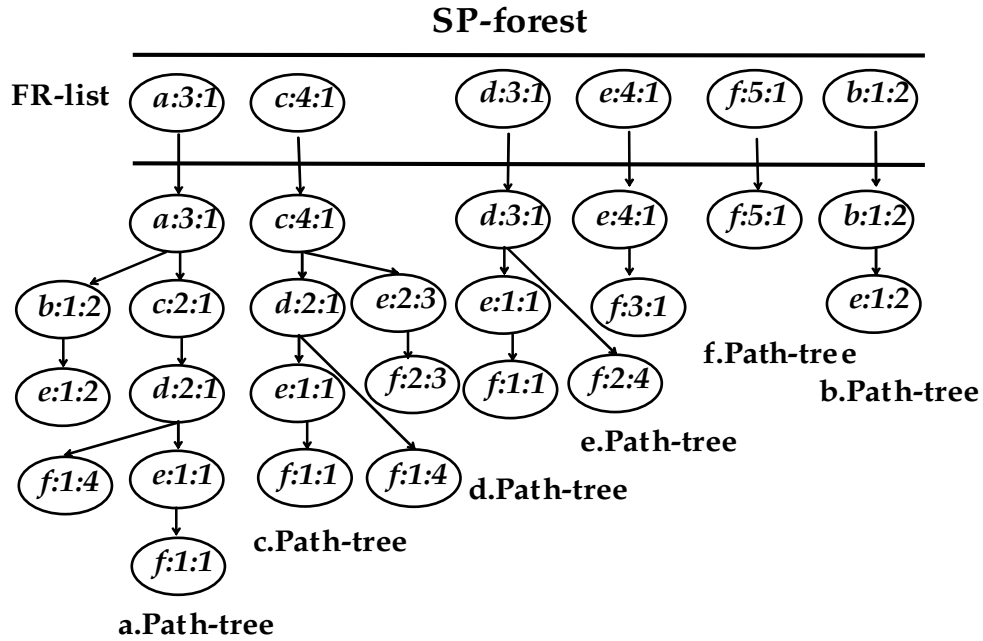


Figure 5- 6. SP-forest after processing the first six maximal forward references



Example 5-1 Let the first six maximal forward references in the stream of Web click-sequences be $\langle acdef \rangle$, $\langle abe \rangle$, $\langle cef \rangle$, $\langle acdf \rangle$, $\langle cef \rangle$, and $\langle df \rangle$, where a , b , c , d , e , and f are Web references. The SP-forest with respect to the first two MFRs, $\langle acdef \rangle$ and $\langle abe \rangle$, constructed by DSM-PLW algorithm is shown in Figure 5-4 and Figure 5-5, respectively. Note that the dotted-line arrows, *node-links*, in Figure 5-4 are used to link up a node with the next node of the same *fr_id* in the current SP-forest. However, in the following steps, as demonstrated in Figure 5-5 through Figure 5-7, the *node-links* are omitted for concise presentation.

First, DSM-PLW algorithm reads the first maximal forward reference $\langle acdef \rangle$ from the buffer, and projects it into five reference-suffix maximal forward references: $\langle acdef \rangle$, $\langle cdef \rangle$, $\langle def \rangle$, $\langle ef \rangle$, and $\langle f \rangle$. Next, the algorithm inserts $\langle acdef \rangle$, $\langle cdef \rangle$, $\langle def \rangle$, $\langle ef \rangle$, and $\langle f \rangle$ into the empty trees, i.e., *a.Path-tree*, *c.Path-tree*, *d.Path-tree*, *e.Path-tree*, and *f.Path-tree*, respectively. The step results in a single path in each Path-tree: $root(a:1:1) \rightarrow (a:1:1) \rightarrow (c:1:1)$

$\rightarrow (d:1:1) \rightarrow (e:1:1) \rightarrow (f:1:1)$, $root(c:1:1) \rightarrow (c:1:1) \rightarrow (d:1:1) \rightarrow (e:1:1) \rightarrow (f:1:1)$, $root(d:1:1) \rightarrow (d:1:1) \rightarrow (e:1:1) \rightarrow (f:1:1)$, $root(e:1:1) \rightarrow (e:1:1) \rightarrow (f:1:1)$, and $root(f:1:1) \rightarrow (f:1:1)$. The projected result is shown in Figure 5-4.

Then, DSM-PLW inserts the result of $MFR\text{-}projection(\langle abe \rangle)$: $\langle abe \rangle$, $\langle be \rangle$, and $\langle e \rangle$ into $a.Path\text{-}tree$, $b.Path\text{-}tree$, and $e.Path\text{-}tree$, respectively. Hence, $\langle abe \rangle$ leads to one path with a being the common prefix: $root(a:2:1) \rightarrow (a:2:1) \rightarrow (c:1:1) \rightarrow (d:1:1) \rightarrow (e:1:1) \rightarrow (f:1:1)$ and $root(a:2:1) \rightarrow (a:2:1) \rightarrow (b:1:2) \rightarrow (e:1:2)$. Then, $\langle be \rangle$ results in a single path in $b.Path\text{-}tree$: $root(b:1:2) \rightarrow (b:1:2) \rightarrow (e:1:2)$. Finally, DSM-PLW algorithm inserts $\langle e \rangle$ into the SP-forest. At this time, no new node is created, but the first path of $e.Path\text{-}tree$ is changed to: $root(e:2:1) \rightarrow (e:2:1) \rightarrow (f:1:1)$. After processing the second maximal forward reference $\langle abe \rangle$, the result is shown in Figure 5-5. After processing the six maximal forward references, the SP-forest is given in Figure 5-6.

5.3.2 Pruning Mechanism of the Summary Data Structure

According to the *Apriori* principle [3], only the frequent references are used to construct candidate k -RSs (k -reference-sequences) in the next pass, where $k > 1$. Thus, the set of candidates containing the infrequent references stored in SP-forest is pruned. The pruning is usually performed periodically or when it is needed.

Let the user-defined minimum support threshold be s in the range of $[0, 1]$, and the length of Web click-sequence stream be N , i.e., N maximal forward references. In the pruning mechanism of SP-forest, a reference sequence X and its super-sequences are deleted from SP-forest if $X.esup < s \cdot N$. For each entry of form $(fr_id, esup, mfr_id, \rightarrow fr_id)$ in the FR-list, if its $fr_id.esup$ is less than $s \cdot (N - mfr_id + 1)$, it can be regarded as an infrequent reference. Three operations are preformed in sequence. First, DSM-PLW deletes the $fr_id.Path\text{-}tree$. Second, it deletes the reference with id fr_id from the FR-list. Finally, DSM-PLW deletes the

infrequent reference with id fr_id and its suffix paths from other Path-trees by node-links. After pruning all infrequent references from SP-forest, SP-forest contains the set of all frequent path traversal patterns of the stream so far.

Example 5-2 Let the user-specified minimum support threshold be 0.3. Hence, a reference sequence X is called *infrequent* in Figure 5-6 if $X.esup < 0.3 \cdot 6 = 1.8$. At this time, only reference b ($b.esup = 1$) is infrequent by searching the current FR-list. Now, in order to maintain the frequent patterns in the SP-forest, DSM-PLW deletes b .Path-tree, b 's suffix paths from a .Path-tree, and b from the FR-list. The result is shown in Figure 5-8.

The next step of DSM-PLW algorithm is to determine the set of all path traversal patterns from SP-forest constructed so far. The step is performed only when the analytical results of the stream is requested.

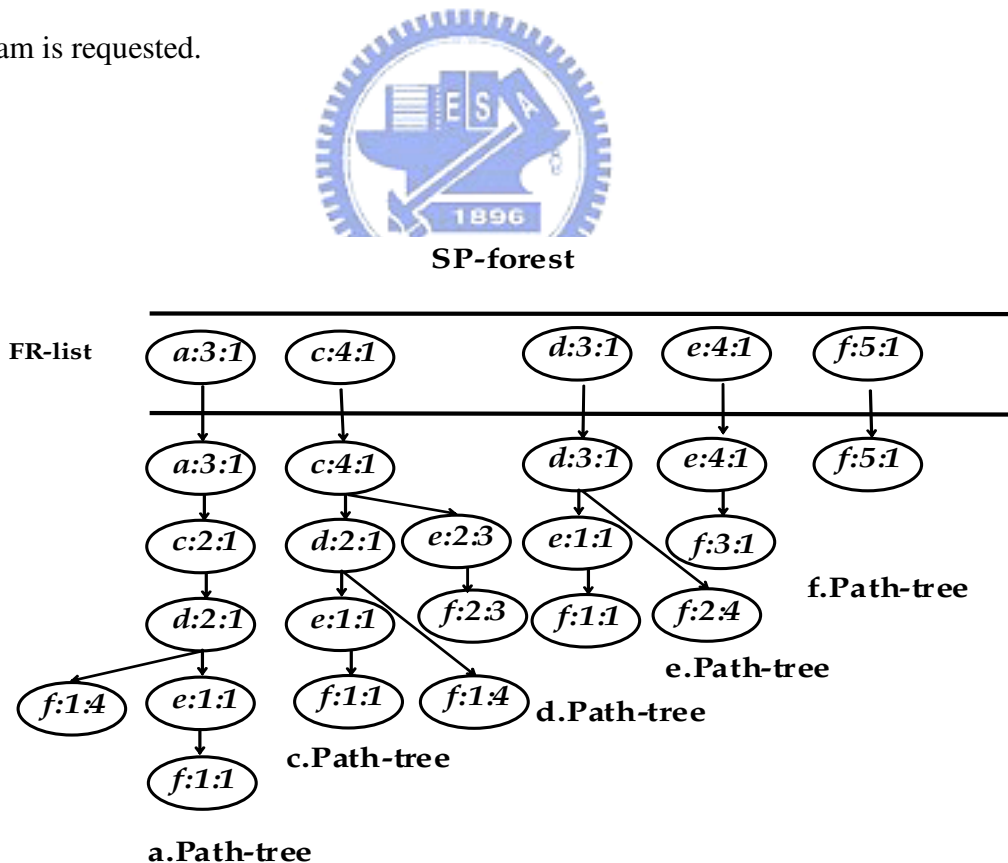


Figure 5- 7. SP-forest after pruning the infrequent reference b

5.3.3 Determination of Path Traversal Patterns from SP-forest

Assume that there are k frequent references, namely r_1, r_2, \dots, r_k , in the current FR-list. Let the minimum support threshold be s in the range of $[0, 1]$, and the current length of stream be N . For each entry $r_i, \forall i = 1, 2, \dots, k$, in the FR-list, DSM-PLW traverses the r_i -Path-tree to find the reference sequences with prefix r_i whose estimated support is greater than $s \cdot N$ in depth-first-search (DFS) manner. Then, DSM-PLW stores the maximal reference sequences in a temporal list, *MRS-list*. Finally, DSM-PLW outputs the set of path traversal patterns stored in the temporal list. Figure 5-8 gives the path traversal pattern mining algorithm, called *MRS-mining* (Maximal Reference Sequence mining).

Example 5-3 The example illustrates the mining of the path traversal patterns from the current SP-forest shown in Figure 5-7. Let the minimum support s be 0.3.

First, MRS-mining algorithm starts the path traversal pattern mining scheme from the first reference a in the FP-list, and generates a frequent reference sequence $\langle acd \rangle$ by DFS. MRS-mining adds $\langle acd \rangle$ into MRS-list because $\langle acd \rangle$ is not a substring of any other patterns stored in the current MRS-list. Next, on the second entry c , MRS-mining algorithm finds two frequent reference sequences: $\langle cd \rangle$ and $\langle cef \rangle$. However, only $\langle cef \rangle$ is added into the MRS-list. This is because $\langle cd \rangle$ is a substring of a generated maximal reference sequence $\langle acd \rangle$. On the third entry d , only one frequent reference sequence $\langle df \rangle$ is generated by MRS-mining, and stored into the MRS-list. On the fourth entry e , only one frequent reference sequence $\langle ef \rangle$ is generated, but it is not a maximal reference sequence. This is because $\langle ef \rangle$ is a substring of $\langle cef \rangle$. On the last entry f , only one frequent reference sequence $\langle f \rangle$ is obtained, but $\langle f \rangle$ is not a maximal reference sequence. This is because $\langle f \rangle$ is a substring of $\langle cef \rangle$. Finally, MRS-list contains the set of maximal reference sequences, i.e., *path traversal patterns*: $\langle acd \rangle$, $\langle cef \rangle$, and $\langle df \rangle$.

Algorithm MRS-mining

Input: A *SP-forest* constructed so far, the current length of maximal forward references N , and a user-defined minimum support threshold s in the range of $[0, 1]$.

Output: A temporal list of maximal reference sequences, *MRS-list*,

1. $MRS\text{-list} = \emptyset$;
2. **foreach** entry r_i in the current FR-list **do**
3. **do** *Depth-First-Search* to find the *esup* of each reference sequence Y with prefix r_i in the r_i .Path-tree;
4. **if** $Y.esup \geq s.N$ and Y is not a substring of any other frequent reference sequences stored in the MRS-list **then**
5. add Y into the MRS-list;
6. **end if**
7. **end for**
8. **if** $MRS\text{-list} \neq \emptyset$ **then**
9. **output** patterns from the MRS-list;
10. **end if**



Figure 5- 8. Algorithm MRS-mining

5.4 Performance Evaluation

To evaluate the performance of DSM-PLW algorithm, two experiments were performed. The experiments were carried out on the synthetic Web traversal path data generator proposed by Chen et al. [13]. In these experiments, a traversal tree is constructed to mimic a Web site structure whose starting position is a root node of the tree. The traversal tree is composed of internal nodes and leaf nodes. A traversal path consists of nodes accessed by a Web user. The size of each traversal path is picked from a Poisson distribution with mean equal to $|P|$, where $|P|$ is the average size of reference paths. With the first node being the root node, a

traversal path is generated probabilistically within the traversal tree as follows. Each edge connecting to an internal node is assigned with a weight. The weight corresponds to the probability that each edge will be accessed next by the Web user. The weight to its parent node is assigned with p_0 , which is generally $1/(n+1)$ where n is the number of child nodes. The probability of traveling to each child node, p_i , is determined from an exponential distribution with unit mean. Moreover, the probability is normalized that the sum of the weights for all child nodes is equal to $1-p_0$. When the path arrives at a leaf node, the next move would be either to its parent node in backward (with a default probability 0.25) or to any internal node (with an aggregate probability 0.75). More detail about the generation of synthetic traversal paths can be found in [13].

Three synthetic data streams, H10P5.D200K, H10P10.D200K, and H10P15.D200K, of size 200,000 reference paths are studied. HxPy means that x is the height of a traversal tree, and y is the average size of the reference paths. D200K means that the number of reference paths is 200,000. A traversal tree for H10 was obtained when the height of the tree is 10, and the fanout at each internal node is between 4 and 7. The root node consists of 7 child nodes. Moreover, the number of internal nodes is 16,200 and the number of leaf nodes is 73,006. In all experiments, the click-sequences of each datasets are looked up in sequence to simulate the environment of a data stream. All the experiments are performed on a 1.80 GHz Pentium 4 processor with 512 megabytes main memory, running on Microsoft Windows 2000. In addition, all the programs are written in Microsoft/Visual C++ 6.0.

5.4.1 Experimental Results of Synthetic Data

We first evaluated the effect of various minimum support threshold s for synthetic data streams having a typical value of 200,000 (200K) reference paths. In Figure 5-9, we plot total execution time taken by our algorithm for minimum support threshold s ranging from 0.2% to

1%. The figure shows how decreasing s leads to increase in running time. Figure 5-10 shows how decreasing s leads to increase in memory usage. The memory usage shown in Figure 5-10 (a) is the memory requirement in Steps 2 and 3 of DSM-PLW algorithm, and Figure 5-10 (b) is the total memory requirement of DSM-PLW algorithm in Steps 2, 3, and 4.

To measure the relative accuracy of DSM-PLW algorithm, an *average support error ASE* proposed in [10] is used. Figure 5-11 shows the average support error of the mining results of the proposed algorithm with respect to that of the FS algorithm [13] performed on the synthetic streaming data by varying the user-specified minimum support threshold s . Generally, the average support error increases as the value of s increases in Figure 5-11.

To assess the scalability of our algorithm, scale-up experiments were conducted. Figure 5-12 shows that the execution time of DSM-PLW increases linearly as the streaming data size increases, ranging from 200K to 1000K. Different minimum support thresholds s yield similar and consistent results. The result of $s = 0.2\%$ is shown in Figure 5-12, and it exhibits good linearity in scale-up.

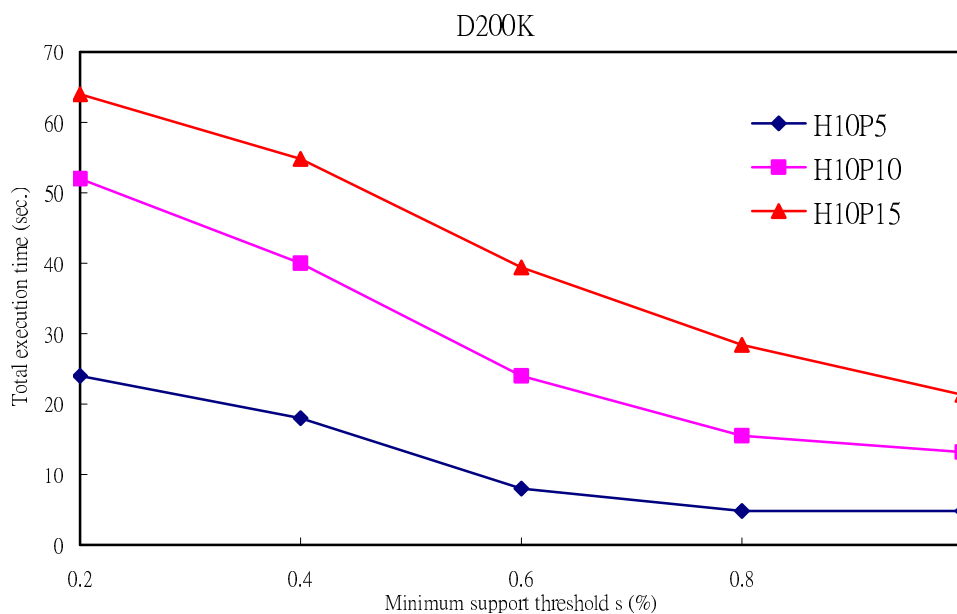
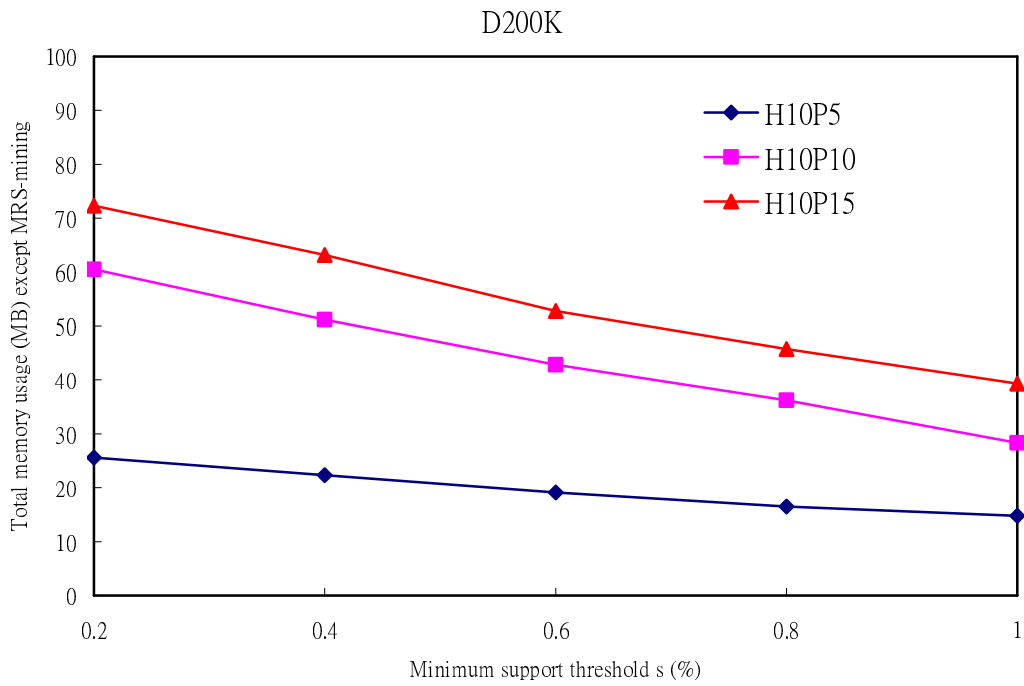
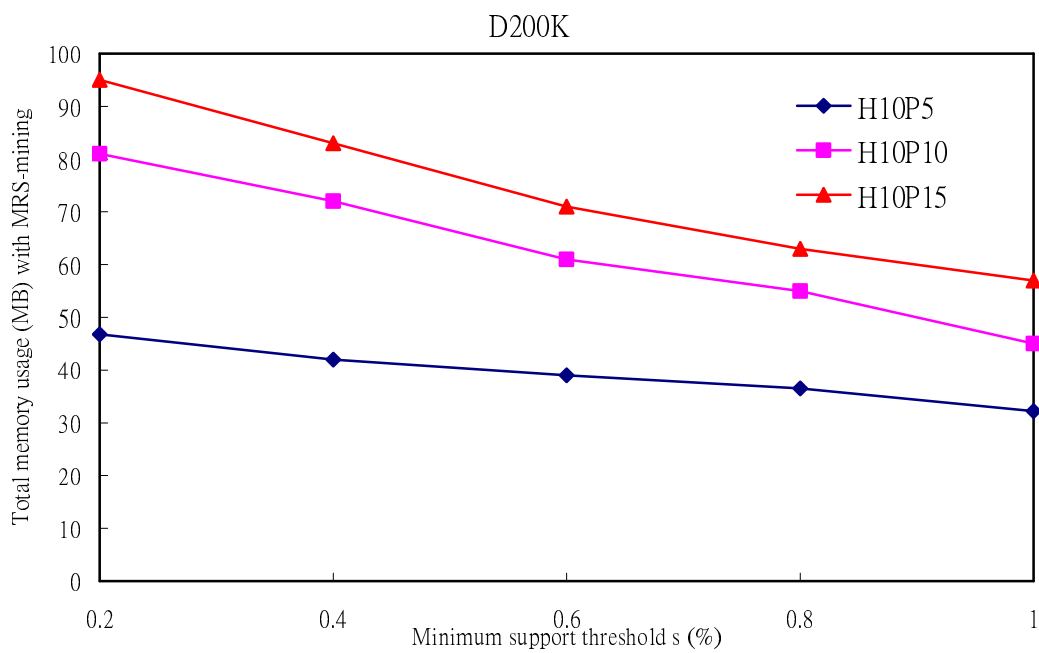


Figure 5- 9. Performance comparisons of total execution time over various minimum support thresholds



(a) without MRS-mining



(b) with MRS-mining

Figure 5- 10. Performance comparisons of memory usage over various minimum support thresholds

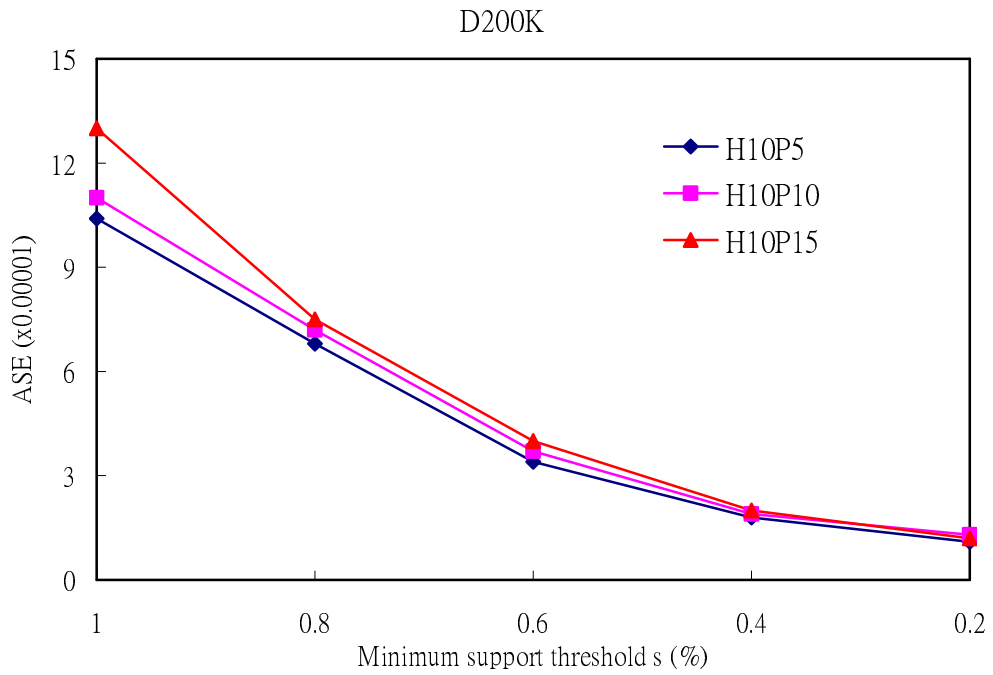


Figure 5- 11. Accuracy of mining results

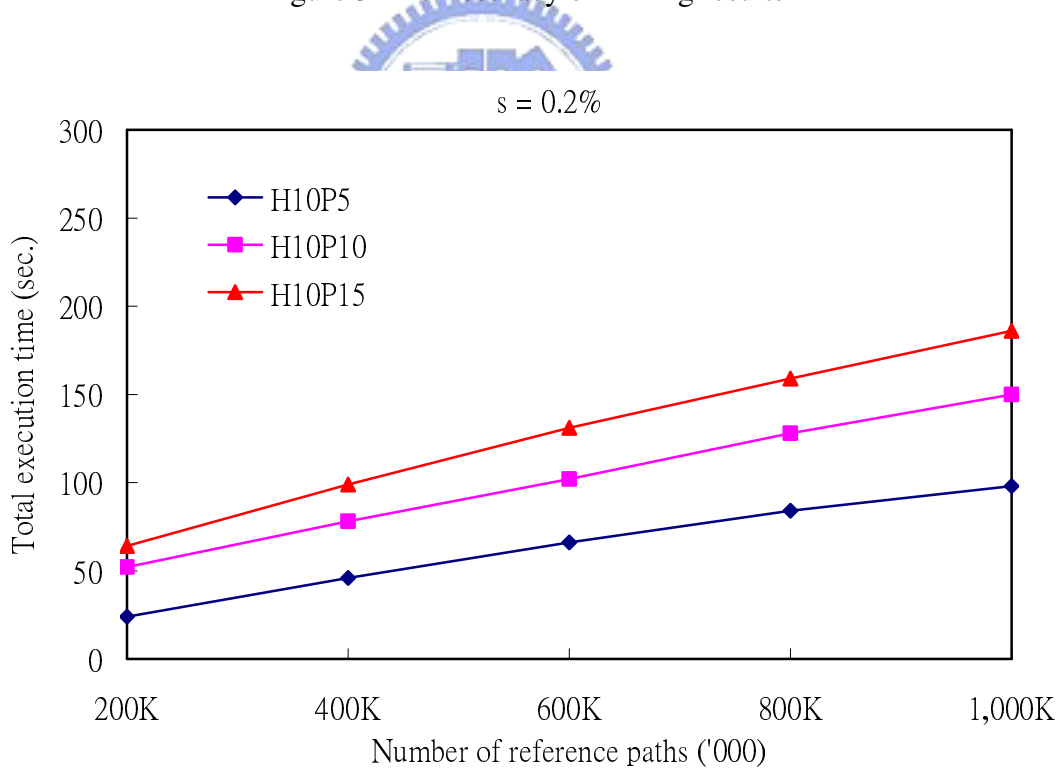
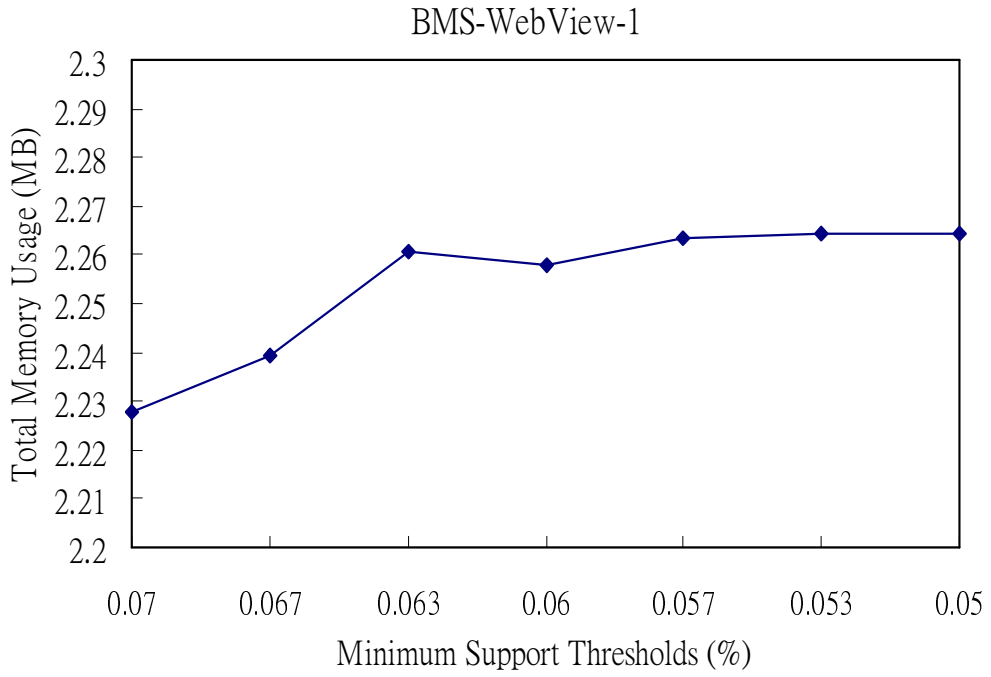
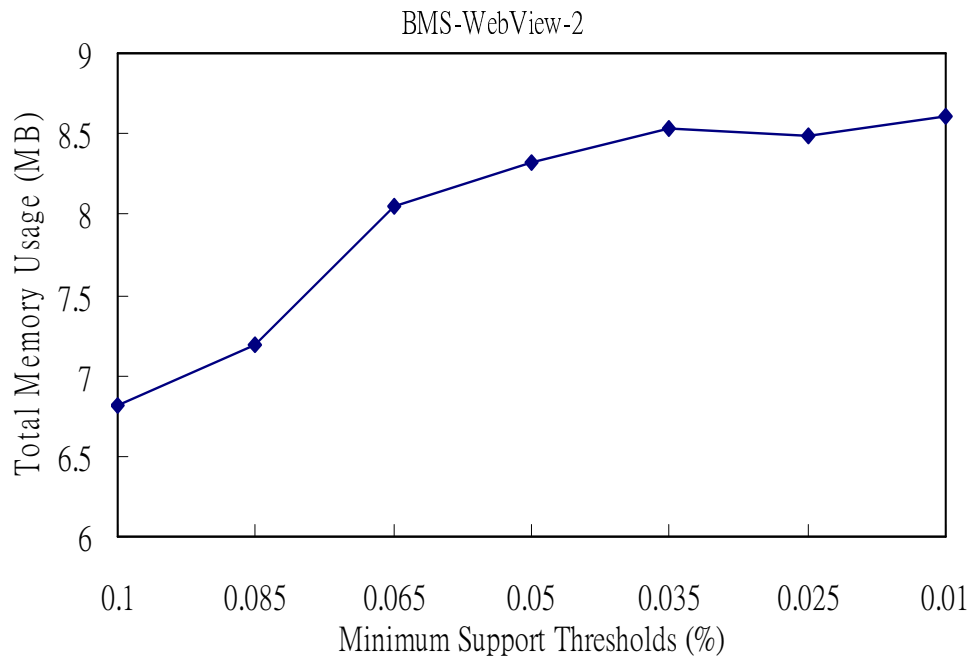


Figure 5- 12. Linear scalability of the streaming data size

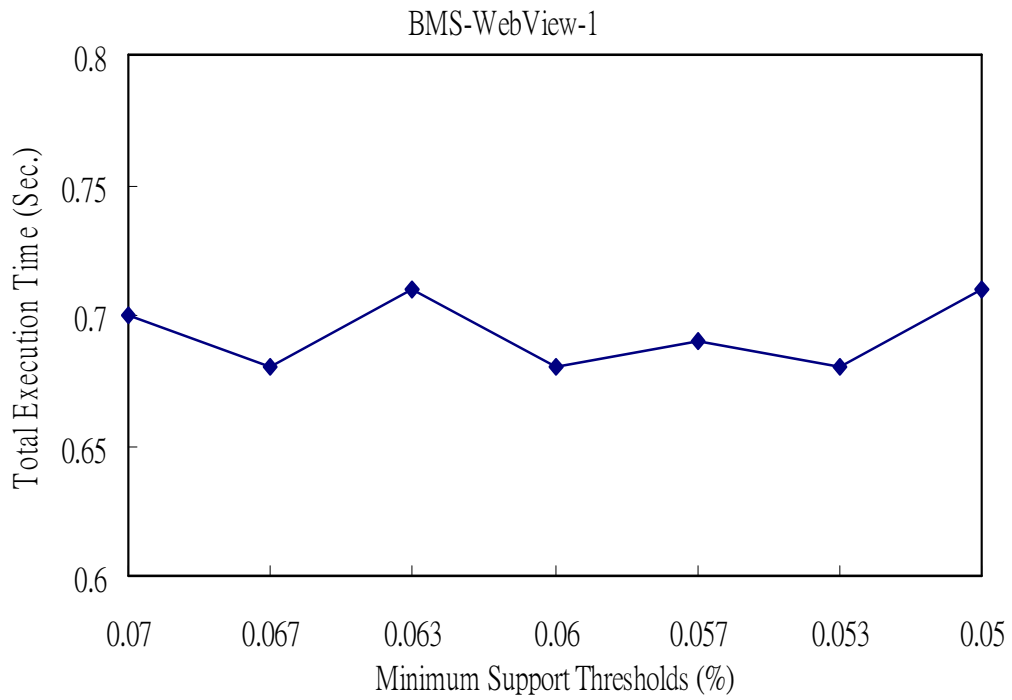


(a) Memory usage on BMS-WebView-1

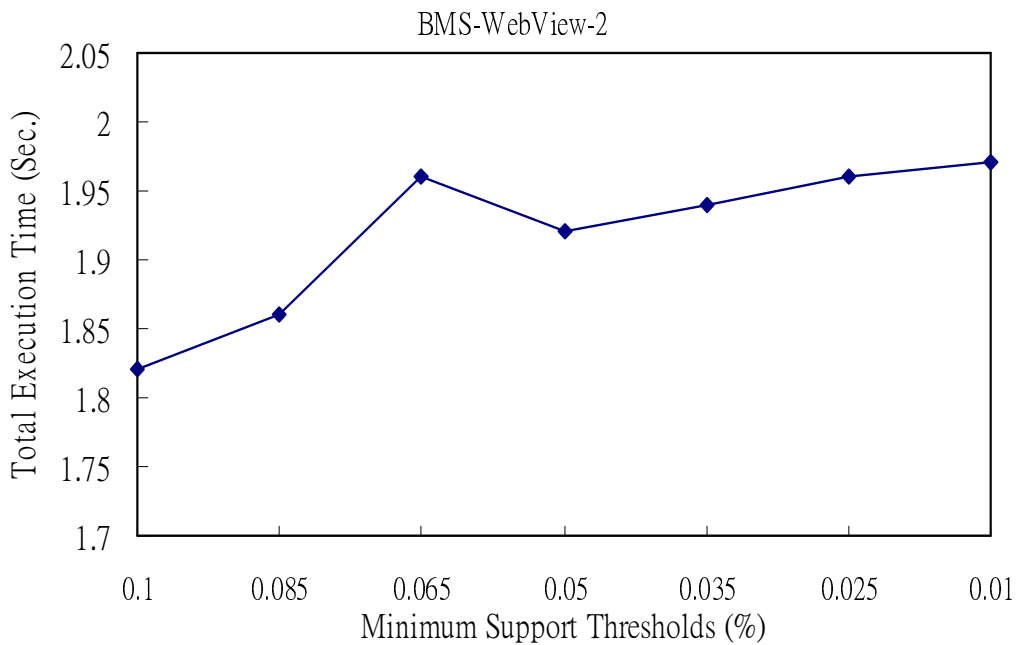


(b) Memory usage on BMS-WebView-2

Figure 5- 13. Memory usage of DSM-PLW on BMS-WebView-1 and BMS-WebView-2 over various minimum support thresholds



(a) Execution time on BMS-WebView-1



(b) Execution Time on BMS-WebView-2

Figure 5- 14. Execution time of DSM-PLW on BMS-WebView-1 and BMS-WebView-2 over various minimum support thresholds

5.4.2 Experimental Results of Real Data

Two real click-stream datasets, BMS-WebView-1 and BMS-WebView-2, which contain several months worth of click-stream data from two e-commerce web sites, are used to evaluate the performance of the DSM-PLW algorithm. The real data was provided by Blue Martini Software [69], and is available from the KDD Cup 2000 home page [71]. The BMS-WebView-1 dataset consists of 497 items and 59,602 transactions. The maximum transaction size of BMS-WebView-1 is 267 distinct items and the average transaction size is 2.5 items. The BMS-WebView-2 dataset consists of 3,340 distinct items and 77,512 transactions. The maximum transaction size of BMS-WebView-2 is 161 items and the average transaction size is 5 items. Note that an item is regarded as a reference and a transaction is regarded as a maximal forward reference in these experiments.

In the experiments, two major factors, *memory* and *execution time*, are examined in the online, single-pass mining path traversal patterns of streaming Web click-sequences, since both should be bounded online as time advances. As shown in Figure 5-13, the memory usage of DSM-PLW algorithm is relatively insensitive to the minimum support thresholds. As the support decreases, the memory consumption of DSM-PLW increases stably, indicating the feasibility of the proposed algorithm. In Figure 5-14, the execution time of DSM-PLW grows smoothly as the support decreases for both real datasets. Hence, the experiments show that DSM-PLW algorithm is a practical scheme to mine the set of path traversal patterns in real data.

5.5 Conclusions

In this chapter, a new interesting research problem of Web usage mining, namely, *online single pass mining path traversal patterns in streaming Web click-sequences* is presented. A new single-pass algorithm, called DSM-PLW (Data Stream Mining for Path traversal patterns

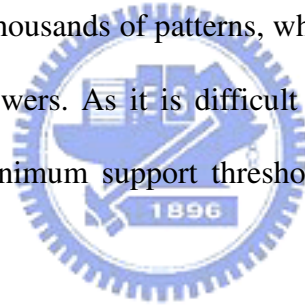
in a Landmark Window), is developed to discover the set of all path traversal patterns over the entire history of continuous stream of Web click-sequences. In the proposed DSM-PLW algorithm, an effective in-memory summary data structure, called SP-forest (Summary Path traversal pattern forest), is developed to maintain the essential information of all maximal reference sequences in the stream so far. The set of all maximal reference sequences, i.e., path traversal patterns, is determined from the SP-forest by a depth-first-search mechanism, called MRS-mining (Maximal Reference Sequence mining). Experimental results show that DSM-PLW can meet the performance requirements of data stream mining: *single-pass*, *bounded space*, and *real time*.



Chapter 6 Online Mining of Top-K Path Traversal Patterns over Web Click-Streams

In this chapter, we study the problem of mining top- k path traversal patterns over Web click-streams. In the framework of DSM-PLW algorithm as discussed in Chapter 5, it requires a user-specified minimum support threshold $minsup$, and then mines path traversal patterns with estimated support values that are higher than the minimum support threshold. Unfortunately, the setting of minimum support threshold is quite tricky and it leads to the following problem that may hinder its popular use.

If the value of minimum support threshold is too small, the pattern mining algorithm may lead to the generation of thousands of patterns, whereas a too big one may often generate a few patterns or even no answers. As it is difficult to predict how many patterns will be mined with a user-defined minimum support threshold, the top- k pattern mining has been proposed.



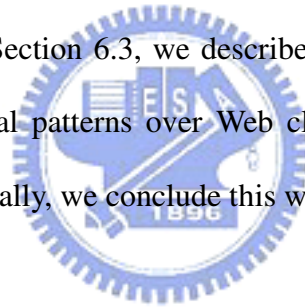
6.1 Introduction

The first top- k pattern mining algorithm Itemset-Loop was proposed by Fu et al. [28]. Itemset-Loop algorithm mines the k most frequent itemsets with lengths shorter than a user-defined value of m . LOOPBACK and BOMO [17] are top- k pattern mining algorithms based on a FP-tree structure, and uses the same estimated mechanism of Itemset-Loop. Moreover, experiments in [17] show that LOOPBACK and BOMO outperform the Itemset-Loop. TFP algorithm [66] is a FP-tree-based algorithm and mines the top- k closed frequent itemsets with lengths longer than a user-specified value of min_l . TSP [65] is the first algorithm to mine the top- k closed sequential patterns of lengths no less than the user-defined

minimum length of mined patterns min_l .

Recently, Metwally et al. [54] proposed a single-pass algorithm to mine the top- k elements over data streams. However, the top- k elements are top- k items. In this chapter, we propose an efficient single-pass algorithm, called DSM-TKP (Data Stream Mining for Top-K Path traversal patterns), to mine the top- k path traversal patterns over Web click streams. An effective summary data structure, called TKP-forest (Top-K Path forest), and an efficient structure pruning mechanism, called KP (K Pruning), are proposed to overcome the data stream mining issues such as bounded space requirement and approximation. Based on our knowledge, DSM-TKP is the first single-pass algorithm for mining top- k path traversal pattern over streaming click-data.

The remainder of the chapter is organized as follows. The problem definition is introduced in Section 6.2. In Section 6.3, we describe the design of our proposed algorithm for mining top- K path traversal patterns over Web click-sequence streams. We discuss the experiments in Section 6.4. Finally, we conclude this work in Section 6.5.



6.2 Problem Definition

Let S be a continuous steam of Web clicks, where a Web click wc consists of Web user identifier (Uid) and a Web page reference r accessed by the user, i.e., $wc = (Uid, r)$. In a steaming environment, a segment of Web click stream arrived at timestamp t_i can be divided into a set of Web click-sequences (or *click-sequences* in short). For example, a fragment of stream, $S = [t_i, (100, a), (100, b), (200, a), (100, c), (200, b), (200, c), (100, d), (100, e), (200, a), (200, e)]$, arrived at timestamp t_i , can be divided into two click-sequences: $\langle 100, abcde \rangle$, and $\langle 200, abcae \rangle$, where 100, 200 are identifiers of Web users, and a, b, c, d, e are references accessed by these users. A **(Web) click-sequence** CS consists of a sequence of forward

references and backward references accessed by a Web user. A **backward reference** means revisiting a previously visited reference by the same user. A **maximal forward reference** (MFR) is a forward reference path without any backward references. Hence, a click-sequence can be divided into several maximal forward references, i.e., $CS = MFR_1, MFR_2, \dots, MFR_i$, where $i \geq 1$. For example, a click-sequence $\langle abcae \rangle$ can be divided into two MFRs: $\langle abc \rangle$ and $\langle ae \rangle$. Therefore, we can map the problem of mining top- k path traversal patterns into the problem of finding top- k occurring consecutive sequences, called **reference sequences** (RSs), among all maximal forward references. The **support** of a reference sequence RS , denoted as $sup(RS)$, is the number of maximal forward references in the stream containing RS as a substring. A reference sequence is called **maximal** if it is not a substring of any other reference sequences. A maximal reference sequence is also called a **path traversal pattern**. A reference sequence RS is a **top- k maximal reference sequence** if there exists no more than $(k-1)$ maximal reference sequences whose support is higher than that of RS . In this chapter, our task is to mine the top- k maximal reference sequences by one scan of a continuous stream of Web clicks when the value of k is given.

6.3 The Proposed Algorithm: DSM-TKP

The proposed algorithm DSM-TKP (Data Stream Mining for Top-K Path traversal patterns) is composed of four steps.

- (a) Read a maximal forward reference from the buffer in the main memory (Step 1).
- (b) Construct an in-memory summary data structure (Step 2).
- (c) Prune and maintain the summary data structure (Step 3).
- (d) Find the path traversal patterns from the summary data structure so far (Step 4).

Steps 1 and 2 are performed in sequence for a new maximal forward reference. Steps 3 and 4 are usually performed periodically or when it is needed. Since the step 1 is

straightforward, we shall henceforth focus on steps 2, 3, and 4, and devise algorithms for effective construction and maintenance of summary data structure, and efficient determination of path traversal patterns.

6.3.1 Effective Construction of the Summary Data Structure

In this section, we describe an algorithm which constructs the in-memory summary data structure, called *Top-K Path forest*.

Definition 6-1 A *Top-K Path forest* (abbreviated as **TKP-forest**) is a prefix tree-based summary data structure defined below.

1. *TKP-forest* consists of a *K-References list* (abbreviated as **KR-list**), such as $\langle r_1 r_2 \dots r_k \rangle$, and a set of *Local Path traversal pattern trees* (abbreviated as **LP-trees**) of references, denoted by r_i .LP-tree, $\forall i = 1, 2, \dots, k$, where r_i is the root node of r_i .LP-tree.
2. Each node in the r_i .LP-tree, $\forall i = 1, 2, \dots, k$, consists of four fields: *fid*, *esup*, *mfr_id*, and *node-link*, where *fid* is the identifier of the incoming maximal forward reference, *esup* registers the number of maximal forward references represented by a portion of the path reaching the node with the *fid*, the value of *mfr_id* assigned to a new node is the identifier of current maximal forward reference, and *node-link* links up a node with the next node with the same *fid* in the same LP-tree or null if there is none.
3. Each entry in the *KR-list* consists of four fields: *fid*, *esup*, *mfr_id*, and *head-link*, where *fid* registers which reference identifier the entry represents, *esup* records the number of maximal forward references containing the reference carrying the reference id, the *mfr_id* assigned to a new entry is the identifier of current maximal forward reference, and *head-link* is a pointer, and points to the root node of the *fid*.LP-tree.

The construction algorithm of TKP-forest is shown in Figure 6-1. The scenario of TKP-forest construction is described as follows. First of all, DSM-TKP reads a maximal forward reference $MFR = \langle r_1 r_2 \dots r_m \rangle$, from the buffer, projects the MFR into m sub-maximal forward references (abbreviated as **sub-MFRs**), and inserts these sub-MFRs into the TKP-forest as branches. Note that m is the number of references in the maximal forward reference. The projection of each incoming maximal forward reference is described as follows. Each maximal forward reference, $MFR = \langle r_1 r_2 \dots r_m \rangle$, is converted into m sub-MFRs; that is, $\langle r_1 r_2 \dots r_m \rangle$, $\langle r_2 r_3 \dots r_m \rangle$, ..., and $\langle r_m \rangle$. These m sequences are called *reference-suffix maximal forward references* (abbreviated as **rs-MFRs**), since the first reference of each sequence is a suffix of the original maximal forward reference. The projection step is called *maximal forward reference projection*, and denoted by **MFR-projection** (MFR) = $\{r_1|MFR, r_2|MFR, \dots, r_i|MFR, \dots, r_m|MFR\}$, where $r_i|MFR = \langle r_i r_{i+1} \dots r_m \rangle$, $\forall i = 1, 2, \dots, m$. The cost of this projection is $(m^2+m)/2$, i.e., $m + (m-1) + \dots + 2 + 1$.

After performing the MFR-projection, DSM-TKP algorithm inserts the MFR into the KR-list, and then removes it from the buffer in the main memory. Next, the set of rs-MFRs are inserted into the r_i -LP-trees ($\forall i = 1, 2, \dots, m$) as branches. If a MFR shares a prefix with a MFR already in the LP-tree, the new MFR will share a prefix of the branch representing that MFR. Moreover, an estimated support counter is associated with each node in the tree. The counter is updated when a rs-MFR causes the insertion of a new branch. The step is called the *rs-MFR insertion*.

Example 6-1. Let the first six maximal forward references be $\langle abcde \rangle$, $\langle acd \rangle$, $\langle cef \rangle$, $\langle acdf \rangle$, $\langle cef \rangle$, and $\langle df \rangle$, where a, b, c, d, e and f are references in the stream. The TKP-forest with respect to the first two MFRs, $\langle abcde \rangle$ and $\langle acd \rangle$, constructed by DSM-TKP algorithm is shown in Figure 6-2 and Figure 6-3, respectively.

Algorithm TKP-forest construction

Input: A continuous stream of maximal forward references, $S = [MFR_1, MFR_2, \dots, MFR_N]$, a user-specified value k .

Output: A TKP-forest generated so far.

```
1: KR-list = {}; /*initialize the KR-list to empty.*/
2: foreach  $MFR_i = \langle x_1 x_2 \dots x_m \rangle$ , do
    /*  $m \geq 1, i=1, 2, \dots, N$  */
3:   foreach reference  $x_j \in MFR_i$  do
4:     if  $x_j \notin$  KR-list then
5:       create a new entry of form  $(x_j, 1, i, head-link)$ 
        into the KR-list;
6:     else /* the entry already exists in the KR-list */
7:        $x_j.esup = x_j.esup + 1$ ;
8:     end if
9:   end for
10:  call MFR-Projection( $MFR_i$ );
11:  call rs-MFR insertion;
12: end for
13: call TKP-forest-pruning( $TKP-forest, k$ );
    /* Step 3 of DSM-TKP algorithm: prune and maintain the summary data structure */
14: end for
```

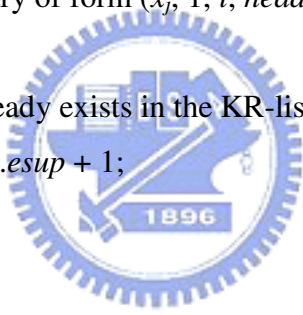


Figure 6- 1. Algorithm of TKP-forest Construction

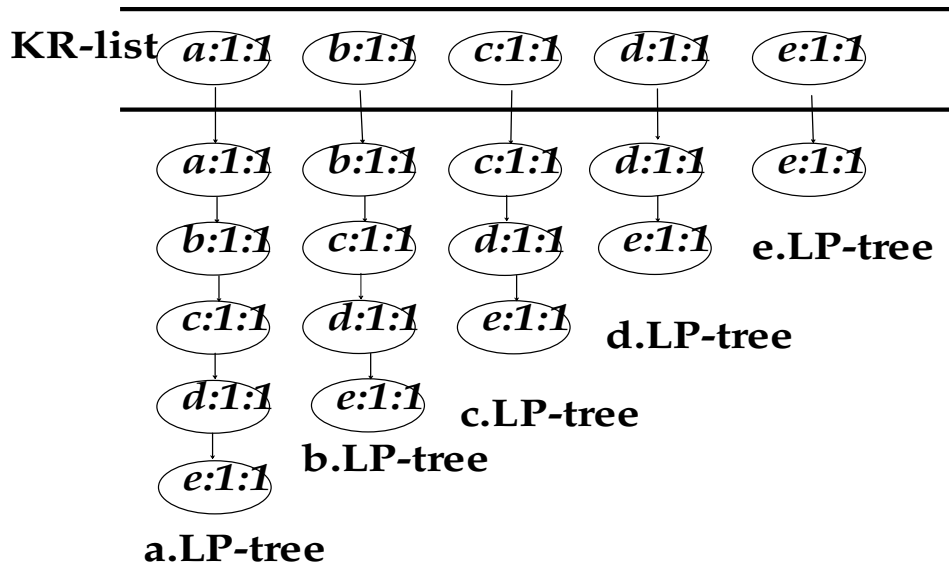


Figure 6- 2. TKP-forest construction after processing the first maximal forward reference $\langle abcde \rangle$

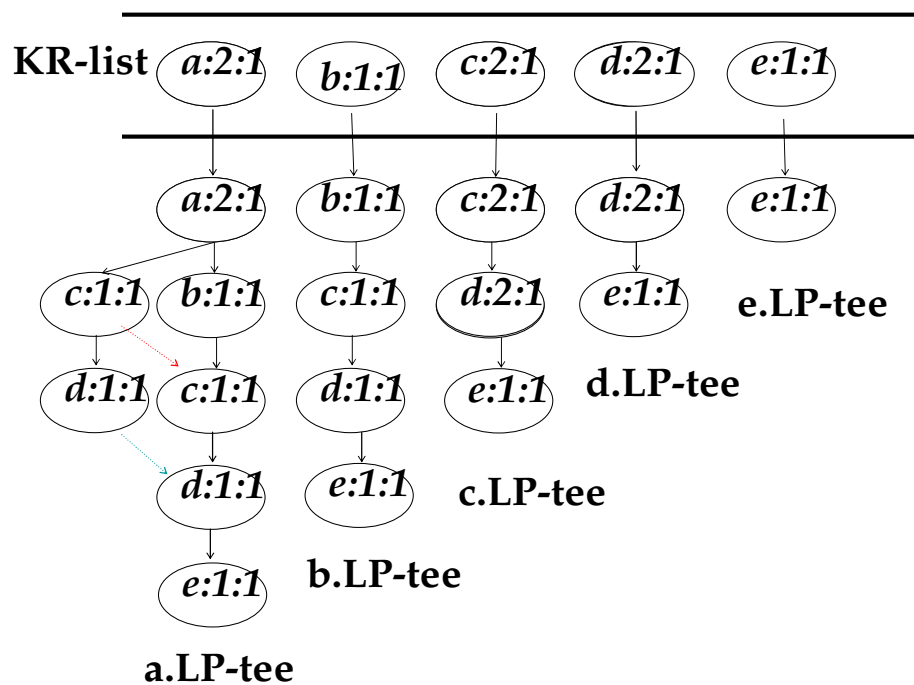


Figure 6- 3. TKP-forest construction after processing the second maximal forward reference $\langle acd \rangle$

6.3.2 Effective Pruning of the Summary Data Structure

The TKP-forest pruning mechanism used in DSM-TKP is performed when the number of references in the KR-list is greater than the value k . The pruning mechanism used in DSM-TKP algorithm is shown in Figure 6-4.

The next step of DSM-TKP algorithm is to determine the top- k path traversal patterns from the current TKP-forest. The step is performed only when the analytical results of the stream is requested.

Subroutine TKP-forest-pruning(*TKP-forest*, k)

- 1: **sort** the references, r_1, r_2, \dots, r_k , in the KR-list and **reorder** the references in an *estimated support decreasing order*, i.e., $r_1', r_2', \dots, r_{k'}$, where $sup(r_1') \geq sup(r_2') \geq \dots \geq sup(r_{k'})$;
- 2: **find** r_{KL}' in the reordered KR-list;
/* r_{KL}' be a reference whose estimated support is the k -th largest one in the KR-list; */
- 3: **foreach** $r_i' \in$ KR-list, $\forall i = 1, 2, \dots, KL$ **do**
- 4: $esup(r_i') = esup(r_i') - esup(r_{KL-1}')$;
- 5: **endfor**
- 6: **foreach** $r_j' \in$ KR-list, $\forall j = KL+1, KL+2, \dots, k'$ **do**
- 7: delete r_j' from the current KR-list;
- 8: delete r_j' .LP-tree;
- 9: **endfor**

Figure 6- 4. Algorithm of TKP-forest pruning

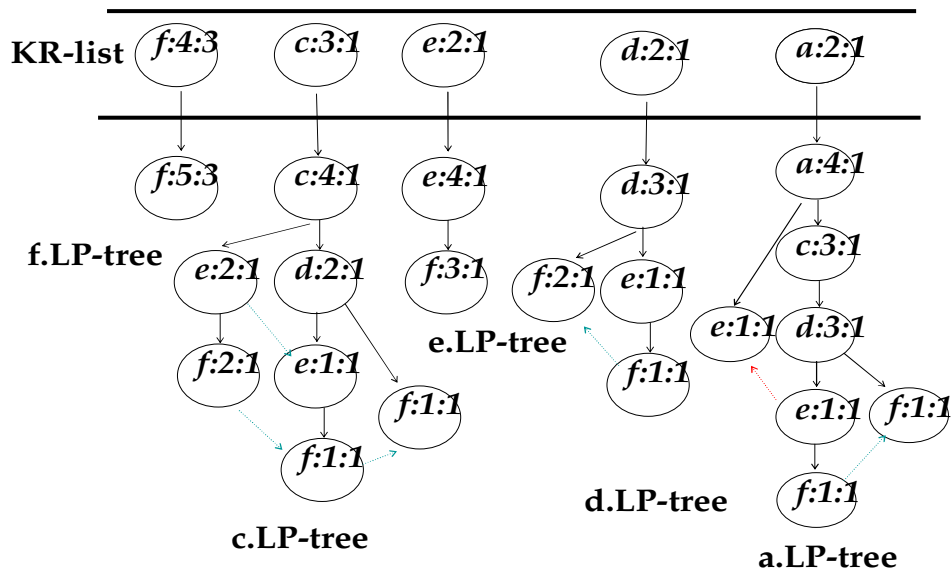


Figure 6- 5. Example of TKP-forest

6.3.3 Determination of the Top-K Path Traversal Patterns

Assume that there are k references, namely r_1, r_2, \dots, r_k , in the current KR-list. For each entry $r_i, \forall i = 1, 2, \dots, k$, in the KR-list, DSM-TKP algorithm traverses the r_i .LP-tree to find the estimated support of each reference sequence with a prefix r_i in a depth-first-search (DFS) manner. Then, DSM-TKP stores these reference sequences into a temporal list of candidate maximal reference sequences, i.e., path traversal patterns, in a support decreasing order. Finally, DSM-TKP outputs the first k maximal reference sequences from the temporal list. For example, in Figure 6-5, the top-3 path traversal patterns are $\langle acd: 3 \rangle$, $\langle cef: 2 \rangle$, and $\langle df: 2 \rangle$, where the 3-th largest estimated support in the reordered KR-list is 2.

6.4 Performance Evaluation

All the experiments are performed on a 1.80 GHz Pentium 4 processor with 512 megabytes main memory, running on Microsoft Windows 2000. In addition, all the programs are written

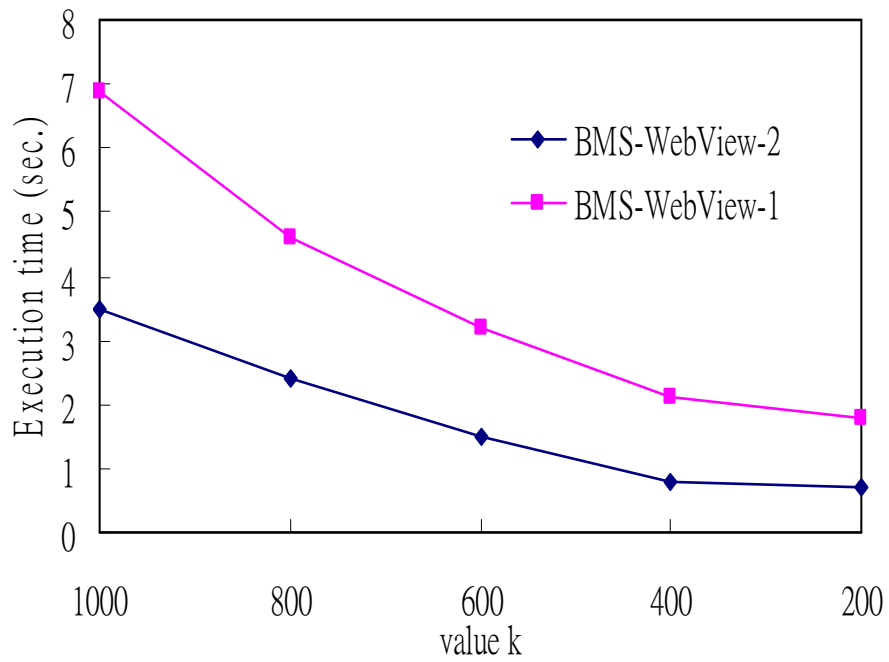
in Microsoft/Visual C++ 6.0.

Two real click-stream datasets, BMS-WebView-1 and BMS-WebView-2, which contain several months worth of click-stream data from two e-commerce web sites, are used to evaluate the performance of the DSM-TKP algorithm. The real data was provided by Blue Martini Software [69], and is available from the KDD Cup 2000 home page [71]. The BMS-WebView-1 dataset consists of 497 items and 59,602 transactions. The maximum transaction size of BMS-WebView-1 is 267 distinct items and the average transaction size is 2.5 items. The BMS-WebView-2 dataset consists of 3,340 distinct items and 77,512 transactions. The maximum transaction size of BMS-WebView-2 is 161 items and the average transaction size is 5 items.

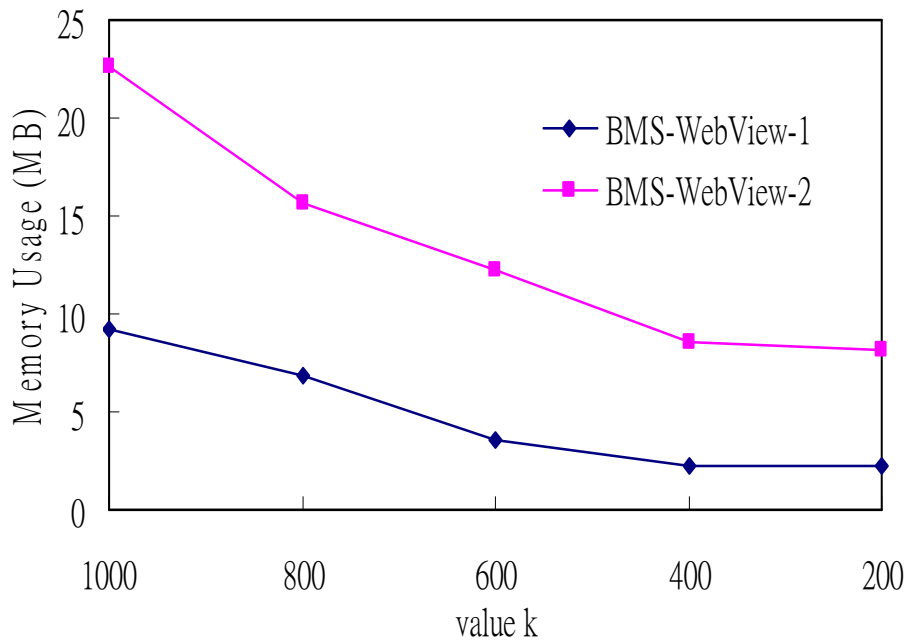
We evaluate the effect of various k values for BMS-WebView-1 and BMS-WebView-2. Figure 6-6 (a) plots the total execution time taken by our algorithm for values of k ranging from 1000 to 200. The figure shows how decreasing k leads to decrease in running time. Figure 6-6 (b) shows how decreasing k leads to decrease in memory usages of DSM-TKP in BMS-WebView-1 and BMS-WebView-2, respectively.

6.5 Conclusions

In this chapter, we proposed an online single-pass algorithm, DSM-TKP, for mining top- k maximal reference sequences in an infinite sequence of Web click-sequences. An effective summary data structure TKP-forest is developed to store the essential information about the set of top- k path traversal patterns of the Web click-stream so far. An efficient pruning mechanism of TKP-forest is presented to guarantee that the upper bound of the summary data structure is predictable. Experiments show that DSM-TKP is efficient and exhibits good scalability.



(a) Execution Time under various k values



(b) Memory usage under various k values

Figure 6- 6. Execution time and memory usage of DSM-TKP on BMS-WebView-1 and BMS-WebView-2 under various k values

Chapter 7 Conclusions and Future Work

In this chapter, summaries of our works are given. Some possible future works are also discussed. For mining of frequent itemsets from data streams, we study the problems involving landmark window-base mining of frequent itemsets and sliding window-base mining of frequent itemsets. For mining of path traversal patterns from Web click streams, we focus on single-pass mining of path traversal patterns and online mining of top-k path traversal patterns without minimum support threshold. For mining of changes of items across two data streams, two one-pass mining algorithms are proposed. All the proposed algorithms are verified by experiments of mining continuous streams of various characteristics. In the experiments comprising comprehensive comparisons, the proposed algorithms outperforms several related algorithms, and they all show excellent linear scalability with respect to the size of the streaming data.



7.1 Conclusions

7.1.1 Summary of Mining of Frequent Itemsets in Data Streams

For the mining of frequent itemsets over the entire history of data streams, we propose an efficient single-pass algorithm, called DSM-FI (Data Stream Mining for Frequent Itemsets), to discover the set of all frequent itemsets over data streams. An effective projection scheme is developed to extract the essential information of frequent itemsets from data streams. Experiments show that DSM-FI outperforms BTS [53], a state-of-the-art single-pass algorithm, by one order of magnitude for discovering the set of all frequent itemsets over data streams with a landmark window.

7.1.2 Summary of Mining of Frequent Itemsets over Stream Sliding Windows

For the mining of frequent itemsets over data streams with a transaction-sensitive sliding window, we develop an efficient one-pass algorithm, called MFI-TransSW (Mining Frequent Itemsets over a Transaction-sensitive Sliding Window) based on bit-vectors, to mine the set of frequent itemsets from only the latest w transactions. Experiments show that MFI-TransSW outperforms SWFI-stream [11] for discovering the set of frequent itemsets in data streams with a transaction-sensitive sliding window.

For the mining of frequent itemsets over data streams with a time-sensitive sliding window, we proposed the first one-pass algorithm, called MFI-TimeSW (Mining Frequent Itemsets over a Time-sensitive Sliding Window), based on the MFI-TransSW to mine the set of frequent itemsets from only the latest w time units. Experiments show that MFI-TimeSW is efficient and exhibits good scalability.

7.1.3 Summary of Mining of Changes of Items across Two Data Streams

We define a new interesting research problem of mining changes of items from data streams in data mining. For the mining of two append-only data streams, we propose a single-pass algorithm, called MFC-append (Mining Frequency Changes of append-only data streams), to find the set of changes of items across two append-only data streams. A new summary data structure, called *Change-Sketch*, is developed to store the essential changed patterns of data streams. The space complexity of Change-Sketch is $O(m\log(n/m))$. For mining of two dynamic data streams, an one-pass algorithm, called MFC-dynamic (Mining Frequency Changes of dynamic data streams), is developed to mine the changes of items across two dynamic data stream. The proposed algorithms take $O(\log(n/m))$ time in the worst case to compute each newly arrived item, but only $O(1)$ amortized time per item.

7.1.4 Summary of Mining of Path Traversal Patterns over Web Click-Streams

For the mining of path traversal patterns over Web click-streams, we propose the first single-pass algorithm, called DSM-PLW (Data Stream Mining for Path traversal patterns in a Landmark Window), to discover the set of all path traversal patterns over streaming maximal forward references. The comprehensive experiments demonstrate that DSM-PLW is efficient and exhibits good scalability.

7.1.5 Summary of Mining of Top-K Path Traversal Patterns

We define a new interesting research problem of mining of top-k path traversal patterns over Web click streams, and propose the first one-pass algorithm, called DSM-TKP (Data Stream Mining for Top-K Path traversal patterns), for mining of top-k path traversal patterns without the user-defined minimum support threshold. An efficient pruning mechanism of the proposed summary data structure is presented to guarantee that the upper bound of the summary data structure is predictable. Experiments show that DSM-TKP is efficient and exhibits good scalability.

7.2 Future Work

With the mining capabilities of the proposed algorithms, there are several interesting extensions on frequent pattern mining and change mining, as listed below.

- **Resource-aware mining of frequent patterns over data streams.**

Resource such as CPU, memory space, and sometimes energy, are very precious in a stream mining environment. They are very likely to be used up when processing data streams which arrive with rapid speed and a huge amount. How to use these resources when we use the proposed algorithms for mining frequent itemsets and changes is an important research issue in our future work.

- **Online mining of sequential patterns over data streams with a sliding window.**

Online mining of sequential patterns in data streams is more complicated than mining of frequent itemset. There are several challenges of mining of sequential patterns from data streams, such as how to define the meaning of sequential patterns in a stream environment, how to define the model of sliding window for mining sequential patterns of data streams, and how to design an efficient single-pass algorithm for mining the set of sequential patterns from data streams.

- **Online mining of high utility itemsets over data streams with a sliding window.**

Although mining itemsets correlations is important in some applications, in many applications people are more interested in finding out how a set of items that is useful by some measure, such as utility. The frequent itemsets do not reflect the impact of any other factor except frequency of the presence or absence of an item. Frequent itemsets may only contribute a small portion of the overall profit, whereas infrequent itemsets may contribute a large portion of the profit. Hence, utility mining is likely to be useful in a wide range of practical application. There are several challenges on mining high utility itemsets over data streams, such as how to define the model of sliding window for mining high utility itemsets of data streams, how to define the meaning of high utility itemsets in a stream environment, and how to design an efficient one-pass algorithm for discovering the set of high utility itemsets from data streams with a sliding window.

References

- [1] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu, A framework for clustering evolving data streams, in: Proc. VLDB, 2003, pp.81-92.
- [2] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases, in: Proc. SIGMOD, 1994, pp. 207-216.
- [3] R. Agrawal and R. Srikant, Fast algorithms for mining association rules, in: Proc. VLDB, 1994, pp. 487-499.
- [4] N. Alon, P. Gibbons, Y. Matias, and M. Szegedy, Tracking join and self-join sizes in limited storage, in: Proc. PODS, 1999, pp. 10-20.
- [5] N. Alon, Y. Matias, and M. Szegedy, The space complexity of approximating the frequency moments, in: Proc. STOC, 1996, pp. 20-29.
- [6] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, Models and issues in data stream systems, in: Proc. PODS, 2002, pp.1-16.
- [7] B. Babcock and C. Olston, Distributed top-k monitoring, in: Proc. ACM SIGMOD, 2003, pp. 28-39.
- [8] S. Babu and J. Widom, Continuous queries over data streams, SIGMOD Record, 30(3) (2001) 109-120.
- [9] J. Borges and M. Levene, Data mining of user navigation patterns, in: Proc. WEBKDD, 1999, pp. 92-111.
- [10] J. H. Chang & W. S. Lee. Finding recent frequent itemsets adaptively over online data streams, in: Proc. ACM SIGKDD, 2003, pp. 487-492.
- [11] J. Chang and W. Lee. A sliding window method for finding recently frequent itemsets over online data streams, Journal of Information Science and Engineering (JISE) 20 (4) (2004).
- [12] M. Charilar, K. Chen, and M. Farach-Colton, Finding frequent items in data streams, in:

- Proc. ICALP, 2002, pp. 693-703.
- [13] M.-S. Chen, J.-S. Park and P. S. Yu, Efficient data mining for path traversal patterns, IEEE TKDE, 10 (2) (1998) 209-221.
- [14] Y. Chen, G. Dong, J. Han, B. W. Wah, and J. Wang, Multi-dimensional regression analysis of time-series data streams, in: Proc. VLDB, 2002, pp. 323-334.
- [15] X. Chen and X. Zhang, A popularity-based prediction model for web prefetching, IEEE Computer 36 (3) (2003) 63-70.
- [16] W. Cheung and O. R. Zaïane, Incremental mining of frequent patterns without candidate generation or support constraint, in: Proc. IDEAS, 2003, pp 111-116.
- [17] Y.L. Cheung, A. W.-C. Fu, Mining association rules without support threshold: with and without item constraints, IEEE TKDE, 16(9), 2004, pp 1052-1069.
- [18] Y. Chi, H. Wang, P. Yu, and R. Muntz. MOMENT: Maintaining closed frequent itemsets over a stream sliding window, in: Proc. ICDM, 2004, pp. 59-66.
- [19] R. Cooley, B. Mobasher, and J. Srivastava, Web mining: information and pattern discovery on the World Wide Web, in: Proc. ICTAI, 1997, pp. 558-567.
- [20] G. Cormode and S. Muthukrishnan, What's hot and what's not: tracking most frequent items dynamically, ACM Trans. Database Syst. 30(1) (2005) 249-278.
- [21] M. Datar, A. Ginois, P. Indyk, and R. Motwani, Maintaining stream statistics over sliding windows, in: Proc. SODA, 2002, pp. 635-644.
- [22] E. Demaine, A. López-Ortiz, and J. I. Munro, Frequent estimation of internet packet streams with limited space, in: Proc. ESA, 2002, pp. 348-360.
- [23] P. Domingos and G. Hulten, Mining high-speed data streams, in: Proc. ACM SIGKDD, 2000, pp. 71-80.
- [24] G. Dong, J. Han, L.V.S. Lakshmanan, J. Pei, H. Wang and P.S. Yu, Online mining of changes from data streams: Research problems and preliminary results, in: Proc. ACM

SIGMOD MPDS, 2003.

- [25] G. Dong and J. Li, Efficient mining of emerging patterns: discovering trends and differences, in: Proc. ACM SIGKDD, 1999, pp. 43-52.
- [26] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Moteani, and J. D. Ullman, Computing iceberg queries efficiently, in: Proc. VLDB, 1998, pp. 299-310.
- [27] J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan, An approximate L1-difference algorithm for massive data streams (extended abstract), in: Proc. IEEE FOCS, 1999, pp. 501-511.
- [28] A. W.-C. Fu, R. W.-W. Kwong, and J. Tang, Mining n-most interesting itemsets, in: Proc. ISMIS, 2000.
- [29] V. Ganti., J. Gehrke, and R. Ramakrishnan, A framework for measuring changes in data characteristics, in: Proc. PODS, 1999, pp. 126-137.
- [30] V. Ganti., J. Gehrke, and R. Ramakrishnan, Mining data streams under block evolution, SIGKDD Explorations, 3(2), 2002, pp. 1-10.
- [31] C. Giannella, J. Han, J. Pei, X. Yan, and P.S. Yu. Mining frequent patterns in data streams at multiple time granularities, in: Data Mining: Next Generation Challenges and Future Directions, AAAI/MIT, H. Kargupta, A. Joshi, K. Sivakumar, and Y. Yesha (eds.), 2003.
- [32] P. B. Gibbons and Y. Matias, Synopsis data structures for massive data sets, in: Proc. SODA, 1999, pp. 909-910.
- [33] L. Golab and M. T. Ozsu, Issues in data stream management, SIGMOD Record 32 (2) (2003) 5-14.
- [34] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan, Clustering data streams, in: Proc. FOCS, 2000, pp. 359-366.
- [35] J. Han, J. Pei, Y. Yin and R. Mao, Mining frequent patterns without candidate generation:

- a frequent-pattern tree approach, *Data Mining and Knowledge Discovery*, 8 (1) (2004) 53-87.
- [36] J. M. Hellerstein, P. J. Haas, and H. Wang, Online aggregation, in: *Proc. ACM SIGMOD*, 1997, pp. 171-182.
- [37] M. R. Henzinger, P. Raghavan, and S. Rajagopalan, Computing data streams, Technical Report 1998-011, Digital Equipment Corporation, Systems Research Center, May, 1998.
- [38] G. Hulten, L. Spencer, and P. Domingos, Mining time-changing data streams, in: *Proc. ACM SIGKDD*, 2001, pp. 97-106.
- [39] R. Jin and G. Agrawal. An algorithm for in-core frequent itemset mining on streaming data. In: *Proc. IEEE ICDM*, 2005.
- [40] R. Karp, C. Paradimitriou, and S. Shenker, A simple algorithm for finding elements in sets and bags, *ACM TODS*, 28 (1) (2003) 51-55.
- [41] H.-F. Li and S.-Y. Lee, Single-pass algorithms for mining frequency change patterns with limited space in evolving append-only and dynamic transaction data streams, in: *Proc. IEEE*, 2004.
- [42] H.-F. Li, C.-C. Ho, M.-K. Shan, and S.-Y. Lee, Efficient Maintenance and Mining of Frequent Itemsets over Online Data Streams with a Sliding Window, in: *Proc. IEEE SMC*, 2006.
- [43] H.-F. Li, S.-Y. Lee, and M.-K. Shan, An efficient algorithm for mining frequent itemsets over the entire history of data streams, in: *Proc. IWKDD*, 2004.
- [44] H.-F. Li, S.-Y. Lee, and M.-K. Shan, Online mining maximal frequent structures in continuous landmark melody streams, *Pattern Recognition Letters*, 26(11), August 2005, pp. 1658-1674.
- [45] H.-F. Li, S.-Y. Lee, and M.-K. Shan, On mining webclick streams for path traversal patterns, in: *Proc. WWW*, 2004, pp. 404-405.

- [46] H.-F. Li, S.-Y. Lee, and M.-K. Shan, Online mining (recently) maximal frequent itemsets over data streams, in: Proc. RIDE, 2005.
- [47] H.-F. Li, S.-Y. Lee, and M.-K. Shan, DSM-TKP: mining top-k path traversal patterns over web click-streams, in: Proc. WI, 2005.
- [48] H.-F. Li, S.-Y. Lee, and M.-K. Shan, DSM-PLW: Single-pass mining of path traversal patterns over streaming web click-sequences, Computer Networks: Special Issue on Web Dynamics, accepted, to appear.
- [49] H.-F. Li, S.-Y. Lee, and M.-K. Shan, Online mining changes of items over continuous append-only and dynamic data streams, Journal of Universal Computer Science: Special Issue on Knowledge Discovery in Data Streams, 11(8), 2005, pp. 1411-1425.
- [50] M.-Y. Lin and S.-Y. Lee, Fast discovery of sequential patterns through memory indexing and database partitioning, Journal of Information Sciences and Engineering (JISE), 21 (1) (2005) 109-128.
- [51] C.H. Lin, D.Y. Chiu, Y.H. Wu and A.L.P. Chen, Mining frequent itemsets from data streams with a time-sensitive sliding window, in: Proc. SIAM SDM, 2005.
- [52] B. Liu, W. Hsu, H.-S. Han, and Y. Xia, Mining changes for real-life applications, in: Proc. DaWaK, 2000, pp. 337-346.
- [53] G. S. Manku and R. Motwani. Approximate frequency counts over data streams, in: Proc. VLDB, 2002, pp. 346-357.
- [54] A. Metwally, D. Agrawal, A. E. Abbadi, Efficient computation of frequent and top-k elements in data streams, in: Proc. ICDT, 2005, pp. 398-412.
- [55] L. O'Callaghan, N. Mishra, A. Meyerson, S. Guha, and R. Motwani, Streaming-data algorithms for high-quality clustering, in: Proc. ICDE, 2002, pp. 685-.
- [56] Z. Pabarskaite, Decision trees for web log mining, Intelligent Data Analysis, 7 (2) (2003) 141-154.

- [57] J. Pei, J. Han, B. Mortazavi-Asl, and H. Zhu, Mining access patterns efficiently from Web logs, in: Proc. PAKDD, 2000, pp. 396-407.
- [58] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu, Mining sequential patterns by pattern-growth: the PrefixSpan approach, IEEE Trans. Knowl. Data Eng. 16 (10) (2004) 1424-1440.
- [59] S. Schechter, M. Krishnan, and M. D. Smith, Using path profiles to predict HTTP requests, Computer Networks, 30 (1-7) (1998).
- [60] M.-K. Shan and H.-F. Li, Fast discovery of structure navigation patterns from web user traversals, in: Proc. SPIE DMKD, 2002, pp. 272-283.
- [61] M. Spiliopoulou, L. C. Faulstich, and K. Winkler, A data miner analyzing the navigational behaviour of web users, in: Proc. ACAI, 1999, pp. 588-589.
- [62] J. Srivastava, R. Cooley, M. Deshpande, P.-N. Tan, Web usage mining: discovery and applications of usage patterns from web data, SIGKDD Explorations, 1 (2) (2000) 12-23.
- [63] W.G. Teng, M.-S. Chen, and P. S. Yu, A regression-based temporal pattern mining scheme for data streams, in: Proc. VLDB, 2003, pp. 93-104.
- [64] W.-G. Teng, M.-S. Chen, and P. S. Yu. Using wavelet-based resource-aware mining to explore temporal and support count granularities in data streams, in Proc: SIAM SDM, 2004.
- [65] P. Tzvetkov, X. Yan, J. Han, TSP: Mining top-k closed sequential patterns, in: Proc. ICDM, 2003, pp. 347-354.
- [66] J. Wang, J. Han, Y. Lu, and P. Tzvetkov, TFP: An efficient algorithm for mining top-k frequent closed itemsets, IEEE TKDE, 17(5), 2005, pp. 652-664.
- [67] D. Xing and J. Shen, Efficient data mining for web navigation patterns, Information and Software Technology, 46 (1) (2004) 55-63.
- [68] J.-X. Yu, Z. Chong, H. Lu, and A. Zhou. False Positive or False Negative: Mining

frequent itemsets from high speed transactional data streams, in: Proc. VLDB, 2004, pp. 204-215.

[69] Z. Zheng, R. Kohavi, and L. Mason, Real world performance of association rule algorithms, in: Proc. ACM SIGKDD, 2001, pp.401-406.

[70] Y. Zhu and D. Shasha, StatStream: statistical monitoring of thousands of data streams in real time, in: Proc.VLDB, 2002, pp. 358-369.

[71] <http://www.ecn.purdue.edu/KDDCUP/>



Publication List

Journal Papers

1. **Hua-Fu Li**, Suh-Yin Lee, and Man-Kwan Shan (2006), "DSM-PLW: Single-Pass Mining of Path Traversal Patterns over Streaming Web Click-Sequences," *Computer Networks: Special Issue on Web Dynamics*, Volume 50, Issue 10, July 2006, pp. 1474-1487. (SCI-E, JCR 2004 IF = 1.226)
2. **Hua-Fu Li**, Suh-Yin Lee, and Man-Kwan Shan (2005), "Online Mining Changes of Items over Continuous Append-only and Dynamic Data Streams," *Journal of Universal Computer Science: Special Issue on Knowledge Discovery in Data Streams*, Volume 11, No. 8, 2005, pp. 1411-1425. (SCI-E, JCR 2004 IF=0.456)
3. **Hua-Fu Li**, Suh-Yin Lee, and Man-Kwan Shan (2005), "Online Mining Maximal Frequent Structures in Continuous Landmark Melody Streams," *Pattern Recognition Letters*, Volume 26, Issue 11, August 2005, pp. 1658-1674 (SCI-E, JCR 2004 IF=0.576)
4. **Hua-Fu Li**, Man-Kwan Shan, and Suh-Yin Lee (2006), "DSM-FI: An Efficient Algorithm for Mining Frequent Itemsets in Data Streams," *Knowledge and Information Systems: An International Journal*, under revision. (SCI-E & EI)
5. **Hua-Fu Li**, Man-Kwan Shan, and Suh-Yin Lee (2006), "DSM-TKP: Mining Top-K Path Traversal Patterns over Web Click-Streams," *in preparation*.
6. **Hua-Fu Li**, Man-Kwan Shan, and Suh-Yin Lee (2006), "Efficient Maintenance and Mining of Frequent Itemsets over Stream Sliding Windows," *in preparation*.
7. **Hua-Fu Li**, Man-Kwan Shan, and Suh-Yin Lee, Online Mining of Frequent Query Trees over Data Streams, *in preparation*.
8. **Hua-Fu Li**, Man-Kwan Shan, and Suh-Yin Lee, Mining and Detecting Changes in

User-Centered Music Query Streams, *in preparation*.

Conference Papers

1. **Hua-Fu Li**, Chin-Chuan Ho, Man-Kwan Shan, and Suh-Yin Lee, "Efficient Maintenance and Mining of Frequent Itemsets over Stream Sliding Windows," in Proc. of IEEE International Conference on Systems, Man, and Cybernetic (IEEE SMC-2006), Taipei, Taiwan, October 8-10, 2006. (EI)
2. **Hua-Fu Li**, Man-Kwan Shan, and Suh-Yin Lee, "Detecting Changes in User-Centered Music Query Streams," in Proc. of IEEE International Conference on Multimedia and Expo (ICME-2006), Toronto, Ontario, Canada, July 9-12, 2006. (EI)
3. **Hua-Fu Li**, Chin-Chuan Ho, Man-Kwan Shan, and Suh-Yin Lee, "Online Mining of Recent Music Query Streams," in Proc. of IEEE International Conference on Multimedia and Expo (ICME-2006), Toronto, Ontario, Canada, July 9-12, 2006. (EI)
4. **Hua-Fu Li**, Man-Kwan Shan, and Suh-Yin Lee, "Online Mining of Frequent Query Trees over Data Streams," in Proc. of the 15th World Wide Web Conference (WWW-2006), Edinburgh, Scotland, May 23-26, 2006. (EI) (poster)
5. **Hua-Fu Li**, Suh-Yin Lee, and Man-Kwan Shan, "DSM-TKP: Mining Top-K Path Traversal Patterns over Web Click-Streams," in Proc. of the 2005 IEEE/WIC/ACM International Conference on Web Intelligence (WI 2005), France, September 19-22, 2005. (EI)
6. **Hua-Fu Li**, Suh-Yin Lee, and Man-Kwan Shan, "Online Mining (Recently) Maximal Frequent Itemsets over Data Streams," in Proc. of the 15th IEEE International Workshop on Research Issues on Data Engineering (RIDE2005),

- Tokyo, Japan, April 3-4, 2005. (EI)
7. **Hua-Fu Li**, Suh-Yin Lee, and Man-Kwan Shan, "Mining Maximal Frequent Itemsets in Data Streams," in Proc. of 2004 International Computer Symposium (ICS2004), Taipei, Taiwan, December 15-17, 2004. (Best Paper Award)
 8. **Hua-Fu Li**, Suh-Yin Lee, and Man-Kwan Shan, "An Efficient Algorithm for Mining Frequent Itemsets over the Entire History of Data Streams," in the Proc. of First International Workshop on Knowledge Discovery in Data Streams, to be held in conjunction with the 15th European Conference on Machine Learning (ECML 2004) and the 8th European Conference on the Principals and Practice of Knowledge Discovery in Databases (PKDD 2004), Pisa, Italy, September 20-24, 2004.
 9. **Hua-Fu Li**, Suh-Yin Lee, and Man-Kwan Shan, "On Mining Webclick Streams for Path Traversal Patterns," in Proc. of the 13th World Wide Web Conference (WWW2004), New York, May 17-22, 2004. (EI)
 10. **Hua-Fu Li**, Suh-Yin Lee, and Man-Kwan Shan, "Mining Frequent Closed Structures in Streaming Melody Sequences," in Proc. of IEEE International Conference on Multimedia and Expo (ICME 2004), Taipei, Taiwan, 2004.(EI)
 11. **Hua-Fu Li** and Suh-Yin Lee, "Single-Pass Algorithms for Mining Frequency Change Patterns with Limited Space in Evolving Append-only and Dynamic Transaction Data Streams", in the Proc. of the 2004 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE-04), Taipei, Taiwan, 2004.
 12. Man-Kwan Shan and **Hua-Fu Li**, "Fast Discovery of Structure Navigation Patterns from Web User Traversals," in Proc. of SPIE Conference on Data Mining and Knowledge Discovery: Theory, Tools, and Technology IV, Orlando, Florida, USA, 2002. (EI)

13. **Hua-Fu Li** and Man-Kwan Shan, "PNP: Mining of Profile Navigational Patterns," in Proc. of SPIE Conference on Data Mining and Knowledge Discovery: Theory, Tools, and Technology IV, Orlando, Florida, USA, 2002. (EI)
14. **Hua-Fu Li** and Man-Kwan Shan, "Mining Non-Simple Traversal Paths from Web Access Logs," in Proc. of 2000 Workshop on Internet and Distributed Systems, Tainan, Taiwan, 2000.



Vita



Hua-Fu Li (李華富) was born on February 24, 1976 in Taoyuan, Taiwan, Republic of China. He received the BS degree in Computer Science and Engineering from Tatung Institute of Technology and the MS degree in Computer Science from National Chengchi University, in 1998 and 2000, respectively. He is currently working towards the Ph.D. degree in National Chiao-Tung University. He coauthored with his advisor Dr. Suh-Yin Lee for their works which received the 2004 ICS (International Computer Symposium) Best Paper Award. His research interests include data mining, data stream management, multimedia information systems and bioinformatics.