

國立交通大學

資訊工程學系

博士論文

有效率之關聯規則勘測與循序樣式勘測方法

Efficient Algorithms for Association Rule Mining and
Sequential Pattern Mining

研究生：林明言

指導教授：李素瑛 教授

中華民國九十二年十一月

授權書

(博士論文)

本授權書所授權之論文為本人在 國立交通大學 資訊工程學系

九十二 學年度第 一 學期取得 博士 學位之論文。

論文名稱：有效率之關聯規則勘測與循序樣式勘測方法

1. 同意 不同意

本人具有著作財產權之論文全文資料，授予行政院國家科學委員會科學技術資料中心、國家圖書館及本人畢業學校圖書館，得不限地域、時間與次數以微縮、光碟或數位化等各種方式重製後散布發行或上載網路。本論文為本人向經濟部智慧財產局申請專利的附件之一，請將全文資料延後兩年後再公開。(請註明文號：)

2. 同意 不同意

本人具有著作財產權之論文全文資料，授予教育部指定送繳之圖書館及本人畢業學校圖書館，為學術研究之目的以各種方法重製，或為上述目的再授權他人以各種方法重製，不限地域與時間，惟每人以一份為限。

上述授權內容均無須訂立讓與及授權契約書。依本授權之發行權為非專屬性發行權利。依本授權所為之收錄、重製、發行及學術研發利用均為無償。上述同意與不同意之欄位若未鈎選，本人同意視同授權。

指導教授姓名：李素瑛博士

研究生簽名：
(親筆正楷)

學號：8517815

日期：民國 92 年 11 月 14 日

-
1. 本授權書請以黑筆撰寫並影印裝訂於書名頁之次頁。
 2. 授權第一項者，所繳的論文本將由註冊組彙總寄交國科會科學技術資料中心。
 3. 本授權書已於民國 85 年 4 月 10 日送請內政部著作權委員會（現為經濟部智慧財產局）修正定稿。
 4. 本案依據教育部國家圖書館 85.4.19 台(85)圖編字第 712 號函辦理。

有效率之關聯規則勘測與循序樣式勘測方法
Efficient Algorithms for Association Rule Mining and
Sequential Pattern Mining

研究生：林明言

Student：Ming-Yen Lin

指導教授：李素瑛博士

Advisor：Dr. Suh-Yin Lee

國立交通大學

資訊工程學系



A Dissertation
Submitted to Department of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Chiao Tung University
in Partial Fulfillment of the Requirements
for the Degree of
Philosophy
in
Computer Science and Information Engineering

November 2003

Hsinchu, Taiwan, Republic of China

中華民國九十二年十一月

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 林 明 言 君

所提論文 有效率之關聯規則勘測與循序樣式勘測方法合於博士資格

水準業經本委員會評審認可。

口試委員：

張明峰

陳銘憲

蔡中川

陳銘憲

陳長鈞

李素瑛

李 傑

蔡錫鈞

指導教授：

李素瑛

系主任：

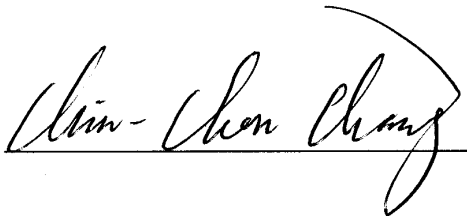
張明峰

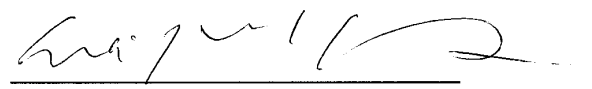
中華民國九十二年十一月十四日

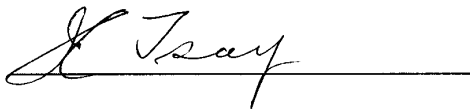
Department of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Chiao Tung University
Hsinchu, Taiwan, R.O.C.

Date: Nov. 14, 2003

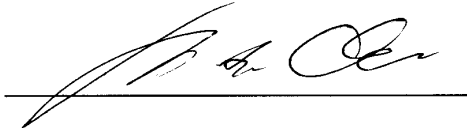
We have carefully read the dissertation entitled **Efficient Algorithms for Association Rule Mining and Sequential Pattern Mining** submitted by **Ming-Yen Lin** in partial fulfillment of the requirements of the degree of Doctor of Philosophy and recommend its acceptance.

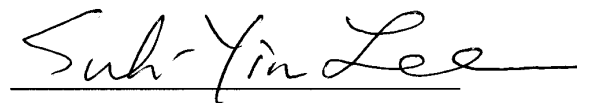




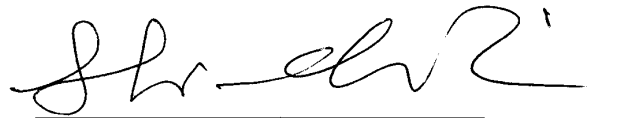


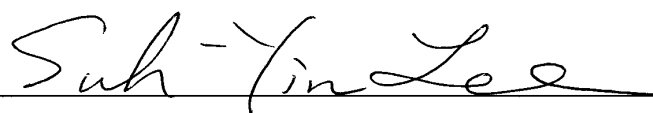










Thesis Advisor: 

Chairman: 

有效率之關聯規則勘測與循序樣式勘測方法

學生：林明言

指導教授：李素瑛 教授

國立交通大學資訊工程學系

摘要

自動化與電腦的應用，讓資料收集無時不刻、隨時隨地的進行，也造成資料大量且迅速增長。隱含於這些巨量資料中的豐富資訊，吸引各領域的學者研發各種粹取其中有用知識的方法。在眾多資料探勘的目標中，頻繁樣式的勘測一直是資料庫中知識挖掘的研究焦點。本論文主旨在於研發有效率的關聯規則及循序樣式探勘方法。

首先，我們提出 *LexMiner* 演算法以找出推演關聯規則的頻繁項目集。為了免除 hash-tree 擺置可能頻繁項目集時的缺點，有些方法將可能項目集依項目的 prefix-order 擺放。*LexMiner* 進一步利用項目集的字典序特性與字典式比較以加速探勘演算法中的核心運算—尋找交易紀錄中包含之可能頻繁項目集。

探勘循序樣式是本論文所探討的第二個主題。我們提出一個記憶體索引的方法，稱為 *MEMISP*，利用「尋找再索引」的技巧來快速探勘循序樣式。無論資料庫大小、無論支持度多小，*MEMISP* 最多僅需檢視資料庫兩回合即可完成探勘。*MEMISP* 優於其他方法的因素在於不產生可能樣式、也不產生暫時的中間資料庫。

探勘具有時間限制的循序樣式，包括時間差與滑動時間窗，可以強化結果的精確性。過去僅有 Apriori 架構可以解決此問題。近來許多研究顯示 pattern-growth 方法可以有效改善探勘速度。因此，我們提出 *DELISP* 演算法，在 pattern-growth 方法論下，利用 divide-and-conquer 策略，整合限制於子資料庫投射，更有效率

的完成具時間限制循序樣式之探勘。

知識挖掘原就是一種挖掘、檢視、再挖掘反覆進行互動的過程。如何減少使用者找到合意結果之交談過程中的反應時間相當重要。我們所提 *KISP* 演算法利用進行過程中所得的資訊，累積計數的資訊以促成有效率之樣式計數運算，並加速整個互動式序列探勘程序。

目前循序樣式的探勘往往假設勘測的資料庫是不變動的。然而，資料庫會有資料更新變動，以致過去找出的樣式會變成無效或新樣式可能會產生。本論文所提的 *IncSP* 方法不需因資料變動而整個從頭開始重新探勘。我們透過隱含式合併與對新增序列有效率分開計數，將過去的樣式漸進式地更新。

我們進行了大規模完整的實驗以評估所提各方法的效能。在我們的實驗範圍中，結果顯示，對於各個不同探勘參數及不同特性的資料集，我們的方法都優於許多著名的方法。針對資料量擴充的實驗也顯示我們探勘頻繁樣式的方法具有線性擴充能力。



Efficient Algorithms for Association Rule Mining and Sequential Pattern Mining

Student: Ming-Yen Lin

Advisor: Prof. Suh-Yin Lee

Department of Computer Science and Information Engineering

National Chiao Tung University

Abstract

Tremendous amount of data being collected is increasing speedily by computerized applications around the world. Hidden in the vast data, the valuable information is attracting researchers of multiple disciplines to study effective approaches to derive useful knowledge from within. Among various data mining objectives, the mining of frequent patterns has been the focus of knowledge discovery in databases. This thesis aims to investigate efficient algorithms for mining frequent patterns including association rules and sequential patterns.

We propose the *LexMiner* algorithm to deal with frequent item-set discovery for association rules. To alleviate the drawbacks of hash-tree placement of candidates, some algorithms store candidate patterns according to prefix-order of itemsets. *LexMiner* utilizes the lexicographic features and lexicographic comparisons to further speed up the kernel operation of mining algorithms.

A memory indexing approach called *MEMISP* is proposed for fast sequential pattern mining using a find-then-index technique. *MEMISP* mines databases of any size, with respect to any support threshold, in just two passes of database scanning. *MEMISP* outperforms other algorithms in that neither candidate patterns nor

intermediate databases are generated.

Mining sequential patterns with time constraints, such as time gaps and sliding time-window, may reinforce the accuracy of mining results. However, the capabilities to mine the time-constrained patterns were previously available only within Apriori framework. Recent studies indicate that pattern-growth methodology could speed up sequence mining. We integrate the constraints into a divide-and-conquer strategy of sub-database projection and propose the pattern-growth based *DELISP* algorithm, which outperforms other algorithms in mining time-constrained sequential patterns.

In practice, knowledge discovery is an iterative process. Thus, reducing the response time during user interactions for the desired outcome is crucial. The proposed *KISP* algorithm utilizes the knowledge acquired from individual mining process, accumulates the counting information to facilitate efficient counting of patterns, and accelerates the whole interactive sequence mining process.

Current approaches for sequential pattern mining usually assume that the mining is performed with respect to a static sequence database. However, databases are not static due to update so that the discovered patterns might become invalid and new patterns could be created. Instead of re-mining from scratch, the proposed *IncSP* algorithm solves the incremental update problem through effective implicit merging and efficient separate counting over appended sequences. Patterns found in prior stages are incrementally updated rather than re-mining.

Comprehensive experiments have been conducted to assess the performance of the proposed algorithms. The empirical results show that these algorithms outperform 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.

誌謝

恩師 李素瑛教授是我人生的明燈。在我工作多年、日漸沈浸於瑣碎的公務時，點亮我突破現狀、開創新生涯的道路。在職博士班的學習，沒有恩師的耐心指導與體諒，絕對無法成就這本博士論文。恩師嚴謹的治學態度，在研究與投稿的每個階段屢次再現，屢見恩師斟酌字句，費心審視。每每讚嘆於恩師思慮的周延、用字的精準與理直氣和的學者風範。恩師的指引、照顧與提攜，銘感於心。

博士論文口試委員之一的 楊維邦老師，與李素瑛老師在我碩一時共同啟蒙我的研究生涯。回首博士班的學習與這本論文的完成，不論是研究興趣的啟發或是追根究底的研究精神建立，對於楊老師當年的指導與適時的鼓勵，深深受益也深表感激。

由衷感謝所有口試委員，不吝於提供多年的寶貴研究經驗，充實了本論文的深度與廣度。謝謝 蔡中川老師、張真誠老師、陳良弼老師、陳銘憲老師、李強老師與 蔡錫鈞老師為豐富本論文內涵提供絕佳的意見，在複雜度與方法適用範圍、研究關鍵轉折點、方法評比公平性、研究結果的適切論述、方法的差異性等等見解，使本論文更臻完善。諸位口試委員都是我學術研究的最佳學習典範。

資訊系統實驗室的學長、姊及學弟、妹們，在研究的坦途上與我攜手並進，謝謝大家也祝福學弟、妹們早日收穫豐碩的研究成果。特別要謝謝學長沈錕坤博士的熱心及研究心得的分享，增添許多研究的樂趣。

父、母親從小至今的關懷，雖然不露於言表，但帶給我的動力是造就我完成博士學位的種子。一路走來，大姊、二姊默默的支持與哥哥的關心，是我源源不絕的溫暖泉源。我年少時的病痛，增添家人許多辛勞，沒有他們的犧牲，絕不會有今日的我。感恩家人的支持、也謝謝兩位姊夫、大嫂及其他親友對我的祝福與勉勵。

一直陪伴在我身旁、沒有怨言、不給我壓力、只給我鼓勵的，就是我的太太夙珍。當我後來辭去工作專心研究時，讓我無後顧之憂，給我綿綿不絕的支持，也常常與我一起討論，激發研究的創意與靈感。能夠順利完成博士學位，對於夙珍，我有無盡的感謝。

要感謝的人真的很多，在此向所有曾經幫過我的人，致上我真切的謝意。

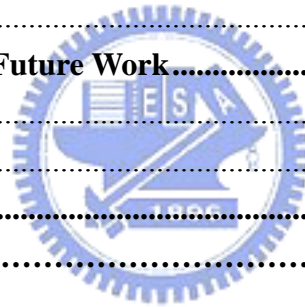
僅以此論文，獻給我摯愛的家人。

Contents

摘要	i
Abstract	iii
誌謝	v
Contents	vi
List of Tables.....	ix
List of Figures.....	x
Chapter 1 Introduction.....	1
1.1 Background	1
1.2 Motivations and Research Objectives.....	4
1.3 Organization of this Thesis	6
Chapter 2 Algorithm LexMiner for Association Rule Mining.....	8
2.1 Overview	8
2.2 Problem Statement	11
2.3 Related Work.....	11
2.4 The Proposed Method	14
2.4.1 <i>LexTree: a lexicographically ordered tree</i>	17
2.4.2 <i>Fast support counting by lexicographic comparisons</i>	19
2.4.3 <i>Candidate generation by leaf joining</i>	24
2.5 Performance Evaluation.....	25
2.5.1 <i>Generation of synthetic data</i>	25
2.5.2 <i>Total execution times of various algorithms</i>	26
2.5.3 <i>Scale-up experiments</i>	29
2.6 Summary	30
Chapter 3 Algorithm MEMISP for Sequential Pattern Mining	33
3.1 Overview	33
3.2 Problem Statement	36
3.3 Related Work.....	37
3.4 The Proposed Method	39
3.4.1 <i>Mining sequential patterns by MEMISP: an example</i>	39
3.4.2 <i>The MEMISP algorithm</i>	43
3.4.3 <i>Dealing with extra-large databases by database partitioning</i>	45
3.4.4 <i>Differences between MEMISP and PrefixSpan</i>	46
3.4.5 <i>Implementation issues</i>	48
3.5 Performance Evaluation.....	49
3.5.1 <i>Generation of experimental data</i>	49
3.5.2 <i>Execution times of GSP, PrefixSpan, and MEMISP algorithms</i>	52

3.5.3 Scale-up experiments	55
3.6 Discussion	56
3.7 Summary	58
Chapter 4 Algorithm DELISP for Sequential Pattern Mining with Time	
Constraints.....	60
4.1 Overview	60
4.2 Problem Statement	63
4.3 Related Work.....	65
4.4 DELISP: Delimited Sequential Pattern Mining.....	67
4.4.1 Terminology used in DELISP.....	67
4.4.2 Mining time-constrained sequential patterns by DELISP: an example	71
4.4.3 The DELISP algorithm	76
4.5 Experimental Results	77
4.5.1 Execution times of GSP and DELISP algorithms	79
4.5.2 Scale up experiments on database size	83
4.6 Discussion	84
4.7 Summary	85
Chapter 5 Algorithm KISP for Interactive Discovery of Sequential Patterns	87
5.1 Overview	87
5.2 Problem Statement	89
5.3 Related Work.....	93
5.3.1. Algorithms for sequential pattern mining.....	93
5.3.2 Algorithms for interactive pattern discovery	96
5.4 The Proposed Algorithm for Interactive Discovery of Sequential Patterns.....	97
5.4.1 The KISP (Knowledge base assisted Incremental Sequential Pattern) mining algorithm.....	98
5.4.2 New-candidate generation by direct computation.....	101
5.4.3 Concurrent support counting and the placement of variable sized candidates	105
5.4.4 Manipulation of the knowledge base	107
5.4.5 Mining efficiency and space utilization with a large knowledge base.....	111
5.5 Performance Evaluation.....	112
5.5.1 Comparisons of KISP and GSP	113
5.5.2 Scale-up experiments	118
5.6 Summary	119
Chapter 6 Algorithm IncSP for Incremental Discovery of Sequential Patterns	121
6.1 Overview	121
6.2 Problem Statement	124

6.2.1 Sequential pattern mining	124
6.2.2 Incremental update of sequential patterns.....	125
6.2.3 Changes of sequential patterns due to database update.....	128
6.3 Related Work.....	129
6.3.1 Algorithms for discovering sequential patterns.....	129
6.3.2 Approaches for incremental pattern updating	132
6.4 The Proposed Algorithm	133
6.4.1 Implicit merging of data sequences with same cids.....	136
6.4.2 The IncSP (Incremental Sequential Pattern Upate) algorithm.....	137
6.4.3 Pattern maintenance on transaction deletion and modification.....	142
6.4.4 Proof of lemmas	143
6.5 Experimental Results	146
6.5.1 Synthetic data generation	146
6.5.2 Comparisons of IncSP and GSP	147
6.5.3 Scale-up experiments	154
6.5.4 Memory requirements	154
6.6 Summary	155
Chapter 7 Conclusions and Future Work.....	157
7.1 Contributions.....	157
7.2 Future work.....	157
References.....	160
Vita.....	172



List of Tables

Table 2-1. Notations used	12
Table 2-2. Itemset grouping by prefixed items	16
Table 2-3. Parameters used in the experiments.....	26
Table 3-1. Example sequence database <i>DB</i> and the sequential patterns.....	37
Table 3-2. Parameters used in the experiments.....	51
Table 4-1. Example sequence database <i>DB</i> and the time-constrained sequential patterns.....	65
Table 4-2. The projected sub-sequences in the <i>-DB</i> sub-databases	73
Table 4-3. Parameters used in the experiments.....	79
Table 5-1. Notations used	90
Table 5-2. The supports of all sequences in an example database.....	91
Table 5-3. User specified minimum supports and the resultant sequential patterns....	91
Table 5-4. Candidates generated by <i>GSP</i> and by <i>KISP</i>	103
Table 5-5. Datasets used in the experiments.....	113
Table 5-6. Number of candidates for the <i>Slen</i> dataset	116
Table 5-7. Effects of concurrent support counting.....	117
Table 5-8. Execution time of <i>KISP</i> when $KB.sup \leq minsup$	118
Table 5-9. Space used by <i>KB</i> with respect to $KB.sup$ (dataset <i>Slen</i>)	118
Table 6-1. Notations used	127
Table 6-2. Sequences and support counts for Example 2	141
Table 6-3. Parameters used in the experiments.....	148

List of Figures

Fig. 1-1. The process of knowledge discovery in databases [5]	2
Fig. 2-1. Example itemsets stored in a <i>LexTree</i>	10
Fig. 2-2. Example candidate itemsets stored in a hash-tree.....	13
Fig. 2-3. Algorithm <i>LexMiner</i>	15
Fig. 2-4. Construction of a <i>LexTree</i> by inserting 3-itemsets: insert (a)(a,c,d) (b)(a,c,j) (c)(a,c,k) (d)(a,d,j) (e)(a,j,o) after (a,d,k) and (a,d,o) inserted (f)(c,d,g)	18
Fig. 2-5. Procedure <i>Find_and_Increment</i>	22
Fig. 2-6. Execution times of various algorithms on the dataset having short patterns	27
Fig. 2-7. Performance comparisons of execution time over various supports.....	28
Fig. 2-8. Execution results of different ordering.....	29
Fig. 2-9. Linear scalability of the database size.....	30
Fig. 3-1. Some index sets and the in-memory <i>DB</i>	41
Fig. 3-2. Algorithm <i>MEMISP</i>	44
Fig. 3-3. Partition the database and discover patterns for extra-large databases	45
Fig. 3-4. Total execution times with respect to various <i>minsup</i> values	53
Fig. 3-5. Comparisons of execution times on dataset <i>C20-T2.5-S4-I1.25</i>	53
Fig. 3-6. Comparisons of execution times on dataset <i>C10-T5-S4-I1.25</i>	54
Fig. 3-7. Comparisons of execution times on dataset <i>C10-T2.5-S8-I1.25</i>	54
Fig. 3-8. Comparisons of execution times on dataset <i>C10-T2.5-S4-I2.5</i>	54
Fig. 3-9. Comparisons of execution times on dataset <i>C10-T7.5-S4-I5</i>	55
Fig. 3-10. Linear scalability of <i>MEMISP</i> vs. <i>PrefixSpan</i>	56
Fig. 4-1. Example of the sequence containment relationship	64
Fig. 4-2. Accessible elements from itemset <i>I</i> in <i>ds</i> with tag-list [$st_1:et_1$, $st_2:et_2$, ..., $st_k:et_k$].....	70

Fig. 4-3. The projected elements of ds with respect to	70
Fig. 4-4. Eliminating items having smaller lexicographic order from projection (Lemma 4-2).....	70
Fig. 4-5. Algorithm <i>DELISP</i>	75
Fig. 4-6. Effect of the <i>mingap</i> constraint	80
Fig. 4-7. Effect of the <i>maxgap</i> constraint	81
Fig. 4-8. Effect of the <i>swin</i> constraint.....	81
Fig. 4-9. Total execution time on datasets of various characteristics	82
Fig. 4-10. Linear scalability of <i>DELISP</i>	83
Fig. 5-1. Proposed Algorithm Basic <i>KISP</i>	99
Fig. 5-2. Structure of the knowledge base	109
Fig. 5-3. Structure of a pattern-support table.....	110
Fig. 5-4. The knowledge base after the second query in Example 5-1	110
Fig. 5-5. Relative execution time and number of candidates on dataset <i>Origin</i>	115
Fig. 5-6. Relative mining performance on datasets of various distributions	115
Fig. 5-7. Relative performance on datasets with longer customer sequences	115
Fig. 5-8. Average execution time vs. number of queries	117
Fig. 5-9. Linear scalability of the database size.....	119
Fig. 6-1. Incremental update versus re-mining.....	126
Fig. 6-2. The original database <i>DB</i> example, $ DB = 6$	128
Fig. 6-3. Data sequences in the increment database and the updated database (a) <i>db</i> with new customers only (b) the updated database <i>UD</i>	129
Fig. 6-4. Data sequences of old and new customers in <i>db</i>	130
Fig. 6-5. Merged data sequences in the updated database <i>UD</i>	130
Fig. 6-6. The architecture of the k -th pass in <i>IncSP</i>	134
Fig. 6-7. Algorithm <i>IncSP</i>	138

Fig. 6-8. The separate counting procedure..... 138

Fig. 6-9. Total execution times over various *minsup* 150

Fig. 6-10. Total execution times over various incremental ratios..... 151

Fig. 6-11. Total execution times over various comeback ratios..... 152

Fig. 6-12. Total execution times over various former ratios..... 153

Fig. 6-13. Linear scalability of the database size..... 153

Fig. 6-14. Maximum required memory with respect to various *minsup*..... 153



Chapter 1 Introduction

Recent developments in computing and automation technologies have resulted in computerizing business and scientific applications in diverse areas. Turning the huge amounts of accumulated data into knowledge is attracting researchers in various domains including databases, machine learning, statistics, and so on. From the perspectives of database researchers, the emphasis is on discovering useful patterns hidden within the large data sets. Hence, a central issue for knowledge discovery in databases, also the focus of this thesis, is to develop efficient and scalable mining algorithms as integrated tools for database management systems.

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 from data in databases [8, 15, 88]. The database used in the mining process generally contains large amounts of data collected by computerized applications. For example, bar-code readers in retail stores, digital sensors in scientific experiments, and other automation tools in engineering often generate tremendous data into databases in a very fast speed. Not to mention the natively computing-centric environments like Web access logs in Internet applications. These databases thus serve as rich and reliable sources for knowledge generation and verification. Meanwhile, the large databases present challenges for effective approaches for knowledge discovery.

The discovered knowledge can be used in many ways in corresponding applications. For example, identifying the frequently appeared sets of items in a retail database can be used to improve the decision making of merchandise placement or

sales promotion. Discovering patterns of customer browsing and purchasing (from either customer records or Web traversals) may assist the modeling of user behaviors for customer retention or personalized services. Given the desired databases, whether relational, transactional, spatial, temporal, or multimedia ones, we may obtain useful information after the knowledge discovery process if appropriate mining techniques are used. A typical process of knowledge discovery in databases is illustrated in Fig. 1-1.

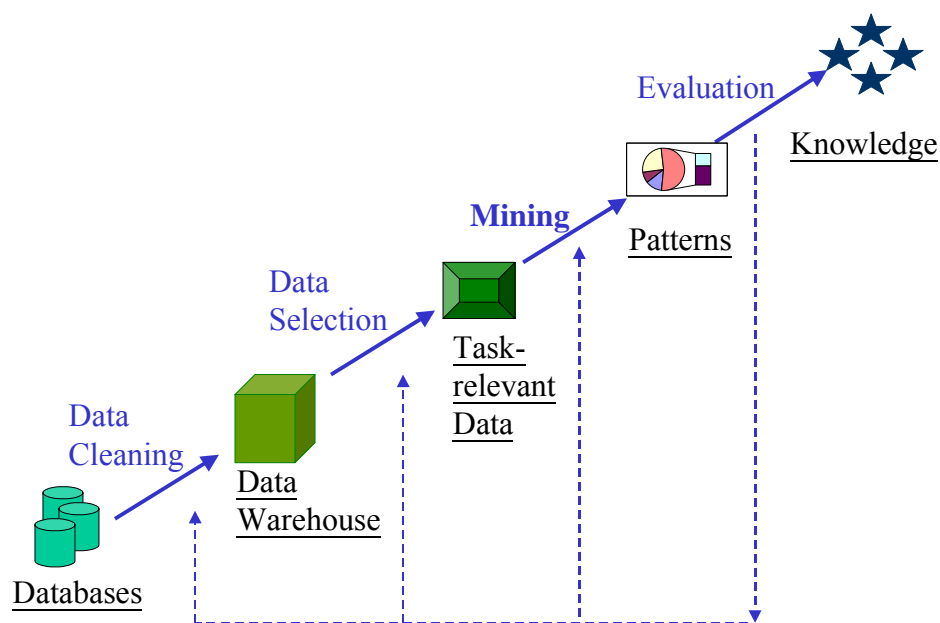


Fig. 1-1. The process of knowledge discovery in databases [5]

Having the databases, relevant prior knowledge, and the goals of the application domain, the target data set is created by *selecting* the data required. The *data cleaning* in Fig. 1-1 may removes those ‘dirty’ data, e.g. data with incomplete fields, missing or wrong values, in the preprocessing stage. The ‘clean’ data is then reduced and/or transformed so that the data is represented by the useful features and actionable dimensions. To find the patterns of interest, the users perform the required *mining* functions, which include summarization/generalization of data characteristics, classification/clustering of data for future prediction, association finding for data

correlation, trend and evolution analysis, etc. The discovered patterns are *evaluated* and presented as knowledge. The process may iterate and contain certain loops between any two steps.

Of all the mining functions in the knowledge discovering process, frequent pattern mining is to find out the frequently occurred patterns. The measure of frequent patterns is a user-specified threshold that indicates the minimum occurring frequency of the pattern. We may categorize recent studies in frequent pattern mining into the discovery of association rules and the discovery of sequential patterns. Association discovery finds closely correlated sets so that the presence of some elements in a frequent set will imply the presence of the remaining elements (in the same set). Sequential pattern discovery finds temporal associations so that not only closely correlated sets but also their relationships in time are uncovered.

Finding all the frequent patterns from the huge data sets is a very time-consuming task. Although the frequency of a pattern can be determined by scanning the database once, the elements of the pattern cannot be known in advance. Take association discovery for example. Given 100 distinct items in the database, the total number of potentially frequent sets is $C(100, 1) + C(100, 2) + C(100, 3) + \dots + C(100, 99) + C(100, 100)$, where $C(m, n)$ represents the combinations to choose n items from m distinct items. The total number of potential patterns is too huge so that validating all the potential patterns in a single database scanning could be impossible. Thus, it is desirable to design efficient algorithms for frequent pattern mining.

In addition, the mining algorithm must be scalable to handle databases of huge size. While the response time may be tolerable for an algorithm to check thousands of potential patterns against a small database having thousands of records, it could be intolerable against a database having millions of records. Similarly, an algorithm that assumes the database has maximum 100 elements might fail to mine any database

having more than 100 elements. In the mining of frequent patterns in database context, the number of elements and the size of the database could be very large. Any improper assumptions on database or main memory could possibly produce an impractical algorithm that works well for small problems only.

1.2 Motivations and Research Objectives

Although there has been a large number of algorithms designed for frequent pattern mining, investigating efficient and scalable algorithms is still very challenging. We first give an overview of the problems, and then describe the motivations and the research objectives of this proposal.

In association rule mining, each record in the database is a set of items (called *itemset*). To generate a rule that associate an itemset X with the itemset Y , the first step is to find all the frequent itemsets, i.e. the itemsets whose occurring frequency is above the user-specified minimum threshold. The second step then uses the discovered frequent itemsets and their frequency to produce all the association rules. In general, most studies in association rule mining generate potential patterns (called *candidates*) and count their frequency in the database to determine the frequent ones. Non-frequent candidates are pruned before counting to reduce the search space, using the property that any candidate having non-frequent sub-sets cannot be frequent. However, not all the properties of itemsets are utilized in the mining process, e.g. the lexicographic property in itemsets. This thesis studies the features presented in itemsets and designs an efficient algorithm to speed up the efficiency of association rule mining.

Previous studies in frequent pattern mining focused on association discovery the most. Nevertheless, sequential pattern mining is even more challenging. In sequential pattern mining, the database is composed of records of data sequences, where each

data sequence is an ordered list of itemsets. The itemsets in a data sequence need not be distinct. The aim is to discover all the frequent sub-sequences in the sequence database.

Considering a sequential pattern having three items, the constitution of the pattern could be a list of: (1) three elements where each element is an item (2) two elements where the first element has one item and the second has two items (3) two elements where the first element has two items and the second has one item (4) one element that has three distinct items. Given the same number of possible items in the itemset database and the sequence database, the potential sequential patterns having three items greatly outnumber the potential itemsets having three items. The total number of candidates, which contains more than patterns having three items, increases exponentially as the number of possible item increases. Searching in the larger and more complex sequence database with the enormous number of candidates demands highly efficient mining algorithms.

Therefore, this thesis investigates an approach that utilizes main memory for indexing sequences and proposes an efficient algorithm for sequential pattern mining.

Common sequence mining considers no constraints for the time-gaps between adjacent elements of a pattern, thereby introducing some uninteresting patterns at times. For example, without specifying the maximum time gap (between adjacent elements), one may discover an example pattern such as “many customers bought *LCD-projector* after purchasing *Laser-pointer*.” Nevertheless, the pattern could be insignificant if the time interval between the two elements is too long such as over years. Typical time constraints include minimum gap, maximum gap, and sliding time-window [80]. In this thesis, we will look into the time-constraint problem and propose an approach that integrates these constraints into a divide-and-conquer strategy for the discovery of sequential patterns with time constraints.

In practice, the mining process is iterative and interactive. The measure of frequent patterns is dependent on the user-specified threshold. Consequently, different thresholds generate different outcomes. Once the mining result is unsatisfactory, the user might try another threshold. Thus, the interactive, time-consuming process usually repeats several times. To reduce the total response time required, an approach employing previous mining results to speed up the whole interactive mining process is investigated in this thesis.

Current approaches for sequential pattern mining usually assume that the mining is performed in a static sequence database. However, databases are not static due to update so that the discovered patterns might become invalid and new patterns could be created. In addition to higher complexity, the maintenance of sequential patterns is more challenging than that of association rules owing to sequence merging. Sequence merging, which is unique in sequence databases, requires the appended new sequences to be merged with the existing ones if their customer ids are the same. Re-mining of the whole database appears to be inevitable since the information collected in previous discovery will be corrupted by sequence merging. Instead of re-mining, we propose an algorithm that solves the maintenance problem through effective merging for incremental pattern updating in this thesis.

1.3 Organization of this Thesis

This rest of the thesis is organized as follows. We describe efficient algorithms for mining association rules in Chapter 2. Fast algorithms for mining sequential patterns are delineated in Chapter 3. Chapter 4 addresses the problem of mining sequential patterns with time constraints and presents related algorithms. The algorithms for interactive sequence mining are introduced in Chapter 5. Chapter 6 extends the sequence mining algorithms over static databases into that over incremental databases.

Finally, Chapter 7 concludes this thesis.



Chapter 2 Algorithm LexMiner for Association Rule Mining

2.1 Overview

Association rule mining has been one of the focusing researches in data mining [4, 5, 9, 14, 22, 28, 31, 37, 58, 95, 101]. The problem is originated from a large transactional database, in which each transaction is *a set of items* (named *itemset*) purchased by a customer [4]. The result of the mining discovers relationships between itemsets (called *association rules*), which can be used for inferring buying patterns of customers, placement of sales items, and so on in many applications.

An association rule $X \Rightarrow Y$ means that the occurrence of itemset X would imply the occurrence of itemset Y . A transaction T is said to *contain* X if and only if $X \subseteq T$. The *support* of itemset X is the number of transactions containing X divided by the total number of transactions in the database. Each rule is associated with two attributes, support and confidence. The *support* of the rule is defined as the support of the itemset $X \cup Y$, and the *confidence* is defined as the support of $X \cup Y$ divided by the support of X . Association discovery aims to find out all association rules with support and confidence greater than the user-specified minimum thresholds.

The discovery usually takes two steps, discovering frequent itemsets and generating rules. Frequent itemsets are those itemsets whose supports are greater than the specified minimum support. Since the desired rules can be easily generated after having the supports of itemsets, the overall performance is dominated by the step of frequent itemset discovering. This issue has been the focus in previous researches [1, 7, 28, 32, 43, 49, 50, 61, 74, 85, 100]. The objective of this chapter is to improve the performance of frequent itemset discovering by fully utilizing the lexicographic property of itemsets.

Most algorithms for frequent itemset finding nearly are variations of the *Apriori* algorithm [5]. *Apriori* generated *potential frequent itemsets* (called *candidates*), stored candidates in a hash-tree, and then located the candidates required for support counting against each transaction. Nevertheless, the hash-tree may introduce ‘irrelevant’ comparisons while counting because the hashing may possibly place candidates with different prefixed items in the same leaf. Some implementations stored candidates in a prefix tree, such as the *SEAR* algorithm [53]. By storing candidates according to the lexicographic order of items, the prefix tree alleviates some drawbacks of hashing.

The lexicographic property of itemsets [1, 2, 12] had also been used in some algorithms. For example, the *TreeProjection* algorithm [1] generated candidates by lexicographic extensions, instead of by join operations as in *Apriori*. The transactions were then projected onto each node of the ordered candidate tree. Lexicographically extending the candidate itemsets were also used in the *Max-Miner* algorithm [12] and the *DepthFirst* algorithm [2] for discovering maximal patterns, i.e. the longest frequent itemsets.

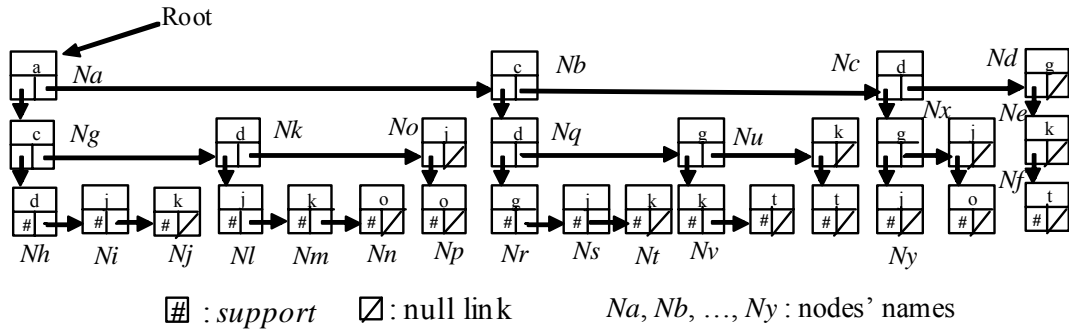
However, the lexicographic property is not fully utilized. In all the mining algorithms, each transaction has to perform *itemset matching*, which checks whether the transaction contains all the items in a candidate, with every located candidate. In general, all the items in each transaction are sorted in dictionary order after a light pre-processing. The lexicographic property in transactions can work with the property in candidates to accelerate itemset matching. In the proposed *LexMiner* algorithm, we break the *itemset matching* into a series of *item matching* (named *lexicographic comparisons*), in addition to storing candidates into a lexicographic tree of items (named *LexTree*). We refer *k-itemset* to an itemset with *k* items. The *LexMiner* algorithm optimizes the discovery of frequent 1-itemsets and 2-itemsets by array

counting, and speeds up the kernel operation, itemset matching, to discover the frequent k -itemsets ($k > 2$).

The proposed *LexTree* is an ordinary trie of k -itemsets, where each node represents an item in an itemset and common items in itemsets share the same nodes. An example *LexTree* is shown in Fig. 2-1 (The detail structure of the *LexTree* is given in Section 2.3.1). *LexMiner* counts the supports efficiently by lexicographic comparisons between transactions and candidates. Each lexicographic comparison effectively obtains the group of promising candidates and prunes the group of irrelevant candidates. Therefore, the support counting is accelerated due to the reduction in the matching required for every transaction, in every database scanning.

The set = $\{(a, c, d), (a, c, j), (a, c, k), (a, d, j), (a, d, k), (a, d, o), (a, j, o), (c, d, g), (c, d, j), (c, d, k), (c, g, k), (c, g, t), (c, k, t), (d, g, j), (d, j, o), (g, k, t)\}$

(a) The set of all 3-itemsets



(b) The 3-itemset *LexTree*

Fig. 2-1. Example itemsets stored in a *LexTree*

The generation of candidates even benefits from the *LexTree* structure. In *Apriori*-like algorithms, the superset of candidate k -itemsets were generated by self-joining frequent $(k-1)$ -itemsets with common prefix $(k-2)$ items, and then pruning those having non-frequent $(k-1)$ -subsets. Common implementations usually store frequent $(k-1)$ -itemsets in a hash table to assist fast pruning. Consequently, either a

traversal over the entire hash table or a pre-sorting of frequent $(k-1)$ -itemsets is required in the join operation. On the other hand, *LexMiner* generates candidates faster without any table searching or sorting since itemsets having common prefixed $(k-2)$ items are already linked by the leaf-pointers.

In this chapter, we present a scalable mining algorithm for the discovery of association rules. The extensive experiments on well-known synthetic data show that our algorithm outperformed *Apriori*, *TreeProjection* and *FP-growth* algorithms. Scale-up experiments also promise the linear scalability with the number of transactions. The rest of the chapter is organized as follows. Section 2.2 introduces the problem. Section 2.3 reviews the related work. The proposed algorithm and the new data structure are described in Section 2.4. Comparative results of the experiments are shown in Section 2.5. Section 2.6 summarizes this chapter.

2.2 Problem Statement



Let $\Psi = \{i_1, i_2, \dots, i_n\}$ be a set of literals, called *items*. A transaction T with m items is denoted by $T = \{x_1, x_2, \dots, x_m\}$, such that $T \subseteq \Psi$. Items within an itemset are kept in lexicographic order. A k -itemset is represented by (x_1, x_2, \dots, x_k) , where $x_1 < x_2 < \dots < x_k$. Given the database D of transactions and the user specified minimum support $minsup$, the mining of frequent itemsets is to find out all the itemsets having support greater than $minsup$. Table 2-1 summarizes the notations used in this chapter

2.3 Related Work

Level-wised algorithms like *Apriori* discover frequent k -itemsets in k -th pass of database scanning by generating candidate k -itemsets and identifying the frequent ones. Key factors of mining performance thus are determined by the number of database scans, the number of transactions needed to be processed in a pass, the

number of candidates generated in a pass, and the efficiency of support counting.

Table 2-1. Notations used

D	The database of transactions
T	A transaction, $T = \{x_1, x_2, \dots, x_p, \dots, x_m\}$
x_1, x_2, \dots, x_k	Items
X, Y	k -itemsets, $X = (x_1, x_2, \dots, x_k)$, $Y = (y_1, y_2, \dots, y_k)$
$X.support$	The support of itemset X
$minsup$	The minimum support specified by the user
C_k	The set of candidate k -itemsets, see Section 2.4
L_k	The set of frequent k -itemsets, see Section 2.4
Γ_{C_k}	The candidate k -itemset <i>LexTree</i> , see Section 2.4
Γ_{L_k}	The frequent k -itemset <i>LexTree</i> , see Section 2.4
${}_pT_m$	The partial transaction of T , ${}_pT_m = \{x_p, x_{p+1}, \dots, x_m\}$, see Section 2.4.2
${}_pT_m^k$	The k -subsets of ${}_pT_m$, see Section 2.4.2

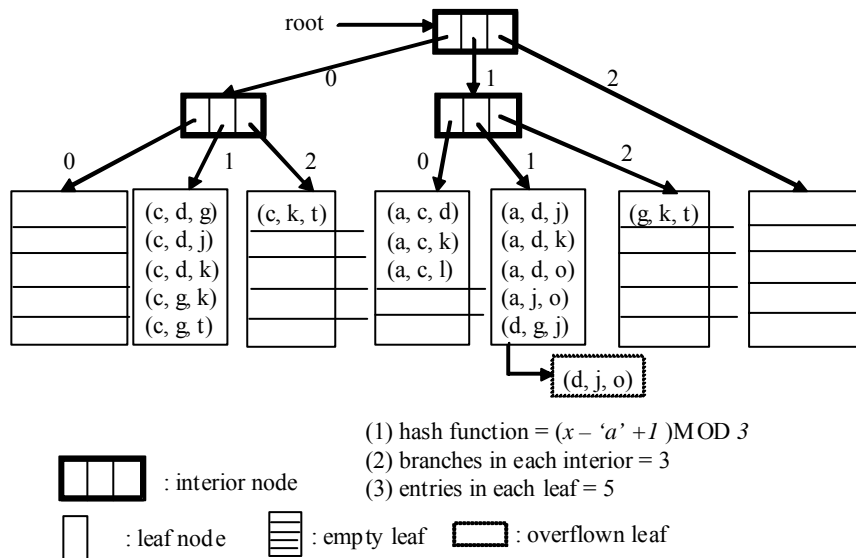
For the reduction of database scans, **DIC (Dynamic Itemset Counting)** algorithm starts counting just the 1 -itemsets and then quickly adds counters of 2 -itemsets, ..., and of k -itemsets, provided that all its subsets have been determined being frequent [14]. *Partition* algorithm generates all the candidates by memory-sized partitions of the database [73]. Besides, **DLG (Direct Large itemset Generation)** algorithm uses large bit vectors for transformation and traversal to reduce database scans [86]. Sampling approaches can effectively reduce the number of database passes too [93].

For the reduction of transactions to be processed in a pass, *AprioriTid* algorithm replaces itemsets in a transaction T by potentially frequent k -itemsets present in T [7]. **DHP (Dynamic Hashing with Pruning)** algorithm substantially minimizes the number of transactions by applying a hashing scheme, which also eliminates some candidates in advance [58].

For fast support counting, *Apriori* stores candidates in a *hash-tree*, where each interior node contains a hash table and each leaf contains a list of candidates. Candidates are placed by hashing on consecutive items in the candidate until a leaf is

reached. Inserting a candidate to a leaf without empty entry introduces a leaf-to-interior conversion and a re-distribution of the candidates. As an example, Fig. 2-2 shows the hash-tree of candidate itemsets in Fig. 2-1(a).

The *TreeProjection* algorithm [1] generates candidates into a lexicographical tree of itemsets. After the transaction projecting (i.e. intersecting all transactions with each node), the supports are obtained by matrix counting. Similar lexicographic extensions are also used in the *Max-Miner* [12] and the *DepthFirst* [2] algorithms to find the maximal itemsets. Note that these algorithms typically generate more candidates than *Apriori* does since the pruning is no longer suitable.



(b) The hash-tree of candidate 3-itemsets

Fig. 2-2. Example candidate itemsets stored in a hash-tree

The *FP-growth* discovers frequent patterns without generating candidates in advance [28]. The database is first compressed into an in-memory data structure called *FP-tree* (**F**requent **P**attern **t**ree). Frequent patterns are then derived by “growing” patterns incrementally on the *FP-tree* by a partitioning-based, divide-and-conquer method [28]. Based on the similar concept, the *CLOSET* algorithm finds out the closed frequent itemsets [66], and the *H-mine* algorithm projects transactions to a hyper-link structure for frequent itemset discovering [65].

In addition, some ‘vertical’ algorithms, such as *Eclat* (**E**quivalence **c**lass and bottom-up), speed up the discovery by lattice-traversal with vertical database layout, which associates each candidate with transaction-id lists [74, 101]. Such scheme is also extended to mine closed frequent itemsets in the *CHARM* algorithm [100].

To summarize, level-wised mining approaches are generally more scalable than other approaches with respect to the database size. Projection-based algorithms like *TreeProjection* might suffer from memory shortage (for keeping transaction sets in each node) and it is costly to project volumes of transactions. Algorithms using the pattern-growth framework like *FP-growth* might be limited by the available memory since transactions are compressed into the main memory. Besides, *FP-growth* might not compress well to achieve good performance with a non-dense database [19]. Given a ‘horizontal’ database, vertical approaches have to transform the horizontal layout into vertical. The storage for storing item-oriented transaction lists will also cost too much for a very large database. Therefore, the *Apriori* framework still has competitive advantage in scalable association mining.

2.4 The Proposed Method

Fig. 2-3 lists the proposed *LexMiner* algorithm using the *LexTree* to speed up the kernel operations in frequent itemset discovering. In brief, the *LexMiner* finds out frequent 1-itemsets and 2-itemsets by an optimized counting technique. The frequent k -itemsets ($k > 2$) are discovered by fast support counting through *efficient lexicographic comparisons*, and rapid candidate generation through *effective leaf joining*, enabled by *prefixed itemset grouping* with the *LexTree*.

Using a one-dimensional array of counters of all items is the fastest way to compute frequent 1-itemsets since every item is a potentially frequent 1-itemset. Let C_k be the set of candidate k -itemsets and L_k the set of frequent k -itemsets. The C_2 is

generated by joining L_1 with L_1 . Since all the subsets of C_2 are frequent, none of the candidate in C_2 can be deleted before counting. We use a two-dimensional array of counters to store the supports of candidates in C_2 . In order to minimize the required storage, we map items in L_1 to contiguous integers and the non-frequent items to zero. A two-level for-loop over each transaction accomplishes the efficient counting and determines L_2 .

```

 $L_1 = \{\text{frequent 1-itemsets}\}$  ;
if  $L_1 \neq \emptyset$  then
   $C_2 = L_1 \times L_1$  ; // stored in a 2-dimensional array
  for each transaction  $T \in D$  do
    for each 2-subset  $X$  of  $T$  do if  $(X \in C_2)$   $X.support++$  ;
  end
   $L_2 = \{X \in C_2 \mid X.support \geq \text{minsup}\}$  ;
end
for  $(k=3; L_{k-1} \neq \emptyset, k++)$  do
  Construct  $L_{k-1}$  into a frequent LexTree,  $\Gamma_{L_{k-1}}$  // LexTree construction – see Section 2.4.1
  Generate  $C_k$  from  $\Gamma_{L_{k-1}}$  to a candidate LexTree,  $\Gamma_{C_k}$  // Candidate generation – see Section 2.4.3
  for each transaction  $T \in D$  do
    Find_and_increment( $T, \Gamma_{C_k}$ ) // Fast support counting – see Section 2.4.2
  end
   $L_k = \{X \in C_k \mid X.support \geq \text{minsup}\}$  ;
end
Answer =  $\cup_k L_k$  ;

```

Fig. 2-3. Algorithm *LexMiner*

The fact that frequent itemsets usually have common items inspires the concept of *prefixed itemset grouping*, which sorts itemsets into groups according to the same prefixed items. For example, assume that L_3 is the set of frequent 3-itemsets as listed in Fig. 2-1(a). Since we generate C_4 by joining those frequent 3-itemsets having the same prefixed 2 items. If we perform the *prefixed itemset grouping* as shown in Table 2-2, the C_4 would be simplified into self-joining over the same group. We use *LexTree* to group the same prefixed itemsets under the same node. The leaf-linked $(k-1)$ -itemsets would have the same prefixed $(k-2)$ items. Therefore, a rapid candidate generation is achieved through *effective leaf joining* in the *LexTree* of the frequent itemsets. For example, candidates (a, c, d, j), (a, c, d, k), (a, c, j, k) are easily obtained by *leaf joining* on nodes N_h, N_i , and N_j of the *LexTree* in Fig. 2-1(b). The construction

of *LexTree* is described in Section 3.1.

Prefixed itemset grouping also provides a quick way to identify the promising group of candidates in support counting. Assume that we are updating the supports of C_3 in Table 2-2, i.e. candidate *LexTree* in Fig. 2-1(b), with transaction $T = \{c, d, g, j, k\}$. By a simple *item comparison* of ‘c’ (first item of the transaction T) and ‘a’ (the item of node Na), we may skip all the candidates in Group 1 (also Groups 2 and 3) since ‘c’ \neq ‘a’. On the other hand, since the result of item comparison of ‘c’ (first item of the transaction T) and ‘c’ (the item of node Nb) is equal, we proceed the comparison on the second item to find which candidate is eventually contained in T . In the *LexTree* of candidate itemsets, the itemsets are grouped and linked by the items of each level. Therefore, we can speed up the support counting by *a series of efficient item matching*, called *lexicographic comparisons*. We present the details of lexicographic comparisons in Section 2.4.2.

Table 2-2. Itemset grouping by prefixed items

Group	Itemsets
1	(a, c, d), (a, c, j), (a, c, k)
2	(a, d, j), (a, d, k), (a, d, o)
3	(a, j, o)
4	(c, d, g), (c, d, j), (c, d, k)
5	(c, g, k), (c, g, t)
6	(c, k, t)
7	(d, g, j)
8	(d, j, o)
9	(g, k, t)
Note: Group the itemsets in Fig. 2-1(a) according to the first and then the second item	

In short, starting from pass three and beyond, three major steps are performed in the *LexMiner* algorithm. At first, frequent $(k-1)$ -itemsets are stored in lexicographic order into a frequent *LexTree*, denoted by $\Gamma_{L_{k-1}}$. Candidate k -itemsets are then generated and stored into a candidate *LexTree*, denoted by Γ_{C_k} . Finally, all the

candidates in each transaction can be efficiently found by lexicographic comparisons. The following sections give the details of these procedures.

2.4.1 *LexTree: a lexicographically ordered tree*

LexTree is a compact, trie-like tree structure for hierarchically storing itemsets. *LexTree* groups itemsets by same prefixed items and stores itemsets in dictionary order (*lexicographic order*). We use $X < Y$ to denote that itemset X precedes itemset Y in lexicographic order. The item in Y that determines $X < Y$ is called the ***pivot item*** of Y . For example, $(a, b, f) < (\mathbf{b}, c, d)$ and $(a, b, c) < (a, \mathbf{b}, e)$, where items in boldface are the pivot items. We insert itemsets in $C_k(L_k)$ to a *candidate LexTree* (*frequent LexTree*) one by one in lexicographic order. The *LexTree* corresponding to the itemsets in Fig. 2-1(a) is shown in Fig. 2-1(b). The definition of *LexTree* is given below.

Definition 2-1. A **LexTree** is a tree such that:

- (i) A *leaf node* comprises three fields, the item identifier (abbreviated as *ID*), the sibling pointer (abbreviated as *sibling*), and the support counter (abbreviated as *support*).
- (ii) An *internal node* comprises three fields, *ID*, *sibling*, and the next pointer (abbreviated as *next*).
- (iii) Nodes linked through the *sibling* pointer are of the same depth. The depth of a node is $(d+1)$, if the node is linked by the *next* pointer of another node whose depth is d .
- (iv) The *Root* is a pointer, which points to the first node of the tree. For convenience, the node pointed by the *Root* is called the *Root node*. The *depth* of the *Root node* is 1.
- (v) A k -itemset (x_1, x_2, \dots, x_k) in a *LexTree* is represented by a group of nodes at depth $1, 2, \dots, k$. In this group, the *ID* of the node at depth d is x_d and the

support of the leaf node is the support of this itemset. □

Various fields of a node Nd are referred to by symbols $Nd.ID$, $Nd.sibling$, $Nd.next$, and $Nd.support$. The ID of the $Root$ node is $Root.ID$, for instance.

Auxiliary $Last$ pointers are used to ease the fast construction of $LexTree$. A k -itemset $LexTree$ has k $Last$ pointers, where each $Last$ points to the last node in that level ($depth$) of the tree. We use $Last[k]$ to denote the node pointed by the $Last$ pointer at level k . Hence, the latest k -itemset inserted can be described by $(Last[1].ID, Last[2].ID, \dots, Last[k].ID)$.

An example, which shows the construction of $LexTree$ by inserting the ordered 3-itemsets in Fig. 2-1(a) is illustrated in Fig. 2-4. Note that the $Root$ pointer is not shown in Figures 2-4(b) to 2-4(e). One can see that starting from the pivot item of the current itemset (we are inserting), a series of new nodes are allocated and the corresponding $Last$ pointers are moved.

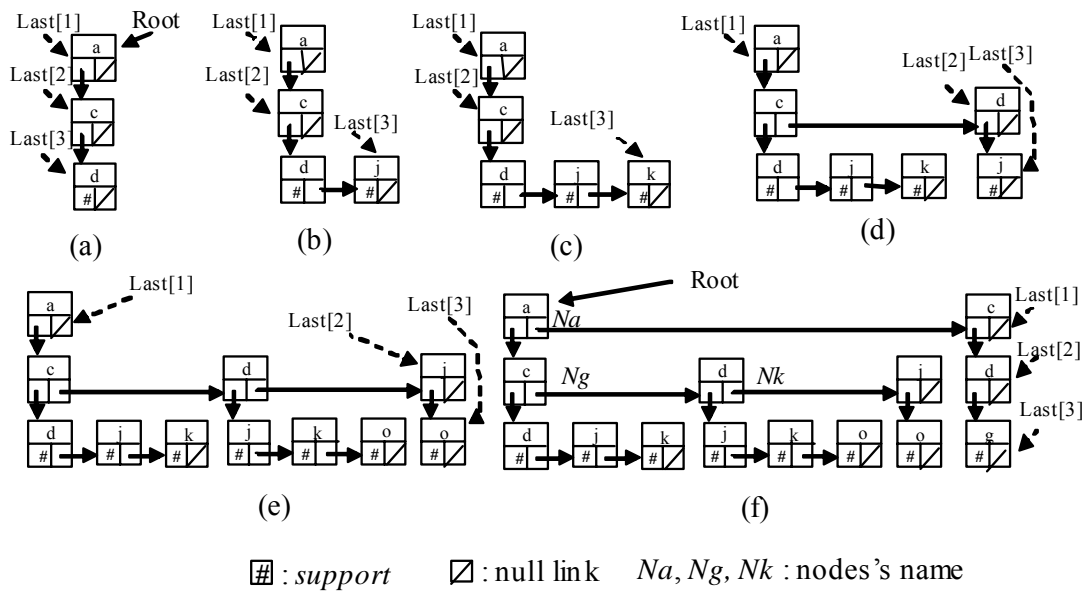


Fig. 2-4. Construction of a $LexTree$ by inserting 3-itemsets: insert (a)(a,c,d) (b)(a,c,j) (c)(a,c,k) (d)(a,d,j) (e)(a,j,o) after (a,d,k) and (a,d,o) inserted (f)(c,d,g)

By inserting itemsets to $LexTree$ in lexicographic order, we can group itemsets

by the same prefixed items. In addition, the *LexTree* is compact since common items share the same nodes. Take itemsets (a, c, d) , (a, c, j) and (a, c, k) in Fig. 2-4(f) for example. They share the same two nodes, Na and Ng . Similarly, itemsets (a, d, j) , (a, d, k) , (a, d, o) share the same two nodes, Na and Nk . When we take the above six itemsets into consideration, node Na is shared by these itemsets. In other words, node Na groups the itemsets with the same prefixed item 'a'; nodes Na and Ng group the itemsets with the same prefixed items 'a' and 'c'; also itemsets with the same prefixed items 'a' and 'd' are grouped by nodes Na and Nk .

2.4.2 Fast support counting by lexicographic comparisons

For every transaction in the database, the supports of those candidates contained in the transaction must be updated. Accordingly, during pass k , all the k -subsets of a transaction are compared with candidates in C_k . Without structuring candidates on item basis like *LexTree*, common implementations processed the kernel operation on an itemset matching basis. In general, all the items in each transaction are sorted in lexicographic order after a light pre-processing. Therefore, we may utilize the lexicographic property (in transactions and in *LexTree*) to break itemset matching into lexicographic comparisons. We describe the lexicographic comparison and the total number of comparisons below.

LexMiner minimizes the number of k -subsets (of a transaction) required matching by generating only those promising k -subsets. Promising k -subsets are composed of a *heading item* x_i and some *partial transaction* that generates the $(k-1)$ -subsets. The definition of *partial transaction* is given in Definition 2-2 below. The *heading item* x_i must appear in the first level of the candidate *LexTree* to make the k -subset promising. Similarly, the *partial transaction* only generates promising $(k-1)$ -subsets having a *heading item* x_j that appears in the second level of the candidate

LexTree.

Definition 2-2. Given a transaction $T = \{x_1, x_2, \dots, x_m\}$, the *partial transaction* ${}_pT_m$ is the set of ordered items from item x_p to item x_m in T . That is, ${}_pT_m = \{x_i \mid x_i \in T, p \leq i \leq m\}$. \square

For example, ${}_1T_m = \{x_1, x_2, \dots, x_m\}$, ${}_4T_m = \{x_4, x_5, \dots, x_m\}$, ${}_{m-1}T_m = \{x_{m-1}, x_m\}$, and ${}_mT_m = \{x_m\}$. The *partial transaction* ${}_pT_m$ is an empty set if $p > m$. Let ${}_pT_m^k$ be the k -subsets of a partial transaction ${}_pT_m = \{x_p, x_{p+1}, \dots, x_m\}$. We have Theorem 2-1.

Theorem 2-1. The k -subsets of a transaction T are ${}_1T_m^k = \prod_{i=1}^{m-k+1} \{x_i\} \times {}_{i+1}T_m^{k-1}$.

Proof. We have ${}_1T_m^1 = \{x_1\} \cup \{x_2\} \cup \dots \cup \{x_m\} = \{x_1\} \cup {}_2T_m^1$.

Also ${}_1T_m^2 = \{x_1\} \times {}_2T_m^1 \cup {}_2T_m^2$, and ${}_1T_m^3 = \{x_1\} \times {}_2T_m^2 \cup {}_2T_m^3$.

So ${}_1T_m^k = \{x_1\} \times {}_2T_m^{k-1} \cup {}_2T_m^k$. — Formula 2-1

Similarly, ${}_2T_m^k = \{x_2\} \times {}_3T_m^{k-1} \cup {}_3T_m^k$. — Formula 2-2

So ${}_1T_m^k = \{x_1\} \times {}_2T_m^{k-1} \cup \{x_2\} \times {}_3T_m^{k-1} \cup {}_3T_m^k$ by substitution using Formula 2-2.

Finally, the formula ${}_1T_m^k = \prod_{i=1}^{m-k+1} \{x_i\} \times {}_{i+1}T_m^{k-1} \cup {}_{m-k+2}T_m^k$ is obtained by iterative

substitution of the last item. Since the last item, ${}_{m-k+2}T_m^k$ is an empty set, the theorem is proved. \square

LexMiner uses Theorem 2-1 to eliminate the generation of many impossible k -subsets of a transaction. Since candidates are grouped by the same prefixed items in *LexTree*, if some item x_i , where $1 \leq i \leq m-k+1$, is not found in the first level of the candidate *LexTree*, no k -subsets comprising x_i as the first item are generated for comparison. Again, whether a partial transaction should generate $(k-1)$ -subsets or not

is determined by the existence of some item x_j , where $i+1 \leq j \leq m$. If item x_j cannot be found in the second level of the sub-tree headed by the matched x_i , these $(k-1)$ -subsets are excluded. In this way, by comparisons between ordered items in the transaction and the nodes in the candidate *LexTree* level by level, those candidates contained in a transaction are found. For example, while updating the supports of Γ_{C_3} in Fig. 2-1(b) with transaction $T_1 = \{g, k, t, c', k'\}$, the 3-subsets of T_1 having k or t as the heading items as well as 3-subsets $\{g, t, c'\}$, $\{g, t, k'\}$ and $\{g, c', k'\}$ never engage in the computation.

Support counting in *LexMiner* is accomplished by breaking the searching of candidates to lexicographic comparisons of sub-items, and then incrementing the supports of matched candidates, as outlined in Fig. 2-5. Whenever a candidate is found, its support is added by one. Two pointers, a transaction pointer tp and a candidate pointer cp , are employed to assist fast matching. Let tp move along T and cp walk through Γ_{C_k} . Once tp or cp reaches the end of the corresponding list or structure, the finding stops. In each pass, starting from the first item of the transaction and the root of candidate tree, fast support counting is accomplished by the *Find_and_Increment* procedure.

Assume that we are comparing the q -th item of a transaction $T = \{x_1, \dots, x_m\}$ with a node Np at level p , where $1 \leq p \leq k-1$ and $1 \leq q \leq m$. The matched prefix $(p-1)$ items can be described by $(N_1.ID, N_2.ID, \dots, N_{p-1}.ID)$ if we reach Np via node N_i at level i , $1 \leq i \leq p-1$. The x_q is the q -th item of T and $Np.ID$ is the item of node Np .

If $x_q < Np.ID$, we advance tp (transaction pointer) so that all the k -subsets of T having prefix $(N_1.ID, N_2.ID, \dots, N_{p-1}.ID, x_q)$ are pruned. If $x_q > Np.ID$, we advance cp (candidate pointer) to eliminate the comparisons of those nodes reached via $Np.next$.

```

Procedure Find_and_Increment(tp, cp)
// input: tp points to the head of a list, cp points to a node in  $\Gamma_{c_k}$ 
// let item[tp] denote the item pointed by tp
if cp = leaf then
  while (not end_of_the_list ) and (cp ≠ null)
    if item[tp] < cp.ID then tp++ ; // advance tp
    else if item[tp] > cp.ID then cp = cp.sibling ; // advance cp
    else // matched candidate found
      cp.support++ ; // increment the support
      tp++ ; // advance tp
      cp = cp.sibling ; // advance cp
  end_while
else // cp is an internal node */
  if item[tp] < cp.ID then
    tp++ ; // advance tp
    Find_and_Increment(tp, cp) ;
  else if item[tp] > cp.ID then
    cp = cp.sibling ; // advance cp
    Find_and_Increment(tp, cp) ;
  else // matched
    Find_and_Increment(tp+1, cp.sibling) ;
    Find_and_Increment(tp+1, cp.next) ;
End_Procedure

```

Fig. 2-5. Procedure *Find_and_Increment*

If $x_q = Np.ID$, it means that candidates with same prefixed p items are found in the sub-list $\{x_1, \dots, x_q\}$ of T . Finding candidates whose p -th item is greater than $Np.ID$ can be done by comparing the sub-list $\{x_{q+1}, \dots, x_m\}$ with the sub-tree headed by $Np.sibling$. Next, we recursively apply *Find_and_Increment* on the sub-list $\{x_{q+1}, \dots, x_m\}$ and $Np.next$, which links the remaining $(k-p)$ items of candidate k -itemsets with same prefixed p items, for further sub-item comparisons. When p is $(k-1)$, it turns out to be a fast ordered list-matching between the sub-list $\{x_{q+1}, \dots, x_m\}$, and the list made of leaf $Np.next$ and the $(Np.next).sibling$ linked leaves.

Through eliminating impossible itemsets at each level, the number of comparisons is minimized in *LexMiner*. Moreover, the supports are efficiently updated for transactions containing many candidate k -itemsets, as demonstrated in Example 2-1. The notation $\langle T.x, Nd.ID: v \rangle$ means that the item 'x' in a transaction T is compared with v , the value of $Nd.ID$.

Example 2-1. Updating the supports of candidate 3-itemsets in $T = \{c, d, g, j, k\}$ is illustrated. In the beginning, cp points to the *Root* node Na .

- 1) $\langle T.c, Na.ID:a \rangle$, advance cp to Nb through $sibling$.
- 2) $\langle T.c, Nb.ID:c \rangle$ matched. We first apply *Find_and_Increment* on $cp.sibling$ to see whether there is any candidate in the sub-list, $\{d, g, j, k\}$, then apply on $cp.next$ to match the second and the third items of candidates having 3 as the first item.
- 3) Apply *Find_and_Increment* on $\{d, g, j, k\}$ and $Nb.sibling$: $\langle T.d, Nc.ID:d \rangle$, is matched. Again, *Find_and_Increment* is applied on $\{g, j, k\}$ with $Nc.sibling$ and on $\{g, j, k\}$ with $Nc.next$.
 - (a) Apply *Find_and_Increment* on $\{g, j, k\}$ and $Nc.sibling$: The matched $\langle T.g, Nd.ID:g \rangle$ induces two findings, on $\{j, k\}$ with $Nd.sibling$ and on $\{j, k\}$ with $Nd.next$. The former stops due to the null sibling pointer. The latter, though $\langle T.k, Ne.ID:k \rangle$ matches, stops since the end of the list is reached.
 - (b) Apply *Find_and_Increment* on $\{g, j, k\}$ and $Nc.next$: The matched $\langle T.g, Nx.ID:g \rangle$ recursively calls the procedure on $\{j, k\}$ with $Nx.sibling$ and on $\{j, k\}$ with $Nx.next$. The former eventually stops after reaching the end of the list. The latter finds the leaf Ny with ID the same as $T.j$, and increments $Ny.support$ by one. It means that the support of candidate (d, g, j) is incremented.
- 4) Apply *Find_and_Increment* on $\{d, g, j, k\}$ and $Nb.next$: The matched $\langle T.d, Nq.ID:d \rangle$ induces two findings on $\{g, j, k\}$ with $Nq.sibling$ and on $\{g, j, k\}$ with $Nq.next$.
 - (a) Apply *Find_and_Increment* on $\{g, j, k\}$ with $Nq.sibling$: This procedure with $\langle T.g, Nu.ID:g \rangle$ eventually will increment $Nv.support$ by one. That is, (c, g, k) will be updated.
 - (b) Apply *Find_and_Increment* on $\{g, j, k\}$ with $Nq.next$: Since Nr is a leaf node, it turns out to be a fast ordered list-matching between $\{g, j, k\}$ and the list made of $Nr.ID, Ns.ID$ and $Nt.ID$. The supports of (c, d, g) , (c, d, j) , and

(c, d, k) are incremented and the process is terminated at last. \square

As shown in this example, grouping candidates under *LexTree* enables fast list-matching at each level. In fact, the itemset matching in other algorithms, whether they explored lexicographic property or not, is broken down to a series of item matching in *LexMiner*. Efficient candidates matching by lexicographic comparisons and immediate increment of supports make *LexMiner* a faster approach for support counting.

2.4.3 Candidate generation by leaf joining

The *Apriori* generates C_k in three steps, collecting L_{k-1} by traversing candidate hash-tree, self-joining itemsets in L_{k-1} having same prefixed ($k-2$) items, and pruning those having any ($k-1$)-subset that is not in L_{k-1} . Most approaches like *Apriori* collect L_{k-1} in a hash table for fast searching in the pruning step. Nevertheless, a complete traversal over the hash table to find common prefixed ($k-2$) itemsets in the joining step is unavoidable. Such an inefficiency is removed by *leaf-join* in *LexMiner*.

In *LexMiner*, the frequent L_{k-1} *LexTree* is obtained by a traversal on a candidate C_{k-1} *LexTree* with a removal of leaves having insufficient supports. The traversal-with-removal results in a frequent L_{k-1} *LexTree*. In *LexTree*, all ($k-1$)-itemsets with the same prefixed ($k-2$) items are grouped together and are linked through the *sibling* pointers. Candidates are efficiently produced by making self-joins with these *sibling*-linked leaves. Before placing a newly generated candidate C in the new candidate C_k *LexTree*, we search in frequent L_{k-1} *LexTree* for the existence of all the ($k-1$)-subsets of C . If any of the searches fails, C is pruned. The search utilizes the similar technique used in fast support counting. Therefore, *LexMiner* generates the same number of candidates as *Apriori* does, in a faster speed. Note that some approaches exploring the lexicographic property, like *TreeProjection*, often generate a

slightly larger number of candidates [1].

2.5 Performance Evaluation

In order to evaluate the performance of the proposed algorithm, we conducted extensive experiments using an 866 MHz Pentium-III PC with 1024MB-memory running Windows NT. The databases are synthetic datasets of various characteristics. The method used to generate these datasets is described in Section 2.5.1. Section 2.5.2 compares the results of executions by various algorithms. Results on some scale-up experiments are presented in Section 2.5.3.

2.5.1 Generation of synthetic data

The synthetic data were generated by the well-known method in [5]. For completeness, we briefly review the method here. The datasets mimic the real world transactions. The total number of possible items for all transactions is $|N|$. The total number of transactions in the database D is $|D|$. $|T|$ is the average number of items in transactions. The size of each transaction is picked from a Poisson distribution with mean equal to $|T|$. The generation of transactions and the generation of potentially frequent itemsets (abbreviated as *PFI*s) are described in the following.

Each *PFI* comprises randomly picked items from the $|N|$ items. L is the set of *PFI*s and its size is $|L|$. The size of each *PFI* is determined following a Poisson distribution with mean equal to $|T|$. In order to model that there are common items in frequent itemsets, subsequent itemsets in L are related. In subsequent *PFI*s, a fraction of items are chosen from the previous *PFI*, the other items are picked at random. The fraction *corr*, called *correlation level*, is decided by an exponentially distributed random variable with mean equal to μ_{corr} . Items in the very first *PFI* are randomly chosen.

To model that all the items in a frequent itemset do not always jointly appear, each transaction consists of a series of fractions of *PFI*s [5]. Each *PFI* in L is assigned a weight, which corresponds to the probability that this itemset will be picked. Each weight is exponentially distributed and then normalized in such a way that the sum of all the weights is equal to one. In addition, each *PFI* is associated with a *corruption level* (abbreviated as *crup*). When adding items from a *PFI* to a transaction, an item is dropped as long as a uniformly distributed random number between 0 and 1 is less than *crup*. The *crup* is a normally distributed random variable with mean μ_{crup} and variance σ_{crup} .

The parameters used in the experiments are summarized in Table 2-3. All datasets used here are generated from 1000 items ($|N|=1000$), and the number of *PFI*s is 5000 ($|L|=5000$). Like most studies in association rule mining, the μ_{crup} , σ_{crup} , and μ_{corr} are set to 0.5, 0.1, and 0.5, respectively.

Table 2-3. Parameters used in the experiments

Parameter	Description	Value
$ D $	Number of transactions in database D	100K~10000K
$ N $	Number of possible items	1000
$ T $	Average number of items of transactions	10, 15, 18, 25
$ L $	Number of potentially frequent itemsets	5000
$ I $	Average size of potentially frequent itemsets	2, 6, 10, 12, 18

2.5.2 Total execution times of various algorithms

Extensive experiments were performed to realize the performance improvements of *LexMiner*. We implemented algorithms including *FP-growth* [28] and *TreeProjection* [1], and used a well-known version of *Apriori*, “GNU Lesser General Public License” available at <http://fuzzy.cs.uni-magdeburg.de/~borgelt/>, for comparisons. Algorithms

such as *Max-Miner* [12], *DepthFirst* [2], *CHARM* [100], and *CLOSET* [65] were not implemented since they discover only the sub-set, instead of the complete set, of frequent itemsets.

The *TreeProjection* we implemented is a memory-based version of the techniques reported in [1]. The cache-blocking technique to overcome extra disk I/O (when memory cannot hold large matrices) was not implemented since the lexicographic tree, all the matrices, and all the projected transaction sets can fit into the 1024MB memory in the experiments.

We first evaluated the effect of various *minsup*s for datasets having a typical value of 100,000 transactions. The notation $T\alpha-I\beta-D\gamma$ means that the dataset is created with $|T| = \alpha$, $|I| = \beta$, and $|D| = \gamma \times 1000$. The experiments started from the combination of (average size of transactions) $|T|=10$ and (average size of potentially frequent itemsets) $|I|=2$. When $|T|$ and $|I|$ are small, all the frequent itemsets can be found in few passes since most transactions comprise few items and most frequent itemsets have few items. As shown in Fig. 2-6, there is not much difference among these algorithms for *minsup* over 1.25%. The array-counting technique, especially in the pass 2 optimization, makes *LexMiner* faster than all the other algorithms for short patterns.

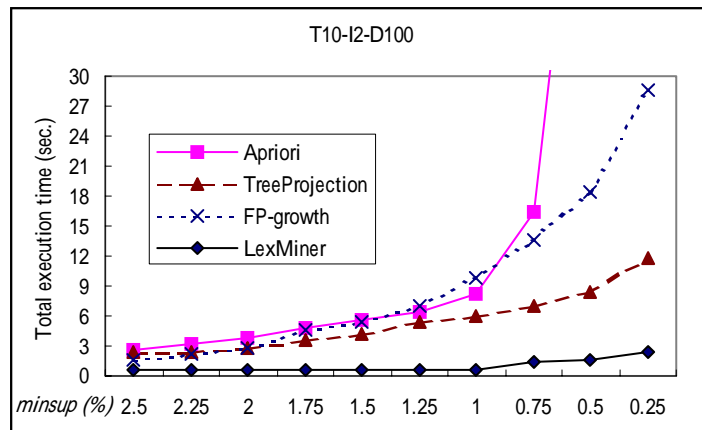
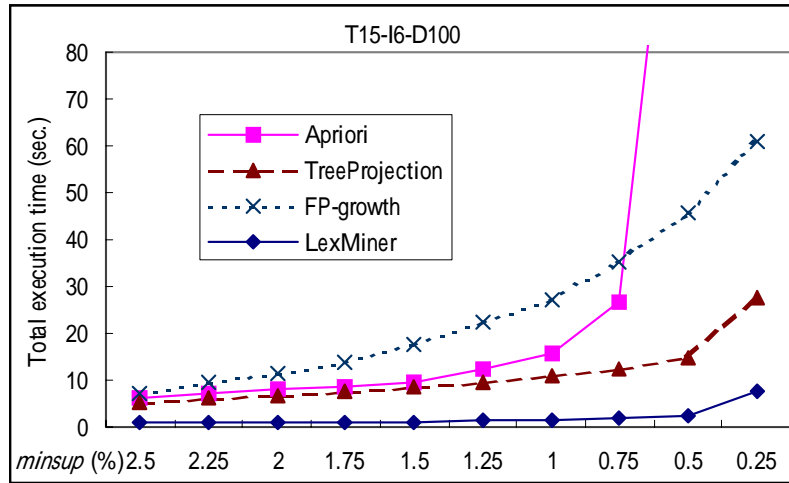
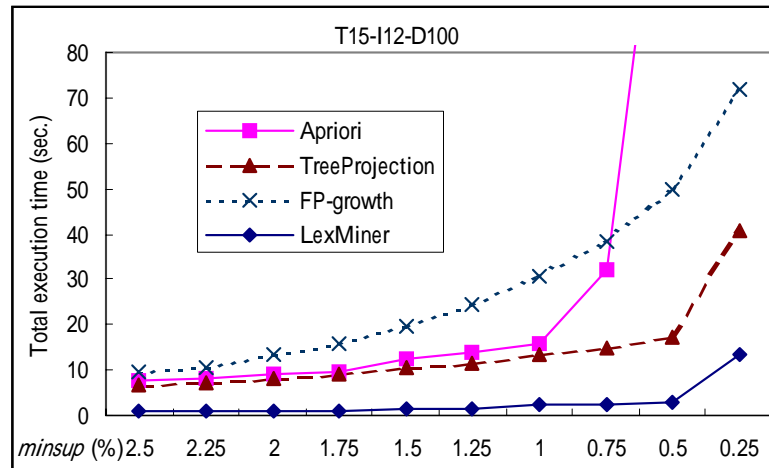


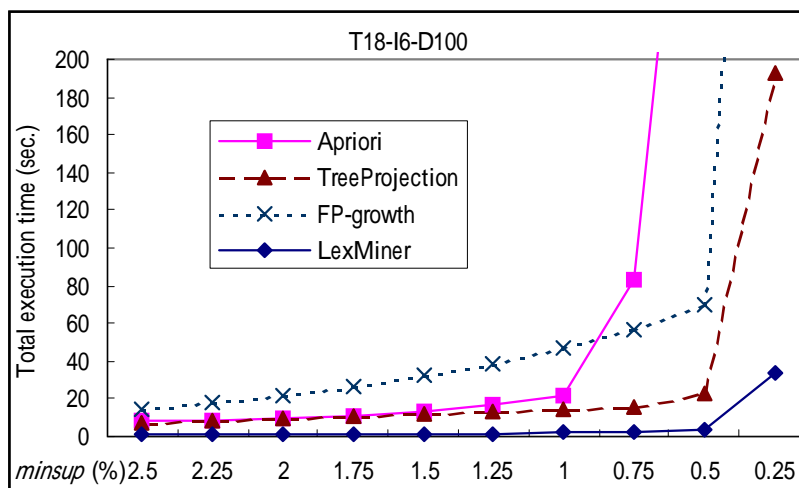
Fig. 2-6. Execution times of various algorithms on the dataset having short patterns



(a) dataset T15-I6-D100



(b) dataset T15-I12-D100



(c) dataset T18-I6-D100

Fig. 2-7. Performance comparisons of execution time over various supports

Next, various combinations of parameters $|T|$ and $|I|$ were used to generate other

datasets. The combinations of $|T|$ and $|I|$ in these experiments are T15-I6, T15-I12, and T18-I6. All the three datasets have frequent itemsets size bigger than two even with large *minsup* values. The relative performance among these algorithms is shown in Fig. 2-7. It can be seen from the figure that *LexMiner* outperforms the others over various minimum supports. The performance improvement is resulted from the fast item-matching, especially in lower levels when *minsup* was smaller than 0.75%.

Fig. 2-8 shows the performance of constructing *LexTree* using three ordering of items, support-ascending, support-descending, and lexicographic order. The tree constructed using support-ascending order is bushier (having more nodes) than the others. The effect of fast list-matching in leaves thus benefits support-ascending order the most. In the experiment, transactions in the dataset were not re-ordered so that item re-ordering (to cope with support ascending/descending ordered nodes) is required for each transaction, in every pass. Therefore, the performance gap is not clear until *minsup* is 0.25%, with the dense dataset T25-I18.

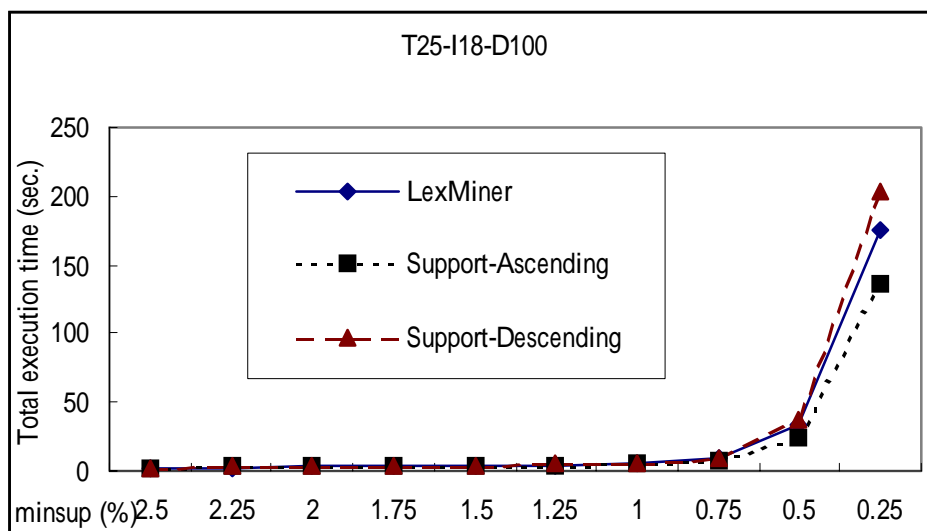
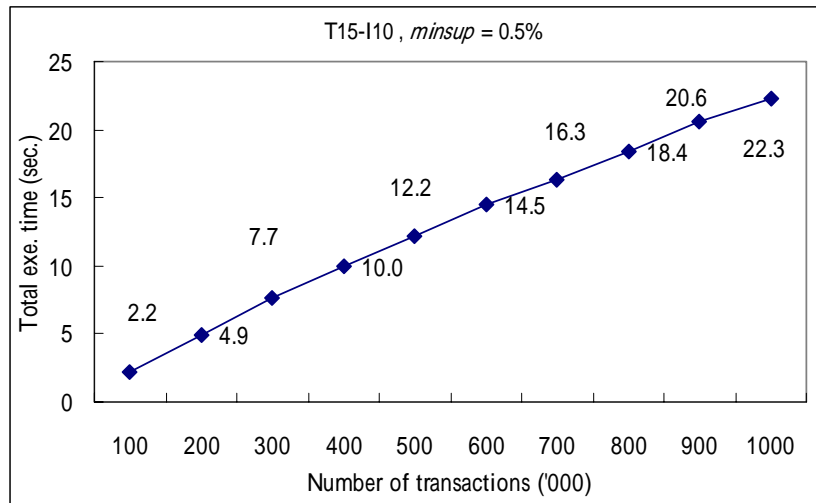


Fig. 2-8. Execution results of different ordering

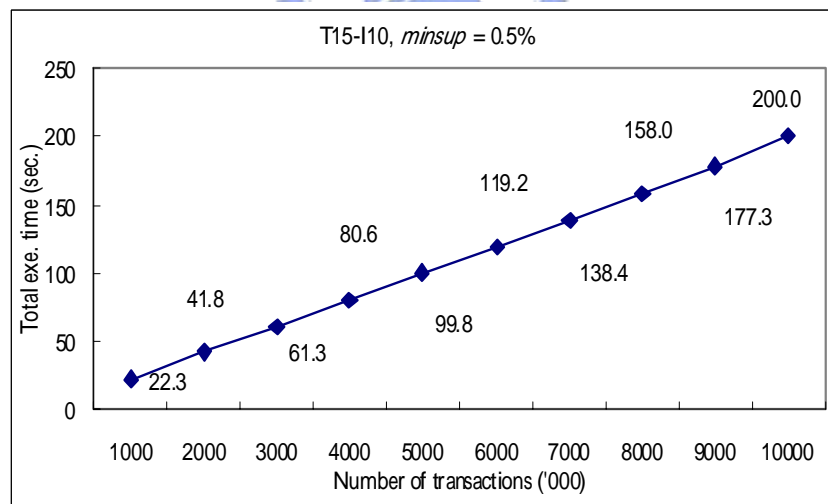
2.5.3 Scale-up experiments

To assess the scalability of our algorithm, several experiments were conducted. Fig.

2-9 shows that the execution time of *LexMiner* increases linearly as the database size increases, ranging from 100K to 10 million. The $|T|$ and the $|I|$ were fixed to see how execution time changes as the database size increases. Different *minsup*s yield similar and consistent results. Fig. 2-9 displays the result of *minsup* = 0.5%, which exhibits good linearity in scale-up.



(a) $|D| = 100,000$ to $1,000,000$



(b) $|D| = 1,000,000$ to $10,000,000$

Fig. 2-9. Linear scalability of the database size

2.6 Summary

The huge amount of data and the complicated interrelationships between data

bring about new challenges in the mining of undiscovered knowledge from large databases. Various algorithms have been developed for the discovery of association rules. However, the ordered property embedded in the transactions has never been fully utilized in existing mining algorithms. Therefore, we take into account the lexicographic nature of data and propose a novel approach for scalable mining of association rules. The proposed approach effectively places itemsets in a *LexTree* structure and discovers frequent itemsets efficiently by the *LexMiner* algorithm.

The *LexTree* structure provides a hierarchical ordering mechanism for storing candidate itemsets and frequent itemsets, and enables fast support counting and rapid candidate generation. In the *LexTree*, an itemset is uniquely represented by a sequential combination of nodes, one node in each level, in the tree. Itemsets having the same prefixed items share the same internal nodes and are grouped by these nodes. Due to sharing, the space used by the candidate *LexTree* is very compact in comparison with methods using hash-tree, which usually allocate additional storage while constructing hash tables. Due to grouping, we can generate candidates more rapidly from the frequent *LexTree* since no traversal is needed like those using hash tables for the storage of frequent itemsets.

In addition, the proposed *LexMiner* algorithm uses the *LexTree* to achieve fast support counting. Our focus is to investigate mechanisms to improve the most time-taking kernel operation of finding candidates in transactions, since the candidate-finding procedure is repeatedly executed for every transaction in every pass. *LexMiner* exploits the orderly placed candidates, breaks the finding into lexicographic comparisons to speed up the matching of candidates and prune the impossible candidates by hierarchical comparisons in each level. The intrinsically ordered transactions and the hierarchically ordered candidates together improve the matching efficiency. The speeding up of kernel computation is the key to performance

improvement. The comprehensive experiments also demonstrate that *LexMiner* coupled with *LexTree* is efficient and exhibits good scalability.

In addition to the discovery of association rules, the problem of sequential pattern mining generalizes the discovery of association rules to relationships of itemsets over time. The ordering property still holds for items in these transactions. It is worthy of further investigation on the mining of sequential patterns, which is explored in Chapter 3.



Chapter 3 Algorithm MEMISP for Sequential Pattern Mining

3.1 Overview

Frequent itemset mining, as discussed in Chapter 2, is extensively studied in data mining. A more complicated issue in data mining is the discovery of sequential patterns, which finds frequent sub-sequences in a sequence database. For example, in the transactional database of an electronic store, each record may correspond to a sequence of a customer's transactions ordered by transaction time. An example sequential pattern might be that customers typically bought *PC* and *printer*, followed by the purchase of *scanner* and *graphics software*, and then *digital camera*. The mining technique is applicable to many applications, including the analysis of Web traversal patterns, telecommunication alarms, DNA sequences, to name a few.

Sequential pattern mining is more difficult than association rule mining because the patterns are formed not only by combinations of items but also by permutations of item-sets. Enormous patterns can be formed as the length of a sequence is not limited and the items in a sequence are not necessarily distinct. Let the *size* of a sequence be the total number of items in that sequence. Given 100 possible items in a sequence database, the number of potential patterns of size two is $100 \cdot 100 + (100 \cdot 99)/2$, that of size three is $100 \cdot 100 \cdot 100 + 100 \cdot [(100 \cdot 99)/2] \cdot 2 + (100 \cdot 99 \cdot 98)/(2 \cdot 3)$, and so on. Owing to the challenge of exponential possible combinations, improving the efficiency of sequential pattern mining has been the focus of recent research in data mining [6, 11, 13, 25, 29, 38, 39, 42, 46, 47, 48, 55, 67, 72, 96, 98, 99].

In general, we may categorize the mining approaches into the generate-and-test framework and the pattern-growth one, for sequence databases of horizontal layout. Typifying the former approaches [6, 51, 80], the *GSP* (**Generalized Sequential Pattern**)

algorithm [80] generates potential patterns (called *candidates*), scans each data sequence in the database to compute the frequencies of candidates (called *supports*), and then identifies candidates having enough supports as sequential patterns. The sequential patterns in current database pass become seeds for generating candidates in the next pass. This generate-and-test process is repeated until no more new candidates are generated. When candidates cannot fit in memory in a batch, *GSP* re-scans the database to test the remaining candidates that have not been loaded into memory. Consequently, *GSP* scans at least k times of the on-disk database if the maximum size of the discovered patterns is k , which incurs high cost of disk reading. Despite that *GSP* was good at candidate pruning, the number of candidates is still very huge that might impair the mining efficiency.

The *PrefixSpan* (**P**refix-projected **S**equential **p**attern mining) algorithm [67], representing the pattern-growth methodology [29, 67, 70], finds the frequent items after scanning the sequence database once. The database is then projected, according to the frequent items, into several smaller databases. Finally, the complete set of sequential patterns is found by recursively growing subsequence fragments in each projected database. Two optimizations for minimizing disk projections were described in [67]. The *bi-level projection* technique, dealing with huge databases, scans each data sequence twice in the (projected) database so that fewer and smaller projected databases are generated. The *pseudo-projection* technique, avoiding physical projections, maintains the sequence-postfix of each data sequence in a projection by a pointer-offset pair. However, according to [67], maximum mining performance can be achieved only when the database size is reduced to the size accommodable by the main memory by employing *pseudo-projection* after using *bi-level* optimization. Although *PrefixSpan* successfully discovered patterns employing the divide-and-conquer strategy, the cost of disk I/O might be high due to the creation and

processing of the projected sub-databases.

Besides the horizontal layout, the sequence database can be transformed into a vertical format consisting of items' id-lists [11, 64, 98]. The id-list of an item is a list of (*sequence-id*, *timestamp*) pairs indicating the occurring timestamps of the item in that *sequence*. Searching in the lattice formed by id-list intersections, the *SPADE* (Sequential **P**attern **D**iscovery using **E**quivalence classes) algorithm [98] completed the mining in three passes of database scanning. Nevertheless, additional computation time is required to transform a database of horizontal layout to vertical format, which also requires additional storage space several times larger than that of the original sequence database.

With rapid cost down and the evidence of the increase in installed memory size, many small or medium sized databases will fit into the main memory. For example, a platform with 256MB memory may hold a database with one million sequences of total size 189MB. Pattern mining performed directly in memory now becomes possible. However, current approaches discover the patterns either through multiple scans of the database or by iterative database projections, thereby requiring abundant disk operations. The mining efficiency could be improved if the excessive disk I/O is reduced by enhancing memory utilization in the discovering process.

Therefore, we propose a memory-indexing approach for fast discovery of sequential patterns, called *MEMISP* (**M**EMory **I**ndexing for **S**equential **P**attern mining). The features of the *MEMISP* approach lie in no candidate generation, no database projection, and high CPU and memory utilization. *MEMISP* reads data sequences into memory in the first pass, which is the sole pass, of database scanning. Through index advancement within an index set comprising pointers and position indices to data sequences, *MEMISP* discovers patterns by a recursive finding-then-indexing technique. When the database is too large to fit into the main

memory, we still can mine patterns efficiently in two database scans by running *MEMISP* with a partition-and-validation technique discussed in Section 3.4.3. The conducted experiments show that *MEMISP* runs faster than both *GSP* and *PrefixSpan* algorithms, whether the main memory can accommodate the database or not.

The rest of the chapter is organized as follows. The problem is formulated in Section 3.2 and related work is reviewed in Section 3.3. Section 3.4 presents the *MEMISP* algorithm. The experimental results of mining memory-accommodable databases and extra-large databases are described in Section 3.5. We discuss the performance factors of *MEMISP* in Section 3.6 and conclude the study in Section 3.7.

3.2 Problem Statement

A *sequence* s , denoted by $\langle e_1 e_2 \dots e_n \rangle$, is an ordered set of n elements where each *element* e_i is an *itemset*. An *itemset*, denoted by (x_1, x_2, \dots, x_q) , is a nonempty set of q items, where each *item* x_j is represented by a literal. Without loss of generality, items in an element are assumed in lexicographic order. The *size* of sequence s , written as $|s|$, is the total number of items in all the elements in s . Sequence s is a *k-sequence* if $|s| = k$. For example, $\langle (a)(c)(a) \rangle$, $\langle (a,c)(a) \rangle$, and $\langle (b)(a,e) \rangle$ are all 3-sequences. A sequence $s = \langle e_1 e_2 \dots e_n \rangle$ is a *subsequence* of another sequence $s' = \langle e_1' e_2' \dots e_m' \rangle$ if there exist $1 \leq i_1 < i_2 < \dots < i_n \leq m$ such that $e_1 \subseteq e_{i_1}'$, $e_2 \subseteq e_{i_2}'$, ..., and $e_n \subseteq e_{i_n}'$. Sequence s' *contains* sequence s if s is a subsequence of s' . For example, $\langle (b,c)(c)(a,c,e) \rangle$ contains $\langle (b)(a,e) \rangle$.

Each sequence in the sequence database DB is referred to as a *data sequence*. The *support* of sequence s , denoted by $s.sup$, is the number of data sequences containing s divided by the total number of data sequences in DB . The *minsup* is the user specified minimum support threshold. A sequence s is a *frequent sequence*, or called *sequential pattern*, if $s.sup \geq minsup$. Given the *minsup* and the sequence

database *DB*, the problem of sequential pattern mining is to discover *the set of all sequential patterns*.

An example database *DB* having 6 data sequences is listed in the first column in Table 3-1. Take the data sequence *C6* for instance. It has three elements (i.e. three itemsets), the first having items *b* and *c*, the second having item *c*, and the third having items *a*, *c* and *e*. The support of $\langle(b)(a)\rangle$ is $4/6$ since all the data sequences, except *C2* and *C3*, contain $\langle(b)(a)\rangle$. The $\langle(a,d)(a)\rangle$ is a subsequence of both *C1* and *C4*, thus $\langle(a,d)(a)\rangle.\text{sup} = 2/6$. Given $\text{minsup} = 50\%$, $\langle(b)(a)\rangle$ is a sequential pattern while $\langle(a,d)(a)\rangle$ is not. The set of all sequential patterns is shown in the second column in Table 3-1.

Table 3-1. Example sequence database *DB* and the sequential patterns

Sequence	Sequential patterns ($\text{minsup}=50\%$)
$C1=\langle(a,d)(b,c)(a,e)\rangle$	$\langle(a)\rangle, \langle(a)(a)\rangle, \langle(a)(b)\rangle, \langle(a,c)\rangle, \langle(a,c)(a)\rangle, \langle(a,e)\rangle,$ $\langle(b)\rangle, \langle(b)(a)\rangle, \langle(b)(a,e)\rangle, \langle(b)(e)\rangle, \langle(b,c)\rangle, \langle(b,c)(a)\rangle,$
$C2=\langle(d,g)(c,f)(b,d)\rangle$	$\langle(b,c)(a,e)\rangle, \langle(b,c)(e)\rangle, \langle(b,d)\rangle,$
$C3=\langle(a,c)(d)(f)(b)\rangle$	$\langle(c)\rangle, \langle(c)(a)\rangle, \langle(c)(a,e)\rangle, \langle(c)(b)\rangle, \langle(c)(e)\rangle,$
$C4=\langle(a,b,c,d)(a)(b)\rangle$	$\langle(d)\rangle, \langle(d)(a)\rangle, \langle(d)(b)\rangle, \langle(d)(c)\rangle,$
$C5=\langle(b,c,d)(a,c,e)(a)\rangle$	$\langle(e)\rangle$
$C6=\langle(b,c)(c)(a,c,e)\rangle$	

3.3 Related Work

The problem of sequential pattern mining is first described and solved in [6] with the *AprioriAll* algorithm. In subsequent work, the same authors proposed the *GSP* algorithm [80] that outperforms *AprioriAll*. The *GSP* algorithm makes multiple passes over the database and finds frequent *k*-sequences at *k*-th database scanning. Initially, each item is a candidate *1*-sequence for the first pass. Frequent *1*-sequences are determined after checking all the data sequences in the database. In succeeding passes, frequent (*k-1*)-sequences are self-joined to generate candidate *k*-sequences, and then any candidate *k*-sequence having a non-frequent sub-sequence is deleted. Again, the

supports of candidate k -sequences are counted by examining all data sequences, and then those candidates having minimum supports become frequent sequences. This process terminates when there is no candidate sequence any more. Owing to the generate-and-test nature, the number of candidates often dominates the overall mining time. However, the total number of candidates increases exponentially as the *minsup* decreases, even with effective pruning techniques. The *PSP* (**P**refix **S**equential **P**attern) algorithm [51] is similar to *GSP*, except that the placement of candidates is improved by prefix tree arrangement to speed up the discovery.

The *FreeSpan* (**F**requent pattern-projected **S**equential **P**attern Mining) algorithm was proposed to mine sequential patterns by a database projection technique [29]. *FreeSpan* first finds the frequent items after scanning the database once. The sequence database is then projected, according to the frequent items, into several smaller databases. Finally, all sequential patterns are found by recursively growing subsequence fragments in each database. Based on the similar projection technique, the authors proposed the *PrefixSpan* algorithm [67]. *PrefixSpan* outperforms *FreeSpan* in that only effective postfixes are projected. The *bi-level* and *pseudo-projection* techniques further enhance *PrefixSpan* to project fewer sub-databases. However, the total size of the projected databases might be several times larger than the size of the original database.

In addition, the *SPADE* algorithm finds sequential patterns using vertical database layout and join-operations [98]. Vertical database layout transforms data sequences into items' id-lists. The id-list of an item is a list of (*sequence-id*, *timestamp*) pairs indicating the occurring timestamps of the item in that sequence-id. The list pairs are joined to form a sequence lattice, in which *SPADE* searches and discovers the patterns [98]. Nevertheless, transforming the naturally horizontal database into vertical demands more space than the original since a sequence-id is repeated in several items'

id-lists. The gain by vertical approach might diminish owing to the additional space and transforming time required while mining large databases.

In order to boost the mining performance, memory utilization should be increased to minimize disk operations, especially when dealing the ever-increasing sequence databases. Therefore, we propose the *MEMISP* algorithm, as described next.

3.4 The Proposed Method

In this section, the proposed method for sequential pattern mining, named *MEMISP*, is described. *MEMISP* uses a recursive find-then-index strategy to discover all the sequential patterns from in-memory data sequences. *MEMISP* first reads all the data sequences into memory and counts the supports of 1-sequences (i.e. sequences having only one item). Next, an index set for each frequent 1-sequence is constructed and then frequent sequences are found using the data sequences indicated by the index set. The algorithm is illustrated by mining an example database in Section 3.4.1. Section 3.4.2 presents the algorithm. The procedure for dealing with extra-large databases beyond main memory space is described in Section 3.4.3. Section 3.4.4 discusses the differences between *MEMISP* and *PrefixSpan*. Some implementation issues are discussed in Section 3.4.5.

3.4.1 Mining sequential patterns by *MEMISP*: an example

Definition 3-1(Type-1 pattern, type-2 **pattern**, **stem**, **P-pat**) Given a pattern ρ and a frequent item x in the sequence database DB , ρ' is a *type-1 pattern* if it can be formed by appending the itemset (x) as a new element to ρ , and is a *type-2 pattern* by extending the last element of ρ with x . The frequent item x is called the *stem-item* (abbreviated as *stem*) of the sequential pattern ρ' and ρ is the *prefix pattern* (abbreviated as *P-pat*) of ρ' .

For example, given a pattern $\langle a \rangle$ and the frequent item b , we have the *type-1* pattern $\langle a(b) \rangle$ by appending (b) to $\langle a \rangle$ and the *type-2* pattern $\langle (a,b) \rangle$ by extending $\langle a \rangle$ with b . The $\langle a \rangle$ is the *P-pat* and the b is the *stem* of both $\langle a(b) \rangle$ and $\langle (a,b) \rangle$. As to a *type-2* pattern $\langle (c)(a,d) \rangle$, its *P-pat* is $\langle (c)(a) \rangle$ and its *stem* is d . Note that the null sequence, denoted by $\langle \rangle$, is the *P-pat* of any frequent 1-sequence. Clearly, any frequent k -sequence is either a *type-1* pattern or a *type-2* pattern of a frequent $(k-1)$ -sequence.

Example 3-1: Given $minsup = 50\%$ and the *DB* in Table 3-1. *MEMISP* mines the patterns by the following steps.

Step 1. Read *DB* into memory and find frequent 1-sequences. We accumulate the count of every item while reading data sequences from *DB* into memory. The in-memory *DB* is referred to as ***MDB*** hereafter. Hence, we have frequent items a (count=5 for appearing in 5 data sequences $C1, C3, C4, C5, C6$), b (count=6), c (count=6), d (count=5), and e (count=3). All these frequent items are *stems* of the *type-1 patterns* with respect to the *P-pat* = $\langle \rangle$. **Loop steps 2 and 3 on each stem to find all the sequential patterns.**

Step 2. Output the sequential pattern ρ formed by current *P-pat* and stem x , and construct the index set ρ -*idx*. We output a sequential pattern ρ generated by current *P-pat* and stem x . Next, we allocate a **(ptr_ds, pos)** pair for each data sequence ds in *MDB* if and only if ds contains x , where **ptr_ds** is a pointer to ds and **pos** is the first occurring position of x in ds . The set of these **(ptr_ds, pos)** pairs is called **index set ρ -*idx***.

Take stem $x = a$ for example. Now, the *P-pat* is $\langle \rangle$. We output the *type-1* sequential pattern $\rho = \langle a \rangle$ and construct the index set $\langle a \rangle$ -*idx* as shown in Fig. 3-1-(1). For instance, the **pos** is **1** for $C1 = \langle (a,d)(b,c)(a,e) \rangle$ and **4** for $C6 = \langle (b,c)(c)(a,c,e) \rangle$.

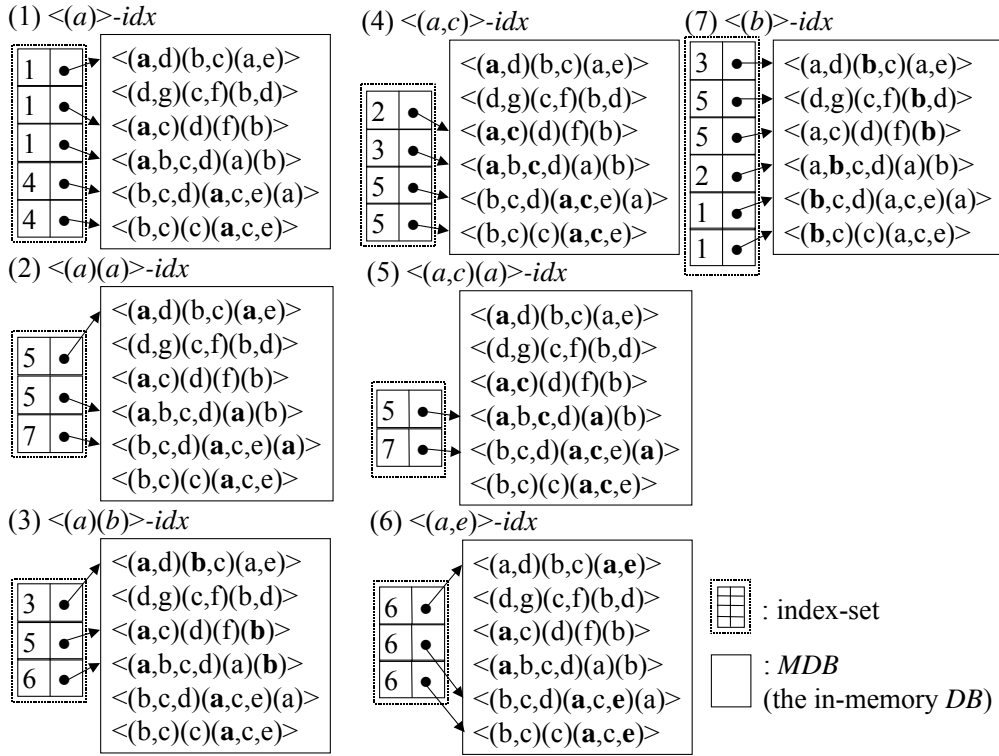


Fig. 3-1. Some index sets and the in-memory DB

Step 3. Use index set $\rho\text{-idx}$ and MDB to find stems with respect to $P\text{-pat} = \rho$. Any sequential pattern having current pattern ρ as its $P\text{-pat}$ will be identified in this step. Now, the ptr_ds of each $(\text{ptr_ds}, \text{pos})$ pair in $\rho\text{-idx}$ points to a data sequence ds that contains ρ . Any item appearing after the pos position in ds could be a potential stem (with respect to ρ). Thus, for every ds existing in $\rho\text{-idx}$, we increase the count of such item (item appearing after the pos in ds) by one, and then identify the stems having sufficient support counts.

Let us continue with $\langle(a)\rangle\text{-idx}$. The pos of the $(\text{ptr_ds}, \text{pos})$ pointing to $C1$ is 1. Only those items occurring after position 1 in $C1$ need counting. We increase the count of potential stem d (for potential type-2 pattern $\langle(a,d)\rangle$) by one (also potential stem e for $\langle(a,e)\rangle$). We also increase the count of potential stem b (also c , a , and e) for potential type-1 pattern $\langle(a)(b)\rangle$ ($\langle(a)(c)\rangle$, $\langle(a)(a)\rangle$, and $\langle(a)(e)\rangle$) by one. Analogously, items occurring after position 1, 1, 4, 4 for data sequences $C3$, $C4$, $C5$, and $C6$ are counted, respectively. After validating the

support counts, we obtain stems a, b of *type-1* patterns and stems c, e of *type-2* patterns with respect to $P\text{-pat} = \langle(a)\rangle$. Steps 2 and 3 will be recursively applied on the stems a, b, c , and e with $P\text{-pat} = \langle(a)\rangle$. We proceed the mining with stem a and $P\text{-pat} = \langle(a)\rangle$ as follows.

Applying step 2 generates and outputs the sequential pattern $\rho = \langle(a)(a)\rangle$. Again, a new **(ptr_ds, pos)** pair for a data sequence ds will be inserted into $\rho\text{-idx}$ ($\langle(a)(a)\rangle\text{-idx}$) if and only if ds contains ρ . While constructing $\langle(a)(a)\rangle\text{-idx}$, we simply check the data sequences indicated by current index set, i.e. $\langle(a)\rangle\text{-idx}$, rather than in MDB . Assume that a pair **(ptr_ds, pos)** in $\langle(a)\rangle\text{-idx}$ points to ds . The search for the occurring position of stem a (with respect to $P\text{-pat} = \langle(a)\rangle$) starts from position **pos+1** in ds . Item a occurs at 5 in $C1$ and in $C4$, and at 7 in $C5$. No entry is created for $C3$ and $C6$ since item a cannot be found after position 1 and 4, respectively. Hence, we have the new index set $\langle(a)(a)\rangle\text{-idx}$ as shown in Fig. 3-1-(2). Note that current index set is ‘pushed’ for later mining before the new index set becomes active.

Applying step 3 with $\langle(a)(a)\rangle\text{-idx}$ and MDB , no stems can form sequential patterns further. Therefore, this mining stops and the previous index set, i.e. $\langle(a)\rangle\text{-idx}$, is popped. The mining goes on with stem b . The creation and mining of $\langle(a)(b)\rangle\text{-idx}$ outputs pattern $\langle(a)(b)\rangle$ but finds no more patterns. Next, the $\langle(a,c)\rangle\text{-idx}$ is constructed. The result of applying step 2 with $\langle(a,c)\rangle\text{-idx}$ generates $\langle(a,c)\rangle$ and discovers next stem a . Thus, $\langle(a,c)\rangle\text{-idx}$ is ‘pushed’ and the $\langle(a,c)(a)\rangle\text{-idx}$ is created.

After the mining with $\langle(a,c)(a)\rangle\text{-idx}$, which stops with nothing found but outputs the pattern $\langle(a,c)(a)\rangle$, the pattern $\langle(a,e)\rangle$ is generated while mining with $\langle(a,e)\rangle\text{-idx}$. All the subsequent find-then-index processes regarding stem a with $P\text{-pat} = \langle\rangle$ now finish.

By collecting the patterns found in the above process, *MEMISP* efficiently discovers all the sequential patterns.

3.4.2 The *MEMISP* algorithm

The central idea of *MEMISP* is to utilize the memory for both data sequences and indices in the mining process. A memory size of 256MB is very common in nowadays computer installation, which can accommodate a sequence database having one million sequences of size 189MB as indicated in our experiments. Processing sequences in-memory is more efficient than disk-based processing, either multiple scans or iterative projections. *MEMISP* scans only one pass over the database, which reads data sequences into memory, in the whole mining process. Starting from sequential patterns of size one, *MEMISP* then discovers all the frequent sequences of larger size recursively by searching the set of in-memory data sequences having common sub-sequences. Fig. 3-2 outlines the proposed *MEMISP* algorithm.

In order to speed up mining by focused search, we construct a *set* grouping the data sequences to check. A data sequence ds participates in the finding of pattern ρ' only when ds contains the *P-pat* (prefix-pattern) ρ of pattern ρ' . Consequently, for each ds containing ρ , we create a pointer ptr_ds pointing to ds in the set for exploring patterns ρ' having *P-pat* ρ . The set is denoted by $\rho\text{-idx}$. For each data sequence ds pointed in the $\rho\text{-idx}$, we associate ptr_ds with a position index pos indicating where (in ds) should we begin to find the potential stems. That is, $\rho\text{-idx}$ is the set of (ptr_ds, pos) pairs for discovering patterns whose *P-pat* = ρ .

Take the data sequence $C6 = \langle (\mathbf{b}, \mathbf{c})(\mathbf{c})(\mathbf{a}, \mathbf{c}, \mathbf{e}) \rangle$ in memory for instance. We may find $\langle (\mathbf{b}) \rangle$ occurring at position 1, $\langle (\mathbf{b}, \mathbf{c}) \rangle$ occurring at composite position (1, 2), and $\langle (\mathbf{b}, \mathbf{c})(\mathbf{a}) \rangle$ occurring at composite position (1, 2, 4). Assume that items b , c , and a are frequent. While mining patterns having *P-pat* $\langle (\mathbf{b}) \rangle$, we include $C6$ in the index set

Algorithm *MEMISP*

Input: DB = a sequence database; $minsup$ = minimum support.

Output: the set of all sequential patterns.

Method:

1. Scan DB into MDB (the in-memory DB), find the set of all frequent items.
2. For each frequent item x ,
 - (i) form the sequential pattern $\rho = \langle x \rangle$ and output ρ .
 - (ii) call $IndexSet(x, \langle \rangle, MDB)$ to construct the index set $\rho\text{-idx}$.
 - (iii) call $Mine(\rho, \rho\text{-idx})$ to mine patterns with index set $\rho\text{-idx}$.

Subroutine $IndexSet(x, \rho, range\text{-set})$

Parameters: x = a stem-item; ρ = a ($P\text{-pat}$) pattern; $range\text{-set}$ = the set of data sequences for indexing. /* If $range\text{-set}$ is an index set, each data sequence for indexing is pointed by the ptr_ds of the (ptr_ds, pos) entry in the index set */

Output: index set $\rho'\text{-idx}$, where ρ' denotes the pattern formed by stem-item x and $P\text{-pat}$ ρ .

Method:

1. For each data sequence ds in $range\text{-set}$,
 - (i) if $range\text{-set} = MDB$ then $start\text{-pos} = 0$; otherwise $start\text{-pos} = pos$.
 - (ii) starting from position ($start\text{-pos}+1$) in ds ,
if the stem-item x is first found at position pos in ds , insert a (ptr_ds, pos) pair to the index set $\rho'\text{-idx}$, where ptr_ds points to ds .
2. Return index set $\rho'\text{-idx}$.

Subroutine $Mine(\rho, \rho\text{-idx})$

Parameter: ρ = a pattern; $\rho\text{-idx}$ = an index set.

Method:

- For each data sequence ds pointed by the ptr_ds of an entry (ptr_ds, pos) in $\rho\text{-idx}$,
- (i) starting from position ($pos+1$) to $|ds|$ in ds , increase the support count of each potential stem x by one.
 2. Find the set of stems x having enough support count to form a sequential pattern.
 3. For each stem x ,
 - (i) form the sequential pattern ρ' with $P\text{-pat}$ ρ and stem x , output ρ' .
 - (ii) call $IndexSet(x, \rho, \rho\text{-idx})$ to construct the index set $\rho'\text{-idx}$.
 - (iii) call $Mine(\rho', \rho'\text{-idx})$ to mine patterns with index set $\rho'\text{-idx}$.

Fig. 3-2. Algorithm *MEMISP*

with $pos=1$, suggesting that only items appearing after position 1 in $C6$ should engage in the mining. Similarly, $C6$ will be included in the index set for patterns having $P\text{-pat}$

$\langle(b,c)\rangle$ with $\mathbf{pos}=2$, $P\text{-pat} \langle(b,c)(a)\rangle$ with $\mathbf{pos}=4$. As the discovered $P\text{-pat}$ becomes longer, the index set will contain fewer data sequences to process. Moreover, the number of items in each data sequence remaining to be processed becomes fewer. Through recursive finding-then-indexing, the proposed *MEMISP* algorithm efficiently discovers sequential patterns.

3.4.3 Dealing with extra-large databases by database partitioning

With more and more memory installed, many databases will fit into the main memory without difficulty. Still, some databases might be too large for the main memory to accommodate in a batch. In this case, the sequential patterns are discovered by a partition-and-validation technique, as shown in Fig. 3-3.

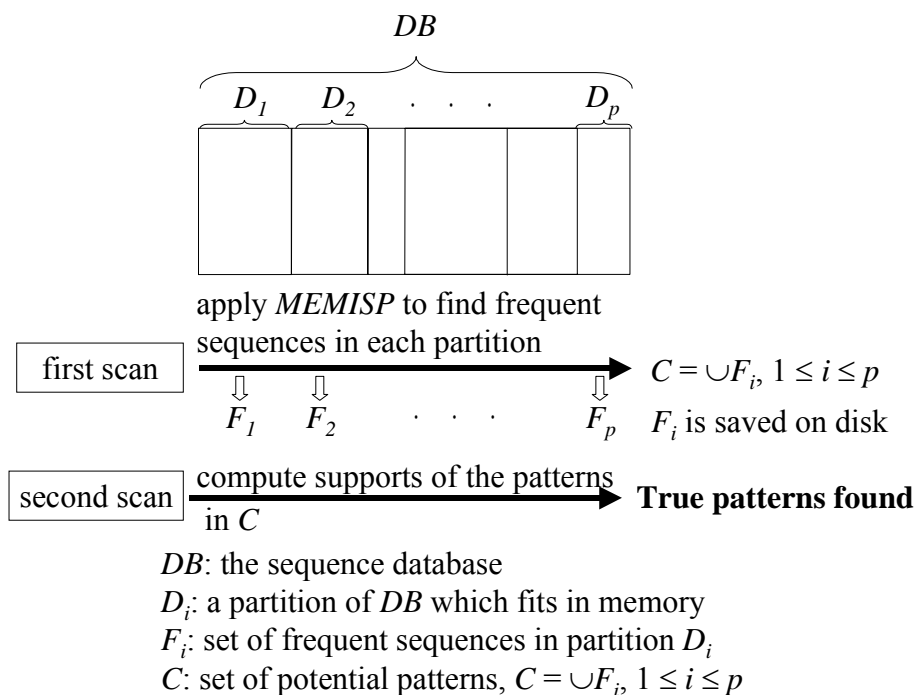


Fig. 3-3. Partition the database and discover patterns for extra-large databases

The extra-large database *DB* is partitioned so that each partition can be handled in main memory by *MEMISP*. The number of partitions is minimized by reading as many data sequences into main memory as possible to constitute a partition. The set of potential patterns in *DB* is obtained by collecting the discovered patterns after

running *MEMISP* on these partitions. The true patterns can be identified with only one extra database pass through support counting against the data sequences in *DB* one at a time. Therefore, we may employ *MEMISP* to mine databases of any size, of any minimum support, in two passes of database scanning.

In comparison with other approaches, *MEMISP* minimizes the total number of complete database passes to two without requiring any additional storage space. *SPADE* needs to scan the database three times and demands disk storage for the transformed vertical database. *GSP* repeats at least k times to discover the frequent k -sequences. *PrefixSpan* often creates and processes the projected databases that amount to several times the original database size.

3.4.4 Differences between *MEMISP* and *PrefixSpan*

The *PrefixSpan* algorithm proposed in [67] can be optimized with bi-level and pseudo-projection techniques. Pseudo-projection technique avoids redundant pieces of postfixes projected when the database/projected database can be held in main memory. *PrefixSpan* and *MEMISP* do differ, although the two algorithms both utilize memory for fast computation. The differences are illustrated in the following two cases: (1) when the database can be held in main memory (2) when the database cannot be held in main memory.

When the database can be held in main memory, the two algorithms find the patterns in a similar, but still different way. Both algorithms load the database into memory, but disagree with the processing of in-memory sequences. *PrefixSpan* algorithm removes in-frequent items and greatly shrinks projected sequences. The example 3 in [67] clearly demonstrates such projections so that item g is not projected in Table 2 of [67]. Pseudo-projection maintains the sequence-postfix of each data sequence in memory by a pointer-offset pair. The detailed implementation of

PrefixSpan with the pseudo-projection technique is not available in the literature. To sustain the spirit of *PrefixSpan*, in-frequent items are to be removed when pseudo-projection is applied. For the in-memory sequences, removing in-frequent items could be done, for example, by copying only frequent items in postfixes or masking out the in-frequent items. Therefore, besides the index tables, an intermediate in-memory working database must be generated to present the physically projected sub-database in each iteration. No matter what the implementation is, the postfixes (sequences) require rearrangements.

MEMISP algorithm removes no items from the in-memory sequences. No intermediate in-memory database generation and no rearrangement of sequences are required at all. Single sole in-memory sequence database as originally loaded is used throughout the whole process. We shift the index without modifying any in-memory sequence to skip the in-frequent items in each iteration. Indeed, the (ds_ptr, pos) index pairs in *MEMISP* function similarly as the $(pointer, offset)$ index pairs in *PrefixSpan*+pseudo_projection for sequence processing. We believe that fast index advancement eliminate the need to process the in-frequent items.

When the database cannot be held in main memory, *MEMISP* is totally different from *PrefixSpan*. *PrefixSpan*, either with pseudo-projection or not, now generates and scans sub-databases that might amount to several times the original database size. Even with bi-level projection technique, *PrefixSpan* still might suffer from low support value for generating many projected sub-databases before pseudo-projection could help. With respect to any support value, *MEMISP* scans the database only twice, and no more, without generating any intermediate databases.

Bi-level projection is proposed to reduce the number and the size of projected databases, at the cost of doubled scanning to fill the *S-matrix* (see Lemma 3.3 [67]). Dealing extra-large databases with bi-level projection means that the entire database

is scanned at least twice at first. Next, if each projected database could be luckily fit into the memory, pseudo-projection can be applied. This gives the fewest scans, which is more than twice in total, *PrefixSpan* can do. Otherwise, re-applying bi-level projection could result in the total number of scans to be far more than two.

MEMISP partitions the extra-large database to several sub-databases; each sub-database can be fit into the memory. The first scan, which mines each sub-database independently by *MEMISP*, identifies the potential candidates. The second scan verifies whether a candidate has sufficient support to be frequent. *MEMISP* never scans the database, no matter how large the database is, more than twice for any value of support. In addition, *MEMISP* never generates any intermediate database during the mining process. The partition-based approach is used in [73] for association rule mining. However, *MEMISP* is the first algorithm that successfully adapts the partitioning technique to the mining of sequential patterns in the literature.

3.4.5 Implementation issues

In common implementations, a data sequence is usually represented as a linked list of itemsets in memory. Such a structure might be suitable for algorithms that access a single data sequence for support counting at a time. In order to facilitate fast index construction and speed up searching from specific position (in a data sequence), *MEMISP* uses variable-length arrays to hold the data sequences in memory. Data sequence $CI = \langle (a,d)(b,c)(a,e) \rangle$, for instance, is coded as the array = [a, d, \$, b, c, \$, a, e, \$], where \$ indicates the end of an element. Therefore, both data sequences and index sets benefit from the array representation for the reduced storage space. Efficient searching from specific position of a data sequence is also achieved.

When mining databases that require partitioning, a percentage of main memory (like 5%) must be reserved for holding variables, index sets, etc. In order to signal that

main memory cannot accept data sequence any more, the amount of available physical memory is checked periodically while reading the database. Once free memory space is below the predefined percentage, *MEMISP* starts mining the memory partition and the remaining data sequences will be handled in subsequent reading.

3.5 Performance Evaluation

Extensive experiments were conducted to assess the performance of the *MEMISP* algorithm. The experiments used an 866 MHz Pentium-III PC with 256MB memory running the Windows NT. Like most studies on sequential pattern mining [6, 11, 13, 29, 51, 67, 98], the synthetic datasets for these experiments were generated using the conventional procedure described in [6]. We briefly review the generation of experimental data in Section 3.5.1. Section 3.5.2 compares the results of mining by *GSP*, *PrefixSpan*, and *MEMISP* algorithms. To justify that *MEMISP* handles large databases as well, scale-up experiments are presented in Section 3.5.3.

3.5.1 Generation of experimental data

The procedure described in [6] models retailing environment, where each customer purchases a sequence of itemsets. Such a sequence is referred to as a *potentially frequent sequence* (abbreviated as *PFS*). Still, some customers might buy only some of the items from a *PFS*. A customer's data sequence may consist of items from several *PFSs*. The *PFSs* are composed of *potentially frequent itemsets* (abbreviated as *PFI*s). A table of total N_I *PFI*s (denoted by Γ_I) and a table of total N_S *PFS*s (denoted by Γ_S) were generated before picking items for the transactions of customer sequences.

Table 3-2 summarizes the symbols and the parameters used in the experiments.

The procedure of data sequence generation [6] is reviewed here, first the generation of *PFI*s and *PFS*s, and then the customer sequences. The number of itemsets in a *PFS* is generated by picking from a Poisson distribution with mean equal to $|S|$. The itemsets in a *PFS* are picked from table I_I . In order to model that there are common itemsets in frequent sequences, subsequent *PFS*s in I_S are related. In the subsequent *PFS*, a fraction of itemsets are chosen from the previous *PFS* and the other itemsets are picked at random from I_I . The fraction $corr_S$, called *correlation level*, is decided by an exponentially distributed random variable with mean equal to μ_{corr_S} . Itemsets in the first *PFS* in I_S are randomly picked. The generations of *PFI* and I_I are analogous to the generations of *PFS* and I_S , with parameters N items, mean $|I|$, *correlation level* $corr_I$ and mean μ_{corr_I} correspondingly.

Customer sequences are generated as follows. The number of transactions for the next customer and the average size of transactions for this customer are determined first. The size of the customer's data sequence is picked from a Poisson distribution with mean equal to $|C|$. The average size of the transactions is picked from a Poisson distribution with mean equal to $|T|$. Items are then assigned to the transactions of the customer. Each customer is assigned a series of *PFS*s from table I_S .

The assignment of *PFS*s is based on the weights of *PFS*s. The weight of the *PFS*, representing the probability that this *PFS* will be chosen, is exponentially distributed and then normalized in such a way that the sum of all the weights is equal to one. Since all the itemsets in a *PFS* are not always bought together, each sequence in I_S is assigned a *corruption level* $crup_S$. When selecting itemsets from a *PFS* to a customer sequence, an itemset is dropped as long as a uniformly distributed random number between 0 and 1 is less than $crup_S$. The $crup_S$ is a normally distributed random variable with mean μ_{crup_S} and variance σ_{crup_S} . The assignment of *PFI*s (from I_I) to

Table 3-2. Parameters used in the experiments

Parameter	Description	Value
$ DB $	Number of data sequences in database DB	200K, 500K, 1000K, 10000K
$ C $	Average size (number of transactions) per customer	10, 20
$ T $	Average size (number of items) per transaction	2.5, 5
$ S $	Average size of potentially sequential patterns	4, 8
$ I $	Average size of potentially frequent itemsets	1.25, 2.5
N_I	Number of potentially frequent itemsets	25000
N_S	Number of possible sequential patterns	5000
N	Number of possible items	10000
Γ_S	The table of <i>potentially frequent sequences (PFSs)</i>	
Γ_I	The table of <i>potentially frequent itemsets (PFIs)</i>	
$corr_S$	Correlation level (sequence), exponentially distributed	$\mu_{corr_S} = 0.25$
$crup_S$	Corruption level (sequence), normally distributed	$\mu_{crup_S} = 0.75,$ $\sigma_{crup_S} = 0.1$
$corr_I$	Correlation level (itemset), exponentially distributed	$\mu_{corr_I} = 0.25$
$crup_I$	Corruption level (itemset), normally distributed	$\mu_{crup_I} = 0.75,$ $\sigma_{crup_I} = 0.1$

a *PFS* is processed analogously with parameters $crup_I$, mean μ_{crup_I} and variance σ_{crup_I} correspondingly.

All datasets used here were generated by setting $N = 10000$, $N_S = 5000$, $N_I = 25000$. A dataset created with $|C| = \alpha$, $|T| = \beta$, $|S| = \chi$, and $|I| = \delta$ is denoted by the notation $C\alpha-T\beta-S\chi-I\delta$. In addition, μ_{crup_S} and μ_{crup_I} were both set to 0.75, σ_{crup_S} and σ_{crup_I} were both set to 0.1. The μ_{corr_S} and μ_{corr_I} were both set to 0.25.

3.5.2 Execution times of GSP, PrefixSpan, and MEMISP algorithms

The total execution times of sequence mining with various *minsup* values by algorithms *GSP*, *PrefixSpan*, and *MEMISP* using horizontal layout are compared in the experiments. The *PrefixSpan* was implemented without further optimizations like pseudo-projection or bi-level projection. The *SPADE* algorithm was not implemented in the comparison because additional storage space and computation time are required to transform the database to vertical format.

Dataset *C10-T2.5-S4-I1.25* having 200,000 data sequences (37.6MB) was used in the first experiment. Fig. 3-4 shows that the total execution times of the three algorithms are nearly the same for *minsup* = 2% and 1.5% because only few (less than 200) patterns have enough supports. Besides, the discovered patterns were all short patterns of size one. However, the performance gaps become clear as *minsup* decreases. In the experiment, *MEMISP* and *PrefixSpan* are faster than *GSP* for all *minsup* values. *MEMISP* outperforms *PrefixSpan* about 13%~38% for low *minsup*.

Next, the characteristics of datasets are changed. The results of execution on dataset *C20-T2.5-S4-I1.25* ($|DB|=200K$, 76.3MB) is shown in Fig. 3-5. The total execution time of running *GSP* was too long to be shown in Fig. 3-5 and in the subsequent figures. With respect to the same *minsup*s, the doubled $|C|$ generated longer data sequences and produced more patterns, thereby requiring more execution time. The total execution time of running *PrefixSpan* is about 1.2 to 3.3 times of running *MEMISP*. The efficiency of *PrefixSpan* was slowed down by fast growth of the projected databases. For example, *PrefixSpan* processed total 4.9 times, and 21 times the size of *DB* when *minsup* = 2% and *minsup* = 0.75%, respectively. The results of execution by changing $|T|$ from 2.5 to 5, $|S|$ from 4 to 8, and $|I|$ from 1.25 to 2.5 have the similar effects. Fig. 3-6, 3-7, and 3-8 display that *MEMISP* outperforms

PrefixSpan. Fig. 3-9 shows that the performance of running with a bigger $|T|$ and a bigger $|I|$ ($|T|=7.5$, $|I|=5$) is consistent with previous experiments. The performance gain resulted from in-memory processing of the *MEMISP* algorithm. In summary, *MEMISP* is faster than *PrefixSpan*, ranging from 1.2 to 3.3 times, for various data characteristics.

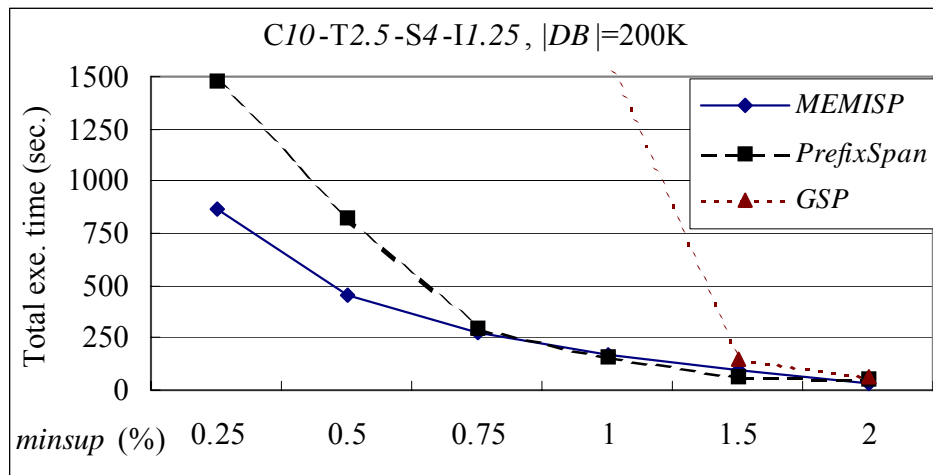


Fig. 3-4. Total execution times with respect to various *minsup* values

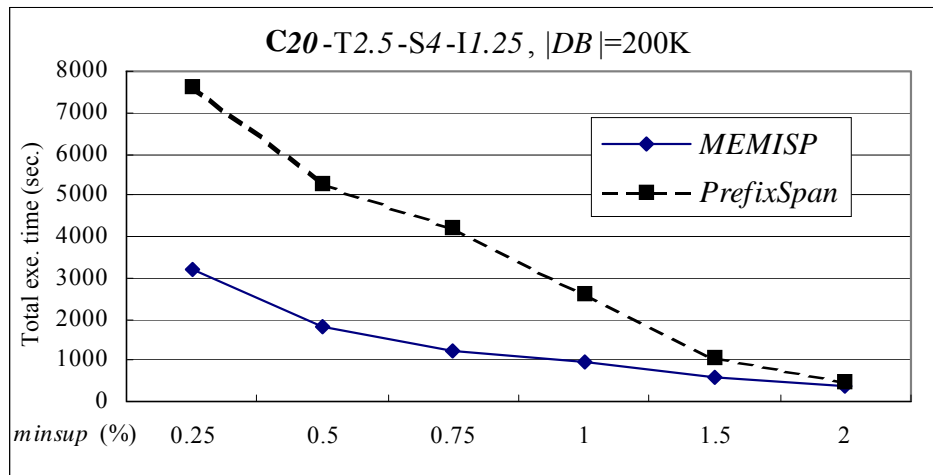


Fig. 3-5. Comparisons of execution times on dataset C20-T2.5-S4-I1.25

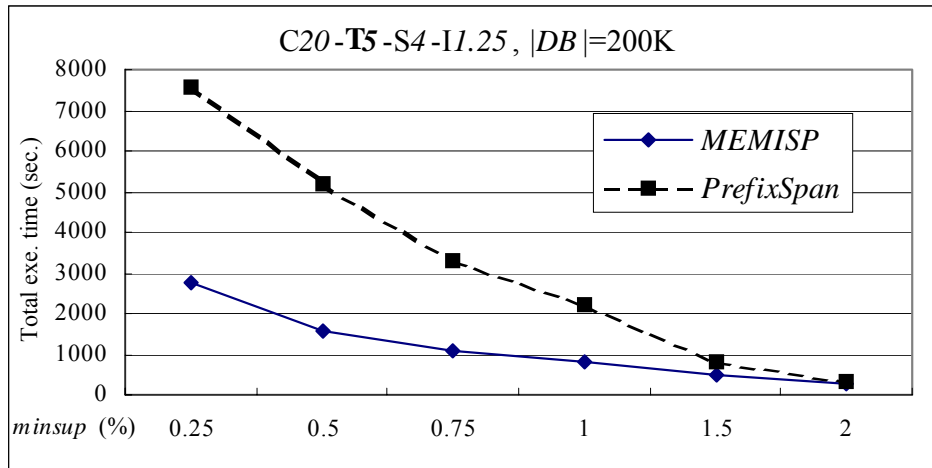


Fig. 3-6. Comparisons of execution times on dataset C10-T5-S4-I1.25

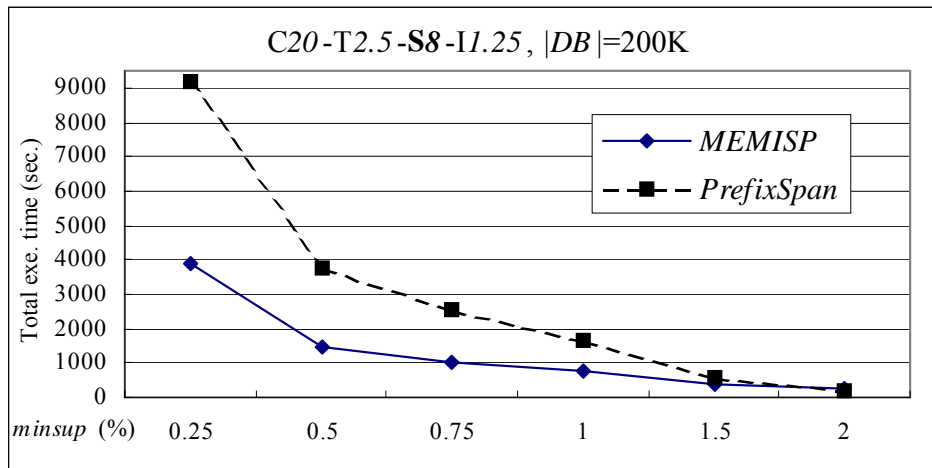


Fig. 3-7. Comparisons of execution times on dataset C10-T2.5-S8-I1.25

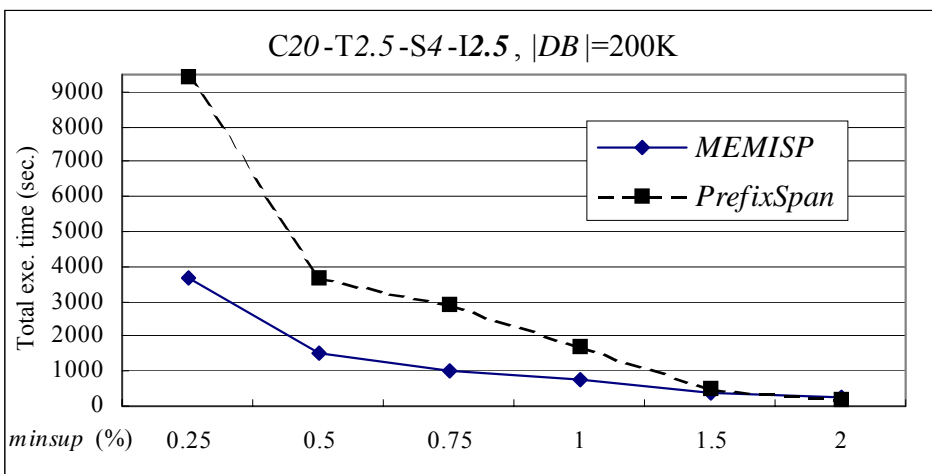


Fig. 3-8. Comparisons of execution times on dataset C10-T2.5-S4-I2.5

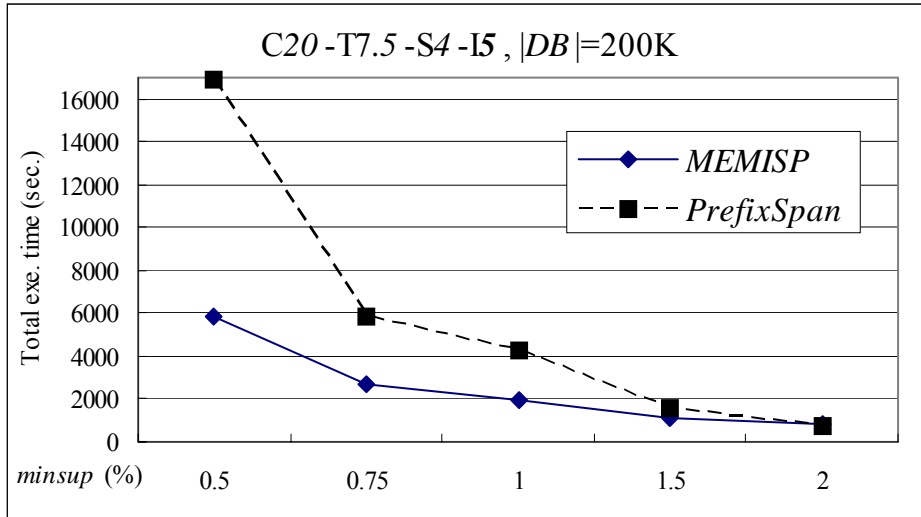


Fig. 3-9. Comparisons of execution times on dataset C10-T7.5-S4-I5

3.5.3 Scale-up experiments

The maximum size of the datasets used in Section 3.5.2 is 76.3MB, the C20-T2.5-S4-I1.25 dataset with 200,000 sequences. Consequently, all the data sequences can fit into the 256MB main memory. The performance of *MEMISP* is very stable even when *minsup* is very low for large databases, if the database can fit into memory. Given *minsup* = 0.25%, *MEMISP* can perform well in processing one million data sequences of total size 189MB with a 256MB main memory in the experiments. Nevertheless, just for the mining of 100K sequences with *minsup* = 0.5%, *GSP* scanned the database 4 times to test the 4.4 million candidates in pass two (more passes to go), and *PrefixSpan* generated sub-databases which amounts to 9.6 times the size of the original database.

In order to justify the scalability of *MEMISP*, the next experiments increased the number of data sequences, from 1000K to 10,000K with C10-T2.5-S4-I1.25. In Fig. 3-10, the total execution times are normalized with respect to the execution time for $|DB| = 1000K$. The size of the dataset having 1000K sequences was 189MB so that *MEMISP* discovered patterns in a single pass without partitioning. Other datasets

were mined by the partition-and-validation technique as described in Section 3.4.3. For example, the dataset of $|DB| = 10,000K$ of size 1.8GB was mined by 10 partitions. Given $minsup = 0.75\%$ with 10 million sequences, *GSP* could not complete the mining in a reasonable time. *PrefixSpan* created the projected databases of size to the amount of 11.4 times the original database size. Though Fig. 3-10 shows that both *PrefixSpan* and *MEMISP* are linearly scalable with the number of data sequences, but *MEMISP* has better scalability.

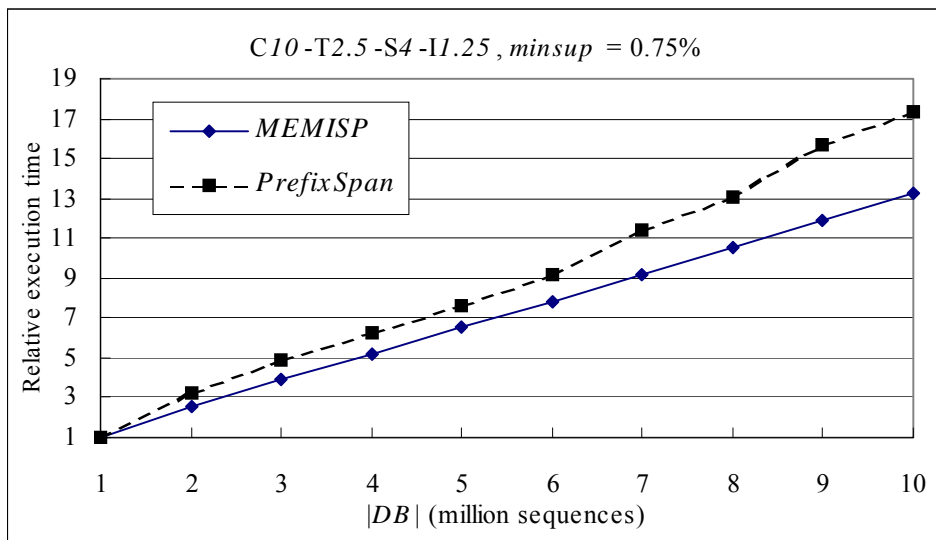


Fig. 3-10. Linear scalability of *MEMISP* vs. *PrefixSpan*

3.6 Discussion

We summarize the factors contributing to the efficiency of the proposed *MEMISP* algorithm by comparing with the well-known *GSP* and *PrefixSpan* algorithms.

- **One pass database scanning.** *MEMISP* reads the original database only once, except for extra-large databases described in Section 3.4.3. In the experiments, a database with one million data sequences can fit into a platform with 256MB memory so that the database was scanned only once by *MEMISP* in the mining. However, *GSP* must read the database at least k times, assuming that

the maximum size of the discovered patterns is k . *PrefixSpan* reads one pass over the original database, and then writes and reads once for each projected sub-database. In some cases such as low *minsup*, the total size of sub-databases might be several times larger than the size of the original database.

- **No candidate generation.** *MEMISP* discovers patterns directly from data sequences in-memory by index advancement. In contrast to *GSP*, *MEMISP* generates no candidates so that the time in candidate generation and testing are saved. Moreover, *MEMISP* works well even with small memory since the unknown sized (and often huge) space for candidate storage is unnecessary.
- **No database projection.** The pure and simple index advancing in *MEMISP* creates no new databases so that the intermediate storage, which *PrefixSpan* needs, is not needed here. Note that *MEMISP* and *PrefixSpan* will have similar performance in mining a memory-accommodable database if the *pseudo-projection* technique [67] is used in *PrefixSpan*. However, according to [67], *pseudo-projection* is not efficient if it is used for disk-based accessing, and should be employed after *bi-level* optimization [67] having reduced the database size to the main memory accommodable size.
- **Focused search and effective indexing.** *MEMISP* considers those data sequences indicated by current index set only instead of searching every data sequence in the database. Furthermore, each position index keeps moving forward along a data sequence as the discovered pattern gets longer. Consequently, fewer and fewer items in a data sequence need to be considered as a prefix pattern getting longer.
- **Compact index storage.** *MEMISP* requires very compact storage for the index sets. In an index set, the maximum number of indices required equals to

the number of data sequences, no matter how small the *minsup* value is. Assume that the database has *m* million sequences. In a 4-byte addressing mode, *MEMISP* demands maximum $(4+4)*m$ MB for an index set. The required total memory would be less than $k*(8*m)$ MB for discovering the frequent *k*-sequences with respect to any *minsup* value. Nevertheless, the memory requirement for storing candidates in *GSP* can hardly be estimated without giving the *minsup*. Similarly, the total size of the projected databases in *PrefixSpan* increases as the *minsup* decreases.

- **High CPU and memory utilization.** *PrefixSpan* needs only little memory space during the mining process. It solved the mining problem successfully by sub-database searching, though, with possible CPU idle while projecting sub-databases. *MEMISP*, by contrast, uses all the available memory and maximizes CPU utilization without extra disk operations.



3.7 Summary

Speeding up the discovery of sequential patterns has been the focus of data mining research. In this chapter, we present a memory indexing approach for fast discovery of sequential patterns, called *MEMISP*. *MEMISP* mines the set of all sequential patterns without generating candidates or sub-databases. The performance study exhibits that *MEMISP* is more efficient than both *GSP* and *PrefixSpan* algorithms, and has good linear scalability even for very low minimum supports. Moreover, *MEMISP* may estimate the total memory required, which is independent of the specified *minsup*. *MEMISP* scans the database at most twice with the partition-and-validation technique even for extra large databases so that the slow disk I/O is minimized. The compact indexing and the effective find-then-index technique together makes *MEMISP* a promising approach for fast discovery of sequential patterns in sequence databases of

any size, even with small memory and low *minsup*.

In addition to sequential pattern mining, the technique could be extended to the discovery of maximum patterns [2], constrained/generalized sequential patterns [80], multi-dimensional patterns [70], and incremental sequence discovery after database updating [102]. It is also interesting to integrate the proposed index sets with database systems for efficient queries.



Chapter 4 Algorithm DELISP for Sequential Pattern Mining with Time Constraints

4.1 Overview

The discovery of sequential patterns is a complicated issue in data mining [6, 11, 25, 47, 72, 80, 89, 98], as described in Chapter 3. A typical example is a retail database where each record corresponds to a customer's purchasing sequence, called *data sequence*. A data sequence is composed of all the customer's transactions ordered by transaction time. Each transaction is represented by a set of literals indicating the set of items (called *itemset*) purchased in the transaction. The objective is to find all the frequent sub-sequences (called *sequential patterns*) in the sequence database.

An example sequential pattern might be that 30% customers bought *PC* and *printer*, followed by the purchase of *scanner* and *graphics-software*, and then *digital camera*. Such a pattern, denoted by $\langle (PC, printer)(scanner, graphics-software)(digital camera) \rangle$, has three *elements* where each element is an itemset. Although the issue is motivated by the retail industry, the mining technique is applicable to domains bearing sequence characteristics, including the analysis of Web traversal patterns, medical treatments, natural disasters, DNA sequences, and so forth [6, 70, 92].

Sequential pattern mining [67, 70, 98] is more complex than association rule mining [14, 84] because the patterns are formed not only by combinations of items but also by permutations of itemsets. The number of potential sequences is by far larger than that of potential itemsets. Given 100 possible items in the database, the total number of possible itemsets is $\sum_{i=0}^{100} \binom{100}{i} = 2^{100}$. Let the *size* of a sequence be the

total number of items in that sequence. The number of potential sequences of size k is

$$\sum_{i_1=1}^k \binom{100}{i_1} \sum_{i_2=1}^{k-i_1} \binom{100}{i_2} \sum_{i_3=1}^{k-i_1-i_2} \binom{100}{i_3} \Lambda \sum_{i_k=1}^{k-i_1-\Lambda-i_{k-1}} \binom{100}{i_k} .$$

The total number of potential sequences, accumulating from size one to size 100 and more, could be enormous.

The issue of mining sequential patterns with time constraints was first addressed in [80]. Three time constraints including minimum gap, maximum gap and sliding time-window are specified to enhance conventional sequence discovery. For example, without time constraints, one may find a pattern $\langle (b, d, e)(a, f) \rangle$. However, the pattern could be insignificant if the time interval between (b, d, e) and (a, f) is too long. Such patterns could be filtered out if the maximum gap constraint is specified.

Analogously, one might discover the pattern $\langle (b, d, e)(a, g) \rangle$ from many data sequences consisting of itemset (a, g) occurring one day after the occurrence of itemset (b, d, e) . Nonetheless, such a pattern is a false pattern in discovering weekly patterns, i.e. the minimum gap of 7 days. In other words, the sale of (b, d, e) might not trigger the sale of (a, g) in next week. Therefore, time constraints including maximum gap and minimum gap should be incorporated in the mining to reinforce the accuracy and significance of mining results.

Moreover, conventional definition of an element of a sequential pattern is too rigid for some applications. Essentially, a data sequence is defined to support a pattern if each element of the pattern is contained in an individual transaction of the data sequence. However, the user may not care whether the items in an element (of the pattern) come from a single transaction or from adjoining transactions of a data sequence if the adjoining transactions occur close in time (within a specified time interval). The specified interval is named *sliding time-window* [80]. For instance, given a sliding time-window of 5, a data sequence $\langle t_1(a, d) t_2(b) t_3(c) \rangle$ can support the pattern $\langle (a, b, d)(c) \rangle$ if the difference between time t_1 and time t_2 is no greater

than 5. Adding sliding time-window constraint to relax the definition of an element will broaden the applications of sequential patterns.

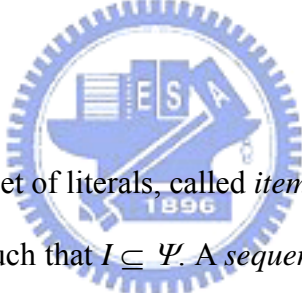
Although there are many algorithms dealing with sequential pattern mining [6, 51, 55, 98], few handle the mining with the addition of time constraints. The *GSP* (**Generalized Sequential Pattern**) algorithm proposed in [80] is the first algorithm that discovers sequential patterns with time constraints within Apriori framework. *GSP* solves the problem by generating and testing candidate patterns in multiple database scans. Candidate patterns having any non-frequent sub-sequence are pruned before testing to reduce the search space. Still, the number of candidates might be huge [67]. Furthermore, in order to check whether a data sequence contains a certain candidate, *GSP* transforms each data sequence into items' transaction-time lists. The transformation speeds up time-constraint related testing but introduces overheads during each database scanning.

Recent studies indicate that pattern-growth methodology could speed up sequence mining. Despite many studies on sequential pattern mining within pattern-growth methodology [29, 67, 68, 69, 70], no algorithm fully functionally equivalent to *GSP* on time constraint issues has been proposed so far. Especially, solving the sliding time-window constraint can be hardly found in the literature (except in the *GSP* context). In this chapter, we propose a new algorithm called *DELISP* (**Delimited Sequential Pattern**) for handling all three time constraints on sequential patterns, introduced in the context of *GSP*, within the pattern-growth framework. *DELISP* solves the problem by recursively growing valid patterns in projected sub-databases generated by sub-sequence projection. To accelerate mining by reducing the size of sub-sequences, the constraints are integrated in the projection to delimit the counting and growing of sequences. In *DELISP*, the *bounded projection* technique eliminates invalid sub-sequence projections caused by unqualified

maximum/minimum gaps, the *windowed projection* technique reduces redundant projections for adjacent elements satisfying the sliding window constraint, and the *delimited growth* technique grows only the patterns satisfying constraints. The conducted experiments show that *DELISP* outperforms the *GSP* algorithm. The scale-up experiments also indicate that *DELISP* has good linear scalability with the number of data sequences.

The rest of the chapter is organized as follows. We formulate the problem in Section 4.2 and review some related work in Section 4.3. Section 4.4 presents the *DELISP* algorithm. The experimental evaluation is described in Section 4.5. We discuss the performance improving factors in Section 4.6. Section 4.7 summarizes this chapter.

4.2 Problem Statement

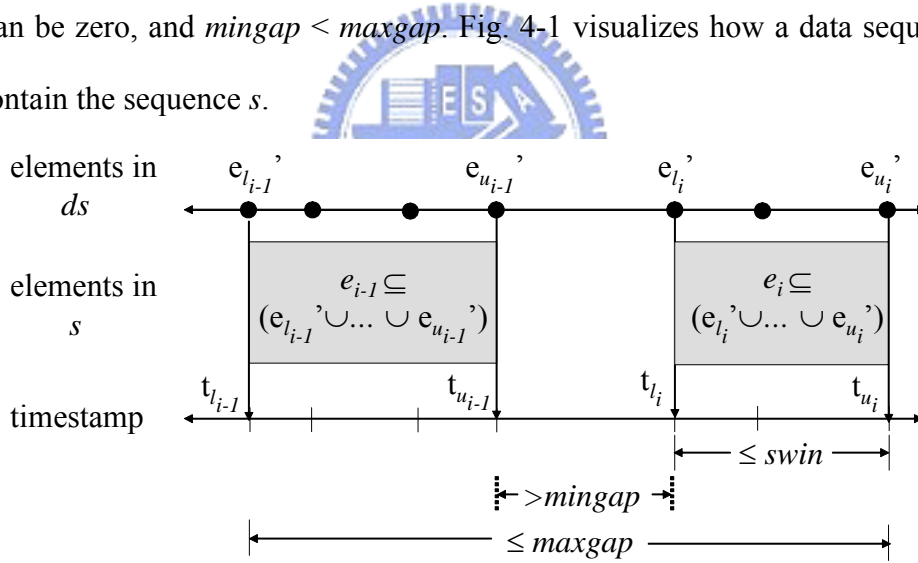


Let $\Psi = \{\alpha_1, \alpha_2, \dots, \alpha_n\}$ be a set of literals, called *items*. An *itemset* $I = (\beta_1, \beta_2, \dots, \beta_q)$ is a nonempty set of q items such that $I \subseteq \Psi$. A *sequence* s , denoted by $\langle e_1 e_2 \dots e_w \rangle$, is an ordered list of w *elements* where each *element* e_i is an itemset. Without loss of generality, we assume the items in an element are in lexicographic order. The *size* of a sequence s , written as $|s|$, is the total number of items in all the elements in s . Sequence s is a k -*sequence* if $|s| = k$. For example, $\langle (a)(c)(a) \rangle$, $\langle (a,c)(a) \rangle$, and $\langle (b)(a,e) \rangle$ are all 3-sequences.

The sequence database DB contains $|DB|$ data sequences. A *data sequence* ds having a unique identifier sid is represented by $sid \langle_{t_1} e_1' \ t_2 e_2' \ \dots \ t_n e_n' \rangle$, where element e_i' occurred at time t_i , $t_1 < t_2 < \dots < t_n$. Four parameters are specified to mine the database DB : (1) *minsup* (**minimum support**) (2) *mingap* (**minimum time gap**) (3) *maxgap* (**maximum time gap**) and (4) *swin* (**sliding time-window**). Given *minsup*, the three constraints *mingap*, *maxgap*, *swin*, and the database DB , the problem is to

discover the set of all time-constrained sequential patterns, i.e. sequential patterns satisfying the three time constraints.

A sequence s is a **time-constrained sequential pattern** if $s.sup \geq minsup$, where $s.sup$ is the *support* of the sequence s and $minsup$ is the user specified minimum support threshold. The *support* of s is the number of data sequences *containing* s divided by $|DB|$. A data sequence $ds = sid/\langle_{t_1}e_1' \ t_2e_2' \dots \ t_n e_n' \rangle$ *contains* a sequence $s = \langle e_1e_2 \dots e_w \rangle$ if there exist integers $l_1, u_1, l_2, u_2, \dots, l_w, u_w$ and $1 \leq l_1 \leq u_1 < l_2 \leq u_2 < \dots < l_w \leq u_w \leq n$ such that the four conditions hold: (1) $e_i \subseteq (e_{l_i}' \cup \dots \cup e_{u_i}')$, $1 \leq i \leq w$ (2) $t_{u_i} - t_{l_i} \leq swin$, $1 \leq i \leq w$ (3) $t_{u_i} - t_{l_{i-1}} \leq maxgap$, $2 \leq i \leq w$ (4) $t_{l_i} - t_{u_{i-1}} > mingap$, $2 \leq i \leq w$. Assume that t_j , $mingap$, $maxgap$, and $swin$ are all positive integers, $mingap$ and $swin$ can be zero, and $mingap < maxgap$. Fig. 4-1 visualizes how a data sequence ds may contain the sequence s .



sequence $s = \langle e_1e_2 \dots e_w \rangle$ is *contained* in data sequence $ds = sid/\langle_{t_1}e_1' \ t_2e_2' \dots \ t_n e_n' \rangle$ if all the items in e_i can be found in the element formed by combining elements between e_{l_i}' and e_{u_i}' , where $1 \leq i \leq w$, and the constraints $swin$, $mingap$, $maxgap$ are satisfied.

Fig. 4-1. Example of the sequence containment relationship

An example database DB is shown in the first column in Table 4-1. The data sequence $C1/\langle_1(c)_{35}(b,f) \rangle$ has two elements (itemsets), one having a single item c occurring at time 1 and the other having items b and f occurring at time 35. Given

Table 4-1. Example sequence database *DB* and the time-constrained sequential patterns

Sequence	Time-constrained sequential patterns (<i>minsup</i> = 40%, <i>mingap</i> = 2, <i>maxgap</i> = 30, <i>swin</i> = 2)	Sequential patterns (<i>minsup</i> =40%)
C1/< ₁ (c) ₃₅ (b,f)>		
C2/< ₂ (b) ₄ (d)>		
C3/< ₁ (a,d) ₅ (c) ₆ (c) ₈ (b) ₃₅ (a,f)>	<(a)>, <(a)(b)>, <(a,d)>, <(a)(f)>, <(b)>, <(b,d)>	<(a)>, <(a)(a)>, <(a)(b)>, <(a)(d)>, <(a)(f)>, <(b)>
C4/< ₂ (a) ₄ (d) ₃₀ (f) ₃₃ (a) ₆₁ (f)>	<(b,f)>, <(b)(f)>, <(c)>, <(d)>, <(f)>	<(b)(d)>, <(b)(f)>, <(c)>, <(c)(b)>, <(c)(f)>, <(d)>
C5/< ₁ (a,b,e) ₄ (e) ₇ (f) ₈ (d) ₉ (b)>		<(d)(a)>, <(d)(b)>, <(d)(f)>, <(f)>

mingap = 2, *maxgap* = 30, *swin* = 2, C1 contains <(c)> and <(b,f)>, but it does not contain either <(c)(b)> or <(c)(f)> since 35-1 > *maxgap*. Similarly, C2/<₂(b)₄(d)> does not contain <(b)(d)> since 4-2 is not greater than *mingap*. Sequence <(a)(b)> is contained in C3/<₁(a,d)₅(c)₆(c)₈(b)₃₅(a,f)> and C5/<₁(a,b,e)₄(e)₇(f)₈(d)₉(b)> so that <(a)(b)>.sup = 2/5. With the specified *swin*, C4/<₂(a)₄(d)₃₀(f)₃₃(a)₆₁(f)> may contain <(a,d)> (4-2 ≤ 2) and C5 may contain <(b,d,f)> (9-7 ≤ 2). Given *minsup* = 40%, both <(a)(b)> and <(a,d)> are time-constrained sequential patterns while <(b,d,f)> is not. The set of all time-constrained sequential patterns is listed in the second column in Table 4-1. Note that the mining of sequential patterns without time constraints, shown in the third column in Table 4-1, is a special case with *mingap* = 0, *maxgap* = ∞, and *swin* = 0 here.

4.3 Related Work

Much research has been focused in sequence mining without time constraints of *mingap*, *maxgap* and *swin* [6, 29, 67, 75, 98]. The *GSP* algorithm is the first algorithm that handles the time constrains in sequential patterns [80]. Based on the Apriori framework [6], the patterns are found in multiple database passes. In every database scan, each data sequence is transformed into items' time-lists for fast finding of

certain element with a time tag. Since the start-time and end-time of an element (may comprise several transactions) must be considered, *GSP* defines ‘contiguous sub-sequence’ for candidate generation, and move between ‘forward phase’ and ‘backward phase’ for checking whether a data sequence contains a certain candidate [80].

A general pattern-growth framework was presented in [69] for constraint-based sequential pattern mining. From the application point of view, seven categories of constraints including *item*, *length*, *super-pattern*, *aggregate*, *regular expression*, *duration*, and *gap* constraints were covered. Among these constraints, *duration* and *gap* constraints are tightly coupled with the support counting process because they confine how a data sequence contains a pattern. Orthogonally classifying constraints by their roles in mining, *monotonic*, *anti-monotonic*, and *succinct* constraints were characterized and the *prefix-monotone* constraint was introduced. The *prefix-growth* framework which pushes prefix-monotone constraints into *PrefixSpan* was also proposed in [69]. However, with respect to time constraints, *prefix-growth* only mentioned *maxgap* and *mingap* time constraints (though *duration* constraint was addressed) with no implementation details, and sliding time-window was not considered at all.

The *cSPADE* algorithm [97] extends the vertical mining algorithm *SPADE* [98] to deal with time constraints. Vertical mining approaches [11, 97, 98] discovers sequential patterns using join-operations and vertical database layout, where data sequences are transformed into items’ (sequence-id, time-id) lists. The *cSPADE* algorithm checks *mingap* and *maxgap* while doing temporal joins. Nevertheless, the huge sets of frequent 2-sequences must be preserved to generate the required classes for the *maxgap* constraint [97]. While it is possible for *cSPADE* to handle constraints like maximum/minimum gaps by expanding the id-lists and augmenting the

join-operations with temporal information [97], it does not appear feasible to incorporate the sliding time-window. The sliding time-window constraint was not mentioned in *cSPADE*.

A different kind of time constraints, discovering patterns that involve multiple time granularities, was addressed in [13]. Simple or complex event structures, which are episodes [47, 42] with time interval restrictions similar to *mingap/maxgap* constraints, are discovered by the introduced timed automaton with granularities [13]. Nevertheless, we are interested in the discovery of time-constrained sequential patterns forming by itemsets.

4.4 DELISP: Delimited Sequential Pattern Mining

In this section, we describe the proposed pattern-growth mechanism for mining time-constrained sequential patterns, called *DELISP*. The main idea is efficiently ‘finding’ the frequent items, and then effectively ‘growing’ potential patterns in the sub-databases constructed by projecting sub-sequences corresponding to the frequent items. We also project the time-tags into the sub-databases to generate patterns satisfying the time constraints. However, *DELISP* projects fewer but complete combinations by windowed and bounded projections, and grows potential patterns effectively by delimited growth. Section 4.4.1 introduces the terminology used in *DELISP*. In Section 4.4.2, we demonstrate the method by mining an example database. Section 4.4.3 describes the proposed algorithm. For convenience, we refer to a data sequence $ds = sid/\langle_{t_1}e_1' \ t_2e_2' \ \dots \ t_n e_n' \rangle$ as ds in the following context.

4.4.1 Terminology used in *DELISP*

Definition 4-1 (Frequent item) An item x is called a *frequent item* in a sequence database DB if the support of 1-sequence $\langle(x)\rangle$ is greater than or equal to *minsup*.

Definition 4-2 (Stem, type-1 growth, type-2 growth, prefix) Given a sequential pattern ρ and a frequent item x in the sequence database DB , x is called the *stem-item* (abbreviated as *stem*) of the sequential pattern ρ' if ρ' can be formed by (1) appending (x) as a new element to ρ or (2) extending the last element of ρ with x . The formation of ρ' is a *type-1 growth* if it is formed by appending (x), and a *type-2 growth* if it is formed by extending with x . The *prefix pattern* (abbreviated as *prefix*) of ρ' is ρ .

For example, given $\langle(a)\rangle$ and the frequent item b , we may have the *type-1 growth* $\langle(a)(b)\rangle$ by appending (b) to $\langle(a)\rangle$ and the *type-2 growth* $\langle(a,b)\rangle$ by extending $\langle(a)\rangle$ with b . The $\langle(a)\rangle$ is the *prefix* and the b is the *stem* of both $\langle(a)(b)\rangle$ and $\langle(a,b)\rangle$. As to a *type-2 growth* $\langle(c)(a,d)\rangle$, its *prefix* is $\langle(c)(a)\rangle$ and its *stem* is d . Note that the null sequence, denoted by $\langle\rangle$, is the *prefix* of any frequent 1-sequence.

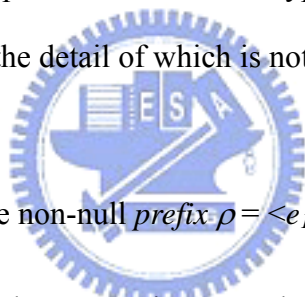
Definition 4-3 (start-time, end-time, tag-list) The timestamp indicating the occurrence of itemset I in ds is marked in the projected database. If itemset I is contained in a single element e_{δ} in ds , the *start-time* (abbreviated as **st**) and *end-time* (abbreviated as **et**) pair **st:et** is marked as $t_{\delta}:t_{\delta}$. If I is contained in $e_{\delta} \cup e_{\delta+1} \cup \dots \cup e_{\varepsilon}$ (in ds), **st:et** is marked as $t_{\delta}:t_{\varepsilon}$. We refer to the list of all the **st:et** pairs as the **tag-list** of I in ds . The tag-list is denoted by $[st_1:et_1, st_2:et_2, \dots, st_k:et_k]$ where $st_i \leq et_i$ for $1 \leq i \leq k$, $st_i < st_{i+1}$ and $et_i < et_{i+1}$ for $1 \leq i \leq k-1$.

Definition 4-4 (Accessible) Let the tag-list of itemset I in ds be $[st_1:et_1, st_2:et_2, \dots, st_k:et_k]$. An element e_a is *accessible* from I in ds if its timestamp t_a satisfies: (1) $et_i - swin \leq t_a \leq st_i + swin$, where $i \in \{1, 2, \dots, k\}$ or (2) $et_i + mingap < t_a \leq st_i + maxgap$, where $i \in \{1, 2, \dots, k\}$ or (3) $t_b + mingap < t_a \leq t_b + maxgap$ where t_b is the timestamp of an *accessible* element e_b from I in ds .

Fig. 4-2 demonstrates the three accessible circumstances. For example, the tag-list of itemset (c) in $C1/\langle_1(c)_{35}(b,f)\rangle$ is $[1:1]$, that of (b) in $C1$ is $[35:35]$. The tag-list of (a) in $C3/\langle_1(a,d)_5(c)_6(c)_8(b)_{35}(a,f)\rangle$ is $[1:1, 35:35]$, that in

$C4/\langle_2(a)_4(d)_{30}(f)_{33}(a)_{61}(f)\rangle$ is [2:2, 33:33], and that in $C5/\langle_1(a,b,e)_4(e)_7(f)_8(d)_9(b)\rangle$ is [1:1]. With respect to (a,d), the tag-list in $C3$ is [1:1] and that in $C4$ is [2:4]. The $_{35}(b,f)$ in $C1$ is not accessible from $_1(c)$ if $maxgap = 30$. Considering $_1(a,d)$ in $C3$, elements $_5(c)$, $_6(c)$, $_8(b)$ are accessible with $mingap = 2$ and $maxgap = 30$. Additionally, $_{35}(a,f)$ is also accessible because it is accessible via $_8(b)$ for $8+2 < 35 \leq 8+30$, or via $_5(c)$ then $_8(b)$.

Note that when an accessible element is extended by condition (1) in Definition 4-4, the extension is checked on not violating $mingap$ or $maxgap$ constraints with respect to the previous itemset of I (in the pattern), denoted by I_p . The checking is to ensure that itemset I , having timestamps satisfying the $mingap/maxgap$ constraint with I_p , does not violate the gap constraint after the type-2 extension. Such a checking requires projecting $st.et$ of I_p , the detail of which is not shown in the following context for clearer illustration.



Lemma 4-1. Let ds contain the non-null prefix $\rho = \langle e_1 e_2 \dots e_p \rangle$. Given the tag-list of e_p in ds , a frequent item x in an element e_a' in ds can be a stem only if e_a' is *accessible* from e_p in ds .

Lemma 4-1 is based on the fact that a valid ‘growth’ must satisfy time constraints. Hence, we may prevent the inaccessible elements from projection to speed up the growing process, as shown in Fig. 4-3. We further reduce projections by eliminating items in an accessible element from projection using Lemma 4-2, as depicted in Fig. 4-4.

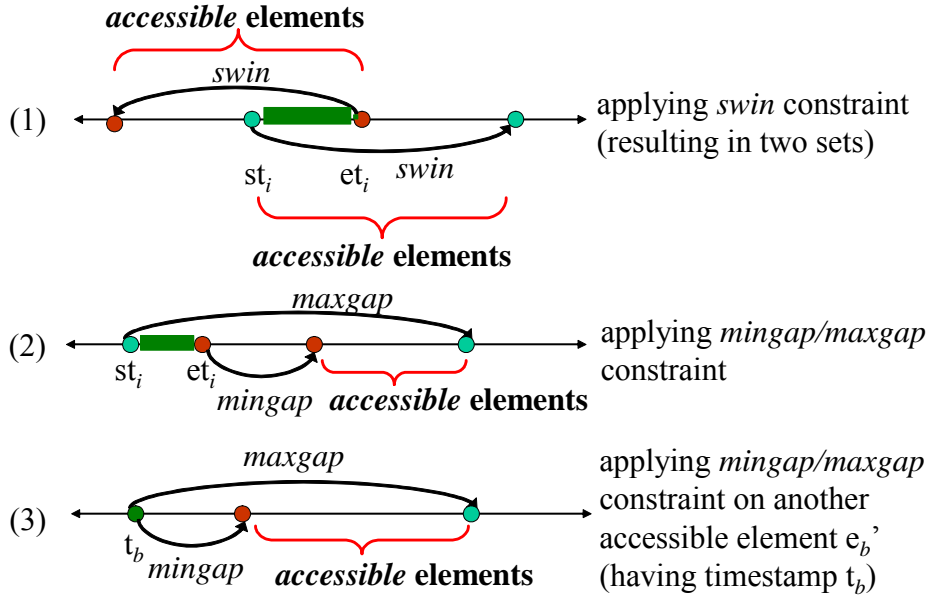


Fig. 4-2. Accessible elements from itemset I in ds with tag-list $[st_1:et_1, st_2:et_2, \dots, st_k:et_k]$

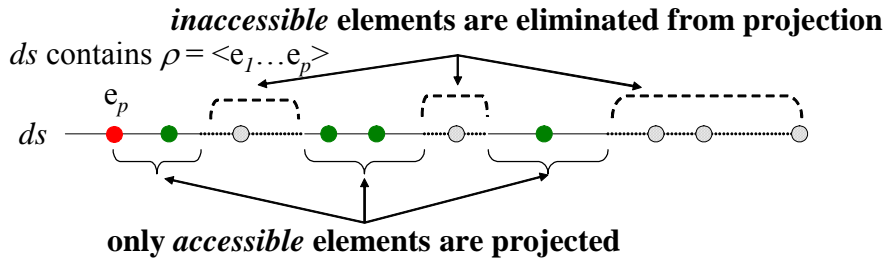


Fig. 4-3. The projected elements of ds with respect to ρ

$$\rho = \langle e_1 \dots e_p \rangle, e_p = (\dots, x) \quad e_p: \text{red bar}$$

any x' in an accessible element e_a' is eliminated from projection if $x' \leq x$

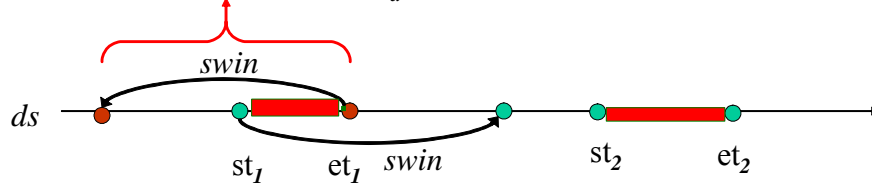


Fig. 4-4. Eliminating items having smaller lexicographic order from projection

(Lemma 4-2)

Lemma 4-2. Let the last element in $prefix \rho$ be e_p , the last item in e_p be x , and the tag-list of e_p in ds be $[st_1:et_1, st_2:et_2, \dots, st_k:et_k]$. Any item x' in an accessible element e_a' cannot be a stem if (1) $x' \leq x$ and (2) $t_a e_a'$ is accessible from ρ by satisfying $et_1 - swin \leq t_a \leq et_1$.

Lemma 4-2 is based on the fact that items are in lexicographic order within elements. Any item to be used as a stem for the type-2 growth having *prefix* ρ should have an order greater than the order of the last item in ρ . Thus, any small-ordered x' (located in $t_a e_a'$, $et_I - swin \leq t_a \leq et_I$) need not be projected.

Note that all the items in an accessible element $t_a e_a'$ having $et_I < t_a \leq st_I + swin$ are projected even their lexicographical orders precede that of the last item in *prefix* ρ . These items can be stems for potential type-1 growth (*prefix* ρ) and cannot be eliminated from projection.

4.4.2 Mining time-constrained sequential patterns by DELISP: an example

All the time-constrained sequential patterns are found by growing frequent sequences from size one to the maximum size. Frequent items in *DB* can be determined after scanning *DB* once. We then use each frequent item as a *stem* with *prefix* $\langle \rangle$ to form the set of all frequent 1-sequences. The sub-sequences satisfying the constraints are then projected into related sub-databases for further 'growing'. The stems of type-1 and type-2 growth can be determined by scanning the sub-databases once. Recursively, the time-constraint integrated projection and growing techniques are applied to discover the frequent 2-sequences, 3-sequences, etc.

Example 1: Given $minsup=40\%$, $mingap=2$, $maxgap=30$, $swin=2$, and the *DB* as shown in Table 4-1, *DELISP* mines the patterns by the following steps.

Step 1. Find frequent items. By scanning *DB* once, we have frequent items a (count = 3 for appearing in 3 data sequences $C3$, $C4$ and $C5$), b (count = 4), c (count = 2), d (count = 4), and f (count = 4). Non-frequent item e is omitted from mining afterward. The five items are stems of type-1 growth having *prefix* $\langle \rangle$.

Step 2. Project corresponding sub-sequences to sub-databases. Considering the time-constrained sequential patterns having *prefix* $\rho = \langle (x) \rangle$, each can be found in the

sub-database (named ρ -DB) generated by projecting all the data sequences having item x in DB . While projecting a data sequence ds into ρ -DB, we omit the non-frequent items, those *inaccessible* elements (using Lemma 4-1), and those ‘lexicographically smaller’ items (using Lemma 4-2).

We tabulate the sub-databases $\langle(a)\rangle$ -DB, $\langle(b)\rangle$ -DB, $\langle(c)\rangle$ -DB, $\langle(d)\rangle$ -DB, and $\langle(f)\rangle$ -DB in part 1 of Table 4-2. Take $\langle(a)\rangle$ -DB for instance. The tag-lists of (a) in $C3$, $C4$, and $C5$ are exemplified in Section 4.4.1. The $_1(d)$ in $C3$ is accessible and is projected with respect to $st:et = 1:1$. Elements $_5(c)$, $_6(c)$, and $_8(b)$ in $C3$ are projected since they are all accessible ($1+2 < 5 \leq 1+30$, $3 < 6 \leq 31$, and $3 < 8 \leq 31$). The $_{35}(a,f)$ in $C3$ is also projected with respect to $st:et = 35:35$. Similarly, we project the accessible elements $_4(d)$, $_{30}(f)$, $_{33}(a)$, and $_{61}(f)$ in $C4$. For $C5$, element $_7(f)$, $_8(d)$, and $_9(b)$ are projected, and $_1(b)$, instead of $_1(a,b,e)$, is projected after dropping non-frequent item e and item a (by Lemma 4-2).

Note that the tag-list of (b) in $C3$ is [8:8], so $_6(c)$ in $C3$ is projected into $\langle(b)\rangle$ -DB since $8-2 \leq 6 \leq 8+2$. The $_{35}(a,f)$ in $C1$ does not appear in $\langle(c)\rangle$ -DB because it is inaccessible from [1:1] ($35 > 1+30$), hence the tag-list and the entire sub-sequence of $C1$ are eliminated. Similarly, $C2$ is removed from $\langle(d)\rangle$ -DB. In addition, the $_2(a)$ in $C4$ is not projected into $\langle(d)\rangle$ -DB using Lemma 4-2 ($a < d$). However, the $_7(f)$ in $C5$ must be included in $\langle(d)\rangle$ -DB because it is accessible from [8:8].

Step 3. Mine each sub-database for the subsets of time-constrained sequential patterns. In each sub-database, we grow the patterns in each sequence according to the time constraints, and determine which pattern is a valid time-constrained sequential pattern. Assume that we are growing patterns from *prefix* ρ whose last element is e_p and the tag-list of e_p in ds is $[st_1:et_1, st_2:et_2, \dots, st_k:et_k]$. The stems of potential type-1 growth come from the accessible e_a whose timestamp t_a satisfying et_i

Table 4-2. The projected sub-sequences in the ρ -DB sub-databases

ρ -DB	Projected sub-sequences
Part 1: sub-databases of DB	
<(a)>-DB	C3/[1:1,35:35]/< ₁ (d) ₅ (c) ₆ (c) ₈ (b) ₃₅ (a,f)>; C4/[2:2,33:33]/< ₄ (d) ₃₀ (f) ₃₃ (a) ₆₁ (f)>; C5/[1:1]/< ₁ (b) ₇ (f) ₈ (d) ₉ (b)>
<(b)>-DB	C1/[35:35]/< ₃₅ (f)>; C2/[2:2]/< ₄ (d)>; C3/[8:8]/< ₆ (c) ₃₅ (a,f)>; C5/[1:1,9:9]/< ₇ (f) ₈ (d) ₉ (b)>
<(c)>-DB	C3/[5:5,6:6]/< ₆ (c) ₈ (b) ₃₅ (a,f)>
<(d)>-DB	C3/[1:1]/< ₅ (c) ₆ (c) ₈ (b) ₃₅ (a,f)>; C4/[4:4]/< ₃₀ (f) ₃₃ (a) ₆₁ (f)>; C5/[8:8]/< ₇ (f) ₉ (b)>
<(f)>-DB	C4/[30:30,61:61]/< ₃₃ (a) ₆₁ (f)>; C5/[7:7]/< ₈ (d) ₉ (b)>
Part 2: sub-databases of <(a)>-DB	
<(a)(b)>-DB	C3/[8:8]/< ₃₅ (f)>
<(a)(f)>-DB	C5/[7:7]/< ₈ (d) ₉ (b)>
<(a,d)>-DB	C3/[1:1]/< ₈ (b) ₃₅ (f)>; C4/[2:4]/< ₃₀ (f) ₆₁ (f)>
Part 3: sub-databases of <(b)>-DB	
<(b)(f)>-DB	C5/[7:7]/< ₈ (d) ₉ (b)>
<(b,d)>-DB	None
<(b,f)>-DB	None
Note: the notation 'st:et' prior to a data sequence denotes the start-time and the end-time of the data sequence with respect to ρ projection.	

+ $mingap < t_a \leq st_i + maxgap$, where $i \in \{1, 2, \dots, k\}$. The stems of potential type-2 growth come from the accessible e_a ' satisfying $et_i - swin \leq t_a \leq st_i + swin$, where $i \in \{1, 2, \dots, k\}$. We may obtain the occurrence counts (i.e. supports) of stems after scanning ρ -DB once. Recursively, we then generate the corresponding ρ' -DB (having prefix ρ) for each stem having sufficient support count.

We mine <(a)>-DB as follows. Potential stems of type-1 growth in C3 (tag-list [1:1,35:35]) are c and b since $_5(c)$ and $_8(b)$ are accessible within $(1 + 2, 1 + 30]$. In C4 (tag-list [2:2, 33:33]), f and a are potential stems of type-1 growth since the accessible ranges are $(2 + 2, 2 + 30]$ and $(33 + 2, 33 + 30]$. Similarly, f , d , and b are potential stems of type-1 growth in C5. Thus, b (count = 2) and f (count = 2) are the valid stems of type-1 growth in <(a)>-DB.

Potential stems of type-2 growth in C3 (tag-list [1:1,35:35]) are d (within [1-2, 1+2]) and f (within [35-2, 35+2]), and that in C4 is d (within [2-2, 2+2]),

and that in $C5$ is b (within $[1-2, 1+2]$). Therefore, d is the valid stem of type-2 growth in $\langle(a)\rangle\text{-DB}$. Consequently, the time-constrained sequential patterns are $\langle(a)(b)\rangle$ (count = 2), $\langle(a)(f)\rangle$ (count = 2), and $\langle(a,d)\rangle$ (count = 2) by mining $\langle(a)\rangle\text{-DB}$.

Step 4. Find all patterns by applying step 2 and step 3 on the sub-databases recursively. Considering the time-constrained sequential patterns having *prefix* $\rho = \langle(a)(b)\rangle$, each can be found in the sub-database (named $\langle(a)(b)\rangle\text{-DB}$) generated by projecting all the data sequences having (b) in $\langle(a)\rangle\text{-DB}$. Again, we eliminate the non-frequent items, those *inaccessible* elements (using Lemma 4-1), and those ‘lexicographically smaller’ items (using Lemma 4-2).

Next, we apply step 2 to project the sub-sequences in $\langle(a)\rangle\text{-DB}$ further into sub-databases $\langle(a)(b)\rangle\text{-DB}$, $\langle(a)(f)\rangle\text{-DB}$, and $\langle(a,d)\rangle\text{-DB}$. The projected sub-databases of $\langle(a)\rangle\text{-DB}$ are shown in part 2 of Table 4-2. Similarly, inaccessible elements and non-frequent items (with respect to $\langle(a)\rangle\text{-DB}$) are not projected. The $\langle(a)(b)\rangle\text{-DB}$ is constructed by projecting the tag-list of (b) and the accessible elements in each sub-sequence as follows. In $\langle(a)\rangle\text{-DB}$ of Table 4-2, the tag-list of (b) in $C3$ is $[8:8]$, that in $C5$ is $[9:9]$. Only $C3/8:8/\langle_{35}(f)\rangle$ is projected since there is no accessible element in $C5$. Neither type-1 nor type-2 growth in the $\langle(a)(b)\rangle\text{-DB}$ finds any pattern so the growth is stopped. The $\langle(a)(f)\rangle\text{-DB}$ contains only one sequence after projection so that the growth in $\langle(a)(f)\rangle\text{-DB}$ is also stopped. Again, constructing $\langle(a,d)\rangle\text{-DB}$ is accomplished by projecting tag-lists of (a,d) and the accessible elements. We project $C3$ as $1:1/\langle_8(b)_{35}(f)\rangle$ instead of $1:1/\langle_5(c)_6(c)_8(b)_{35}(a,f)\rangle$ by removing non-frequent items a and c . Growing pattern in $\langle(a,d)\rangle\text{-DB}$ is stopped without forming any pattern. The mining with sub-databases of $\langle(a)\rangle\text{-DB}$ thus terminates.

We then recursively apply the steps on $\langle(b)\rangle\text{-DB}$ for patterns having *prefix* $\langle(b)\rangle$,

Algorithm *DELISP*

Input: DB = a sequence database; $minsup$ = minimum support; $mingap$ = minimum time gap; $maxgap$ = maximum time gap; $swin$ = sliding time-window.

Output: the set of all time-constrained sequential patterns.

Method:

1. Scan DB once, find the set of all frequent items.
2. For each frequent item x ,
 - (a) form a time-constrained sequential pattern $\rho = \langle(x)\rangle$ and output ρ .
 - (b) call $ProjectDB(\rho, DB)$ to construct sub-database ρ - DB .
 - (c) call $Mine(\rho$ - $DB)$.

Subroutine $ProjectDB(\rho, Db)$

Parameters: ρ = pattern; Db = the sub-database.

Output: the sub-database ρ - DB .

Method:

1. For each data sequence $ds = sid/\langle_{t_1}e_1' \ t_2e_2' \ \dots \ t_n e_n'\rangle$ in Db ,
 - (a) record the tag-list $[st_1:et_1, st_2:et_2, \dots, st_k:et_k]$ of ρ in ds , where each $st_i:et_i$ marks the *start-time:end-time* of the last element of ρ in ds .
 - (b) (**Bounded-projection**) mark the list of accessible elements in ds . /* See Definition 4-4 (accessible) in Section 4.4.1 */
 - (c) (**Windowed-projection**) drop item x' in an accessible element e_a' where $et_1 - swin \leq t_a \leq et_1$ and $x' \leq x$. The item x is the last item in $e_p \in \rho = \langle e_1 e_2 \dots e_p \rangle$. /* Use Lemma 4-2 in Section 4.4.1 */
 - (d) if the list of accessible elements is not empty, drop the non-frequent items in ds and project $sid/[st_1:et_1, st_2:et_2, \dots, st_k:et_k]/\langle$ the list of accessible elements \rangle to ρ - DB .

Subroutine $Mine(\rho$ - $DB)$

Parameter: ρ - DB = the sub-database.

Output: time-constrained sequential patterns having prefix ρ .

Method:

1. For each data sequence $ds = sid/[st_1:et_1, st_2:et_2, \dots, st_k:et_k]/\langle_{t_1}e_1' \ t_2e_2' \ \dots \ t_n e_n'\rangle$ in ρ - DB ,
 - (a) for each element e_i' with timestamp t_i in ds , insert the items in e_i'
 - (i) to the stem set of potential type-1 growth if $et_j + mingap < t_i \leq st_j + maxgap$ where $j \in \{1, 2, \dots, k\}$. (**Delimited-growth/type-1**)
 - (ii) to the stem set of potential type-2 growth if $et_j - swin \leq t_i \leq st_j + swin$ where $j \in \{1, 2, \dots, k\}$. (**Delimited-growth/type-2**)
 - (b) for each stem in the two sets, increase its support count by one.
2. Find the frequent items in the two sets by comparing the supports with $minsup$.
3. For each frequent item x in the two sets,
 - (a) form a time-constrained sequential pattern ρ' (prefix ρ and stem x) and output ρ' .
 - (b) call $ProjectDB(\rho', \rho$ - $DB)$ to construct sub-database ρ' - DB .
 - (c) call $Mine(\rho'$ - $DB)$.

Fig. 4-5. Algorithm *DELISP*

on $\langle(c)\rangle$ - DB for patterns having *prefix* $\langle(c)\rangle$, ..., and on $\langle(f)\rangle$ - DB for patterns having *prefix* $\langle(f)\rangle$. By collecting the patterns found in the above process, *DELISP* efficiently

discovers all the sequential patterns satisfying the time constraints.

4.4.3 The DELISP algorithm

Fig. 4-5 presents the proposed *DELISP* algorithm. Analogous to *PrefixSpan* algorithm, *DELISP* decomposes the mining problem by recursively growing patterns, one item longer than the current patterns, in the projected sub-databases. However, the potential items used to grow are subjected to *mingap* and *maxgap* constraints, called *de-limited growth*. Therefore, we perform type-1 growth with items in each element $t_a e_a'$ within range $(et_i + mingap < t_a \leq st_i + maxgap)$, where $i \in \{1, 2, \dots, k\}$, and type-2 growth with items in each element $t_a e_a'$ within range $(et_i - swin \leq t_a \leq st_i + swin)$, where $i \in \{1, 2, \dots, k\}$. The $[st_1:et_1, st_2:et_2, \dots, st_k:et_k]$ is the tag-list of element $e_p \in prefix \langle e_1 e_2 \dots e_p \rangle$ in ds . On projecting sub-databases, we avoid the bi-directional growth by imposing the item-order in the type-2 growth. We always add a new item (in e_p) whose order is lexicographically larger than the order of the existing items for type-2 growth. Considering an example element (b, d, e) formed by combining $t_1(d, e)$ and $t_2(b)$, i.e. $|t_1 - t_2| \leq swin$. When the time t_2 is earlier than time t_1 , (b, d, e) will be discovered in the projected $\langle(b)\rangle$ -DB since $t_1 \leq t_2 + swin$. In case $t_1 < t_2$, (d, e) will be kept in $\langle(b)\rangle$ -DB since it is accessible for $t_2 - swin \leq t_1$. We refer to such projection as *windowed-projection*.

Theorem 4-1. Algorithm *DELSIP* discovers the set of all time-constrained sequential patterns.

Proof. Obviously, *DELISP* discovers the set of all frequent 1-sequences in step 1. Clearly, a frequent k -sequence is formed by either a *type-1 growth* or a *type-2 growth* from a frequent $(k-1)$ -sequence. Thus, the set of all time-constrained sequential patterns can be obtained by *type-1* and *type-2 growth*, from size one to the maximum size. Any item to be used as a stem must come from an accessible element; otherwise,

the corresponding growth would violate either *swin* or *mingap/maxgap* constraint. In Subroutine *ProjectDB*, by Lemma 4-1 and Lemma 4-2, those inaccessible items need not be projected so they are eliminated. Subroutine *Mine* counts the supports of time-constraint satisfied items for type-1 and type-2 growth, respectively. By recursively applying *ProjectDB* and *Mine*, *DELISP* discovers the set of all time-constrained sequential patterns.

4.5 Experimental Results

Extensive experiments were conducted to assess the performance of the *DELISP* algorithm. We compared the total execution times of *DELISP* and *GSP* [80] by varying the parameters of *mingap*, *maxgap*, and *swin*. The scalability of the algorithm was also evaluated over different database sizes. The experiments were performed on an 866 MHz Pentium-III PC with 1024MB memory running the Windows NT.

The *PrefixSpan* [67] does not handle the time constraints and therefore is not considered. However, note that for gap constraints (*mingap* and *maxgap*) *PrefixSpan* could be applied with an extra pattern counting step. In the step, patterns discovered without time constraints can be verified in an extra scan of the whole database. Nevertheless, such an extension cannot be applied for sliding time-window. The *prefix-growth* [69] gives no implementation details of gap constraints and no descriptions on sliding time-window, so that *prefix-growth* is not compared in our experiments.

The *cSPADE* algorithm [97], though accepts the minimum and maximum gap constraints, was not implemented in the comparison because it uses vertical database layout. Additional storage space and computation time are required to transform the natively horizontal databases into vertical. In addition, the *swin* constraint is not

handled in *cSPADE*. Revision of *cSPADE* to handle *swin* constraint is not trivial. One possible implementation is to incorporate sliding time-window by incrementing the support for each distinct window in the vertical representation. Nevertheless, the join operation has to be extended, beyond temporal and equality join, to allow ‘window join’. For example, joining the id-list of item x with that of item y , even their timestamps are not equal, now might generate itemset (x, y) if the time difference is no greater than *swin*. Such an extension could generate many combinations that turn out to be rejected after invoking another round of validating *mingap* and/or *maxgap*. The structure of the id-list also needs to be expanded to indicate the timestamps of previous elements to enable the counting of validating *mingap* gap.

Like most studies on sequential pattern mining [6, 29, 67, 98, 102], synthetic datasets were used and were generated using the procedure described in [80] for these experiments. The transaction IDs were used to represent the transaction times. As to the details of generating synthetic data, please refer to Section 3.5.1. The datasets mimic the real world transactions by using various parameters. Table 4-3 shows the meaning and the values of the parameters used in the experiments. A dataset generated with $|C| = 10$, $|T| = 2.5$, $|S| = 4$, $|I| = 1.25$ is denoted by *C10-T2.5-S4-I1.25*. It indicates that, in average, each customer has 10 transactions, each transaction has 2.5 items, sequences are generated from a sample having 4 transactions per pattern, and 1.25 items per transaction. The sample was generated with 1000 possible items, 5000 possible sequential patterns, and 25000 possible frequent itemsets. In Section 4.5.1, experimental results on varying the *minsup*, *mingap*, *maxgap*, and *swin*, and the results on various datasets are described. Section 4.5.2 shows the results of the experiments on scaling up the database size.

Table 4-3. Parameters used in the experiments

Parameter	Description	Value
$ DB $	Number of data sequences in database DB	100K, 200K, 400K, 800K, 1000K
$ C $	Average size (number of transactions) per customer	10, 15
$ T $	Average size (number of items) per transaction	2.5, 5
$ S $	Average size of potentially sequential patterns	4, 8
$ I $	Average size of potentially frequent itemsets	1.25, 2.5

4.5.1 Execution times of *GSP* and *DELISP* algorithms

First, we report the results on dataset *C10-T2.5-S4-I1.25* having 100,000 sequences. The execution times of *GSP* and *DELISP* in mining time-constrained sequential patterns are compared. In these experiments, *DELISP* is about 3 times faster than *GSP*. Various values of $minsup$, $mingap$, $maxgap$, and $swin$ are used. Note that the mining of sequential patterns without time constraints is a special case with $mingap = 0$, $maxgap = \infty$, and $swin = 0$ here. The results of varying $minsup$ (2%, 1.5%, 1%, 0.75%, 0.5%) are consistent. We set the $minsup$ to 0.75% and focus on the comparisons of varying time constraints in the following.

The result of varying $mingap$ with fixed $maxgap$ and fixed $swin$ is shown in Fig. 4-6. When $mingap = 0$, $maxgap = \infty$, and $swin = 0$, the resulting patterns are the same as common sequential pattern discovery without time constraints. As $mingap$ increases, the number of qualified patterns existing in data sequences decreases, and thereby the total execution time decreases. The total execution time of *GSP* is 2.8 ($mingap = 0$) up to 3.3 ($mingap = 8$) times than that of *DELISP*. It shows that *DELISP* removes more inaccessible elements with larger $mingap$.

Fixing both $mingap$ and $swin$ to zero, Fig. 4-7 shows the result of varying $maxgap$. The number of time-constrained sequential patterns will decrease when the $maxgap$ value increases, since larger $maxgap$ restricts more data sequences to contain

certain patterns. In Fig. 4-7, the line depicting the execution time of *GSP* starts to fall steeply at $maxgap = 4$, because the sample sequences have 4 transactions ($|S| = 4$) in average. Note that *GSP* runs slightly faster without constraints (673 seconds) than with $maxgap = 12$ since most checks eventually are useless and introduce overheads. *DELISP* consistently outperforms *GSP*, from 2.9 ($maxgap = 12$) down to 1.4 ($maxgap = 1$) times, in the experiments.

Next, the *swin* was varied from 0 up to 4 without setting *mingap* and *maxgap* constraints. The *swin* allows adjoining transactions to combine either way to form an element so that each data sequence may contain more patterns. Consequently, more execution time is required with the increased *swin*. When $swin = 0$, it took *GSP* 673 seconds and *DELISP* 238 seconds, respectively, for the discovery. To mine the additional patterns appeared with $swin = 1$, *GSP* spent 815 seconds and *DELISP* spent 272 seconds. Fig. 4-8 displays the effect on performance when constraint *swin* is increased. Both algorithms scale up with the increased *swin*, *DELISP* performs the better.

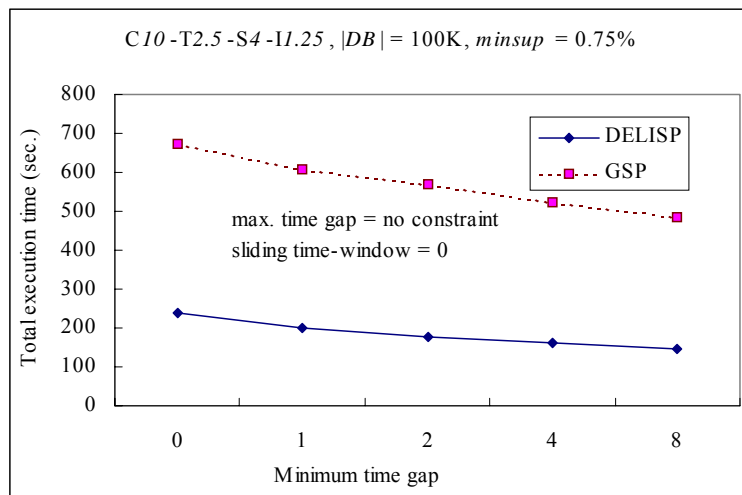


Fig. 4-6. Effect of the *mingap* constraint

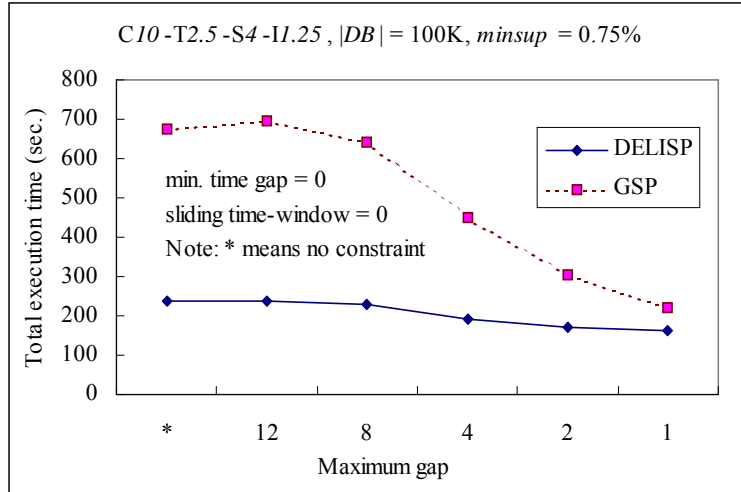


Fig. 4-7. Effect of the *maxgap* constraint

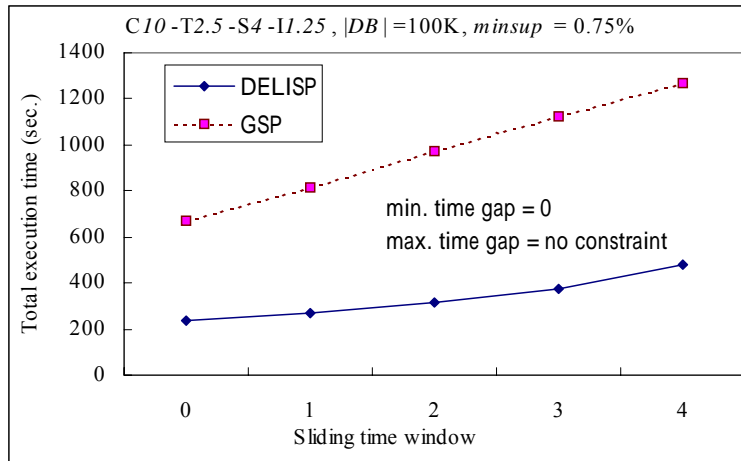
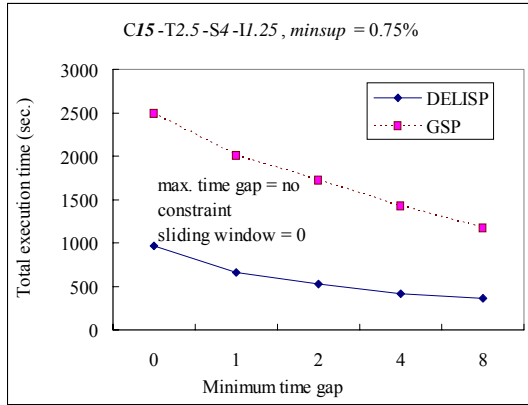


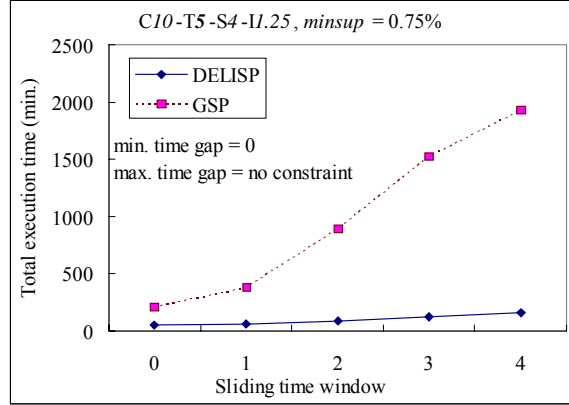
Fig. 4-8. Effect of the *swin* constraint

To evaluate the performance with respect to datasets of different characteristics, the series of experiments were applied on dataset *C15-T2.5-S4-II.25* (varying *mingap*), *C10-T5-S4-II.25* (varying *swin*), *C10-T2.5-S8-II.25* (varying *maxgap*), and *C10-T2.5-S4-II.25* (varying *mingap*). The results for sensitivity analysis, displayed in Fig. 4-9, demonstrate that *DELISP* algorithm consistently outperforms the *GSP* algorithm for various data characteristics.

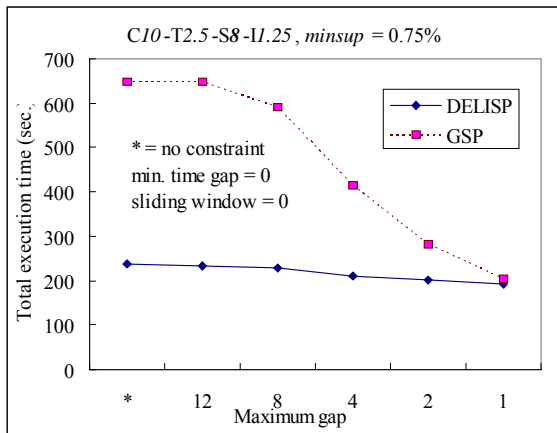
The effects of varying the three constraints on performance are summarized below. With respect to *mingap* constraint, *GSP* effectively prunes the impossible candidates utilizing the monotonic property of candidate generation. For instance, if



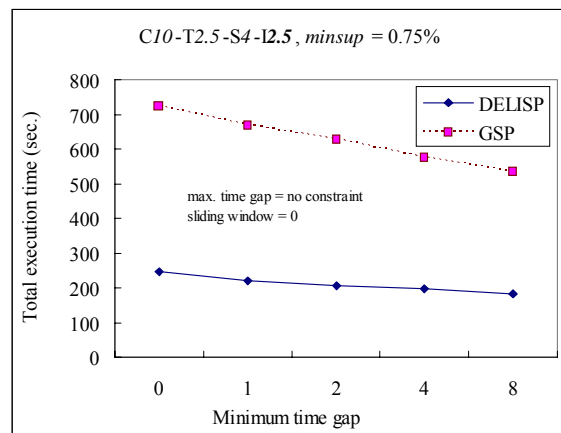
(a) Increase $|C|$ from 10 to 15, varying $mingap$



(b) Increase $|T|$ from 2.5 to 5, varying $swin$



(c) Increase $|S|$ from 4 to 8, varying $maxgap$



(d) Increase $|I|$ from 1.25 to 2.5, varying $mingap$

Fig. 4-9. Total execution time on datasets of various characteristics

(a)(b) fails to be a candidate due to $mingap$, then (a)(b)(c) cannot be a candidate. *DELISP* utilizes $mingap$ constraint to effectively remove the inaccessible items within pattern-growth framework. Both *DELISP* and *GSP* can effectively handles the mining with $mingap$ constraint, while *DELISP* outperforms *GSP* at least two times faster.

In *GSP*, there is performance degradation when $maxgap$ or $swin$ specified. With respect to $maxgap$ constraint, the time for the containment test increases when $maxgap$ is specified. Besides, the number of candidates increases when $maxgap$ is used, since we can no longer prune non-contiguous subsequences [80]. The time for the containment test also increases when $swin$ is specified. In addition, the hash-tree is less effective in reducing the number of candidates that need to be checked against a

data sequence when the user specifies a larger *swin* value.

However, *DELISP* effectively handles all the three constraints by integrating them in sequence projecting and growing within the pattern-growth framework. Thus, the performance difference between *DELISP* and *GSP* increases when *maxgap* or *swin* increases, as shown in Fig. 4-9.

4.5.2 Scale up experiments on database size

In order to justify the scalability of *DELISP*, the number of data sequences was increased from 100K to 1000K with *C10-T2.5-S4-I1.25*. In Fig. 4-10, the total execution times are normalized with respect to the execution time for $|DB| = 100K$. As indicated in Fig. 4-10, the execution time of *DELISP* scales up sub-linearly with the number of data sequences. When $|DB|$ increases to very large size like 800K or 1000K and the average number of items per transaction might be large, the projected sub-databases increase tremendously, which incurs larger overhead in disk accessing. In the experiment, the execution time ratio scaled up sub-linearly. The execution time for *maxgap* = 12 and *swin* = 1 is 271 seconds, and that for *maxgap* = 8, *swin* = 2 is 304 seconds. It reflects that relaxing *swin* has stronger influence than restricting *maxgap* on the number of patterns discovered.

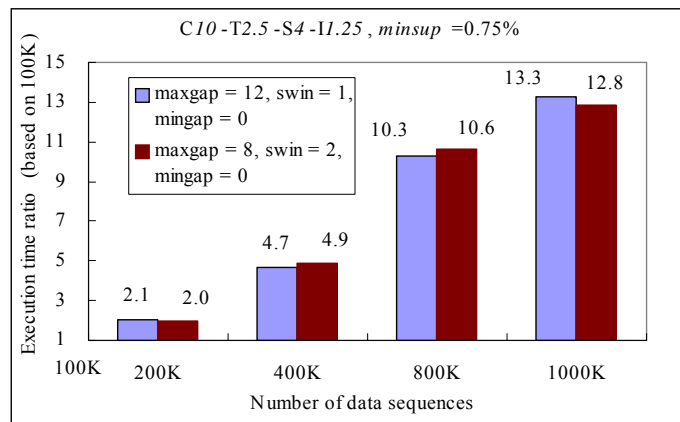


Fig. 4-10. Linear scalability of *DELISP*

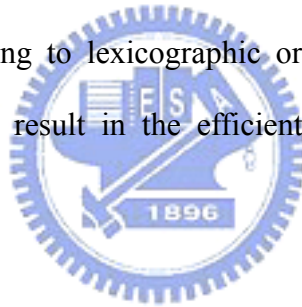
4.6 Discussion

We summarize the factors contributing to the efficiency of the proposed *DELISP* algorithm, by comparing with the well-known *GSP* algorithm below.

- **No candidate generation.** *DELISP* generates no candidates and saves the time for not only candidate generation but also candidate testing. Moreover, the huge space required for candidate hash-tree is eliminated entirely. Such an advantage is shared by all pattern-growth approaches like *PrefixSpan* or *prefix-growth*.
- **Focused search.** *DELISP* projects the accessible elements and grows patterns by considering only constraint satisfied elements in sub-sequences. We search and grow longer patterns in the smaller, promising subspace. In contrast, *GSP* takes every data sequence (the entire sequence) for support calculation in each pass.
- **Constraint integration.** The *maxgap* constraint makes candidate reduction less powerful in *GSP* since some candidates cannot be pruned in advance. For instance, given *maxgap* constraint, a data sequence which supports candidate (a)(e)(f) may not contain candidate (a)(f). Thus, *GSP* suffers from *maxgap* constraint as candidate pruning is less restrictive. Nevertheless, *DELISP* benefits from the *maxgap* constraint by incorporating the constraint in growing and projecting shorter sequences. Some posterior elements of a sequence, once they are inaccessible, need not be considered because of the *maxgap* constraint.
- **Containment checking and sequence shrinking.** In each pass, *GSP* transforms every data sequence into items' transaction-time lists, and switches between alternative phases with excess "pull up" of elements to check whether a data sequence contains a candidate [8]. For instance, *GSP* having found (a)(b) in a data sequence, noticing that adding (c) would violate *maxgap*, has to "pull-up" (b) and maybe then (a), considering their later occurrences. Without any

transformation, at each recursion, *DELISP* shrinks a data sequence by removing non-frequent items, ‘small’ items, and the inaccessible elements. Moreover, *DELISP* finds elements incrementally by checking time-valid subsequences only. The delimited growth technique further assures each growth satisfies the constraints and makes pattern-growth more efficient.

- *DELISP* benefits from the properties of pattern-growth approaches for factors like “no candidate generation” and “focused search”. However, *DELISP* eliminates the need for "switching between forward and backward phases" of *GSP* by extending concurrently all valid occurrences of the pattern used for projection. In addition, *DELISP* preserves the property of growing longer patterns from prefixes (i.e., avoiding the bi-directional growth) by extending pattern elements according to lexicographic order. These core techniques are specific to *DELISP* and result in the efficient discovery of time-constrained sequential patterns.



4.7 Summary

We have presented the *DELISP* algorithm to provide the full functionality of the classic *GSP* algorithm in terms of time constraints. The conducted experiments confirm that with good scalability, the pattern-growth based *DELISP* outperforms the Apriori-based *GSP* algorithm.

However, pattern-growth based algorithms usually require the intermediate storage for the projected sub-databases while mining. Sometimes, the total size of the sub-databases might amount to several times larger than that of the original sequence database. It is desirable to employ the pseudo-projection and bi-level projection techniques [67], described in *PrefixSpan*, in *DELISP* to minimize disk projections. Future improvements may include sharing common sub-sequences among

sub-databases, projecting sub-sequences into memory, or extending the memory indexing approach in Chapter 3 to mine sequential patterns with time constraints. It is also interesting to extend the approach to deal with other time constraints like overall time span [97] and various constraints [25, 47, 56, 97] for effective and efficient sequential pattern mining.



Chapter 5 Algorithm KISP for Interactive Discovery of Sequential Patterns

5.1 Overview

An important issue in data mining is the discovery of sequential patterns, which finds out temporal associations among items in the sequence database [6, 29, 48, 51, 75, 98]. A classic application of the problem is the market basket analysis whose database contains purchase records, where each record is an ordered sequence of *itemsets* (sets of items) bought by a customer. The mining is to discover the itemsets in future purchase after certain itemsets were bought. For example, a discovery might find out a sequential pattern “ $(a, c, d) \Rightarrow (b, e)$ [support=30%]”, which means that 30% of customers who purchase items a, c and d at the same time would buy items b and e at some later time. The technique can be applied to various domains such as discovering the relationships between the symptoms and certain diseases in medical applications.

In order to find the interesting patterns, a user specifies a minimum support threshold (abbreviated *minsup*) for the mining. The result of the mining lists all patterns, named *sequential patterns* or *frequent sequences*, whose *supports* are greater than or equal to the *minsup*. The *support* of a pattern is the percentage of sequences (in the database) containing the pattern. In general, we would generate potential sequential patterns (called *candidates*), count the occurrence of each candidate, and then determine the sequential patterns among these candidates.

The mining process is very difficult and time-consuming due to several factors. First, the formation of a pattern is not limited to single items but itemsets. Second, neither the number of itemsets in a pattern nor the number of items in an itemset is known a priori. Third, patterns could be formed by any permutation, of any

combination of possible items in the database. Most approaches focused on minimizing the search space of candidates [6, 80], or on minimizing the required disk I/O due to the multiple database scanning [75, 98]. Each time a user specifies a *minsup*, all these approaches discover the resultant patterns by executing their mining algorithms with respect to this *minsup*.

However, a user may specify a *minsup* value that results in too many or too few patterns. When the specified *minsup* is too large, either no patterns or only few patterns might satisfy the threshold. On the contrary, the user might have difficulty in distinguishing the interesting patterns from a large number of patterns due to a very small *minsup*. Usually, the user must try various *minsup*s until the result is satisfactory. Nevertheless, most approaches for mining sequential patterns are not designed to deal with repeated mining under such circumstance. For such interactive sequence discovery, these approaches consider no prior results so that the mining process must start over again for every newly specified *minsup*. However, keeping knowledge obtained from the time-consuming process is beneficial to further queries. For example, the result of mining with *minsup* = 0.1 could be used to extract the sequential patterns for *minsup* = 0.3 without re-examining the sequence database.

Therefore, we propose a novel approach, named *KISP* (**k**nowledge base assisted **i**ncremental **s**equential **p**attern mining), to improve the efficiency of sequential pattern discovery with changing supports. Instead of re-mining from scratch for each discovery, *KISP* utilizes the knowledge obtained from prior minings, and generates a knowledge base for further queries about sequential patterns of various *minsup*s. When the sequential patterns cannot be directly derived from the knowledge base, *KISP* incorporates the knowledge base into a fast sequence discovery. The candidates existing in the knowledge base are spared in the support counting process. In addition, the knowledge base could be used to support OLAP since the knowledge, sufficient

for users' interests, of current database is accumulated by *KISP*. The conducted experiments on synthetic data also show that the proposed algorithm effectively improves the performance of interactive sequence discovery.

The rest of the chapter is organized as follows. We formulate the problem of interactive sequential pattern mining in Section 5.2 and review some related algorithms in Section 5.3. Section 5.4 presents the proposed approach for the interactive discovery problem. The experimental evaluation is described in Section 5.5. Section 5.6 summarizes this chapter.

5.2 Problem Statement

Table 5-1 summarizes the notations used in this chapter. Let $\Psi = \{\alpha_1, \alpha_2, \dots, \alpha_z\}$ be a set of literals, called *items*. A set of items is referred to as an *itemset*. An itemset I with m items is denoted by $I = (\beta_1, \beta_2, \dots, \beta_m)$, such that $I \subseteq \Psi$. A *sequence* x , denoted by $\langle a_1 a_2 \dots a_n \rangle$, is an ordered set of n *elements* where each *element* a_j is an itemset. The *size* of the sequence x , denoted by $|x|$, is the total number of items in all the elements in x . Sequence x is a k -sequence if $|x| = k$. For example, $\langle (a)(c)(e) \rangle$, $\langle (b)(c,d) \rangle$, and $\langle (a)(b)(a) \rangle$ are all 3-sequences. A sequence $\omega = \langle a_1 a_2 \dots a_n \rangle$ is a *subsequence* of another sequence $\varpi = \langle b_1 b_2 \dots b_w \rangle$ if there exist $1 \leq i_1 < i_2 < \dots < i_n \leq w$ such that $a_1 \subseteq b_{i_1}$, $a_2 \subseteq b_{i_2}$, ..., and $a_n \subseteq b_{i_n}$. Sequence ϖ *contains* sequence ω if ω is a subsequence of ϖ . For instance, $\langle (b)(e) \rangle$ is a subsequence of $\langle (d)(b)(a)(c,e) \rangle$ since $(b) \subseteq (b)$ and $(e) \subseteq (c,e)$.

Each customer record in the database DB is referred to as a *data sequence*, which is a sequence of purchased itemsets ordered by transaction time. The *support* of sequence x , denoted by $x.sup$, is the number of data sequences containing x divided by the total number of data sequences in database DB . The *minsup* is the user specified minimum support threshold. A sequence x is a *frequent sequence* if $x.sup \geq minsup$.

The sequence x is also called a *sequential pattern*. Given the *minsup* and the database DB , the problem of sequential pattern mining is to discover *the set of all sequential patterns*, denoted by $S[\text{minsup}]$.

The interactive sequence discovery process is described as follows. Given the database DB , the user queries with several *minsup* values interactively, and finds out the desired set of sequential patterns with respect to the final *minsup*. The objective of interactive discovery is to respond to each query quickly and to reduce the overall mining time for the whole process accordingly.

Table 5-1. Notations used

$\Psi = \{\alpha_1, \alpha_2, \dots, \alpha_z\}$	The set of all items.
$\alpha_1, \alpha_2, \dots, \alpha_z, \beta_1, \beta_2, \dots, \beta_m$	Items.
$I = (\beta_1, \beta_2, \dots, \beta_m)$	An m -itemset, $I \subseteq \Psi$.
$x = \langle a_1 a_2 \dots a_n \rangle$	A sequence. Each a_i is an itemset.
$x.\text{sup}$	The support of sequence x .
DB	The database of data sequences.
<i>minsup</i>	The user specified minimum support.
$S[\text{minsup}]$	The set of all sequential patterns in DB with respect to <i>minsup</i> .
$S_k[\text{minsup}]$	The set of frequent k -sequences with respect to <i>minsup</i> . See Section 5.3.1.
$X_k[\text{minsup}]$	The set of candidate k -sequences with respect to <i>minsup</i> . See Section 5.3.1.
KB	The knowledge base. See Section 5.4.1.
$KB.\text{sup}$	The <i>minsup</i> used in the construction of KB . See Section 5.4.1.
$N_k[\text{minsup}]$	The set of new frequent k -sequences with respect to <i>minsup</i> . See Section 5.4.2.
Xk'	The reduced set of candidate k -sequences. See Section 5.4.2.

Example 5-1: Interactive sequence discovery without knowledge base. Table 5-2 shows the supports of all sequences in an example database. The sequences are grouped by sequence-size and are listed in ascending order of supports. The interactive sequence discovery is described below. For convenience, we list the results of the four example queries in Table 5-3.

Table 5-2. The supports of all sequences in an example database

Sequence	Support	Sequence	Support
<(a)>	0.90	<(a,b)>	0.80
<(b)>	0.82	<(a)(c)>	0.70
<(c)>	0.75	<(a)(e)>	0.60
<(e)>	0.62	<(b)(b)>	0.55
<(d)>	0.40	<(a)(b)>	0.53
<(a,c)(e)>	0.40	<(c)(e)>	0.51
<(a)(c)(e)>	0.30	<(a,c)>	0.45
<(c)(b,e)>	0.27	<(c)(b)>	0.30
<(a,c)(b)>	0.18	<(b,e)>	0.29
<(a)(b,e)>	0.12	<(b)(c)>	0.20
<(a,c)(b,e)>	0.10	<(a)(d)>	0.10
<*>	Less than 0.10		

Note: <*> represents the sequence whose support < 0.1, e.g. <(b)>.sup = 0.08.

Table 5-3. User specified minimum supports and the resultant sequential patterns

Query	The <i>minsup</i> value and the set of all sequential patterns	Frequent <i>k</i> -sequences and new <i>k</i> -sequences
First	$minsup = 0.7,$ $S[0.7] = S_1[0.7] \cup S_2[0.7] \cup S_3[0.7] \cup S_4[0.7].$	$S_1[0.7] = \{<(a)>, <(b)>, <(c)>\}.$ $S_2[0.7] = \{<(a,b)>, <(a)(c)>\}.$ $S_3[0.7] = S_4[0.7] = \emptyset.$
Second	$minsup = 0.4,$ $S[0.4] = S_1[0.4] \cup S_2[0.4] \cup S_3[0.4] \cup S_4[0.4].$	$S_1[0.4] = S_1[0.7] \cup N_1[0.4],$ $N_1[0.4] = \{<(e)>, <(d)>\}.$ $S_2[0.4] = S_2[0.7] \cup N_2[0.4],$ $N_2[0.4] = \{<(a)(e)>, <(b)(b)>, <(a)(b)>, <(c)(e)>, <(a,c)>\}.$ $S_3[0.4] = \{<(a,c)(e)>\}.$ $S_4[0.4] = \emptyset.$
Third	$minsup = 0.1,$ $S[0.1] = S_1[0.1] \cup S_2[0.1] \cup S_3[0.1] \cup S_4[0.1].$	$S_1[0.1] = S_1[0.4] \cup S_1[0.1],$ $N_1[0.1] = \emptyset.$ $S_2[0.1] = S_2[0.4] \cup N_2[0.1],$ $N_2[0.1] = \{<(c)(b)>, <(b,e)>, <(b)(c)>, <(a)(d)>\}.$ $S_3[0.1] = S_3[0.4] \cup N_3[0.1],$ $S_3[0.1] = \{<(a)(c)(e)>, <(c)(b,e)>, <(a,c)(b)>, <(a)(b,e)>\}.$ $S_4[0.1] = \{<(a,c)(b,e)>\}.$
Final	$minsup = 0.3,$ $S[0.3] = S[0.1] - \{x x \in S[0.1] \wedge x.sup < 0.3\}$ $= \{<(a)>, <(b)>, <(c)>, <(d)>, <(e)>, <(a,b)>, <(a)(c)>, <(a)(e)>, <(b)(b)>, <(a)(b)>, <(c)(e)>, <(a,c)>, <(c)(b)>, <(a,c)(e)>, <(a)(c)(e)>\}.$	

At first, the user specified 0.7 as the *minsup* and mined the database. Only five

patterns were found so that the user decided to discover more patterns. The second query with a smaller *minsup* ($minsup = 0.4$) found out more patterns (total 13 patterns) than the first discovery. Running the third time of the mining algorithm, 22 patterns in total were returned for the third trial with $minsup = 0.1$. Finally, the user located all sequential patterns whose supports are at least 0.3 by the fourth execution. The overall response time for the interactive process is the total time spent for the four rounds of execution. Although the result of the final mining could be obtained by retrieving qualified patterns after the third query, current approaches generally re-execute the mining algorithm without utilizing previous results. On the contrary, Example 5-2 shows that the knowledge base helps to reduce the time for the last three example queries. That is, the total response time is reduced. \square

Example 5-2: Interactive sequence discovery using discovered patterns. The example database is the same as in Example 5-1. The process for the same four queries is as follows.

There was no advantage for the first mining with an empty knowledge base. A *knowledge base* (abbreviated *KB*) containing patterns whose supports are at least 0.7 was built after the first query. For the second trial, patterns in the *KB*, such as $\langle(b)\rangle$ or $\langle(a)(c)\rangle$, need not be counted again since they are still frequent with respect to $minsup = 0.4$. Only the newly generated candidates, such as $\langle(a)(e)\rangle$ or $\langle(d)(e)\rangle$, were counted against the sequence database. After the second query, the *KB* has more information by accumulating the new patterns such as $\langle(c)(e)\rangle$. Similarly, the support counting of patterns kept in the *KB* were eliminated in the third mining. The employment of the *KB* can accelerate the support counting process by reducing the number of candidates. The *KB* contains all the patterns whose supports are at least 0.1 after the third query. At last, no counting is necessary since $S[0.3]$ can be directly extracted from the *KB* without any database access. The response time for every query

is reduced by the use of a knowledge base, except for the first query requiring the same execution time, and consequently the overall response time is reduced. □

Therefore, we propose the *KISP* mining algorithm to effectively utilize the discovered knowledge for interactive sequence discovery. In fact, the knowledge base built by *KISP* keeps not only the supports of sequential patterns, but also the supports of all candidates generated in prior minings. The fast response time of interactive sequence discovery is achieved by the use of the knowledge base, which is incrementally built by accumulating the information obtained in the mining processes.

5.3 Related Work

Few researches are directly related to interactive sequence discovery. In Section 5.3.1, we review some algorithms for sequential pattern mining. Section 5.3.2 presents related approaches for interactive pattern discovery.

5.3.1. Algorithms for sequential pattern mining

The *AprioriAll* [6] is the first algorithm dealing with sequential pattern discovery [6, 48, 89]. *AprioriAll* splits sequential pattern mining into three phases: itemset phase, transformation phase, and sequence phase. The itemset phase uses *Apriori* to find all frequent itemsets. The database is transformed by replacing each transaction by the set of all frequent itemsets contained in the transaction in the transformation phase. In the third phase, *AprioriAll* makes multiple passes over the database to generate candidates and to count the supports of candidates. In subsequent work, the same authors proposed the *GSP* (**G**eneralized **S**equential **P**attern) algorithm that outperforms *AprioriAll* [80]. Both algorithms use the similar techniques for candidate generation and support counting, as described in the following.

GSP algorithm makes multiple passes over the database and finds out frequent

k -sequences at k -th database scanning. In each pass, every data sequence is examined to update the support counts of the candidates contained in this sequence. Initially, each item is a candidate 1 -sequence for the first pass. Frequent 1 -sequences are determined after checking all the data sequences in the database. In succeeding passes, frequent $(k-1)$ -sequences are self-joined to generate candidate k -sequences. Again, the supports of these candidate sequences are counted by examining all data sequences, and then those candidates having minimum supports become frequent sequences. This process terminates when there is no candidate sequences any more. In the following, we further describe two essential sub-processes in *GSP*, the candidate generation and the support counting.

Candidate generation: Let $S_k[minsup]$ denote the set of all frequent k -sequences and $X_k[minsup]$ denote the set of all candidate k -sequences with respect to *minsup*. *GSP* generates $X_k[minsup]$ by two steps. The first step joins $S_{k-1}[minsup]$ with $S_{k-1}[minsup]$ and obtains a superset of the final $X_k[minsup]$. Those candidates having any $(k-1)$ -subsequence which is not in $S_{k-1}[minsup]$ are deleted in the second step. In the first step, we join a sequence x with another sequence y if the subsequence obtained by dropping the first item of x is the same as the subsequence obtained by dropping the last item of y . The resultant candidate from this join is the sequence x extended with the last item of y . The added item becomes the last element (of the candidate) if the last item of y itself is an element. Otherwise, the added item becomes the last item of the last element (of the candidate). For example, the candidate $\langle(a)(c)(e)\rangle$ is generated by joining $\langle(a)(c)\rangle$ with $\langle(c)(e)\rangle$, and the candidate $\langle(a)(c,e)\rangle$ is generated by joining $\langle(a)(c)\rangle$ with $\langle(c,e)\rangle$. Besides, the $X_k[minsup]$ produced from this procedure is a superset of $S_k[minsup]$ as proved in [80]. That is, $X_k[minsup] \supseteq S_k[minsup]$.

Specifically note that for candidate 2-sequences, the generation of $X_2[minsup]$ is

described by the formula: $X_2[minsup] = \{ \langle (x_1, x_2) \rangle \mid \forall x_1, \forall x_2 \in S_1[minsup], x_1 \neq x_2 \} \cup \{ \langle (x_1)(x_2) \rangle \mid \forall x_1, \forall x_2 \in S_1[minsup] \}$. Take the database in Table 5-2 for instance, $S_1[0.8] = \{ \langle (a) \rangle, \langle (b) \rangle \}$, so that $X_2[0.8] = \{ \langle (a,b) \rangle, \langle (a)(a) \rangle, \langle (a)(b) \rangle, \langle (b)(a) \rangle, \langle (b)(b) \rangle \}$.

Support counting: *GSP* adopts a hash-tree structure [7, 80] for storing candidates to reduce the number of candidates that need to be checked for each data sequence. Candidates would be placed in the same leaf if their leading items, starting from the first item, were hashed to the same node. The next item is used for hashing when an interior node, instead of a leaf node, is reached [80]. The candidates required for checking against a data sequence are located in leaves reached by applying the hashing procedure on each item of the data sequence [80]. The support of the candidate is incremented by one if it is contained in the data sequence.

The *SPADE* (**S**equential **P**attern **D**iscovery using **E**quivalence classes) algorithm finds out sequential patterns using vertical database layout and join-operations [98]. Vertical database layout transforms data sequences into item-oriented lists. For example, the transformation of a sequence $\langle (a,c)(e) \rangle$ with sequence id = *C310* would generate an entry (*C310*, a) in the list of item 'a', an entry (*C310*, c) in the list of item 'c', and an entry (*C310*, e) in the list of item 'e'. The lists are joined to form a sequence lattice, in which *SPADE* searches and discovers the patterns [98].

Recently, the *FreeSpan* (**F**requent pattern-projected **S**equential **P**attern Mining) algorithm was proposed to mine sequential patterns by a database projection technique [29]. *FreeSpan* first finds the frequent items after scanning the database once. The sequence database is then projected, according to the frequent items, into several smaller databases. Finally, all sequential patterns are found by recursively growing subsequence fragments in each projected database. Based on the similar projection technique, *PrefixSpan* (**P**refix-projected **S**equential **p**attern mining)

algorithm [67] efficiently mines the complete set of patterns employing a divide-and-conquer strategy with the PatternGrowth methodology.

However, as mentioned above, all these algorithms re-execute the mining procedure every time a new *minsup* is specified during the interactive process. Therefore, the response time would be longer for subsequent queries with smaller *minsup* values with all these algorithms.

5.3.2 Algorithms for interactive pattern discovery

The objective of interactive pattern discovery is to reduce the response time for users' online queries. In general, the discovery of frequent patterns in large databases is categorized into association discovery and sequence discovery. The problem of interactive association discovery, also called online association generation, was addressed in [3]. The method in [3] preprocesses the data in the transactional database, and stores frequent itemsets in an adjacency lattice. Each vertex in the adjacency lattice is labeled with the support of the corresponding itemset. A directed edge in the lattice links from a 'parent' itemset to one of its 'child' itemsets. An itemset Y is a 'child' of itemset X if Y can be obtained from X by dropping a single item from X . Online repeated queries about association rules are answered by graph theoretic searching on the lattice.

Similarly, a knowledge cache storing the discovered frequent itemsets and the non-frequent itemsets is used for interactive discovery of association rules [54]. It is indicated that their *benefit replacement* algorithm using B+-tree to store cache buckets is the best caching algorithm [54].

Although on-line association discovery is close to our problem, the aim of these approaches [3, 30, 54, 63] is to interactively find frequent itemsets rather than frequent sequences. Sequence discovery is more complicated than association

discovery because with n frequent items, the total number of candidates in pass k is

$$\binom{n}{k} \text{ for association discovery and } \sum_{i_1=1}^k \binom{n}{i_1} \sum_{i_2=1}^{k-i_1} \binom{n}{i_2} \sum_{i_3=1}^{k-i_1-i_2} \binom{n}{i_3} \Lambda \sum_{i_k=1}^{k-i_1-\dots-i_{k-1}} \binom{n}{i_k} \text{ for}$$

sequence discovery. One related work of interactive sequence mining is described below.

The *SPADE* algorithm [98] was extended into the *ISM* (Incremental Sequence Mining) algorithm for incremental sequence mining and interactive sequence mining [64]. All queries are performed on a pre-processed in-memory data structure, the Increment Sequence Lattice (*ISL*). Therefore, a ‘small enough’ *minsup* must be selected in advance to mine all patterns by executing *SPADE* and save the results in the *ISL*. Nevertheless, if a query involves a support smaller than the pre-selected *minsup*, another (more) lengthy mining process must be performed to generate another new *ISL* sufficient for the new query. Moreover, the *ISM* might encounter memory problem if the number of the potentially frequent patterns is too large [64].

Without any assumption on the possible values of *minsup*, our algorithm aims to reduce the response time for each query for sequential patterns in a large database. In the proposed algorithm, subsequent mining is assisted with the information accumulated from prior mining processes and an efficient interactive sequence discovery is achieved.

5.4 The Proposed Algorithm for Interactive Discovery of Sequential Patterns

The proposed *KISP* algorithm is described in Section 5.4.1. The algorithm speeds up the mining process by eliminating the counting efforts required for those candidates already existing in the knowledge base. Two optimizations are proposed for further improvements. In Section 5.4.2, the generation of the remaining ‘new’ candidates is

optimized by direct computation. Enabled by candidate reduction and assisted by the information in the knowledge base, the optimization by current support counting is depicted in Section 5.4.3. Section 5.4.4 presents the manipulation of the knowledge base. Section 5.4.5 discusses the mining efficiency and space utilization with a large knowledge base.

5.4.1 The *KISP* (Knowledge base assisted Incremental Sequential Pattern) mining algorithm

Fig. 5-1 outlines the proposed Basic *KISP* algorithm for interactive discovery of sequential patterns. We adopt the *GSP* algorithm as the basis for constructing the knowledge base assisted mining algorithm. *KISP* uses similar procedures of candidate generation and support counting as used in *GSP*. Nevertheless, *KISP* speeds up support counting by reducing considerable amounts of candidates. It reduces the number of database passes by concurrent counting of variable-sized candidates. Consequently, *KISP* makes a significant performance improvement for interactive discovery.

During the interactive process, the *knowledge base* (denoted by *KB*) is empty only in the very first mining. Once *KISP* is executed, the information about the supports of counted candidates would be inserted into *KB*. The *KB.sup* is the *minsup* used when *KB* is constructed or expanded. Although *KISP* would degenerate into the *GSP* algorithm with an empty *KB*, *KISP* will enrich *KB* from every counting effort in later minings. The details are given below.

In the beginning, *KB* contains no information since no mining has been performed. *KISP* works similar to *GSP* for the very first mining. Initially every item in the database is a candidate *I*-sequence. The fundamental *KB* is built, only once, by a simple scan over the database to count the supports of candidate *I*-sequences (line

1). After that, the supports of all candidate l -sequences are included in KB , and $S[minsup]$ contains the frequent l -sequences (line 2). Since the supports of candidates having size other than one is unavailable from KB at the time being, no candidate counting can be spared by $KISP$. At the end of this mining, KB would collect the supports of all the candidates in each pass (line 13), and $KB.sup$ is the $minsup$ designated for this mining (line 19).

```

Algorithm KISP ( $DB, KB, minsup$ )
Input:  $DB$  = the database of data sequences;  $minsup$  = user specified minimum support ;
       $KB$  = knowledge base having the supports of all the candidates in prior minings
Output :  $S[minsup]$  = sequential patterns with respect to  $minsup$ ;  $KB$  = (new) knowledge base

// Let  $x.sup$  be the support of a candidate  $x$ ,  $X_k[minsup]$  be the set of candidate  $k$ -sequence in  $DB$  with
// respect to  $minsup$ , and  $KB.sup$  be the smallest  $minsup$  used in the construction of  $KB$ 
1) if  $KB = \phi$  then  $KB = \{x \text{ and } x.sup, \forall x \in X_1\}$  ;
2)  $S[minsup] = \{x | x \in KB \wedge x.sup \geq minsup\}$  ; // obtain valid sequential patterns from knowledge base
3) if  $minsup < KB.sup$  then // mine new patterns and accumulate new knowledge
4)    $k = 2$  ;
5)   generate  $X_k[minsup]$  from the frequent  $(k-1)$ -sequences in  $S[minsup]$  ;
6)    $X_k' = X_k[minsup] - \{x | x \in KB\}$  ; // eliminate those candidate  $k$ -sequences in  $KB$ 
7)   while  $X_k' \neq \phi$  do // there exists candidate  $k$ -sequences, obtains their supports
8)     for each data sequence  $ds$  in database  $DB$  do
9)       for each candidate  $x \in X_k'$  do
10)        increase the support of  $x$  if  $x$  is contained in  $ds$  ;
11)      endfor
12)    endfor
13)    $KB = KB \cup \{x \text{ and } x.sup, \forall x \in X_k'\}$  ; // collect new candidates and their supports
14)    $S[minsup] = S[minsup] \cup \{x | x.sup \geq minsup \wedge x \in X_k'\}$  ; // collect new patterns from  $X_k'$ 
15)    $k = k+1$  ;
16)   generate  $X_k[minsup]$  from the frequent  $(k-1)$ -sequences in  $S[minsup]$  ;
17)    $X_k' = X_k[minsup] - \{x | x \in KB\}$  ; // the reduced set eliminates candidate  $k$ -sequences in  $KB$ 
18) endwhile
19)  $KB.sup = minsup$  ; // update the smallest  $minsup$  of  $KB$ 
20)endif

```

Fig. 5-1. Proposed Algorithm Basic $KISP$

Note that in KB besides the sequential patterns we also keep the supports of all candidates regardless of their values for two reasons. First, several currently non-frequent candidates might turn out to be frequent when a smaller $minsup$ is specified in subsequent queries. We can immediately obtain these patterns from KB without any database access. Second, to find out the true patterns, the mining process

generally counts a large number of candidates although they are eventually rejected. We can get rid of the ‘useless counting’ for the ‘commonly non-frequent’ candidates if their supports were kept. For example, those candidates ever counted with the support value of zero would not be inserted into the candidate hash-tree afterward. Consequently, a faster counting is enabled due to the smaller hash-tree of the reduced set of candidates.

For subsequent queries, KB is not empty. Assume that the user specifies $minsup$ to $KISP$ with a non-empty KB . KB now contains the supports of all the generated candidates while mining with $KB.sup$ as the support threshold. Since the supports of all the candidates in KB are available, whether new counting is required or not depends on the values of $minsup$ and $KB.sup$. If the $minsup$ is greater than $KB.sup$, we simply search in KB for patterns whose supports satisfy the new $minsup$, and return all patterns in $S[minsup]$ (line 2). KB and $KB.sup$ stay intact since no counting is performed. In this case, the employment of KB eliminates the need of re-mining completely in comparison with GSP . Tremendous gains in performance can be resulted from direct retrieval of valid patterns without re-counting the huge database. In fact, $KISP$ would output all the valid patterns in constant time independent of the database size when $KB.sup$ is less than the user specified $minsup$. On the contrary, other re-mining based algorithms such as GSP need to re-scan the database.

In case the $minsup$ is less than $KB.sup$, we have to mine the database for new patterns that are not in KB . The fundamental difference between $KISP$ and GSP is that $KISP$ only needs to count the supports of the ‘new’ candidates by sparing the counting of the candidates already existing in KB (line 6 and line 17). Even the modest technique spares the counting of a substantial amount of candidates, as confirmed by our experiments. Take the number of candidates in pass 2 for example. Assume that in query Q_i , there are 100 frequent 1-sequences so that $(100*100)+(100*99)/2 = 14950$

candidate 2-sequences are generated and counted in pass 2. Assume that the number of frequent 1-sequences is 110 for query Q_{i+1} . In pass 2 of Q_{i+1} , GSP must count in total $(110*110)+(110*109)/2 = 18095$ candidates, while $KISP$ counts only $(18095-14950)= 3145$ candidates. In each pass of a query, we first generate the candidates and then remove those existing in KB . Next, we expand KB with the support of every new candidate for reuse in future mining processes (line 13). The sequential patterns are collected (line 14). Finally, $KB.sup$ is replaced by the new $minsup$ since the counting base is changed (line 19). Thereafter, $KISP$ uses KB to answer all queries whose $minsup$ are greater than or equal to $KB.sup$. The ‘new pattern’ mining part (line 3 through line 20), which is also the part of new information acquisition step, of the procedure is activated again only when $minsup < KB.sup$ occurs in subsequent queries.

In fact, instead of generating all candidates and then removing the counted ones (line 5 then line 6, and line 16 then line 17), the optimized $KISP$ directly generates the new candidates requiring counting with the assistance of KB , as presented in Section 5.4.2. In the following context, $KISP$ stands for the optimized $KISP$.

5.4.2 New-candidate generation by direct computation

The first optimization in $KISP$ is the direct generation of new candidates. In GSP , the joining-then-pruning procedure generates the set of required candidate k -sequences in pass k . $KISP$ further removes the candidates existing in KB from the set before counting. The remaining candidates are referred to as *new-candidates* in $KISP$. The candidates to be removed from counting are those generated by the self-join of the frequent $(k-1)$ -sequences in $S_{k-1}[KB.sup]$. Therefore, any formation of the *new-candidates* must involve one of the new frequent $(k-1)$ -sequences. These new-candidates can be directly generated as follows.

Let $X_k[minsup]$ be the set of candidate k -sequences, $S_k[minsup]$ be the set of frequent k -sequences, and X_k' be the reduced set of candidate k -sequence, i.e. the new-candidates in pass k . We use $N_k[minsup]$ to designate the new frequent k -sequences (due to *minsup*) in contrast to the frequent k -sequences in KB . Recall that *KISP* mines the database for new patterns only when $minsup < KB.sup$. Hence, $S_k[minsup] = S_k[KB.sup] \cup N_k[minsup]$. The X_k' is the union of the two sets; one obtained from joining the frequent $(k-1)$ -sequences in KB with the new frequent $(k-1)$ -sequences, the other is obtained from self-joining the new frequent $(k-1)$ -sequences. Theorem 5-1 derives the X_k' .

Theorem 5-1. $X_k' = (S_{k-1}[KB.sup] \otimes N_{k-1}[minsup]) \cup (N_{k-1}[minsup] \otimes N_{k-1}[minsup])$, where “ \otimes ” represents the join operation described in Section 3.1.

Proof. $X_k[minsup] = S_{k-1}[minsup] \otimes S_{k-1}[minsup]$,

- 1) $X_k[minsup] = (S_{k-1}[KB.sup] \cup N_{k-1}[minsup]) \otimes (S_{k-1}[KB.sup] \cup N_{k-1}[minsup])$.
- 2) $X_k[minsup] = (S_{k-1}[KB.sup] \otimes S_{k-1}[KB.sup]) \cup (S_{k-1}[KB.sup] \otimes N_{k-1}[minsup]) \cup (N_{k-1}[minsup] \otimes S_{k-1}[KB.sup]) \cup (N_{k-1}[minsup] \otimes N_{k-1}[minsup])$.
- 3) $X_k[minsup] = X_k[KB.sup] \cup (S_{k-1}[KB.sup] \otimes N_{k-1}[minsup]) \cup (N_{k-1}[minsup] \otimes N_{k-1}[minsup])$ due to $X_k[KB.sup] = S_{k-1}[KB.sup] \otimes S_{k-1}[KB.sup]$ and $N_{k-1}[minsup] \otimes S_{k-1}[KB.sup] = S_{k-1}[KB.sup] \otimes N_{k-1}[minsup]$.
- 4) By definition $X_k' = X_k[minsup] - X_k[KB.sup]$, so $X_k' = (S_{k-1}[KB.sup] \otimes N_{k-1}[minsup]) \cup (N_{k-1}[minsup] \otimes N_{k-1}[minsup])$ since $X_k[KB.sup] \cap [(S_{k-1}[KB.sup] \otimes N_{k-1}[minsup]) \cup (N_{k-1}[minsup] \otimes N_{k-1}[minsup])] = \emptyset$. □

The direct generation of new-candidates eliminates the searching and the removing of candidates in KB , and speeds up the mining process. Example 5-3 contrasts the number of candidates requiring support counting in *GSP* and in *KISP*. It also shows that *KISP* might generate very few candidates even for a low *minsup*. The counting effort of each mining incrementally expands KB so that *KISP* is gradually

enhanced with greater candidate reduction capability during the interactive process.

Example 5-3: Number of candidates generated by GSP, and by KISP. The database and the queries are the same as in Example 5-1. Assume that the set of items $\Psi = \{a, b, c, d, e, f\}$. Table 5-4 tabulates the candidates generated by GSP and those generated by KISP.

Table 5-4. Candidates generated by GSP and by KISP

Query	Candidate k -sequences	
	GSP	KISP
First (<i>minsup</i> = 0.7)	$X_1[0.7] = \{ \langle (a) \rangle, \langle (b) \rangle, \langle (c) \rangle, \langle (d) \rangle, \langle (e) \rangle, \langle (f) \rangle \}$ Number of candidates in $X_1[0.7] = 6$	The same as in GSP
	$X_2[0.7] = S_1[0.7] \otimes S_1[0.7]$ Number of candidates in $X_2[0.7] = 12$	The same as in GSP
Second (<i>minsup</i> = 0.4)	$X_1[0.4] = X_1[0.7]$ Number of candidates in $X_1[0.4] = 6$	0
	$X_2[0.4] = S_1[0.4] \otimes S_1[0.4]$ Number of candidates in $X_2[0.4] = 35$	$X_2' = (S_1[0.7] \otimes N_1[0.4]) \cup (N_1[0.4] \otimes N_1[0.4])$ Number of candidates in $X_2' = 23$
	$X_3[0.4] = S_2[0.4] \otimes S_2[0.4]$ Number of candidates in $X_3[0.4] = 5$	$X_3' = (S_2[0.7] \otimes N_2[0.4]) \cup (N_2[0.4] \otimes N_2[0.4])$ Number of candidates in $X_3' = 5$
Third (<i>minsup</i> = 0.1)	$X_1[0.1] = X_1[0.7]$ Number of candidates in $X_1[0.1] = 6$	0
	$X_2[0.1] = X_2[0.4]$ Number of candidates in $X_2[0.1] = 35$ (Note: $S_1[0.1] = S_1[0.4]$)	$X_2' = (S_1[0.4] \otimes N_1[0.1]) \cup (N_1[0.1] \otimes N_1[0.1])$ Number of candidates in $X_2' = 0$
	$X_3[0.1] = S_2[0.1] \otimes S_2[0.1]$ Number of candidates in $X_3[0.1] = 14$ (Note: after 4 candidates pruned)	$X_3' = (S_2[0.4] \otimes N_2[0.1]) \cup (N_2[0.1] \otimes N_2[0.1])$ Number of candidates in $X_3' = 9$ (Note: after 4 candidates pruned)
	$X_4[0.1] = S_3[0.1] \otimes S_3[0.1]$ Number of candidates in $X_4[0.1] = 1$	$X_4' = (S_3[0.4] \otimes N_3[0.1]) \cup (N_3[0.1] \otimes N_3[0.1])$ Number of candidates in $X_4' = 1$
Final (<i>minsup</i> = 0.3)	$X_1[0.3] = X_1[0.4]$ Number of candidates in $X_1[0.3] = 6$	0
	$X_2[0.3] = X_2[0.4]$ Number of candidates in $X_2[0.3] = 35$ (Note: $S_1[0.3] = S_1[0.4]$)	0
	$X_3[0.3] = S_2[0.3] \otimes S_2[0.3]$ Number of candidates in $X_3[0.3] = 8$	0

(a) Candidates generated by GSP. GSP generates 6+12 = 18 candidates for the

first query. For the second and the third query, there are $46(6+35+5)$ and $56(6+35+14+1)$ candidates requiring support counting by *GSP*, respectively. Note that four candidates are pruned before counting in the third mining. For instance, candidate $\langle a, b, e \rangle$ is pruned because it contains a non-frequent subsequence $\langle a, e \rangle$. As described in Example 5-1, in total 49 candidates still need support counting by *GSP* for the final query without a knowledge base.

(b) Candidates generated by *KISP*. For the first query, *KISP* generates the same number of candidates as in *GSP* since *KB* is empty. For the remaining queries, *KB* already has the supports of the entire candidate *l*-sequences so that no candidate *l*-sequence is generated. As shown in Table 5-4, *KISP* generates only 28 candidates for $minsup = 0.4$. Moreover, only 10 candidates are generated for $minsup = 0.1$. Finally, no candidate is generated for the last query since all the desired patterns are available from *KB*. \square

With the assistance of *KB*, *KISP* directly generates fewer candidates for support counting in comparison with *GSP*. The capability of candidate reduction becomes more powerful as the minimum support threshold getting smaller gradually. In *GSP*, the number of candidates is proportional to the value of $minsup$, while the number of new-candidates is not necessarily proportional to $minsup$ in *KISP*. *KISP* might have only very few new-candidates at a very low $minsup$ value since the information gathered from each mining during the interactive process all contribute to the candidate reduction. In each pass, the number of candidates inserted into the hash-tree is smaller. Therefore, *KISP* is enabled to accommodate more candidates, even candidates of different size, in the same hash-tree during the same pass of database scanning. The improved counting technique and the placement of variable sized candidates are described in Section 5.4.3.

5.4.3 Concurrent support counting and the placement of variable sized candidates

The second optimization in *KISP* is the technique of concurrent support counting. Being a multi-pass based instead of a memory-based mining algorithm, without optimization, the number of database passes required in *KISP* (also in *GSP*) is equal to the size of the longest pattern. Concerning mining efficiency, reducing the number of database scanning is thus as important as minimizing the search space of candidates. Concurrent support counting is used to achieve database-pass reduction while preserving the completeness of pattern discovery. Specifically, we can reduce the number of database accesses if we count not only the supports of candidate k -sequences but also that of length longer than k in pass k . An intuitive way is to generate candidates of all sizes simultaneously. Nevertheless, the support counting would be slowed down if the space for storing candidates exceeds the memory limit so that the candidates have to be fetched from the disk rather than the memory. Therefore, the available memory restricts the generation of all sized candidates at the same time.

In general, *KISP* counts the supports of candidate k -sequences in pass k . One situation is that the available memory is not enough for the generation of candidate k -sequences. For example, the number of candidate 2-sequences might be huge for the very first mining. Analogous to *GSP*, if the set of frequent $(k-1)$ -sequences, i.e. $S_{k-1}[KB.sup]$ and $N_{k-1}[minsup]$, cannot fit into the memory, the reduced candidate set X_k' is generated by the relational merge-join technique without pruning [80]. If the memory can completely hold $S_{k-1}[KB.sup]$ and $N_{k-1}[minsup]$, but not the entire X_k' , then *KISP* generates as many candidates of the X_k' as possible. The supports of these candidates are counted and are written out to disk. This procedure is repeated until all the candidates are processed.

On the contrary, the X_k' is more likely to occupy just a small part of the memory at pass k as illustrated in Section 5.4.2. In *KISP*, we maximize memory utilization to reduce the number of database passes by inserting as many candidates as possible into the same hash-tree. We continuously generate the candidates of longer size until the memory space is nearly full. All the candidates of the same size are inserted in a batch at the same time. With the information about $S_{k-1}[KB.sup]$ and the $N_{k-1}[minsup]$, *KISP* can estimate the number of new-candidates, which indicates the space required. Therefore, we can place variable-sized candidates in the same hash-tree and concurrently count the supports against the data sequences in the same database pass. This technique reduces the total number of database scanning. The estimation procedure and the placement of variable-sized candidates that enables concurrent counting are described in the following.

Considering the number of candidates generated in each pass, the number of candidates in X_2' is greater than that in other X_k' because none in the candidate superset of size two can be pruned. Every frequent 1-sequence must join with other frequent 1-sequence since the subsequence of any frequent 1-sequence is an empty sequence. For candidates of X_k' where $k > 2$, some frequent $(k-1)$ -sequences are not joined if their subsequences do not match. Assume the number of patterns in $S_1[KB.sup]$ is p and the number of patterns in $N_1[minsup]$ is q . The number of new-candidates in pass 2 is $[3(p+q)^2-(p+q)]/2-(3p^2-p)/2 = 3pq+(3q^2-q)/2$. This formula can be applied to roughly estimate the maximum number of candidates in other passes. Whenever there is room for the next set of candidates (of longer size), *KISP* continuously generates and inserts the candidates into the same hash-tree. Thus, *KISP* can generate as many candidates as possible in the same pass.

Originally, the hash-tree in *GSP* is designed to store the same sized candidates in the leaves. The leaf where a candidate should be placed in is the leaf reached by

consecutive hashing on the items of the candidate. Since the *GSP*-generated candidates are of the same size in the same hash-tree, the item for hashing is always available while determining the branch to be followed. Nonetheless, the accommodation of variable sized candidates in the same hash-tree might have the problem of having no item for hashing. For example, inserting a candidate 4-sequence might cause the re-distribution of an overflowed node, while the re-distribution might need to hash on the fourth item of a candidate 3-sequence in the node. We modify the hashing procedure slightly to store the same prefixed candidates, despite their sizes, in the same leaf. In case there is no item for hashing any more, the candidate is stored in one of the descendent leaves (due to the splitting of the overflowed leaf). We select the leaf having the fewest number of candidates stored to maximize memory utilization. Since candidates of different size are stored in the same hash-tree, we can check the variable sized candidates against a data sequence at the same time. Therefore, the concurrent support counting minimizes the number of database passes required in *KISP*.

Note that a similar technique named *pass bundling* is described for association mining in [53]. However, *pass bundling* statically sets a limit to determine whether the generation should be continued or not, while *KISP* dynamically estimates and computes the available memory for maximum utilization. The next section will describe the structure and the manipulation of the knowledge base, which is the key to facilitate the above stated improvements.

5.4.4 Manipulation of the knowledge base

The knowledge base is essential to the proposed algorithm since it is the groundwork for all improvements. Thus, the knowledge base should be manipulated effectively to supply necessary information. We store the knowledge base in disk so that *KISP* is

independent of the main memory size. The information about the candidate supports in *KB* helps to eliminate all database access. The candidate information also enables direct new-candidate generation and concurrent support counting. In addition, the knowledge base is incrementally increased as new support information is acquired. Therefore, the knowledge base should provide fast access to the counting information of patterns, carry quick estimation of required candidate storage, and be able to expand incrementally. Fig. 5-2 shows the logical structure of the knowledge base designed based on these requirements.

A knowledge base is composed of a **minimal *KB.sup***, and one or more ***KB* heads**. The minimal *KB.sup* is the smallest *KB.sup* among all the *KB.sups* in the *KB* heads. We create a *KB* head to store the newly acquired information only when the ‘new pattern’ mining part of *KISP* is executed (i.e. when the user-specified *minsup* is less than the minimal *KB.sup*). A *KB* head comprises a ***KB.sup***, the **number of pattern-support heads (*ps_heads*)**, the **pattern-support heads**, and the **position of next *KB* head**. The ***KB.sup*** indicates the *minsup* used while adding this head. The **number of *ps_heads*** indicates the total number of pattern-support heads in this *KB* head. The **pattern-support heads** summarizes the pattern-support tables, which contain the information of all patterns and their supports as described below. The **position of next *KB* head** links the next *KB* head so that the knowledge base can ‘grow’ incrementally.

The details of pattern information are collected in the **pattern-support tables** after mining. We group all the same sized patterns in the same table so that the pattern information of desired size can be directly found through the **position of pattern-support** in the corresponding *ps_head*. The *ps_head* also contains a summary of **the size of the patterns**, the total **number of counted candidates** (of that size), and the total **number of non-zero patterns**. The total number of counted candidates

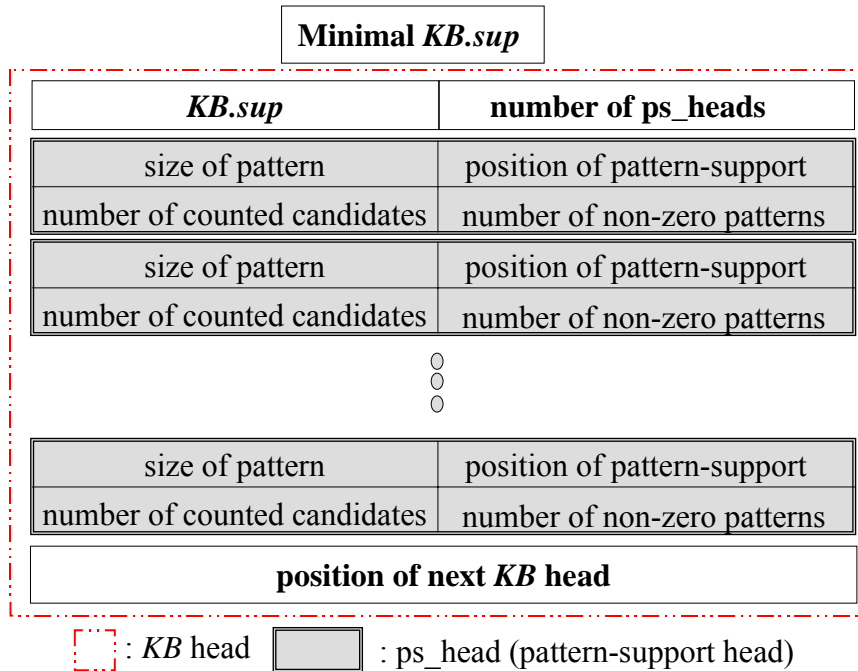


Fig. 5-2. Structure of the knowledge base

and the total number of non-zero patterns are used for estimating the number of new-candidates. During the interactive process, *KISP* can obtain effectively the full pattern information of certain size by accessing the **pattern-support table** (of that size) in every ***KB* head**. The **position of pattern-support**, in the **ps_head** within a ***KB* head**, indicates the location of the pattern-support table.

Fig. 5-3 shows the pattern-support table. Note that we keep only the patterns with non-zero support value to minimize the total size of each pattern-support table. The supports of patterns (of the same size) are stored in support-descending order in the structure. The descending ordered patterns eases the searching of valid patterns on answering an online query. Sorting pattern-supports before writing to the disk might increase the response time if the number of candidates is very huge. An option to eliminate support sorting is writing the supports in the order of hash-tree traversal. Even when the pattern supports are directly stored without sorting, searching within the knowledge base is still more efficient than re-mining.

support value of pattern	pattern
support value of pattern	pattern
.	.
.	.
.	.
support value of pattern	pattern

Fig. 5-3. Structure of a pattern-support table

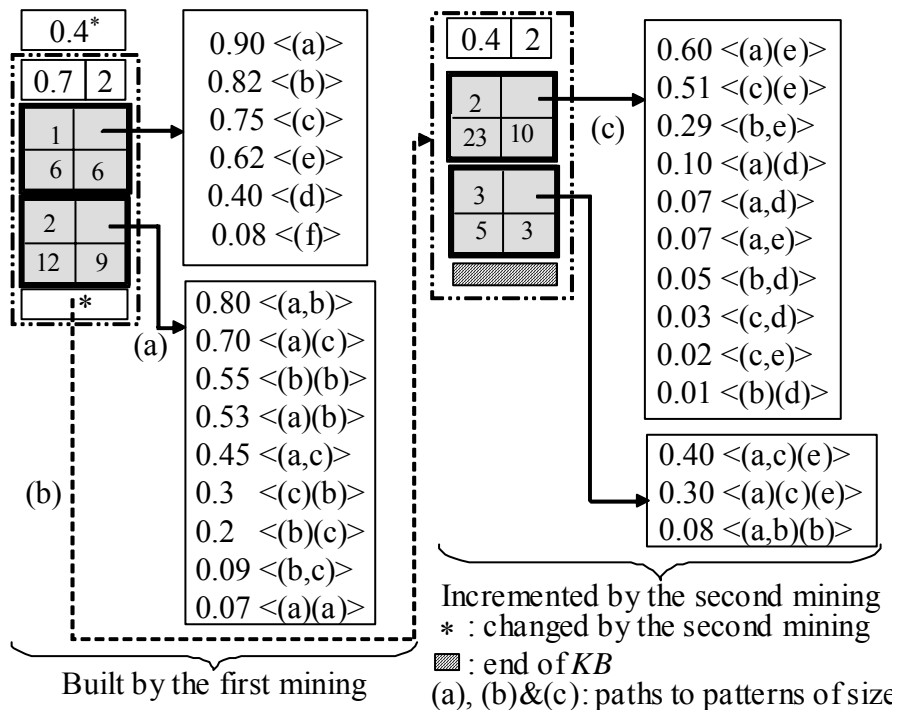


Fig. 5-4. The knowledge base after the second query in Example 5-1

Fig. 5-4 illustrates a sample knowledge base, showing the incrementally expanded support information after the second query in Example 5-1. In this figure, the **minimal KB.sup** and the **position of next KB head** are changed, and a new **KB head** is built by the second mining. The minimal **KB.sup** is changed from 0.7 to 0.4. The **position of next KB head** in the left-hand-side (built after the first mining) are changed to indicate the position of the new knowledge base (for **KB.sup** = 0.4). For instance, the supports of all size-2 patterns can be easily found by path (a), and path (b) then (c). The second **KB head** also shows that only 10 non-zero patterns out of the 23

counted size-2 candidates are stored after mining with $minsup = 0.4$.

5.4.5 Mining efficiency and space utilization with a large knowledge base

Given a very low $KB.sup$ value, one might concern that the space used by KB could be so large that $KISP$ might not sustain the high level of performance. Although KB may increase as a result of accumulating more pattern information, $KISP$ still could efficiently answer the interactive query request with new $minsup$. We analyze the overall performance affected when KB is getting very large below.

$KISP$ retrieves two kinds of data from KB , the KB heads and the stored patterns with associated supports (i.e. pattern-support tables). Relatively small space is required by a KB head for recording merely pattern summaries. Accessing these linked KB heads is so easy and there is no influence. The performance could be affected only by the reading of the pattern-support tables. However, the reading is confined to qualified patterns only, instead of every pattern, in the tables. $KISP$ may sustain the good performance by skipping a large number of unqualified patterns in KB , even if the KB is large.

The pattern-support tables are utilized to assist $KISP$ in either directly answering a query (when $KB.sup \leq minsup$) or generating the ‘new candidates’ by Theorem 5-1 in Section 5.4.2 (when $KB.sup > minsup$). In both circumstances, not every pattern needs to be scanned. Given a support-descending ordered table, when the first pattern whose support is smaller than $minsup$ is encountered, we stop reading the rest of the patterns in that table. Such an operation is also used in retrieving $S_{k-1}[KB.sup]$ for new-candidate generation. Thus, by sparing the reading of many unqualified patterns, $KISP$ may effectively retrieve the desired patterns and outperform the re-mining based approaches. In fact, $KISP$ would output all the valid patterns in constant time independent of the database size when $KB.sup \leq minsup$. Note that when patterns are

stored in the hash-tree traversal order initially, we may re-arrange the tables in support-descending order, periodically or after several *KB* heads are generated. Therefore, the overall performance affected due to a large *KB* is quite limited.

We now examine the space utilization of *KB*, which comprises *KB heads* and the pattern-support tables. When the requested new query with $KB.sup > minsup$ invokes new pattern generation in the interactive mining, one and only one *KB* head will be added to *KB*. Otherwise, *KB* stays intact because *KISP* simply responds by retrieving patterns from *KB*. The total number of *KB* heads hence is the total number of ‘new-pattern’ generation triggered. As described in Section 5.4.4, a *KB* head contains *KB.sup*, the position of next *KB* head, the number of *ps_heads*, and the *ps_heads*. A major portion of *KB* is the *ps_heads*, i.e. the pattern-support tables. The others need only negligible space. The size of a pattern-support table is proportional to the number of stored patterns where a pattern takes typically 19 to 22 bytes according to our experiments (details in Table 5-6, Section 5.5.1). The size of *KB*, as a consequence, might be larger than that of the original database. Appropriate compression on the pattern-support tables, being collections of the same sized patterns, could be employed to reduce the storage consumption for better storage utilization. Nevertheless, how compression would affect the performance needs further investigations.

5.5 Performance Evaluation

In order to assess the performance of the *KISP* algorithm, we conducted comprehensive experiments. All experiments were performed with an 866 MHz Pentium-III PC having 1024MB memory, running the Windows NT. In these experiments, the databases are composed of synthetic data. Please refer to Section 3.5.1 for the method used to generate these data. The performance of interactive

sequence discovery using the *KISP* and the *GSP* algorithms are compared in Section 5.5.1. Results of scale-up experiments are presented in Section 5.5.2. Table 5-5 lists the datasets used in the experiments. A dataset created with $|C| = \alpha$, $|T| = \beta$, $|S| = \chi$, and $|I| = \delta$ is denoted by the notation $C\alpha.T\beta.S\chi.I\delta$. For instance, the *Origin* dataset is denoted by $C10.T2.5.S4.I1.25$. The last four datasets are used for scalability tests in Section 5.5.2.

Table 5-5. Datasets used in the experiments

Name	$ DB $	$ C $	$ T $	$ S $	$ I $	N	N_S	N_I	Size (MB)
<i>Origin</i>	100K	10	2.5	4	1.25	10,000	2500	25,000	18.8
<i>NItem</i>	100K	10	2.5	4	1.25	5,000	2500	25,000	18.8
<i>SNpat</i>	100K	10	2.5	4	1.25	10,000	5000	25,000	18.8
<i>LNpat</i>	100K	10	2.5	4	1.25	10,000	2500	12,500	18.8
<i>Slen</i>	100K	20	2.5	4	1.25	10,000	2500	25,000	28.4
<i>Tlen</i>	100K	10	5	4	1.25	10,000	2500	25,000	28.0
<i>SPLen</i>	100K	10	2.5	8	1.25	10,000	2500	25,000	20.0
<i>LPLen</i>	100K	10	2.5	4	2.5	10,000	2500	25,000	18.5
<i>DB250k</i>	250K	10	2.5	4	1.25	10,000	2500	25,000	46.9
<i>DB500k</i>	500K	10	2.5	4	1.25	10,000	2500	25,000	94.0
<i>DB750k</i>	750K	10	2.5	4	1.25	10,000	2500	25,000	140.9
<i>DB1000k</i>	1000K	10	2.5	4	1.25	10,000	2500	25,000	187.9

5.5.1 Comparisons of *KISP* and *GSP*

Extensive experiments were performed to compare the execution times of *KISP* and *GSP*. The effect of using knowledge base **without concurrent support counting optimization** is studied first. The interactive discovery comprises five consecutive queries, with *minsup* values varying from 2.5% down to 0.5%.

Fig. 5-5 compares the relative performance of *KISP* and *GSP* on the *Origin* dataset with respect to various *minsup*s. The total number of candidates and the total number of database scanning required for each query in *GSP* are also shown in the bottom of Fig. 5-5. The number of passes is the same in *GSP* and in *KISP* without

concurrent support counting. The total execution time with *KISP* and *GSP* are 6652 and 8028 seconds, respectively. As to individual mining, *KISP* is faster than *GSP* for the last two queries with smaller *minsup* since considerable amount of candidates were eliminated. Fig. 5-5 also depicts the ratios of the number of candidates in *GSP* to those in *KISP*. Since the mining time reduced from the size-1 patterns in *KB* is very little in comparison with the pattern-outputting time increased, the overhead of *KISP* accounted for this phenomenon in the first three queries with larger *minsup*. In the first three queries, *KISP* runs slower than *GSP* due to the extra time spent for writing pattern information to *KB* being relatively larger than the time saved for the reduction in candidates. For instance, the mining stopped after pass two for the second query. Not much time was saved by the assistance of *KB* since the size-1 patterns occupied 77% of the reduced candidates.

Keeping the number of customers and the distribution of customer database the same, the series of queries were applied on the datasets *NItem*, *LNpat*, *SNpat*, *SPLen*, and *LPLen* to evaluate the impact of different sequence space for sampling. Similar results were obtained as shown in Fig. 5-6. The total execution time ratios of *KISP* to *GSP* are 67%, 74%, 97%, 89%, and 93%, respectively for the datasets *NItem*, *LNpat*, *SNpat*, *SPLen*, and *LPLen*. Due to the rush increase of qualified frequent 1-sequences which incurred the mass production of new candidates in the third query, the performance drops for *minsup* = 1% in Fig. 5-6. Note that for dataset *SNpat*, the sizes of the longest patterns are respectively 2, 2, 2, 3, and 5 for the five queries. Therefore, the reduction of total execution time is not apparent since the *KB* manifests much effect on candidate reduction only for the last two queries.

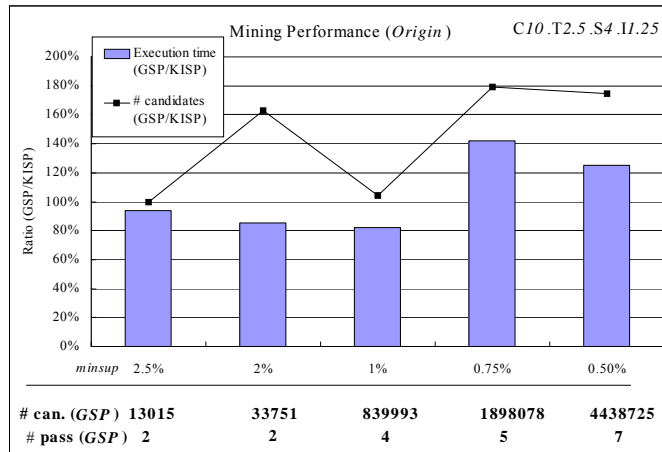


Fig. 5-5. Relative execution time and number of candidates on dataset *Origin*

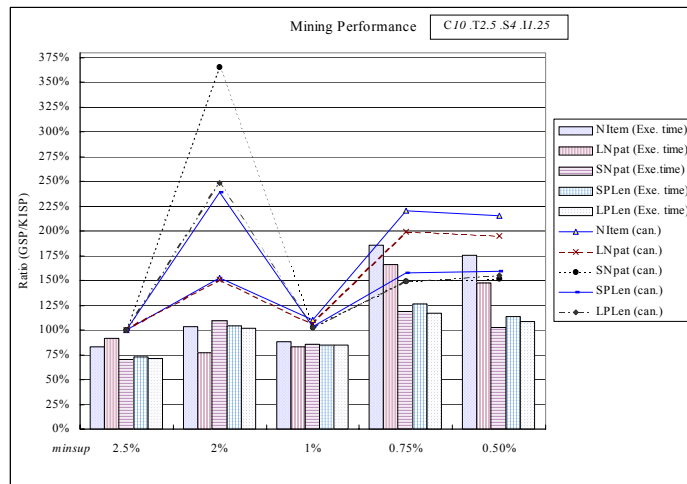


Fig. 5-6. Relative mining performance on datasets of various distributions

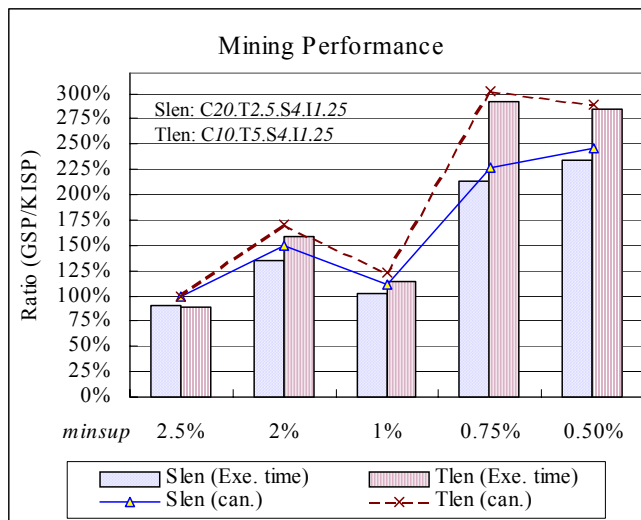


Fig. 5-7. Relative performance on datasets with longer customer sequences

Table 5-6. Number of candidates for the *Slen* dataset

Number of candidates			Pass number							
			1	2	3	4	5	6	7	8
<i>Minsup</i> value	2.5%	<i>GSP</i>	<i>GSP</i>	<i>GSP</i>	terminated					
		<i>KISP</i>	10000	78547	terminated					
	2%	<i>GSP</i>	10000	259376	1	terminated				
		<i>KISP</i>	0	180829	1	terminated				
	1%	<i>GSP</i>	10000	2534350	463	105	8	terminated		
		<i>KISP</i>	0	2274974	462	105	8	terminated		
	0.75%	<i>GSP</i>	10000	4550975	2045	413	80	6	terminated	
		<i>KISP</i>	0	2016625	1582	308	72	6	terminated	
	0.5%	<i>GSP</i>	10000	7673835	7986	2800	1339	430	63	3
		<i>KISP</i>	0	3122860	5941	2387	1259	424	63	3

Next, the distributions of customer sequences were changed. The *Slen* dataset increases the average sequence size of customers (from 10 to 20), and the *Tlen* dataset increases the average transaction size of customers (from 2.5 to 5). In general, both changes would allow the databases to have more (and longer) sequential patterns with respect to the above *minsup* values. As indicated in Fig. 5-7, *KISP* runs faster than *GSP* for each individual mining except for the very first mining. *KISP* benefits from the accumulated information so that the individual discovery could be accelerated. Take *minsup* = 0.75% for example, the execution time ratio of *GSP* to *KISP* is 2.9 times for dataset *Tlen*. The time saved by *KISP* resulted from the reduced number of candidates. In contrast, *GSP* generated three times the number of candidates. Additionally, the total execution time ratios of *KISP* to *GSP* are 54% for dataset *Slen*, and 48% for dataset *Tlen*. To illustrate the accumulating power of *KB*, the number of candidates in each pass generated by *GSP* and by *KISP* for the *Slen* dataset are enumerated in Table 5-6.

KISP exhibits excellent mining capability for query intensive applications, as demonstrated in Fig. 5-8. The average execution time (also the time required for posterior queries) decreases as the number of queries increased. That is, users might have shorter response time in each query by decreasing *minsup* value gradually to

reach the desirable *minsup* value, which generates the desired patterns. Similar results were obtained for the same series of queries applying on datasets *Slen* and *Tlen*.

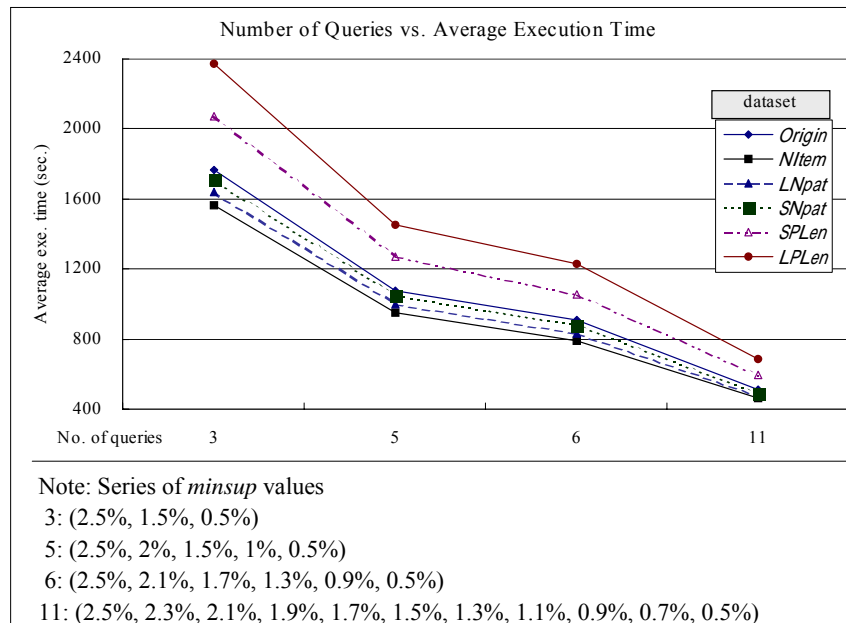


Fig. 5-8. Average execution time vs. number of queries

All the preceding experiments were performed **without optimization by concurrent support counting** so that the number of database passes is the same in *GSP* and in *KISP*. Table 5-7 illustrates the number of database scanning reduced by concurrent support counting, and the reduced execution times for all the datasets with respect to *minsup* = 0.5% and *KB.sup* = 0.75%. The first pass for support counting of candidate *l*-sequences is not required for all minings in *KISP* in comparison with *GSP*. In general, the number of size-2 candidates is so many that the concurrent optimization is effective from the second pass of database scanning (which counts candidates of size-3 and above). However, most scans were combined in pass two so that the total number of passes and the total execution times were reduced.

Table 5-7. Effects of concurrent support counting

<i>minsup</i> = 0.5%	<i>Origin</i>	<i>NItem</i>	<i>LNpat</i>	<i>SNpat</i>	<i>SPLen</i>	<i>LPLen</i>	<i>Slen</i>	<i>Tlen</i>
Reduced execution time (sec.)	29	39	40	4	8	5	94	157
Reduced number of passes	5	5	5	3	5	3	6	8

When users need to find the appropriate set of patterns by reducing the number of sequential patterns found in a query, the next specified *minsup* would be greater than the counting base of *KB* (*KB.sup*). *KISP* is faster than all the other re-mining based algorithms for such queries since the answer set is already in *KB*. In the next experiment, all *KB.sup*s of the *KB*s were 0.5%, and 100 *minsup*s ranging from 0.5% to 2.5% were randomly selected. As shown in Table 5-8, the mining results are all available in very short time for all datasets. For most queries, the execution time of *KISP* is several orders of magnitude faster than *GSP*, which always re-mines from scratch.

However, one drawback of *KISP* is that the size of *KB* might be larger than the size of the original database, due to the space increased for storing supports. The size of *KB* is proportional to the number of patterns existing in *KB*. The maximum sizes of *KB* are also shown in Table 5-8. Table 5-9 shows that, in worst case, *KB* might need as much as five times the space of the sequence database for low *KB.sup*

Table 5-8. Execution time of *KISP* when $KB.sup \leq minsup$

Exe. Time (sec.)	Origin	Slen	Tlen	SPLen	LPLe
Minimum	0	4	10	0	0
Maximum	22	29	13	14	16
Average	4.3	11.8	10.8	5.1	4.4

Table 5-9. Space used by *KB* with respect to *KB.sup* (dataset *Slen*)

<i>KB.sup</i>	2%	1%	0.5%
Worst case size of KB (MByte)	5.6	51.7	140.9
Number of patterns stored	269377	2544926	7696456
Average cost of a pattern (Byte)	21.9	21.3	19.2

5.5.2 Scale-up experiments

To assess the scalability of the proposed algorithm, several experiments were conducted. Since the basic construct of *KISP* is similar to that of *GSP*, similar scalable

results could be expected. In the scale-up experiments, the total number of customers was increased from 100K to 1000K and other parameters were the same as the *Origin* dataset. Again, *KISP* were faster than *GSP* for all the datasets. The execution times were normalized with respect to the time for 100,000 customers here. Fig. 5-9 shows that the execution time of *KISP* increases linearly as the database size increases, which demonstrates good scalability of *KISP*.

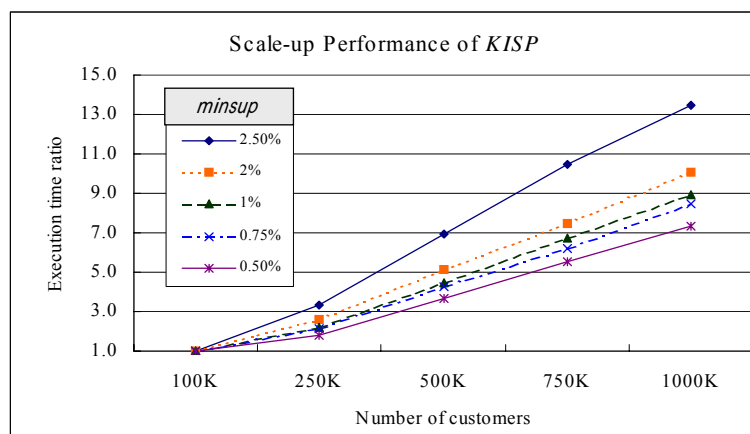


Fig. 5-9. Linear scalability of the database size

5.6 Summary

In this chapter, we propose an efficient knowledge base assisted mining algorithm for interactive discovery of sequential patterns. For online queries, manual tuning on mining parameters such as the minimum support is inevitable since no one can predict the best parameter and the corresponding outcome. A result driven discovery requires many times of repeated mining in an interactive process. A fast mining algorithm that always re-mines from scratch is not good enough for interactive query in practice. Knowledge obtained from each mining should be utilized to accelerate the entire process.

The proposed *KISP* algorithm constructs a knowledge base for minimizing the total response time for online queries. Neither database access nor counting is

required if the query result is a subset of patterns in the knowledge base. In case some resultant patterns are new to the knowledge base, we speed up the mining process by the assistance of the knowledge base. The proposed approach directly generates only the new candidates which are not counted before, concurrently counts variable sized candidates in the same database scanning, and incrementally expand the knowledge base by every counting effort for future queries. The knowledge base keeps the patterns grouped by the size to provide fast access to pattern information. The experiments performed shows that the proposed approach is faster than *GSP* by several orders of magnitude, with good linear scalability.



Chapter 6 Algorithm IncSP for Incremental Discovery of Sequential Patterns

6.1 Overview

Sequential pattern discovery, which finds frequent temporal patterns in databases, is an important issue in data mining originated from retailing databases with broad applications [6, 29, 64, 75, 80, 99]. The discovery problem is difficult considering the numerous combinations of potential sequences, not to mention the re-mining required when databases are updated or changed. Therefore, it is essential to investigate efficient algorithms for sequential pattern mining and effective approaches for sequential pattern updating.

A sequential pattern is a relatively frequent sequence of transactions, where each transaction is a set of items (called *itemset*). For example, one might purchase a PC and then purchase a printer later. After some time, he or she could possibly buy some printing software and a scanner. If there exists a sufficient number of customers in the transactional database who have the purchasing sequence of PC, printer, printing software and scanner, then such a frequent sequence is a sequential pattern. In general, each customer record in the transactional database is an *itemset* associated with the transaction *time* and a *customer-id* [6]. Records having the same customer-id are sorted by ascending transaction time into a *data sequence* before mining. The objective of the discovery is to find out all sequential patterns from these data sequences.

A *sequential pattern* is a sequence having *support* greater than or equal to a minimum threshold, called the *minimum support*. The *support* of a sequence is the

percentage of data sequences containing the sequence. Note that the support calculation is different in the mining of association rules [5, 18, 58] and sequential patterns [6, 80]. The former is transaction-based, while the latter is sequence-based. Suppose that a customer has two transactions buying the same item. In association discovery, the customer “contributes” to the support count of that item by two, whereas it counts only once in the support counting in sequential pattern mining.

The discovery of sequential patterns is more difficult than association discovery because the patterns are formed not only by combinations of items but also by permutations of itemsets. For example, given 50 possible items in a sequence database, the number of potential patterns is $50*50+C(50,2)$ regarding two items, and $50*50*50 + 50*C(50,2)*2 + C(50,3)$ regarding three items (formed by 1-1-1, 1-2, 2-1, and 3), ..., etc. Most current approaches assume that the sequence database is static and focus on speeding up the time-consuming mining process. In practice, databases are not static and are usually appended with new data sequences, conducted by either existing or new customers. The appending might invalidate some existing patterns whose supports become insufficient with respect to the currently updated database, or might create some new patterns due to the increased supports. Hence, we need an effective approach for keeping patterns up-to-dated.

However, not much work has been done on the maintenance of sequential patterns in large databases. Many algorithms deal with the mining of association rules [5, 58], the mining of sequential patterns [6, 29, 67, 80, 93, 99], and parallel mining of sequential patterns [75]. Some algorithms discover frequent episodes in a single long sequence [46]. Nevertheless, when there are changes in the database, all these approaches have to re-mine the whole updated database. The re-mining demands more time than the previous mining process since the appending increases the size of the database.

Although there are some incremental techniques for updating association rules [18, 19, 40, 87], few research has been done on the updating of sequential patterns, which is quite different. Association discovery is transaction-based; thus, none of the new transactions appended is related to the old transactions in the original database. Sequential pattern mining is sequence-based; thus, the two data sequences, one in the newly appended database and the other in the original database, must be merged into a data sequence if their customer-ids are the same. However, the *sequence merging* will corrupt previous support count information so that either FUP or FUP2 [19] algorithm could not be directly extended for the maintenance of sequential patterns.

One work dealing with incremental sequence mining for vertical database is the *ISM (Incremental Sequence Mining)* algorithm [64]. Sequence databases of vertical layout comprise a list of $(cid, timestamp)$ pairs for each of all the items. In order to update the supports and enumerate frequent sequences, *ISM* maintains “maximally frequent sequences” and “minimally infrequent sequences” (called *negative border*). However, the problem with *ISM* is that the size of negative border (i.e. the number of potentially frequent sequences) might be too large to be processed in memory. Besides, the size of extra space for transformed vertical lists might be several times the size of the original sequence database.

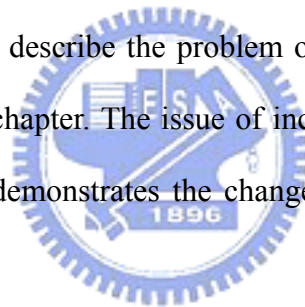
This chapter presents an efficient incremental updating algorithm for up-to-date maintenance of sequential patterns after a nontrivial number of data sequences are appended to the sequence database. Assume that the minimum support keeps the same. Instead of re-mining the whole database for pattern discovery, the proposed algorithm utilizes the knowledge of previously computed frequent sequences. We merge data sequences implicitly, generate fewer but more promising candidates, and separately count supports with respect to the original database and the newly appended database. The supports of old patterns are updated by merging new data sequences implicitly

into the original database. Since the data sequences of old customers are processed already, efficient counting over the data sequences of new customers further optimizes the pattern updating process.

The rest of the chapter is organized as follows. Section 6.2 describes the problem of sequential pattern mining and addresses the issue of incremental update. In Section 6.3, we review some previous algorithms of sequence mining. Section 6.4 presents our proposed approach for the updating of sequential patterns after databases are changed. Comparative results of the experiments by comprehensive synthetic data sets are depicted in Section 6.5. Section 6.6 concludes this chapter.

6.2 Problem Statement

In Section 6.2.1, we formally describe the problem of sequential pattern mining and the terminology used in this chapter. The issue of incremental update is presented in Section 6.2.2. Section 6.2.3 demonstrates the changes of sequential patterns due to database update.



6.2.1 Sequential pattern mining

A *sequence* s , denoted by $\langle e_1 e_2 \dots e_n \rangle$, is an ordered set of n elements where each *element* e_i is an itemset. An *itemset*, denoted by (x_1, x_2, \dots, x_q) , is a nonempty set of q items, where each *item* x_j is represented by a literal. Without loss of generality, we assume the items in an element are in lexicographic order. The *size* of sequence s , written as $|s|$, is the total number of items in all the elements in s . Sequence s is a k -*sequence* if $|s| = k$. For example, $\langle (e)(b)(a) \rangle$, $\langle (a,b)(a) \rangle$, and $\langle (c)(e,f) \rangle$ are all 3-sequences. A sequence $s = \langle e_1 e_2 \dots e_n \rangle$ is a *subsequence* of another sequence $s' = \langle e'_1 e'_2 \dots e'_m \rangle$ if there exist $1 \leq i_1 < i_2 < \dots < i_n \leq m$ such that $e_1 \subseteq e_{i_1}'$, $e_2 \subseteq e_{i_2}'$, ..., and $e_n \subseteq e_{i_n}'$. Sequence s' *contains* sequence s if s is a subsequence of s' . For example,

$\langle (b)(a,e) \rangle$ is a subsequence of $\langle (b,d)(c)(a,c,e) \rangle$.

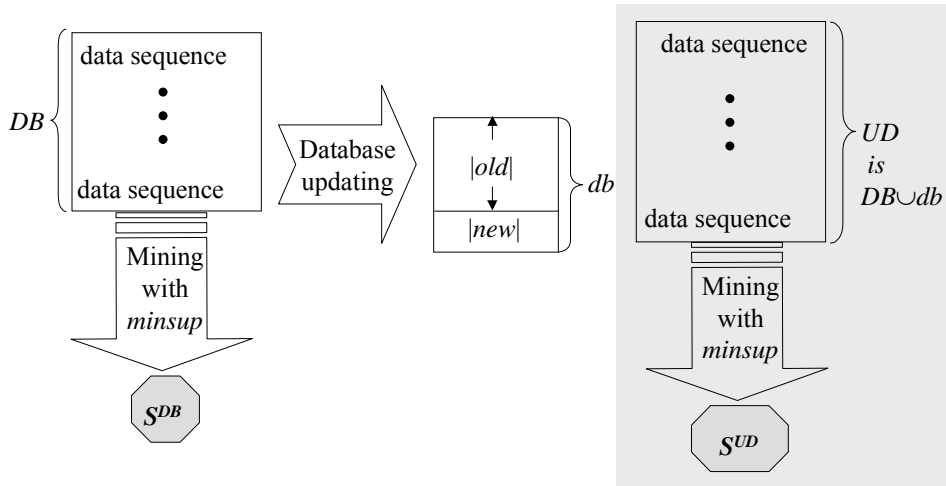
Each sequence in the sequence database DB is referred to as a *data sequence*. Each data sequence is associated with a customer-id (abbreviated as *cid*). The number of data sequences in DB is denoted by $|DB|$. The *support* of sequence s , denoted by $s.sup$, is the number of data sequences containing s divided by the total number of data sequences in DB . The *minsup* is the user specified minimum support threshold. A sequence s is a *frequent sequence*, or called *sequential pattern*, if $s.sup \geq minsup$. Given the *minsup* and the sequence database DB , the problem of sequential pattern mining is to discover *the set of all sequential patterns*, denoted by S^{DB} .

6.2.2 Incremental update of sequential patterns

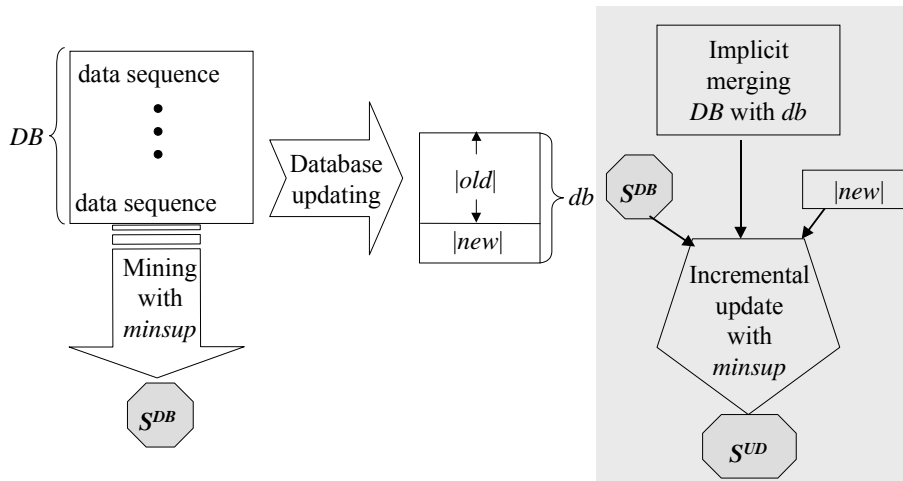
In practice, the sequence database will be updated with new transactions after the pattern mining process. Possible updating includes transaction appending, deletions, and modifications. With respect to the same *minsup*, the incremental update problem aims to find out the new set of all sequential patterns after database updating without re-mining the whole database. First, we describe the issue of incremental updating by taking the transaction appending as an illustrating example. Transaction modification can be accomplished by transaction deletion and appending.

The **original database** DB is appended with a few data sequences after some time. The **increment database** db is referred to as the set of these newly appended data sequences. The *cids* of the data sequences in db may already exist in DB . The whole database combining all the data sequences from the original database DB and the increment database db is referred to as the **updated database** UD . Let the *support count* of a sequence s in DB be s_{count}^{DB} . A sequence s is a frequent sequence in UD if

$s_{count}^{UD} \geq minsup \times |UD|$, where s_{count}^{UD} is the support count of s in UD . Although UD is



(a) Obtain S^{UD} by re-executing mining algorithm on UD



(b) Obtain S^{UD} by incremental updating with S^{DB}

Fig. 6-1. Incremental update versus re-mining

the union of DB and db , $|UD|$ is not necessarily equal to $|DB|$ plus $|db|$. If there are $|old|$ $cids$ appearing both in DB and db , then the number of 'new' customers is $|new| = |db| - |old|$. Thus $|UD| = |DB| + |db| - |old|$ due to sequence merging. When all $cids$ in db are different from those in DB , $|old|$ (the number of 'old' customers) is zero. On the contrary, $|old|$ equals $|db|$ in case all $cids$ in db exist in DB . Let s_{count}^{db} be the increase in support count of sequence s due to db . Whether sequence s in UD is frequent or not depends on s_{count}^{UD} with respect to the same $minsup$ and $|UD|$.

Most approaches re-execute mining algorithms over all data sequences in UD to

Table 6-1. Notations used

x_1, x_2, \dots, x_q	Items.
(x_1, x_2, \dots, x_q)	A q -itemset, each x_i is an item.
$s = \langle e_1 e_2 \dots e_n \rangle$	A sequence with n element.
e_1, e_2, \dots, e_n	Elements (of a sequence). Each e_i is an itemset.
$minsup$	The minimum support specified by the user.
UD	The updated database.
DB	The original database.
db	The increment database.
$ UD , DB , db $	The total number of data sequences in UD , DB , and db respectively.
$ old $	The total number of data sequences of ‘old’ customers in db .
$ new $	The total number of data sequences of ‘new’ customers in db .
S_{DB}, S_{UD}	The set of all sequential patterns in DB and UD respectively.
$s_{count}^{DB}, s_{count}^{UD}$	The support counts of candidate sequence s in DB and UD respectively.
s_{count}^{db}	The increase in support count of candidate sequence s due to db .
S_k	The set of all frequent k -sequences, see Section 6.3.1.
X_k	The set of all candidate k -sequences, see Section 6.3.1.
X_k'	The reduced set of candidate k -sequences, see Section 6.4.
S_k^{DB}	The set of frequent k -sequences in DB , see Section 6.4.2.
$X_k(DB)$	The set of candidates in X_k that are also in S_k^{DB} , see Section 6.4.
$X_k(DB)'$	$X_k(DB)' = X_k - X_k(DB)$, see Section 6.4.
$ds_{UD}, ds_{DB}, ds_{db}$	A data sequence in UD , DB , and db respectively, see Section 6.4.1.
$ds_{DB} \cup ds_{db}$	An implicitly merged data sequence, see Section 6.4.1.
UD_{DB}	Data sequences in UD whose cids appearing in DB only, see Section 6.4.4.
UD_{db}	Data sequences in UD whose cids appearing in db only, see Section 6.4.1.
UD_{Dd}	Data sequences in UD whose cids are in both DB and db , see Section 6.4.1.

obtain s_{count}^{UD} and discover S^{UD} , as shown in Fig. 6-1(a). However, we can effectively calculate s_{count}^{UD} utilizing the support count of each sequential pattern s in S^{DB} . Fig. 6-1(b) shows that we discover S^{UD} through incremental update on S^{DB} after implicit

merging. Table 6-1 summarizes the notations used in this chapter.

6.2.3 Changes of sequential patterns due to database update

Consider an example database DB with 6 data sequences as shown in Fig. 6-2. Assume that $minsup = 33\%$, i.e., minimum support count being 2. The sequential patterns in DB are $\langle(a)\rangle$, $\langle(b)\rangle$, $\langle(c)\rangle$, $\langle(d)\rangle$, $\langle(a,b)\rangle$, $\langle(a)(d)\rangle$, $\langle(b)(b)\rangle$, and $\langle(c)(a)\rangle$. Note that $\langle(f)\rangle$, though appeared twice in the same data sequence C6, is not frequent because its support count is one.

	Data Sequence (ds^{DB})
C1	$\langle(a)(d)\rangle$
C2	$\langle(b)(c,e)(a,b)\rangle$
C3	$\langle(a,b)(b,d)\rangle$
C4	$\langle(d)(c)(a)\rangle$
C5	$\langle(a)\rangle$
C6	$\langle(g)(b,f,g)\rangle$

Fig. 6-2. The original database DB example, $|DB| = 6$

Fig. 6-3(a) shows the data sequences in the increment database db after some updates from new customers only. The updated database UD is shown in Fig. 6-3(b). Corresponding to the nine data sequences and with the same $minsup$, the support count of a frequent sequence must be three or larger. The support counts of previous sequential patterns $\langle(c)\rangle$, $\langle(a)(d)\rangle$, and $\langle(c)(a)\rangle$ are less than three, and are no longer frequent due to the database updates. While $\langle(e)\rangle$, $\langle(b)(e)\rangle$, and $\langle(b,d)\rangle$ become new patterns because they have minimum supports now.

In the cases of updates when the new sequences are from old customers, i.e., the $cids$ of the new sequences appear in the original database. These data sequences must be appended to the old data sequences of the same customers in DB . Assume that two customers, $cid=C4$ and $cid=C8$, bought item 'h' afterward. The data sequences for $cid=C4$ and $cid=C8$ now become $\langle(d)(c)(a)(\underline{h})\rangle$ and $\langle(b,d)(e)(\underline{h})\rangle$, respectively. Fig. 6-4 shows the example of an increment database having data sequences from both old

and new customers. In this example, $|old| = 4$, $|new| = 3$, and $|db| = 7$ where records in shadow are old customers. Fig. 6-5 presents the resulting data sequences in UD . After invalidating the patterns $\langle(e)\rangle$, $\langle(b)(b)\rangle$, $\langle(b)(e)\rangle$, and $\langle(a,b)\rangle$, the up-to-date sequential patterns are $\langle(a)\rangle$, $\langle(b)\rangle$, $\langle(d)\rangle$, $\langle(f)\rangle$, $\langle(b,d)\rangle$, $\langle(b,f)\rangle$ and $\langle(a)(d)\rangle$, for the given $minsup$ 33%.

Cid	Data Sequence (ds^{db})
C7	$\langle(b,d)\rangle$
C8	$\langle(b,d)(e)\rangle$
C9	$\langle(a,b)(e)(b,f)\rangle$

Cid	Data Sequence (ds^{UD})
C1	$\langle(a)(d)\rangle$
C2	$\langle(b)(c,e)(a,b)\rangle$
C3	$\langle(a,b)(b,d)\rangle$
C4	$\langle(d)(c)(a)\rangle$
C5	$\langle(a)\rangle$
C6	$\langle(f)(b,f,g)\rangle$
C7	$\langle(b,d)\rangle$
C8	$\langle(b,d)(e)\rangle$
C9	$\langle(a,b)(e)(b,f)\rangle$

(a) new customers only, $|db|=3$. (b) the updated database, $|UD|=9$.

Fig. 6-3. Data sequences in the increment database and the updated database (a) db with new customers only (b) the updated database UD

6.3 Related Work

In Section 6.3.1, we review some algorithms for discovering sequential patterns. Section 6.3.2 presents related approaches for incremental pattern updating.

6.3.1 Algorithms for discovering sequential patterns

The *Apriori* algorithm discovers association rules [5], while the *AprioriAll* algorithm deals with the problem of sequential pattern mining [6]. *AprioriAll* splits sequential pattern mining into three phases, itemset phase, transformation phase, and sequence phase. The itemset phase uses *Apriori* to find all frequent itemsets. The database is transformed, with each transaction being replaced by the set of all frequent itemsets contained in the transaction, in the transformation phase. In the third phase, *AprioriAll*

makes multiple passes over the database to generate candidates and to count the supports of candidates. In subsequent work, the same authors proposed the *GSP* (Generalized Sequential Pattern) algorithm that outperforms *AprioriAll* [80]. Both algorithms use the similar techniques for candidate generation and support counting, as described in the following.

Cid	Data Sequence (ds^{db})
C2	<(d)>
C4	<(h)>
C5	<(a,d)>
C8	<(h)>
C10	<(b,d,f,h)>
C11	<(a)(g)>
C12	<(b,f)(g)>

Fig. 6-4. Data sequences of old and new customers in *db*

Cid	Data Sequence (ds^{UD})
C1	<(a)(d)>
C2	<(b)(c,e)(a,b)(d)>
C3	<(a,b)(b,d)>
C4	<(d)(c)(a)(h)>
C5	<(a)(a,d)>
C6	<(f)(b,f,g)>
C7	<(b,d)>
C8	<(b,d)(e)(h)>
C9	<(a,b)(e)(b,f)>
C10	<(b,d,f,h)>
C11	<(a)(g)>
C12	<(b,f)(g)>

Fig. 6-5. Merged data sequences in the updated database *UD*

The *GSP* algorithm makes multiple passes over the database and finds out frequent *k*-sequences at *k*-th database scanning. In each pass, every data sequence is examined to update the support counts of the candidates contained in this sequence. Initially, each item is a candidate *1*-sequence for the first pass. Frequent *1*-sequences are determined after checking all the data sequences in the database. In succeeding passes, frequent (*k*-1)-sequences are self-joined to generate candidate *k*-sequences. Again, the supports of these candidate sequences are counted by examining all data

sequences, and then those candidates having minimum supports become frequent sequences. This process terminates when there is no candidate sequence any more. In the following, we further depict two essential sub-processes in *GSP*, the candidate generation and the support counting.

Candidate generation: Let S_k denote the set of all frequent k -sequences, and X_k denote the set of all candidate k -sequences. *GSP* generates X_k by two steps. The first step joins S_{k-1} with S_{k-1} and obtains a superset of the final X_k . Those candidates in the superset having any $(k-1)$ -subsequence which is not in S_{k-1} are deleted in the second step. In the first step, a $(k-1)$ -sequence $s1 = \langle e_1 e_2 \dots e_{n-1} e_n \rangle$ is joined with another $(k-1)$ -sequence $s2 = \langle e_1' e_2' \dots e_n' \rangle$ if $\overline{s1} = \overline{s2}$, where $\overline{s1}$ is the $(k-2)$ -sequence of $s1$ dropping the first item of e_1 and $\overline{s2}$ is the $(k-2)$ -sequence of $s2$ dropping the last item of e_n' . The generated candidate k -sequence $s3$ is $\langle e_1 e_2 \dots e_{n-1} e_n e_n' \rangle$ if e_n' is a 1-itemset. Otherwise, $s3$ is $\langle e_1 e_2 \dots e_{n-1} e_n' \rangle$. For example, the candidate 5-sequence $\langle (a,b)(c,e)(f) \rangle$ is generated by joining $\langle (a,b)(c,e) \rangle$ with $\langle (b)(c,e)(f) \rangle$, and the candidate $\langle (a,b)(c,e,f) \rangle$ is generated by joining $\langle (a,b)(c,e) \rangle$ with $\langle (b)(c,e,f) \rangle$. In addition, the X_k produced from this procedure is a superset of S_k as proved in [80]. That is, $X_k \supseteq S_k$.

Support counting: *GSP* adopts a hash-tree structure [5, 80] for storing candidates to reduce the number of candidates that need to be checked for each data sequence. Candidates would be placed in the same leaf if their leading items, starting from the first item, were hashed to the same node. The next item is used for hashing when an interior node, instead of a leaf node, is reached [80]. The candidates required for checking against a data sequence are located in leaves reached by applying the hashing procedure on each item of the data sequence [80]. The support of the candidate is incremented by one if it is contained in the data sequence.

In addition, the *SPADE* (Sequential **P**attern **D**iscovery using **E**quivalence classes)

algorithm finds out sequential patterns using vertical database layout and join-operations [99]. Vertical database layout transforms customer sequences into items' id-lists. The id-list of an item is a list of $(cid, timestamp)$ pairs indicating the occurring timestamps of the item in that customer-id. The list pairs are joined to form a sequence lattice, in which *SPADE* searches and discovers the patterns [99].

Recently, the *FreeSpan* (**F**requent pattern-projected **S**equential **P**attern Mining) algorithm was proposed to mine sequential patterns by a database projection technique [29]. *FreeSpan* first finds the frequent items after scanning the database once. The sequence database is then projected, according to the frequent items, into several smaller intermediate databases. Finally, all sequential patterns are found by recursively growing subsequence fragments in each database. Based on the similar projection technique, the authors proposed the *PrefixSpan* (**P**refix-projected **S**equential **p**attern mining) algorithm [67].

Nevertheless, all these algorithms have to re-mine the database after the database is appended with new data sequences. Next, we introduce some approaches for updating patterns without re-mining.

6.3.2 Approaches for incremental pattern updating

A work for incremental sequential pattern updating was proposed in [90]. The approach uses a dynamic suffix tree structure for incremental mining in a single long sequence. However, the focus of research here is on multiple sequences of itemsets, instead of a single long sequence of items.

Based on the *SPADE* algorithm, the *ISM* (**I**ncremental **S**equence **M**ining) algorithm was proposed for incremental sequence mining [64]. An *Increment Sequence Lattice* consisting of both frequent sequences and the nearly frequent ones (called *negative border*) is built to prune the search space for potential new patterns.

However, the *ISM* might encounter memory problem if the number of the potentially frequent patterns is too large [64]. Besides, computation is required to transform the sequence database into vertical layout, which also requires additional storage several times the original database.

In order to avoid re-mining from scratch with respect to database updates with both old and new customers, we propose a pattern updating approach that incrementally mines sequential patterns by utilizing the discovered knowledge. Section 6.4 gives the details of the proposed algorithm.

6.4 The Proposed Algorithm

In sequence mining, frequent patterns are those candidates whose supports are greater than or equal to the minimum support. In order to obtain the supports, every data sequence in the database is examined, and the support of each candidate contained in that data sequence is incremented by one. For pattern updating after database update, the database *DB* was already mined and the supports of the frequent patterns with respect to *DB* are known. Intuitively, the number of data sequences need to be examined in current updating with database *UD* seems to be $|UD|$. However, we can utilize the prior knowledge to improve the overall updating efficiency. Therefore, we propose the *IncSP* (**I**ncremental **S**equential **P**attern **U**pate) algorithm to speed up the incremental updating problem. Fig. 6-6 depicts the architecture of a single pass in the *IncSP* algorithm. In brief, *IncSP* incrementally updates and discovers the sequential patterns through effective implicit merging, early candidate pruning, and efficient separate counting.

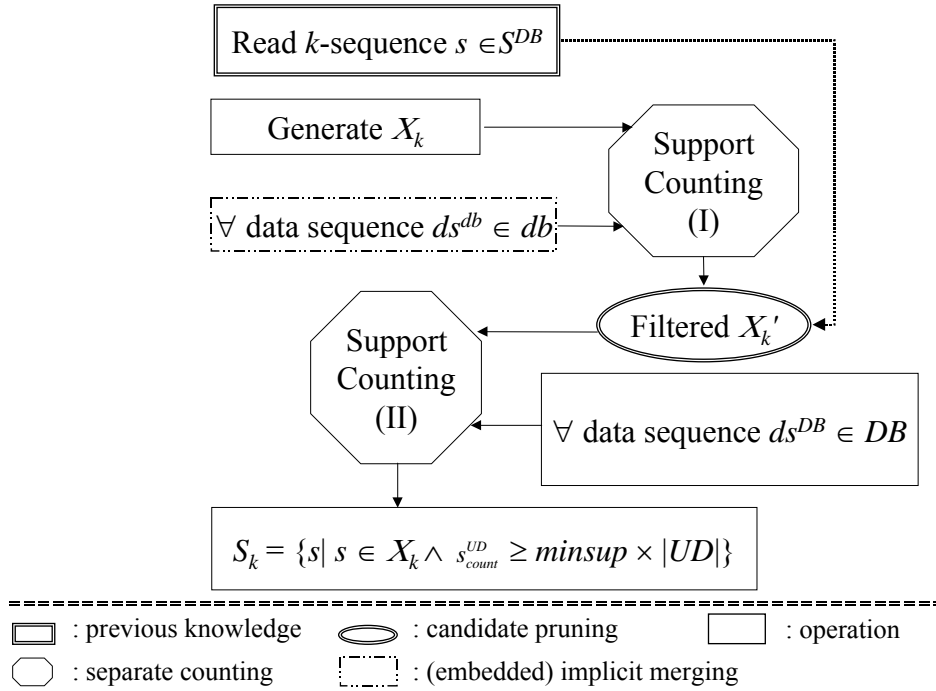


Fig. 6-6. The architecture of the k -th pass in *IncSP*

The data sequence of a customer in DB and the sequence with same cid in db must be merged into the customer's data sequence in UD . If all such sequences are merged explicitly, we have to re-mine and re-count the supports of the candidates contained in the resultant customer sequences from scratch. Hence, *IncSP* deals with the required sequence merging implicitly for incremental pattern updating, which is described in Section 6.4.1.

IncSP further speeds up the support counting by partitioning the candidates into two sets. The candidates with respect to DB which were also frequent patterns before updating are placed into set $X_{\mathcal{K}(DB)}$, and the remaining candidates are placed into set $X_{\mathcal{K}(DB)'}.$ After the partitioning, the supports of the candidates in $X_{\mathcal{K}(DB)}$ can be incremented and updated simply by scanning over the increment database db . During the same scanning, we also calculate the increment supports of the candidates in $X_{\mathcal{K}(DB)'}.$ with respect to db . Since the supports of the candidates in $X_{\mathcal{K}(DB)'}.$ are not available (only the supports of frequent patterns in DB are kept in prior mining over DB), we need to compute their supports against the data sequences in DB . The

number of candidates need to be checked is reduced to the size of set $X_{\mathcal{K}(DB)'}'$ instead of the full set X_k . Thus, *IncSP* divides the counting procedure into separate processes to efficiently count the supports of candidates with respect to DB and db . We show that the support of a candidate is the sum of the two support counts after the two counting processes in Lemma 6-1 (in Section 6.4.2).

Moreover, some candidates in $X_{\mathcal{K}(DB)'}'$ can be pruned earlier before the actual counting over the data sequences in DB . By partitioning the set of candidates into $X_{\mathcal{K}(DB)}$ and $X_{\mathcal{K}(DB)'}'$, we know that all the candidates in $X_{\mathcal{K}(DB)'}'$ are not frequent patterns with respect to DB . If the support of a candidate in $X_{\mathcal{K}(DB)'}'$ with respect to db is smaller than the proportion $minsup \times (|UD| - |DB|)$, the candidate cannot possibly become a frequent pattern in UD . Such unqualifying candidates are pruned and only the more promising candidates go through the actual support counting over DB . Lemma 6-2 (in Section 6.4.2) shows this property. This early pruning further reduces the number of candidates required to be counted against the data sequences in DB . The reduced set of candidates is referred to as $X_{\mathcal{K}'}$.

In essence, *IncSP* generates candidates and examines data sequences to determine frequent patterns in multiple passes. As shown in Fig. 6-6, *IncSP* reduces the size of X_k into $X_{\mathcal{K}'}$ and updates the supports of patterns in S^{DB} by simply checking the increment database db , which is usually smaller than the original database DB . In addition, the *separate counting* technique enables *IncSP* to accumulate candidates' supports quickly because only the *new* candidates, whose supports are unavailable from S^{DB} , need to be checked against DB . The complete *IncSP* algorithm and the separate counting are described in Section 6.4.2. Section 6.4.3 further illustrates other updating operations such as modifications and deletions. In Section 6.4.4, we provide the proof of lemmas used in Section 6.4.

6.4.1 Implicit merging of data sequences with same cids

For the discovery of sequential patterns, transactions coming from the same customer, either in DB or in db , are parts of the unique data sequence corresponding to that customer in UD . Given a customer having one data sequence in DB and another sequence in db , the proper data sequence for the customer (in UD) is the merged sequence of the two. Since the transaction times in db are later than those in DB , the merging appends the data sequences in db to the sequences in DB . Nevertheless, such “explicit merging” might invalidate S^{DB} because the data sequence of the customer becomes a longer sequence. Some patterns in S^{DB} , which are not contained in the data sequence before merging, might become contained in the now longer data sequence so that the support counts of these patterns become larger. In order to effectively keep the patterns in S^{DB} up-to-date, *IncSP* implicitly merges data sequences of the same customers and delays the actual action of merging until pattern updating completes.

Assume that an explicit merging must merge ds^{DB} with ds^{db} into ds^{UD} , where ds^{DB} , ds^{db} , and ds^{UD} represent the data sequences in DB , db , and UD respectively. In each pass, the mining process needs to count the supports of candidate sequences against ds^{UD} . The “implicit merging” in *IncSP* employs ds^{DB} and ds^{db} as if ds^{UD} is produced during mining process. We will describe how “implicit merging” updates the supports of sequential patterns in S^{DB} , and how “implicit merging” counts the supports of candidates contained in the implicitly merged data sequence, represented by $ds^{DB} \cup ds^{db}$.

The “implicit merging” updates the supports of sequential patterns in S^{DB} according to ds^{DB} and ds^{db} . This updating involves only the newly generated (candidate) k -sequences in the k -th pass, which are contained in ds^{UD} but not in ds^{DB} , since ds^{DB} had already engaged in the discovery of S^{DB} . We refer to these candidate

k -sequences as the *new k-sequences*. As indicated in Fig. 6-6, when ds^{db} is checked in Support Counting (I), only the supports of such *new k-sequences* must be counted. If this *new k-sequence* is also a sequential pattern in S^{DB} , we update the support count of the sequence in S^{DB} . Otherwise, supports of *new k-sequences* which are not in S^{DB} , being initialized to zero before counting, are incremented by one for this data sequence ($ds^{DB} \cup ds^{db}$). In this way, *IncSP* correctly maintains S^{DB} with the *new k-sequences* and counts supports with respect to ds^{db} during Support Counting (I).

Example 1: Implicit merging for support updating in pass-1. Take customers in Fig. 6-5 for example, the *DB* is shown in Fig. 6-3(b) and the *db* is shown in Fig. 6-4. The customer with $cid=C2$ has the two sequences, $ds^{DB} = \langle (b)(c,e)(a,b) \rangle$ and $ds^{db} = \langle (d) \rangle$. During pass 1, $\langle (d) \rangle_{count}^{DB}$ is increased by one due to the implicit merging with ds^{db} and ds^{DB} (of C2). Note that implicit merging for the customer with $cid=C5$ whose $ds^{DB} = \langle (a) \rangle$ and $ds^{db} = \langle (a,d) \rangle$ contains only the *new 1-sequence* $\langle (d) \rangle$ because $\langle (a) \rangle$ was already counted when we examined ds^{DB} to produce S^{DB} . Eventually, the support count $\langle (d) \rangle_{count}^{DB}$ is increased by two considering the two implicitly merged sequences of C2 and C5. Similarly, the support count of candidate $\langle (h) \rangle_{count}^{DB}$ is two after the implicit merging on customer sequences whose $cid=C4$ and $cid=C8$. □

6.4.2 The *IncSP* (Incremental Sequential Pattern Update) algorithm

The implicit merging technique preserves the correctness of supports of the patterns and enables *IncSP* to count the supports in *DB* and *db* separately for pattern updating. Fig. 6-7 lists the proposed *IncSP* algorithm and Fig. 6-8 depicts the two separate sub-processes of support counting in the *IncSP* algorithm. Through separate counting, we do not have to check the full candidate set X_k against all data sequences from *db* and *DB*. Only the (usually) smaller *db* must take all the candidates in X_k into

- 1) /* Initially, each item is a candidate 1-sequence */
- 2) X_1 = set of candidate 1-sequences;
- 3) $k = 1$; /* Start from pass 1 */
- 4) repeat /* Find frequent k -sequences in the k -th pass */
- 5) for each $s \in X_k$ do $s_{count}^{DB} = s_{count}^{db} = 0$; /* Initialize counters */
- 6) Read S_k^{DB} ; /* $S_k^{DB} = \{\text{frequent } k\text{-sequences in } DB\}$ */
- 7) Check sequences in db by **Support Counting (I)** ; /* See Fig. 6-8 */
- 8) /* Prune candidates: (1) counted in S_k^{DB} (2) insufficient “new” counts */
- 9) $X'_k = X_k - \{s | s \in S_k^{DB}\} - \{s | s_{count}^{db} < minsup \times (|UD| - |DB|)\}$;
- 10) Check sequences in DB by **Support Counting (II)** ; /* See Fig. 6-8 */
- 11) /* Frequent k -sequences in UD found */
- 12) $S_k = \{s | s \in X'_k \wedge s_{count}^{DB} + s_{count}^{db} \geq minsup \times |UD|\}$;
- 13) $k = k + 1$;
- 14) Generate C_k with S_{k-1} ; /* Generate candidates for next pass */
- 15) until no more candidates
- 16) Answer $S^{UD} = \cup_k S_k$;



Fig. 6-7. Algorithm *IncSP*

Support Counting (I):

/* Updating “old” supports and counting candidates against data sequences in db */

- 1) for each data sequence ds^{db} in db do
- 2) if cid of ds^{db} is not found in DB then /* ds^{db} is a new customer’s sequence */
- 3) /* Increment s_{count}^{db} by 1 if s is contained in ds^{db} */
- 4) for each $s \in X_k \wedge s \subseteq ds^{db}$ do $s_{count}^{db}++$;
- 5) endif
- 6) if cid of ds^{db} is found in DB then /* ds^{db} should be appended to ds^{DB} */
- 7) for each $s \in X_k \wedge s \subseteq (ds^{DB} \cup ds^{db})$ do /* Implicit merging and counting */
- 8) /* Increment s_{count}^{db} by 1 if s is contained in ds^{db} but not in ds^{DB} */
- 9) if $s \notin ds^{DB}$ then $s_{count}^{db}++$;
- 10) endfor
- 11) endif
- 12) endfor

Support Counting (II):

/* Counting “new” candidates against data sequences in DB */

- 1) for each data sequence ds^{DB} in DB do
- 2) /* Increment s_{count}^{DB} by 1 if s is contained in ds^{DB} */
- 3) for each $s \in X'_k \wedge s \subseteq ds^{DB}$ do $s_{count}^{DB}++$; /* X'_k is the reduced candidate set */
- 4) endfor

Fig. 6-8. The separate counting procedure

consideration for support updating. Furthermore, we can prune previous patterns and leave fewer but more promising candidates in X_k' before applying the data sequences in DB for support counting.

The *IncSP* algorithm generates candidates and computes the supports for pattern updating in multiple passes. In each pass, we initialize the two support counts of each candidate in UD to zero, and read the support count of each frequent k -sequence s in DB to s_{count}^{DB} . We then accumulate the increases in support count of candidates with respect to the sequences in db by Support Counting (I). Before Support Counting (II) starts, candidates which are frequent in DB but cannot be frequent in UD according to Lemma 6-4 are filtered out. The full candidate set X_k is reduced into the set X_k' . Next, the Support Counting (II) calculates the support counts of these promising candidates with respect to the sequences in DB . As indicated in Lemma 6-1, the support count of any candidate k -sequence is the sum of the two counts obtained after the two counting processes. Consequently, we can discover the set of frequent k -sequences S_k by validating the sum of the two counts of every candidate. The S_k is used to generate the complete candidate set for the next pass, employing the similar candidate generation procedure in *GSP*. The above process is iterated until no more candidates.

We need to show that *IncSP* updates the supports and discovers frequent patterns correctly. Several properties used in the *IncSP* algorithm are described as follows. The details of the proof of the lemmas are included in Appendix.

Lemma 6-1. The support count of any candidate k -sequence s in UD is equal to s_{count}^{DB} + s_{count}^{db} .

Lemma 6-2. A candidate sequence s , which is not frequent in DB , is a frequent sequence in UD only if $s_{count}^{db} \geq minsup \times (|UD| - |DB|)$.

Lemma 6-3. The separate counting procedure (in Fig. 6-8) completely counts the supports of candidate k -sequences against all data sequences in UD .

Lemma 6-4. The candidates required for checking against the data sequences in DB in Support Counting (II) is the set X_k' , where $X_k' = X_k - \{s \mid s \in S_k^{DB}\} - \{s \mid s_{count}^{db} < minsup \times (|UD| - |DB|)\}$.

Theorem 6-1. *IncSP* updates the supports and discovers frequent patterns correctly.

Proof: In *IncSP*, we use the candidate generation procedure analogous to *GSP* to produce the complete set of candidates in X_k . By Lemma 6-3, the separate counting procedure completely counts the supports of candidate k -sequences against all data sequences in UD . Lemma 6-1 determines frequent patterns in UD and the updated supports. Therefore, *IncSP* correctly maintains sequential patterns. \square

Example 2: Sequential pattern updating using *IncSP*. The data sequences in the original database DB is shown in Fig. 6-3(b). The *minsup* is 33%. S^{DB} is listed in Table 6-2. The increment database db is shown in Fig. 6-4. *IncSP* discovers S^{UD} as follows.

Pass 1:

- 1) Generate candidates for pass 1, $X_1 = \{<(a)>, <(b)>, \dots, <(h)>\}$.
- 2) Initialize the two counts of each candidate in X_1 to zero, and read S_1^{DB} .

After Support Counting (I), the increases in support count are listed in Part (b) of Table 6-2. Note that for customer with $cid=C5$, the increase in support count of $<(a)>$ is not changed. Now $|UD| = 12$ and $|DB| = 9$. Since $S_1^{DB} = \{<(a)>, <(b)>, <(d)>, <(e)>\}$ and the increase in support count of $<(c)>$ are less than $33\% \times (|UD| - |DB|)$, the reduced set X_1' is $\{<(f)>, <(g)>, <(h)>\}$.

Table 6-2. Sequences and support counts for Example 2

Part (a): S^{DB}	Part (b): Pass 1	Part (c): Pass 2	Part (d): S^{UD}
s_{count}^{DB}	Support Counting (I) s_{count}^{db}	Support Counting (I) s_{count}^{db}	s_{count}^{UD}
<(a)> 6	<(a)> 1	<(a)(a)> 1	<(a)> 7
<(b)> 6	<(b)> 2	<(a)(d)> 2	<(b)> 8
<(d)> 5	<(d)> 3	<(b)(d)> 1	<(d)> 8
<(e)> 3	<(f)> 2	<(b,d)> 1	<(f)> 4
<(b)(b)> 3	<(g)> 2	<(b,f)> 2	<(a)(d)> 4
<(b)(e)> 3	<(h)> 3	<(d,f)> 1	<(b,d)> 4
<(a,b)> 3	<(c)> 0	<(a,d)> 1	<(b,f)> 4
<(b,d)> 3	<(e)> 0	Others 0	
	Support Counting (II) s_{count}^{DB}	Support Counting (II) s_{count}^{DB}	
	<(f)> 2	<(a)(d)> 2	
	<(g)> 1	<(b)(d)> 1	
	<(h)> 0	<(b,f)> 2	
		<(a)(a)> 0	
		<(a,d)> 0	
		<(d,f)> 0	

3) After Support Counting (II), the s_{count}^{DB} of <(f)> and <(g)> are 2 and 1 respectively.

The minimum support count is 4 in UD . $IncSP$ obtains the updated frequent 1-sequences, which are <(a)>, <(b)>, <(d)>, and <(f)>. Total 22 candidate 2-sequences are generated with the four frequent 1-sequences.

Pass 2:

4) We read S_2^{DB} after initializing the two support counts of all candidate 2-sequences. Note that the s_{count}^{DB} of <(b)(e)> is useless because <(b)(e)> is not a candidate in UD in this pass.

5) We list the result of Support Counting (I) in Part (c) of Table 6-2. The increases in

support count of some candidates, such as $\langle(a,f)\rangle$ or $\langle(d,f)\rangle$, are all zero and are not listed.

- 6) Again, we compute the X_2' so that the candidates need to be checked against the data sequences in DB are $\langle(a)(a)\rangle$, $\langle(a)(d)\rangle$, $\langle(a,d)\rangle$, $\langle(b)(d)\rangle$, $\langle(b,f)\rangle$, and $\langle(d,f)\rangle$. We filter out 16 candidates (13 candidates with insufficient “support increases” and 3 candidates in S_2^{DB}) before Support Counting (II) starts.
- 7) The s_{count}^{DB} of $\langle(a)(d)\rangle$, $\langle(b)(d)\rangle$, and $\langle(b,f)\rangle$ are 2, 1, and 2 respectively after Support Counting (II). *IncSP* then sums up the counts (s_{count}^{DB} and s_{count}^{db}) to obtain the updated frequent 2-sequences. Finally, *IncSP* terminates since no candidate 3-sequence is generated. Part (d) of Table 6-2 lists the sequential patterns and their support counts in UD .□

In comparison with *GSP*, *IncSP* updates supports of sequential patterns in S^{DB} by scanning data sequences in db only. New sequential patterns, which are not in DB , are generated from fewer candidate sequences comparing with previous methods. The support increases of new candidates are checked in advance and leave the most promising candidates for Support Counting (II) against data sequences in DB . Every candidate in the reduced set is then checked against DB to see if it is frequent in UD . On the contrary, *GSP* takes every candidate and counts over all data sequences in the updated database. Consequently, *IncSP* is much faster than *GSP* as shown in the experimental results.

6.4.3 Pattern maintenance on transaction deletion and modification

Common operations on constantly updated databases include not only appending, but also deletions and modifications. Deleting transactions from a data sequence changes the sequence; thereby changing the supports of patterns contained in this sequence.

The supports of the discovered patterns might decrease but no new patterns would occur. We check patterns in S^{DB} against these data sequences. Assume that a data sequence ds is changed to ds' due to deletion. The ds' is an empty sequence when all transactions in ds are deleted. If a frequent sequence s is contained in ds but not in ds' , s_{count}^{DB} is decreased by one. The resulting sequential patterns in the updated database are those patterns still having minimum supports.

A transaction modification can be accomplished by deleting the old transaction and then inserting the new transaction. In *IncSP*, we delete the original data sequence from the original database, create a new sequence comprising the substituted transaction(s), and then append the new sequence to the increment database.

6.4.4 Proof of lemmas

We provide the proof of lemmas used in Section 6.4. As noted in Table 6-1, s_{count}^{DB} is the support count of candidate sequence s in DB , and s_{count}^{db} denotes the increase in support count of candidate sequence s due to db . The candidate k -sequences in UD is partitioned into $X_{\mathcal{K}(DB)}$ and $X_{\mathcal{K}(DB)'}.$ That is, $X_k = X_{\mathcal{K}(DB)} \cup X_{\mathcal{K}(DB)'}$, where $X_{\mathcal{K}(DB)} = \{s | s \in X_k \wedge s \in S_k^{DB}\}$ and $X_{\mathcal{K}(DB)'} = X_k - X_{\mathcal{K}(DB)}$. The data sequences in UD could be partitioned into three sets: sequences with *cids* appearing in DB only, sequences with *cids* appearing in db only, and sequences with *cids* occurring in both DB and db . The *cid* of a data sequence ds is represented by $ds.cid$. Let $UD = UD_{DB} \cup UD_{db} \cup UD_{Dd}$, where $UD_{DB} = \{ds | ds \in DB \wedge ds \notin db\}$, $UD_{db} = \{ds | ds \in db \wedge ds \notin DB\}$, and $UD_{Dd} = \{ds | ds = ds_1 + ds_2, ds_1 \in DB \wedge ds_2 \in db \wedge ds_1.cid = ds_2.cid\}$.

Lemma 6-1. The support count of any candidate k -sequence s in UD is equal to s_{count}^{DB}

$$+ s_{count}^{db}.$$

Proof: The support count of s in UD is the support count of s in DB , plus the count increase due to the data sequences in db . That is $s_{count}^{DB} + s_{count}^{db}$ by definition. \square

Lemma 6-2. A candidate sequence s , which is not frequent in DB , is a frequent sequence in UD only if $s_{count}^{db} \geq \text{minsup} \times (|UD| - |DB|)$.

Proof: Since $s \notin S^{DB}$, we have $s_{count}^{DB} < \text{minsup} \times |DB|$. If $s_{count}^{db} < \text{minsup} \times (|UD| - |DB|)$, then $s_{count}^{DB} + s_{count}^{db} < \text{minsup} \times |UD|$. That is, $s \notin S^{UD}$. \square

Lemma 6-3. The separate counting procedure (in Fig. 6-8) completely counts the supports of candidate k -sequences against all data sequences in UD .

Proof: Considering a data sequence ds in UD and a candidate k -sequence $s \in X_k$,

(i) For each candidate k -sequence s contained in ds where $ds \in UD_{db}$: The support count increase (due to ds) is accumulated in s_{count}^{db} , by line 4 of Support Counting (I) in Fig. 6-8.

(ii) For each candidate k -sequence s contained in ds where $ds \in UD_{DB}$: (a) If $s \in X_{\mathcal{K}(DB)}$, no counting is required since s had been counted while discovering S^{DB} . The support count of s in DB is read in s_{count}^{DB} by line 6 in Fig. 6-7. (b) If $s \in X_{\mathcal{K}(DB)'}'$, s_{count}^{DB} accumulates the support count of s , by line 3 of Support Counting (II) in Fig. 6-8. Note that in this counting, we reduce $X_{\mathcal{K}(DB)'}'$ to $X_{\mathcal{K}'}$ by Lemma 6-4.

(iii) For each candidate k -sequence s contained in ds where $ds \in UD_{Dd}$: Now ds is formed by appending ds^{db} to ds^{DB} . (a) If $s \not\subseteq ds^{DB}$, i.e., ds^{DB} of the ds does not contain s . We accumulate the increase in s_{count}^{db} , by line 9 of Support Counting (I) in Fig. 6-8. (b) If $s \subseteq ds^{DB} \wedge s \in X_{\mathcal{K}(DB)}$, similar to (ii)-(a), the support count is already read in s_{count}^{DB} so that no counting is required. (c) If $s \subseteq ds^{DB} \wedge s \in X_{\mathcal{K}(DB)}'$,

similar to (ii)-(b), we calculate s_{count}^{DB} by line 3 of Support Counting (II) in Fig. 6-8. Again, $X_{\kappa_{(DB)'}}$ is reduced to $X_{\kappa'}$ by Lemma 6-4 here.

The separate counting considers all the data sequences in UD as described here. Next, we show that the supports of all candidates are calculated. By Lemma 6-1, the support count of s in UD is the sum of s_{count}^{DB} and s_{count}^{db} .

(iv) For any candidate s in $X_{\kappa_{(DB)'}}$: The s_{count}^{DB} is from (ii)-(a) and (iii)-(b), and the s_{count}^{db} is accumulated by (i) and (iii)-(a).

(v) For any candidate s in $X_{\kappa_{(DB)'}}$: The s_{count}^{DB} is counted by (ii)-(b) and (iii)-(c), and the s_{count}^{db} is counted by (i) and (iii)-(a). The separate counting is complete. \square

Lemma 6-4. The candidates required for checking against the data sequences in DB in Support Counting (II) is the set $X_{\kappa'}$, where $X_{\kappa'} = X_k - \{s \mid s \in S_k^{DB}\} - \{s \mid s_{count}^{db} < minsup \times (|UD| - |DB|)\}$.

Proof: Since $UD = UD_{DB} \cup UD_{db} \cup UD_{Dd}$ and UD_{db} contains no data sequence in DB , the data sequences concerned are in UD_{DB} and UD_{Dd} . Considering a candidate s ,

(i) If $s \in S_k^{DB}$: For any data sequence $ds \in UD_{DB}$ or $ds \in UD_{Dd} \wedge s \subseteq ds^{DB}$, s was counted while discovering S_k^{DB} . For $ds \in UD_{Dd} \wedge s \not\subseteq ds^{DB}$, the increase in support count s_{count}^{db} is accumulated by line 9 of Support Counting (I). Therefore, in Support Counting (II), we can exclude any candidate s which is also in S_k^{DB} .

(ii) If $s \notin S_k^{DB}$: After Support Counting (I), the s_{count}^{db} now contains the support count counted for data sequence ds , where $ds \in UD_{db}$ or $ds \in UD_{Dd} \wedge s \not\subseteq ds^{DB}$. By Lemma 6-2, if the s_{count}^{db} is less than $minsup \times (|UD| - |DB|)$, this candidate s

cannot be frequent in UD . Therefore, such candidate s could be filtered out.

(iii) By (i) and (ii), we have $X_{k'} = X_k - \{s \mid s \in S_k^{DB}\} - \{s \mid s_{count}^{db} < minsup \times (|UD| - |DB|)\}$. \square

6.5 Experimental Results

In order to assess the performance of the *IncSP* algorithm, we conducted comprehensive experiments using an 866 MHz Pentium-III PC with 1024MB memory. In these experiments, the databases are composed of synthetic data. The method used to generate these data is described in Section 6.5.1. Section 6.5.2 compares the performance and resource consumption of algorithms *GSP*, *ISM* and *IncSP*. Results of scale-up experiments are presented in Section 6.5.3. Section 6.5.4 discusses the memory requirements of these algorithms.

6.5.1 Synthetic data generation

Updating the original database DB with the increment database db was modeled by generating the update database UD , then partitioning UD into DB and db . Synthetic transactions covering various data characteristics were generated by the well-known method in [6]. As to the details of generating synthetic data, please refer to Section 3.5.1. Since all sequences were generated from the same statistical patterns, it might model real updates very well.

At first, total $|UD|$ data sequences were created as UD . Three parameters are used to partition UD for simulating different updating scenarios. Parameter R_{inc} , called *increment ratio*, decides the size of db . Total $|db| = |UD| \times R_{inc}$ sequences were randomly picked from UD into db . The remaining $|UD| - |db|$ sequences would be placed in DB . The *comeback ratio* R_{cb} determines the number of “old” customers in db . Total $|old| = |db| \times R_{cb}$ sequences were randomly chosen from these $|db|$ sequences

as “old” customer sequences, which were to be split further. The splitting of a data sequence is to simulate that some transactions were conducted formerly (thus in DB), while the remaining transactions were newly appended. The splitting was controlled by the third parameter R_f , the *former ratio*. If a sequence with total $|ds^{UD}|$ transactions was to split, we placed the leading $|ds^{DB}| = |ds^{UD}| \times R_f$ transactions in DB and the remaining $|ds^{UD}| - |ds^{DB}|$ transactions in db . For example, a UD with $R_{inc} = 20\%$, $R_{cb} = 30\%$, and $R_f = 40\%$ means that 20% of sequences in UD come from db , 30% of the sequences in db have $cids$ occurring in DB , and that for each “old” customer, 40% of his/her transactions were conducted before current pattern updating. Note that the calculation is integer-based with ‘ceiling’ function. E.g. $|ds^{UD}| = 4$, $|ds^{DB}| = \lceil 4 * 40\% \rceil = 2$.

Table 6-3 summarizes the symbols and the parameters used in the experiments. A database generated with these parameters is described as follows. The updated database has $|UD|$ customer sequences, each customer has $|C|$ transactions on average, and each transaction has average $|T|$ items. A table of total N_I PFI s and a table of total N_S PFS s were generated before picking items for the transactions of customer sequences. On average, a PFS has $|S|$ transactions and a PFI has $|I|$ items. The total number of possible items for all PFI s is N . All datasets used here were generated by setting μ_{crup_S} and μ_{crup_I} to 0.75, σ_{crup_S} and σ_{crup_I} to 0.1, μ_{corr_S} and μ_{corr_I} to 0.25, $N_S = 5000$, $N_I = 25000$. Two values of N (1000 and 10000) were used. A dataset created with $|C| = \alpha$, $|T| = \beta$, $|S| = \chi$, and $|I| = \delta$ is denoted by the notation $C\alpha.T\beta.S\chi.I\delta$.

6.5.2 Comparisons of *IncSP* and *GSP*

To realize the performance improvements of *IncSP*, we first compare the efficiency of incremental updating with that of re-mining from scratch, and then contrast that with other incremental mining approaches. The well-known *GSP* algorithm [80], which is

Table 6-3. Parameters used in the experiments

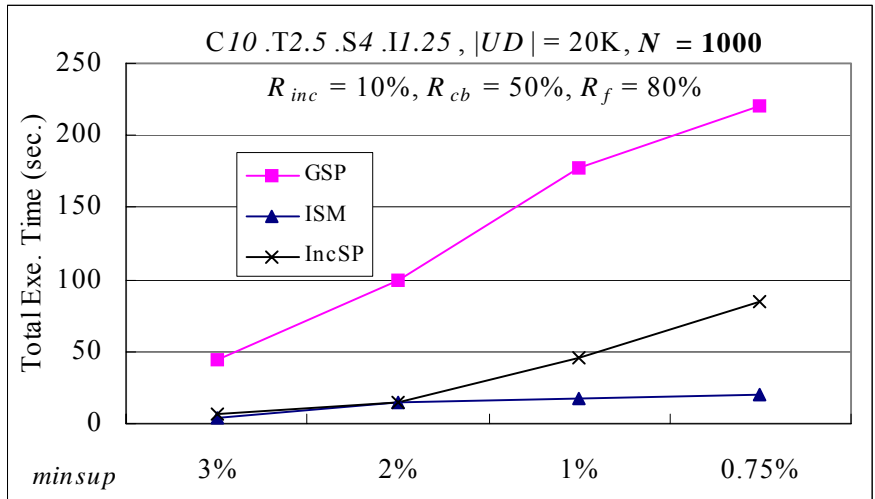
Parameter	Description	Value
$ UD $	Number of data sequences in database UD	10K, 100K, 250K, 500K, 750K, 1000K
$ C $	Average size (number of transactions) per customer	10, 20
$ T $	Average size (number of items) per transaction	2.5, 5
$ S $	Average size of potentially sequential patterns	4, 8
$ I $	Average size of potentially frequent itemsets	1.25, 2.5
N	Number of possible items	1000, 10000
N_I	Number of potentially frequent itemsets	25000
N_S	Number of possible sequential patterns	5000
Γ_S	The table of <i>potentially frequent sequences (PFSs)</i>	
Γ_I	The table of <i>potentially frequent itemsets (PFIs)</i>	
$corr_S$	Correlation level (sequence), exponentially distributed	$\mu_{corr_S} = 0.25$
$crup_S$	Corruption level (sequence), normally distributed	$\mu_{crup_S} = 0.75,$ $\sigma_{crup_S} = 0.1$
$corr_I$	Correlation level (itemset), exponentially distributed	$\mu_{corr_I} = 0.25$
$crup_I$	Corruption level (itemset), normally distributed	$\mu_{crup_I} = 0.75,$ $\sigma_{crup_I} = 0.1$
R_{inc}	Ratio of increment database db to updated database UD	1%, 2%, 5%, 8%, 10%, 20%, 30%, ..., 90%
R_{cb}	Ratio of comeback customers to all customers in increment database db	0%, 10%, 25%, 50%, 75%, 100%
R_f	Ratio of former transactions to all transactions for each “old” customer	10%, 20%, ..., 90%

a re-mining based algorithm, is used as the basis for comparison. The *PrefixSpan* algorithm [67] mines patterns by recursively projecting data sequences to smaller intermediate databases. Starting from prefix-items (the frequent items), sequential patterns are found by recursively growing subsequence fragments in each intermediate database. Except re-mining, mechanisms of modifying *PrefixSpan* to solve incremental updating is not found in the literature. Since it demands a totally different framework to handle the sequence projection of the original database and the increment database, the *PrefixSpan* is not included in the experiments. The *ISM* algorithm [64], which is the incremental mining version of the *SPADE* algorithm [99],

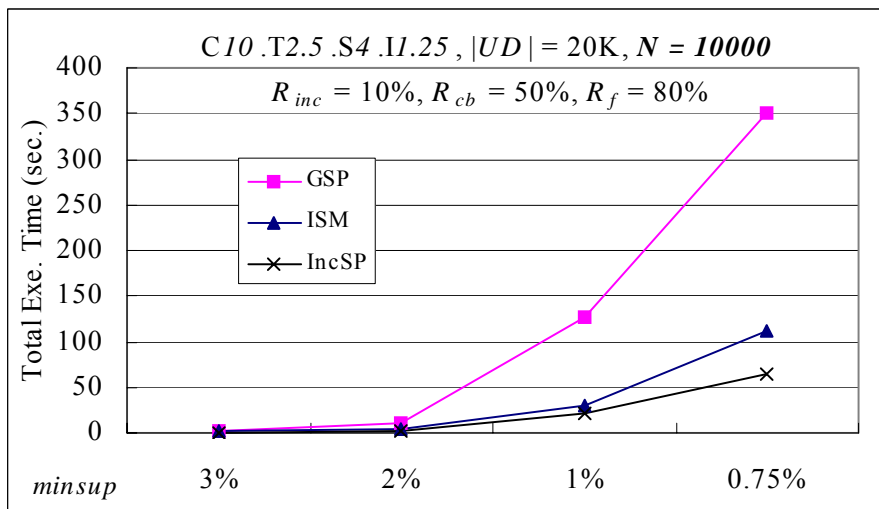
deals with database update using databases of vertical layout. We pre-processed the databases for *ISM* into vertical layout and the pre-processing time is not counted in the following context.

Extensive experiments were performed to compare the execution times of *GSP*, *ISM*, and *IncSP* with respect to critical factors that reflect the performance of incremental updating, including *minsup*, increment ratio, comeback ratio, and former ratio. We set $R_{inc}=10\%$, $R_{cb}=50\%$, and $R_f=80\%$ to model common database updating scenarios. The dataset has 20000 sequences ($|UD| = 20K, 3.8MB$), generated with $|C|=10, |T|=2.5, |S|=4, |I|=1.25$.

The effect on performance with various *minsup*s was evaluated first. Re-mining is less efficient than incremental updating, as indicated in Fig. 6-9. In the experiments, both *ISM* and *IncSP* are faster than *GSP* for all values of minimum supports. Fig. 6-9(a) shows that *ISM* is faster than *IncSP* when the number of items (N) is 1000 and $minsup \leq 1\%$. When N is 10000, *IncSP* outperforms *ISM* for all values of *minsup*, as shown in Fig. 6-9(b). The total execution time is longer for all the three algorithms for smaller *minsup* value, which allows more patterns to pass the frequent threshold. *GSP* suffers from the explosive growth of the number of candidates and the re-counting of supports for each pattern. For example, when *minsup* is 1% and $N = 10000$, the number of candidate 2-sequences in *GSP* is 532526 and that of ‘new’ candidate 2-sequences in *IncSP* is 59. Only 59 candidate 2-sequences required counting over the data sequences in *UD*. The other candidate 2-sequences are updated, rather than re-counted, against the 2000 sequences in *UD* ($UD*10\%$).



(a) $N = 1000$



(b) $N = 10000$

Fig. 6-9. Total execution times over various *minsup*

Comparing Fig. 6-9(a) with Fig. 6-9(b), it indicates that *ISM* is more efficient with a smaller N . *ISM* keeps all frequent sequences, as well as the maximally potential frequent sequences (negative borders), in memory. Take *minsup* = 0.75% for example. The number of frequent sequences is 701 for $N = 1000$ and 1017 for $N = 10000$, respectively. Accordingly, the size of negative borders of size two is 736751 and 1550925, respectively. Those turn-into-frequent patterns that were in negative borders before database updating must intersect with the complete set of frequent patterns. Consequently, with a smaller *minsup* like 0.75%, the larger N provides more possible

items to pass the frequent threshold so that the total execution is less efficient in *ISM*. Instead of frequent-pattern intersection, *IncSP* deals with candidates separately, the explosively increased frequent items (because of the larger N) affect the efficiency of the pattern updating less. This also accounts for the performance gaps between *IncSP* and *ISM*, no matter how increment ratio, comeback ratio or former ratio changes.

The results of varying increment ratio from 1% to 50% are shown in Fig. 6-10. The *minsup* is fixed at 2%. In general, *IncSP* gains less at higher increment ratio because larger increment ratio means more sequences appearing in *db* and causes more pattern updatings. As indicated in Fig. 6-10, the smaller the increment database *db* is, the more time on the discovery *IncSP* could save.

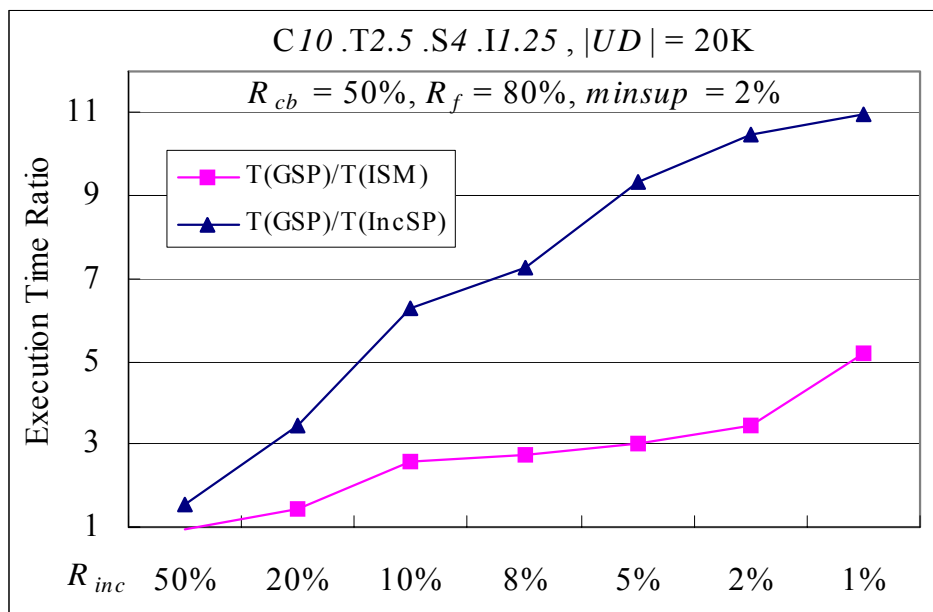


Fig. 6-10. Total execution times over various incremental ratios

IncSP is still faster than *GSP* even when increment ratio is 50%. When increment ratio becomes much larger, say over 60%, *IncSP* is slower than *GSP*. Clearly, when most of the frequent sequences in *DB* turn out to be invalid in *UD*, the information used by *IncSP* in pattern updating might become useless. When the size of the increment database becomes larger than the size of the original database, i.e. the

database has accumulated dramatic change and not incremental change any more, re-mining might be a better choice for the total new sequence database.

The impact of the comeback ratio is presented in Fig. 6-11. *IncSP* updates patterns more efficiently than *GSP* and *ISM* for all the comeback ratios. High comeback ratio means that there are many ‘old’ customers in the increment database. Consequently, the speedup ratio decreases as the comeback ratio increases because more sequence merging is required. Fig. 6-11 shows that *IncSP* was efficient with implicit merging, even when the comeback ratio was increased to 100%, i.e., all the sequences in the increment database must be merged.

Fig. 6-12 depicts the performance comparisons concerning former ratios. It can be seen from the figure that *IncSP* was constantly about 6.5 times faster than *GSP* over various former ratios, ranging from 10% to 90%.

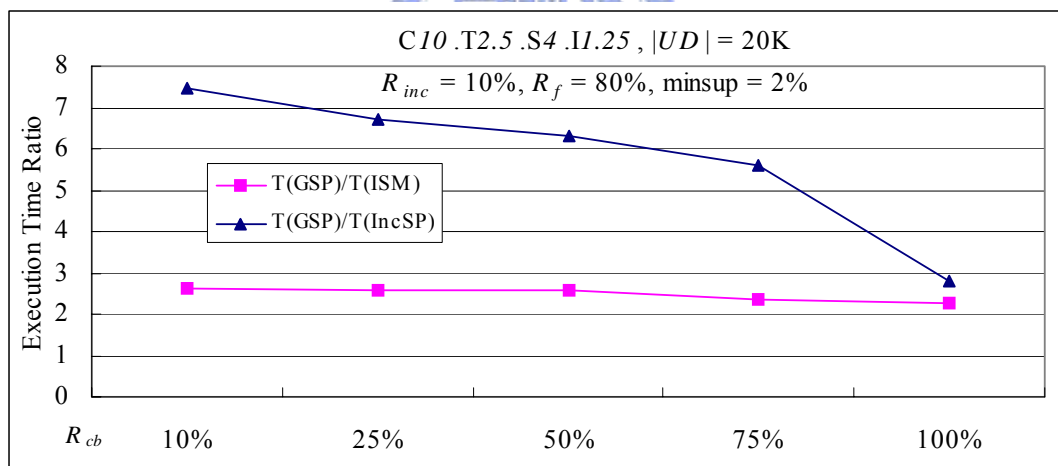


Fig. 6-11. Total execution times over various comeback ratios

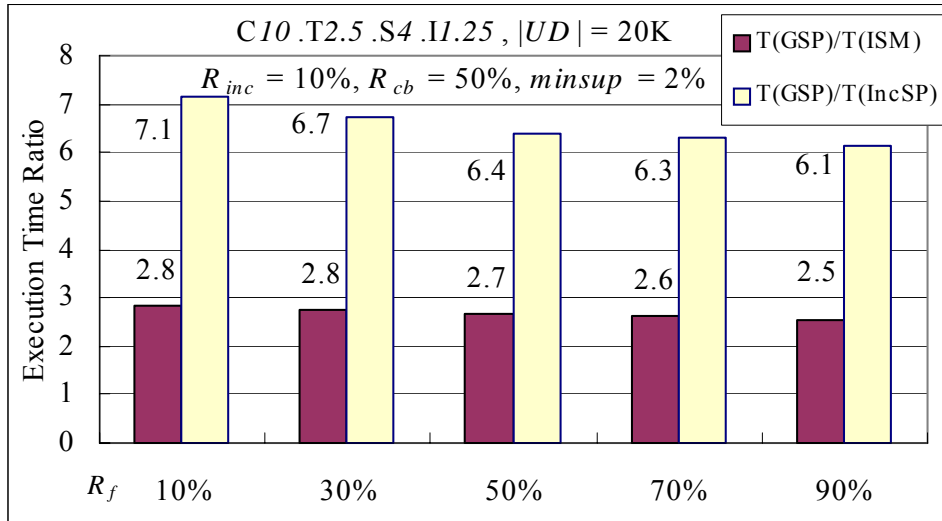


Fig. 6-12. Total execution times over various former ratios

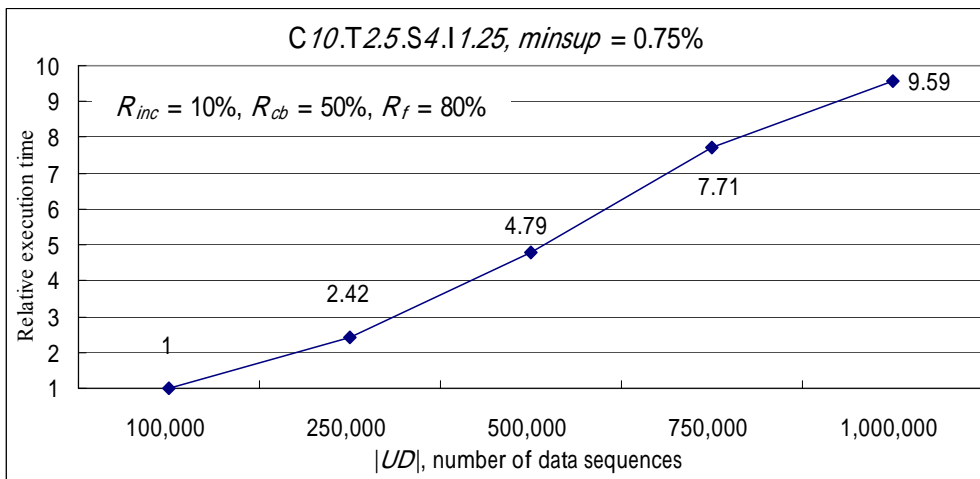


Fig. 6-13. Linear scalability of the database size

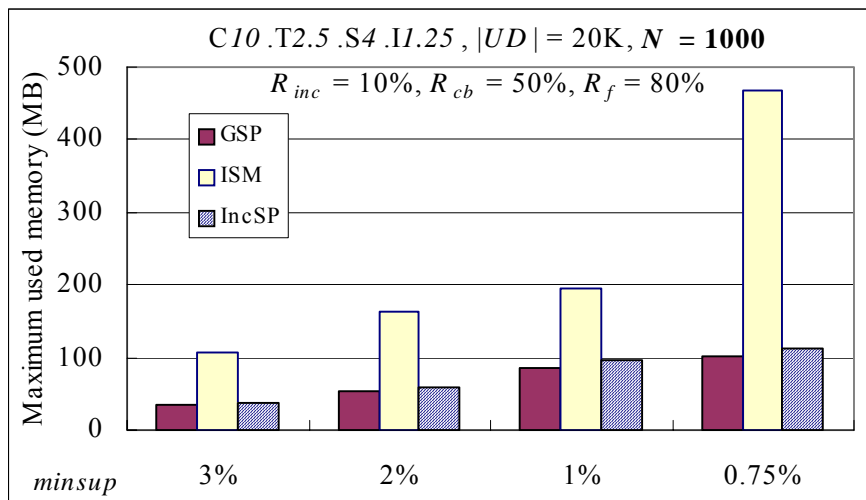


Fig. 6-14. Maximum required memory with respect to various minsup

6.5.3 Scale-up experiments

To assess the scalability of our algorithm, several experiments of large databases were conducted. Since the basic construct of *IncSP* is similar to that of *GSP*, similar scalable results could be expected. In the scale-up experiments, the total number of customers was increased from 100K (18.8MB) to 1000K (187.9MB), with fixed parameters *C10.T2.5.S4.I1.25*, $N = 10000$, $R_{inc} = 10\%$, $R_{cb} = 50\%$, and $R_f = 80\%$. Again, *IncSP* are faster than *GSP* for all the datasets. The execution times were normalized with respect to the execution time for 100K customers here. Fig. 6-13 shows that the execution time of *IncSP* increases linearly as the database size increases, which demonstrates good scalability of *IncSP*.

6.5.4 Memory requirements

Although *IncSP* uses separate counting to speed up mining, it generates candidates and then performs counting by multiple database scanning, like *GSP*. The pattern updating process in *IncSP* reads in the previous discovered patterns and stores them into a hash-tree for fast support updating. Therefore, the maximum size of memory required for both *GSP* and *IncSP* is determined by the space required to store the candidates. A smaller *minsup* often generates a large number of candidates, thereby demanding a larger memory space.

In contrast, *ISM* applies item-intersection in each class for new pattern discovery, assuming that all frequent sequences as well as potentially frequent sequences are stored in a lattice in memory. Storing every possible frequent sequence costs a huge memory space, not to mention those required for lattice links. For instance, the size of negative borders of size two is over 1.5 million with $N = 10000$ ($minsup = 0.75\%$) in the experiment of Fig. 6-9(b). As shown in Fig. 6-14, the required memory for *IncSP* is smaller than that of *ISM*. More memory is required in vertical approaches like

SPADE.

6.6 Summary

The problem of sequential pattern mining is much more complicated than association discovery due to sequence permutation. Validity of discovered patterns may change and new patterns may emerge after updates on databases. In order to keep the sequential patterns current and up-to-dated, re-execution of the mining algorithm on the whole database updated is required. However, it takes more time than required in prior mining because of the additional data sequences appended. Therefore, we proposed the *IncSP* algorithm utilizing previously discovered knowledge to solve the maintenance problem efficiently by incremental updating without re-mining from scratch. The performance improvements result from effective implicit merging, early candidate pruning, and efficient separate counting.

Implicit merging ensures that *IncSP* employs correctly combined data sequences while preserving previous knowledge useful for incremental updating. Candidate pruning after updating pattern supports against the increment database further accelerates the whole process, since fewer but more promising candidates are generated by just checking counts in the increment database. Eventually, efficient support counting of promising candidates over the original database accomplishes the discovery of new patterns. *IncSP* both updates the supports of existing patterns and finds out new patterns for the updated database. The simulation performed shows that the proposed incremental updating mechanism is several times faster than re-mining using the *GSP* algorithm, with respect to various data characteristics or data combinations. *IncSP* outperforms *GSP* with regard to different ratios of the increment database to the original database except when the increment database becomes larger than the original database. It means that it has been long time since last database

maintenance and most of the patterns become obsolete. In such a case, re-mining with new *minsup* over the whole database would be more appropriate since the original *minsup* might not be suitable for current database any more.

The *IncSP* algorithm currently solves the pattern updating problems using previously specified minimum support. Further researches could be extended to the problems of dynamically varying minimum supports. Generalized sequential pattern problems [80], such as patterns with *is-a* hierarchy or with sliding-time window property, are also worthy of further investigation since different constraints induce diversified maintenance difficulties. In addition to the maintenance problem, constantly updated database generally create a pattern-changing history, indicating changes of sequential patterns at different time. It is challenging to extend the proposed algorithm to exploring the pattern changing history for trend prediction.



Chapter 7 Conclusions and Future Work

7.1 Contributions

The objective of this thesis is to investigate efficient and scalable algorithms for mining frequent patterns in large databases. The algorithms proposed in this thesis include:

- *LexMiner*: A fast algorithm for mining frequent itemsets in association rule mining
- *MEMISP*: An efficient algorithm for mining sequential patterns in databases of any size, using only two passes of database scanning at most
- *DELISP*: A divide-and-conquer method for mining sequential patterns with time constraints including minimum gap, maximum gap, and sliding windows
- *KISP*: An interactive algorithm that reduce the total response time
- *IncSP*: An incremental algorithm for updating sequential patterns after a non-trivial updates of the large database.

All the proposed algorithms are verified by experiments of mining large datasets of various characteristics. In the experiments comprising comprehensive comparisons, the proposed algorithms outperform several related algorithms, and they all show excellent linear scalability with respect to the size of the databases.

7.2 Future work

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

- The discovery of sequential patterns with time constraints by memory indexing

The proposed algorithm for sequence mining with constraints in this thesis, though outperforms the other mining algorithms, requires the creation of intermediate sub-databases. The accumulated size of the sub-databases might be several times bigger than that of the original database. The memory indexing approach is efficient for the discovery of common sequential patterns without the need of generating any sub-databases. It is worthy of study on extending the memory indexing approach for efficient mining of generalized sequential patterns.

- Maximal frequent sequence mining

Most sequence mining algorithms aim to find out the set of all frequent sequences. In some applications, we only need to discover those frequent sequences that have no super-sequences. For example, given the longest frequent sequences $\langle(e)(f)\rangle$ and $\langle(b,c)(a,d)\rangle$, the users also learn that all their sub-sequences (like $\langle(e)\rangle$, $\langle(f)\rangle$, $\langle(b,c)\rangle$, $\langle(a)\rangle$, $\langle(b,c)(d)\rangle$, etc.) are frequent. Once we have the maximal frequent sequences, we may start classifying data sequences according to the longest common elements. The lengthy process for mining the complete set of the frequent sequences is no longer needed.

- Integration with database management systems

Given the success of the proposed algorithms, a seamless integration with the database management system is necessary. The benefits for end-users will be maximized only if the trivial process of selecting target data, transforming data, and mining data is integrated as one of the query functions of the database management system. However, such an integration requires not only extensions on data manipulation languages but also effective indexing and accessing mechanisms coupled with the system. It is

challenging to integrate the mining algorithms with the database management systems.



References

- [1] R. C. Agarwal, C. C. Aggarwal, and V.V.V. Prasad, "A Tree Projection Algorithm for Generation of Frequent Item Sets," *Journal of Parallel and Distributed Computing*, Vol. 61, No. 3, pp. 350-371, 2001.
- [2] R. C. Agarwal, C. C. Aggarwal, "Depth First Generation of Long Patterns," *Proceedings of 2000 ACM International Conference on Knowledge Discovery in Databases*, pp. 108-118, 2000.
- [3] C. C. Aggarwal and P. S. Yu, "Online Generation of Association Rules," *Proceedings of the 14th International Conference on Data Engineering*, Orlando, Florida, USA, pp. 402-411, Feb. 1998.
- [4] R. Agrawal, T. Imielinski, A. Swami, "Mining Association Rules between Sets of Items in Large Databases," *Proceedings of the 1993 ACM SIGMOD Conference on Management of Data*, Washington D.C., pp. 207-216, May 1993.
- [5] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo, "Fast Discovery of Association Rules," *Advances in Knowledge Discovery and Data Mining*, U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, eds., AAAI/MIT Press, pp. 307-328, 1996.
- [6] R. Agrawal and R. Srikant, "Mining Sequential Patterns," *Proceedings of the 11th International Conference on Data Engineering*, Taipei, Taiwan, pp. 3-14, March 1995.
- [7] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," *Proceedings of the 20th International Conference on Very Large Data Bases*, Santiago, Chile, pp. 487-499, Sep. 1994.
- [8] R. Agrawal, T. Imielinski, and A. Swami, "Database Mining: A Performance Perspective," *IEEE Transaction on Knowledge and Data Engineering*, special

issue on Learning & Discovery in Knowledge-Based Databases, Chile, Vol.5, No. 6, pp.914-925, Dec. 1993.

- [9] A. Amir, R. Feldman, and R. Kashi, "A New and Versatile Method for Association Generation," *Information Systems*, Vol. 22, No. 6/7, pp. 333-347, 1997.
- [10] N. F. Ayan, A. U. Tansel and E. Arkun, "An Efficient Algorithm to Update Large Itemsets with Early Pruning," *ACM SIGKDD Intl. Conf. on Knowledge Discovery in Data and Data Mining, San Diego, California*, pp. 287-291, Aug. 1999.
- [11] J. Ayres, J. E. Gehrke, T. Yiu, and J. Flannick, "Sequential PAttern Mining Using Bitmaps," *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. Edmonton, Alberta, Canada, July 2002*.
- [12] R. J. Bayardo Jr., "Efficiently Mining Long Patterns from Databases," *Proceedings of the 1998 ACM-SIGMOD International Conference on Management of Data*, pp. 85-93, 1998.
- [13] C. Bettini, X. S. Wang, and S. Jajodia, "Mining Temporal Relationships with Multiple Granularities in Time Sequences," *Data Engineering Bulletin*, Vol. 21, pp. 32-38, 1998.
- [14] S. Brin, R. Motwani, J. Ullman, and S. Tsur, "Dynamic Itemset Counting and Implication Rule for Market Basket Data," *Proceedings of the 1997 SIGMOD Conference on Management of Data*, pp. 255-264, 1997.
- [15] M. S. Chen, J. Han, and P. S. Yu, "Data Mining: An Overview from Database Perspective," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 8, No. 6, pp. 866-883, 1996.
- [16] M. S. Chen, J. S. Park, P. S. Yu, "Efficient Data Mining for Path Traversal Patterns," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 10, No. 2, pp. 209-221, 1998.

- [17] M. S. Chen, J. S. Park and P. S. Yu, "Data mining for path traversal patterns in a web environment," *Proceedings of 16th International Conference on Distributed Computing Systems*, pp. 385-392, May 1996.
- [18] D. W. Cheung, J. Han, V. Ng, and C. Y. Wong, "Maintenance of Discovered Association Rules in Large Databases: An Incremental Updating Technique," *Proceedings of 12th IEEE International Conference on Data Engineering*, pp. 106-114, 1996.
- [19] D. W. Cheung, S. D. Lee, and B. Kao, "A general incremental technique for maintaining discovered association rules," *Proceedings of the 5th International Conference on Database Systems for Advanced Applications*, pp. 185-194, 1997.
- [20] R. Cooley, B. Mobasher, and J. Srivastava, "Web Mining : Information and Pattern Discovery on the World Wide Web," *Proceedings of the 1997 IEEE International Conference on Tools with Artificial Intelligence*, pp. 558-567, 1997.
- [21] R. Feldman, Y. Aumann, A. Amir, and H. Mannila, "Efficient Algorithms for Discovering Frequent Sets in Incremental Databases," *2nd SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, May 1997.
- [22] J. L. Feng and Y. Feng, "Binary partition based algorithms for mining association rules," *Proceedings IEEE International Forum on Research and Technology –Advances in Digital Libraries (ADL'98)*, pp. 30-34, Apr. 1998.
- [23] Y. Fu and J. Han, "Metarule-guided Mining of Association Rules in Relational Databases," *Proceedings of the 1995 International Workshop on Knowledge Discovery and Deductive and Object-Oriented Databases*, Singapore, Dec. 1995.
- [24] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama, "Mining Optimized Association Rules for Numeric Attributes," *Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 182-191, 1996.

- [25] M. N. Garofalakis, R. Rastogi, and K. Shim, "SPIRIT: Sequential Pattern Mining with Regular Expression Constraints," *Proceedings of the 25th International Conference on Very Large Data Bases*, Edinburgh, Scotland, pp. 223-234, 1999.
- [26] J. Han, Y. Cai, and N. Cercone, "Data-Driven Discovery of Quantitative Rules in Relational Databases" *IEEE Transactions on Knowledge and Data Engineering*, Vol. 5, No. 1, pp.29-40, 1993.
- [27] J. Han, Y. Cai, and N. Cercone, "Knowledge Discovery in Databases: An Attribute-Oriented Approach," *Proceeding of the 18th VLDB Conference*, pp.547-559, 1992.
- [28] J. Han, J. Pei, and Y. Yin, "Mining Frequent Patterns without Candidate Generation," *Proceedings of the 2000 ACM SIGMOD Conference on Management of Data*, Dallas, Texas, USA, pp. 1-12, May 2000.
- [29] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal and M.-C. Hsu, "FreeSpan: Frequent Pattern-projected Sequential Pattern Mining," *Proceedings of the 6th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 355-359, 2000.
- [30] C. Hidber, "Online Association Rule Mining," *Technical Report UCB/CSD-98-1004*, Department of Electrical Engineering and Computer Science, University of California at Berkeley, 1998.
- [31] J. Hipp, U. Güntzer, and G. Nakhaeizadeh, "Algorithms for Association Rule Mining – A General Survey and Comparison," *SIGKDD Explorations*, Vol. 2, Issue 1, pp. 58-64, 2000.
- [32] M. Houtsma and A. Swami, "Set-Oriented Mining of Association Rules in Relational Databases," *Int'l Conference on Data Engineering*, pp. 25-33, Taipei, Taiwan, March 1995.
- [33] H. Kan, D. W. Cheung, and S. W. Xia, "Efficient parallel mining of association

- rules on shared-memory multiple-processor machine,” *IEEE International Conference on Intelligent Processing Systems*, pp. 1133-1137, Oct. 1997.
- [34] K. Koperski and J. Han, “Discovery of Spatial Association Rules in Geographic Information Databases,” *SSD*, pp. 47-66, 1995.
- [35] M. Klemettinen, H. Mannila, P. Ronkainen, and H. Toivonen, “Finding Interesting Rules from Large Sets of Discovered Association Rules,” *3rd International Conference on Information and Knowledge Management*, pp. 401-407, Nov. 1994.
- [36] C. M. Kuok, A. Fu, and M. H. Wong, “Mining Fuzzy Association Rules in Databases,” *SIGMOD Record*, pp. 41-46, Mar. 1998.
- [37] G. Lee, K.L. Lee and A.L.P. Chen, “Efficient Graph-Based Algorithms for Discovering and Maintaining Association Rules in Large Databases,” *Knowledge and Information Systems*, Springer-Verlag, Vol. 3, 2001, pp.338-355.
- [38] C.-H. Lee, P. S. Yu and M.-S. Chen, “Causality Rules: Exploring the Relationship between Triggering and Consequential Events in a Database of Short Transactions,” *Proceedings of the 2nd SIAM International Conference on Data Mining (SDM-02)*, April 11-13, 2002, pp. 403-419.
- [39] C.-H. Lee, C.-R. Lin and M.-S. Chen, “On Mining General Temporal Association Rules in a Publication Database,” *Proceedings of the First IEEE International Conference on Data Mining (ICDM-01)*, Nov. 29 – Dec. 2, 2001.
- [40] C.-H. Lee, C.-R. Lin and M.-S. Chen, “Sliding-Window Filtering: An Efficient Algorithm for Incremental Mining,” *Proceedings of the ACM 10th International Conference on Information and Knowledge Management (CIKM-01)*, Nov. 5-10, 2001, pp. 263-270.
- [41] S. D. Lee, D. Cheung, and B. Kao, "A General Incremental Technique For Maintaining Discovered Association Rules," *Proceedings of the 5th International*

Conference On Database Systems For Advanced Applications, pp. 185-194, Melbourne, Australia, Apr. 1997.

[42] C.-R. Lin, C.-H. Yun and M.-S. Chen, "Utilizing Slice Scan and Selective Hash for Episode Mining," *KDD-01 Workshop on Temporal Data Mining*, August 26-29, 2001.

[43] J. L. Lin and M. H. Dunham, "Mining association rules: anti-skew algorithms," *Proceedings 14th International Conference on Data Engineering*, Orlando, FL, USA., pp. 486-493, Feb. 1998.

[44] B. Liu, W. Hsu, and Y. Ma, "Mining Association Rules with Multiple Minimum Supports," *SIGKDD International Conference on Knowledge Discovery and Data Mining*, Aug. 1999.

[45] D. J. Lubinsky, "Discovery from Databases: A Review of AI and Statistical Techniques," *IJCAI-89 Workshop on Knowledge Discovery in Databases*, pp.204-218, Aug. 1989.

[46] H. Mannila, H. Toivonen and A. I. Verkamo, "Discovery of Frequent Episodes in Event Sequences," *Data Mining and Knowledge Discovery*, Vol. 1, Issue 3, pp. 259-289, 1997.

[47] H. Mannila and H. Toivonen, "Discovering Generalized Episodes using Minimal Occurrences," *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96)*, pp. 146-151, Portland, 1996.

[48] H. Mannila, H. Toivonen, and A. I. Verkamo, "Discovering Frequent Episodes in Sequences," *Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD'95)*, pp. 210-215, Montreal, Canada, 1995.

[49] H. Mannila, H. Toivonen, and A. I. Verkamo, "Efficient Algorithms for Discovering Association Rules," *KDD-94: AAAI Workshop on Knowledge Discovery in Databases*, Seattle, Washington, pp.181-192, July 1994.

- [50]H. Mannila, H. Toivonen, and A. I. Verkamo, “Improved Methods for Finding Association Rule,” *Report C-1993-65*,U. Helsinki 1994.
- [51]F. Masegla, F. Cathala, and P. Poncelet, “The PSP Approach for Mining Sequential Patterns,” *Proceedings of 1998 2nd European Symposium on Principles of Data Mining and Knowledge Discovery*, Vol. 1510, Nantes, France, pp. 176-184, Sep. 1998.
- [52]R. J. Miller, Y. Yang, “Association Rules over Interval Data,” *Proceedings ACM SIGMOD International Conference on Management of Data*, pp. 452-461,May 1997.
- [53]A. M. Mueller, *Fast Sequential and Parallel Algorithm for Association Rule Mining: A Comparison*, Technical report CS-TR-3515, University of Maryland, 1995.
- [54]B. Nag, P. M. Deshpande and D. J. DeWitt, “Using a Knowledge Cache for Interactive Discovery of Association Rules,” *Proceedings of Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Aug. 1999.
- [55]T. Oates, M. D. Schmill, D. Jensen, and P. R. Cohen, “A Family of Algorithms for Finding Temporal Structure in Data,” *Proceedings of the 6th International Workshop on AI and Statistics*, Fort Lauderdale, Florida, pp. 371-378, 1997.
- [56]J.-Z. Ouh, P. Wu, and M.-S. Chen, “Constrained Based Sequential Pattern Mining,” *Proceedings of International Workshop on Web Technology*, Dec. 4-6, 2001.
- [57]B. Ozden, S. Ramaswamy and A. Silberschatz, “Cyclic Association Rules,” *International Conference on Data Engineering*, 1998.
- [58]J. S. Park, M. S. Chen, and P. S. Yu, “Using a Hash-Based Method with Transaction Trimming for Mining Association Rules,” *IEEE Transactions on Knowledge and Data Engineering*, Vol. 9, No. 5, pp. 813-825, 1997.

- [59]J. S. Park, M. S. Chen and P. S. Yu, “Mining Association Rules with Adjustable Accuracy,” *IBM Research Report*, 1995.
- [60]J. S. Park, P. S. Yu and M. S. Chen, “Mining Association Rules with Adjustable Accuracy,” *Proceedings of ACM International Conference on Information and Knowledge Management*, pp. 151-160, 1997.
- [61]J. S. Park, M. S. Chen, and P. S. Yu., “An Effective Hash based Algorithm for Mining Association Rules,” *Proceedings ACM SIGMOD International Conference on Management of Data*, pp. 175-186 ,May 1995.
- [62]J. S. Park, M. S. Chen and P. S. Yu, “Efficient Parallel Data Mining for Association Rules,” *Proceedings of 4th International Conf. on Information and Knowledge Management*, pp. 31-36, Baltimore, Maryland, Nov. 1995.
- [63]S. Parthasarathy, S. Dwarkadas, and M. Ogihara, “Active Mining in a Distributed Setting,” *Large-Scale Parallel Data Mining*, Lecture Notes in Computer Science Vol. 1759, Springer- Verlag, pp. 65-82, 2000.
- [64]S. Parthasarathy, M. J. Zaki, M. Ogihara, and S. Dwarkadas, “Incremental and Interactive Sequence Mining,” *Proceedings of the 8th International Conference on Information and Knowledge Management*, Kansas, Missouri, USA, pp. 251-258, Nov. 1999.
- [65]J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang, “H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases,” *Proceedings of 2001 International Conference on Data Mining*, San Jose, CA, Nov. 2001.
- [66]J. Pei, J. Han, and R. Mao, “CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets,” *Proceedings of 2000 ACM-SIGMOD International Workshop on Data Mining and Knowledge Discovery*, Dallas, TX, May 2000.
- [67]J. Pei, J. Han, H. Pinto, Q. Chen, U. Dayal and M.-C. Hsu, “PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-projected Pattern Growth,” *Proceedings*

- of 2001 International Conference on Data Engineering, pp. 215-224, 2001.
- [68] J. Pei and J. Han, "Constrained Frequent Pattern Mining: A Pattern-Growth View," *SIGKDD Explorations*, Vol. 4, Issue 1, pp. 31-39, June 2002.
- [69] J. Pei, J. Han, and W. Wang, "Mining Sequential Patterns with Constraints in Large Databases," *Proceedings of the 11th International Conference on Information and Knowledge Management*, 2002.
- [70] H. Pinto, J. Han, J. Pei, K. Wang, Q. Chen, and U. Dayal, "Multi-Dimensional Sequential Pattern Mining," *Proceedings of the 10th International Conference on Information and Knowledge Management*, pp. 81-88, 2001.
- [71] V. Pudi and J. Haritsa, "Quantifying the Utility of the Past in Mining Large Databases," *Information Systems*, Vol. 25, N. 5, pp. 323-343, Jul. 2000.
- [72] P. Rolland, "FIEXPAT: Flexible Extraction of Sequential Patterns," *Proceedings of the IEEE International Conference on Data Mining 2001*, pp. 481-488, 2001.
- [73] A. Sarasere, E. Omiecinsky, and S. Navathe, "An Efficient Algorithm for Mining Association Rules in Large Databases," *Proceedings of the 21st International Conference on Very Large Data Bases*, Zurich, Switzerland, pp. 432-444, 1995.
- [74] P. Shenoy, J. Haritsa, S. Sudarshan, G. Bhalotia, M. Bawa, and D. Shah, "Turbo-charging Vertical Mining of Large Databases," *Proceedings of the 2000 ACM SIGMOD Conference on Management of Data*, Dallas, Texas, USA, pp. 22-33, May 2000.
- [75] T. Shintani and M. Kitsuregawa, "Mining Algorithms for Sequential Patterns in Parallel: Hash Based Approach," *Proceedings of the Second Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pp. 283-294, 1998.
- [76] T. Shintani and M. Kitsuregawa, "Parallel Mining Algorithms for Generalized Association Rules with Classification Hierarchy," *Proceedings of ACM SIGMOD Conference*, pp. 25-36, 1998.

- [77] T. Shintani, and M. Kitsuregawa, "Hash based parallel algorithms for mining association rules," *Proceedings of 4th International Conference on Parallel and Distributed Information Systems*, FL, USA, pp. 19-30, Dec. 1996.
- [78] Shrividya, "DELTA: A Fast Algorithm for Incremental Mining of Association Rules," *Project report*, Dept. of Computer Science and Automation, Indian Institute of Science, 1997.
- [79] A. Siberschatz and A. Tuzhilin, "On Subjective Measure of Interestingness in Knowledge Discovery," *Proc. 1st Int'l Conf. Knowledge Discovery and Data Mining*, pp.275-281, 1995.
- [80] R. Srikant and R. Agrawal, "Mining Sequential Patterns: Generalizations and Performance Improvements," *Proceedings of the 5th International Conference on Extending Database Technology*, Avignon, France, pp. 3-17, 1996. (An extended version is the IBM Research Report RJ 9994)
- [81] R. Srikant, Q. Vu, and R. Agrawal, "Mining Association Rules with Item Constraints," *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining (KDD'97)*, pp. 67-73, 1997.
- [82] R. Srikant and R. Agrawal, "Mining Quantitative Association Rules in Large Relational Tables," *Proceedings of the ACM SIGMOD Int'l Conference on Management of Data*, pp. 1-12, 1996.
- [83] R. Srikant, R. Agrawal, "Mining Generalized Association Rules," *Proceedings of the 21th VLDB Conference Zurich*, Switzerland, pp. 407-419, 1995.
- [84] S. Thomas, S. Sarawagi, "Mining Generalized Association Rules and Sequential Patterns Using SQL Queries," *Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining (KDD'98)*, pp. 344-348, 1998.
- [85] H. Toivonen, "Discovery of Frequent Patterns in Large Data Collections," *Ph.D. thesis*, University of Helsinki, Finland, 1996.

- [86] H. Toivonen, "Sampling Large Databases for Association Rules," *Proceedings of the 22nd International Conference on Very Large Data Bases*, pp. 134-145, 1996.
- [87] P. S. M. Tsai, C. C. Lee, A. L. P. Chen, "An Efficient Approach for Incremental Association Rule Mining," *Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining 1999*, pp. 74-83.
- [88] S. Tsur, "Data Dredging," *IEEE Data Engineering Bulletin*, Vol. 13, No. 4, pp. 58-63, Dec. 1990.
- [89] K. Wang, "Discovering Patterns from Large and Dynamic Sequential Data," *Journal of Intelligent Information Systems*, Vol. 9, No. 1, pp. 33-56, 1997.
- [90] K. Wang and J. Tan, "Incremental discovery of sequential patterns," *Proceedings of Workshop on Research Issues on Data Mining and Knowledge Discovery*, Montreal, Canada, June 1996.
- [91] M. Wojciechowski, "Interactive Constraint-Based Sequential Pattern Mining," *Proceedings of the 5th East European Conference on Advances in Databases and Information Systems*, pp. 169-181, 2001.
- [92] P. H. Wu, W. C. Peng, and M. S. Chen, "Mining Sequential Alarm Patterns in a Telecommunication Database," *Proceedings of VLDB-01 Workshop on Databases in Telecommunications 2001*, pp. 37-51, 2001.
- [93] S. J. Yen and A. L. P. Chen, "An Efficient Approach to Discovering Knowledge from Large Databases," *Proceedings of 4th International Conference on Parallel and Distributed Information Systems*, pp. 8-18, Dec. 1996.
- [94] S. J. Yen and A. L. P. Chen, "An efficient data mining technique for discovering interesting association rules," *Proceedings of Database and Expert Systems Applications. 8th International Conference (DEXA '97)*, pp. 664-669, Sep. 1997.
- [95] S. J. Yen and A. L. P. Chen, "A Graph-Based Approach for Discovering Various

Types of Association Rules,” *IEEE Transactions on Knowledge and Data Engineering*, Vol. 13, No. 5, pp.839-845.

- [96] M. J. Zaki, "Fast Mining of Sequential Patterns in Very Large Databases," *Technical Report 668*, The University of Rochester, New York, Nov. 1997.
- [97] M. J. Zaki, “Sequence Mining in Categorical Domains: Incorporating Constraints,” *Proceedings of the 9th International Conference on Information and Knowledge Management*, Washington D.C., pp. 422-429, 2000.
- [98] M. J. Zaki, “SPADE: An Efficient Algorithm for Mining Frequent Sequences,” *Machine Learning Journal*, Vol. 42, No. 1/2, pp. 31-60, 2001.
- [99] M. J. Zaki, “Efficient enumeration of frequent sequences,” *Proceedings of the 7th International Conference on Information and Knowledge Management*, Washington, USA, pp. 68-75, Nov.1998.
- [100] M. J. Zaki and C. Hsiao, “CHARM: An Efficient Algorithm for Closed Association Rule Mining,” *RPI Technical Report 99-10*, 1999.
- [101] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li, “New Algorithms for Fast Discovery of Association Rules,” *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, Newport, California, pp. 283-286, Aug. 1997.
- [102] M. Zhang, B. Kao, D. Cheung, and C.-L. Yip, “Efficient Algorithms for Incremental Update of Frequent Sequences,” *Proceedings of the 6th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pp. 186-197, 2002.

Vita

Ming-Yen Lin was born on March 31, 1966 in Kaohsiung, Taiwan, Republic of China. He received the BS degree in Computer Engineering and the MS degree in Computer Science and Information Engineering both from National Chiao Tung University, Taiwan, in 1988 and 1990, respectively. After that time, he was a software engineer, in charge of system/VGA BIOS of PCs, in Mitac Inc. In 1991, he changed the job and worked for CTXOPTO Electronics Co. He had completed the design of the firmware/software of LCD monitors/projectors, established the Internet networking infrastructure of the company, introduced Oracle Database/Applications and EDI systems, formulated the ISO9000 management system of MIS department, and developed new techniques (which becomes an US patent 6,326,961) for the R&D department. Later in 1998, he quitted the job and served as an adjunct lecturer in Providence University for one semester. Starting from 2003, he is an adjunct lecturer both in Feng Chia University and Taichung HealthCare and Management University. He is currently working towards the Ph.D. degree in National Chiao Tung University. His research interests include data mining, database systems, data stream management systems, bioinformatics, and semantic Web.